Multiple Priorities in a Two-Lane Buffered Crossbar

Nikos Chrysos and Manolis Katevenis[◇]
Institute of Computer Science - Foundation for Research and Technology - Hellas (FORTH)
ICS-FORTH, P.O. Box 1385, Vassilika Vouton, Heraklion, Crete, GR-711-10 Greece
http://archvlsi.ics.forth.gr/bufxbar/ - {nchrysos,katevenis}@ics.forth.gr
March 2004

***Abstract*— A significant advantage of buffered crossbar (combined input-crosspoint queueing - CICQ) switches is that they can directly operate on variable-size packets, thus saving the costs and inefficiencies of packet segmentation and reassembly (SAR). However, in order to support multiple priority levels, separate queues per priority are needed at each crosspoint, in order to prevent HOL blocking and buffer hogging; these queues are expensive because they each need a size of at least one maximum-size packet. In this paper we propose a scheme that uses only two queues per crosspoint to effectively support multiple priorities. We adaptively adjust the priority levels of the two queues so that most traffic goes through the "lower" queue, while the "upper" queue remains usually available for higher priority packets to overtake the former. Through simulation, and assuming 8 priority levels, we compare our scheme to an ideal system that uses 8 queues per crosspoint. For realistic traffic, the two systems perform almost identically, although ours uses 4 times less memory in the crossbar. Even under a highly irregular traffic pattern Bursts60, our system will not increase the average delay of any priority level by more than 75 percent compared to the ideal system.**

1 . INTRODUCTION

Combined Input-Crosspoint Queueing (CICQ), also called *Buffered Crossbar*, is a packet switch architecture that has recently become popular, because it features simple and efficient scheduling [1] [2] [3] [4], including weighted fair scheduling [5] [6], and owing to advances in CMOS technology that made its implementation feasible. Compared to the traditional bufferless crossbar with input (virtual output) queueing (VOQ), CICQ is efficient enough (when crosspoint buffers are larger than a few cells each) to the point where internal speedup [7] is no longer needed to compensate for scheduling inefficiencies; note that speedup is still necessary to cope with the throughput increase that results when variable-size packets are segmented into fixed-size cells.

Zhang [1], Christensen [8], and the present authors [9] observed that buffered crossbars have the additional advantage of being able to operate directly on *variable-size* packets, without requiring prior segmentation into fixed-size cells. In [9] we showed that such architectures can dramatically reduce cost: internal speedup is no longer needed –neither to compensate scheduling inefficiencies, nor to cope with packet segmentation and reassembly (SAR); in turn, output queues are not needed, given the lack of speedup and packet reassembly. In exchange for these major cost savings, one has to bear the relatively minor cost of larger crosspoint buffers: each of them needs to fit, at least, not only one flow-control window – as they always needed to– but also one maximum-size packet, additionally. When packet size is limited up to 1500 bytes (e.g. packets that have crossed an Ethernet link), crosspoint buffers in the range of 2 KBytes each are needed; these are feasible in modern technology for crossbars with 24 to 32 ports [9].

Multiple *priority levels* are desirable as a means for service differentiation and quality of service (QoS); e.g. IEEE 802.ID/Q defines eight classes of service by means of priorities [10]. Proper priority mapping, at the application or packet level, can provide efficient utilization of network resources and increase the "user-perceived" utility [11]. A CICQ switch needs a separate queue per priority (class of service), in order to prevent head-of-line (HOL) blocking, and each of these queues needs a minimum reserved space in order to prevent buffer hogging (sec. 2.2).

This paper deals with supporting multiple priority levels in a buffered crossbar that directly operates on *variable-size* packets. Each crosspoint buffer must contain multiple, per-priority queues. However, these queues can*not* dynamically share the buffer space, because that would require queues implemented via linked lists of dynamically-allocated memory blocks [4], thus re-introducing packet segmentation and the need for speedup. The solution is to use circular queues, each of them implemented inside a static partition of the buffer memory. This has the advantage of trivially solving the buffer hogging problem (each queue has a statically allocated and reserved space), but has the disadvantage of requiring large crosspoint buffers: one flow-control window (e.g. 512 bytes) plus one maximum-size packet (e.g. 1500 bytes), *times* the number of priority levels (e.g. 8), amounting to 16 KBytes per crosspoint in this example, or 16 MBytes in a 32×32 switch chip.

The goal of this paper is to solve the above memory space problem: a 32×32 variable-packet-size buffered crossbar chip in a modern 0.13-micron ASIC technology can fit 2 MBytes of 2-port SRAM memory (2 KBytes per crosspoint) [9], but 8 times that much memory (for 8 priority levels) is too expensive. We develop and evaluate an innovative architecture, whereby *just two queues* per crosspoint support as many priority levels as desired, with an efficiency not much worse than that of separate, per-priorirty queues. For the above typical numbers, using our new architecture, a 24×24 multi-priority switch chip is feasible today (4 KB per crosspoint, or

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

© ICS-FORTH, MARCH 2004

2.3 MB total), and a 32×32 chip (4 KB per crosspoint, 4 MB total) can be implemented using embedded DRAM today or will be feasible in 2005-2006 using 2-port SRAM.

The basic idea of our new architecture is to *dynamically and adaptively map* the multiple priority levels onto the two existing queues, for each crosspoint buffer. The mapping is such that the *"lower queue"* usually contains packets of the top-most non-empty priority level, while the *"upper queue"* is usually empty, thus being available for the high-priority packets that *occasionally* appear to quickly bypass the lower-priority traffic. We call this a two-lane system, because its operation is analogous to a two-lane highway, where cars drive in one lane and overtake using the other.

Section 2 starts by describing a baseline architecture, with a separate lane for each priority level, used for comparisons. Section 2.1 reviews the problems of HOL blocking and of buffer hogging. Sections 2.2 and 2.3, present our general methods to reduce their negative effects: (1) Packet *push-out* resolves HOL blocking and buffer hogging, even with one queue per crosspoint, at the expense of increased delay relative to baseline; (2) *wait-to-drain* reduces that delay, by pro-actively policing the traffic sent to the crosspoint queues. Section 2.4 describes our overall system, *2Q*, with two queues per crosspoint. In section 3, we show through simulation, that when eight priorities are supported, even under highly irregular traffic, *2Q* will not increase the average delay of the top-most priority level by more than **75** percent, when compared to a system with a separate crosspoint queue for each priority (i.e. **4** times more buffering); intermediate priorities experience much smaller discrepancies. Under smoother patterns, like typical network traffic, we find that the two systems perform almost identically (the same holds for Poisson arrivals). Section 5 discusses the hardware complexity of our methods and section 6 concludes.

To the best of our knowledge, this is the first published study of how to effectively map multiple priority levels onto a reduced number of queues in a buffered crossbar –and perhaps in any kind of switch. The importance of the paper stems from this novelty and from the importance of buffered crossbars –especially variable-packet-size ones– as a likely emerging architecture of choice for future commercial crossbar products.

2 . System & Methods

The system considered in this paper is a buffered crossbar that supports L priority levels ($L > 2$) and directly switches variable-size packets. Separate, per-priority virtual output queues (VOQ) are maintained at the input line cards: $N \cdot L$ VOQ's per line card for an $N \times N$ switch. Each crosspoint buffer contains two (2) queues, as shown in figure 1, used to support multi-priority operation. We compare this system of ours to a traditional, "*baseline*" system that contains L queues per crosspoint, each of them statically allocated to a specific priority level. We assume that each crosspoint queue –in either system– has S_q buffer memory space statically allocated to it.

Credit-based flow control provides lossless transmission between the input line cards and the crosspoint queues. Credits

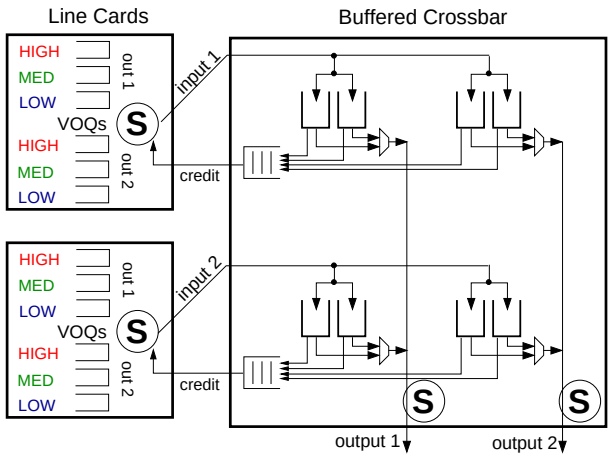

Fig. 1. The system assumed in this paper, supporting three priority levels.

destined to the same input are sent in FIFO order; the credit rate per input line card is one credit every minimum-packet-size (S_{min}) time. For flow control to be deadlock-free, the size of each crosspoint queue must be $S_q \geq S_{max}$, where S_{max} is the size of one maximum-size packet. In order to sustain full output-link utilization even with a single active flow, we need an $S_q \geq S_{max} + R \cdot t_{RTT}$, where R is the line rate and t_{RTT} is the round-trip time from the generation of a credit till the first word of a packet that utilized this credit at the input is transmitted on an output line of the crossbar [9].

Each input and output line has a non-preemptive, priority, round-robin scheduler: when queues of different priorities are eligible for service, the highest priority is selected; if multiple queues are eligible at that priority, we use round-robin to select one of them; round-robin is on a byte-count basis (equalizes, in the long-term, the number of bytes served from each back-logged queue). Crosspoints provide cut-though.

2.1. HOL Blocking and Buffer Hogging

At the crosspoints, separate queues per priority are desired in order to provide flow isolation and protection. When a crosspoint queue is shared among multiple priorities, a high-priority packet queued behind a low-priority one may suffer excessive delay, because the output scheduler may postpone serving the low-priority packet, due to its low priority, not knowing that a high-priority packet is waiting behind it; this is known as *head of line (HOL) blocking* –figure 2. Even if each priority level has a dedicated *logical* queue, but these queues *share* a common buffer space, a similar effect can occur if low-priority packets fill up the entire shared space: high-priority packets will have to wait at the line card, due to unavailability of buffer space (no flow-control credits), and thus the high-priority crosspoint queue will appear empty to the output scheduler; this is known as *buffer hogging*. The multi-priority buffered crossbar in [4] uses multiple queues in a shared space, at each crosspoint, and is thus subject to buffer hogging.

In the worst case, either HOL blocking or buffer hogging may lead to *starvation* for a high-priority flow, if its packets

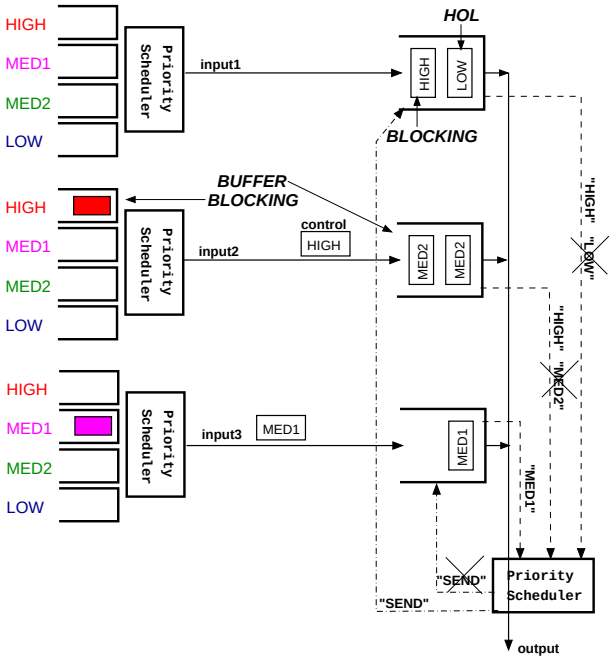

Fig. 2. HOL blocking, buffer hogging, and how we resolve them with packet push-out and special control packets.

happen to wait behind low-priority packets at the relevant crosspoint: if sufficient medium-priority traffic exists from other inputs to the same output, the output scheduler will keep selecting the latter, thus starving the low-priority packets – which is OK– but also, indirectly, starving the high-priority flow too. Our system queues variable-size packets; thus, as mentioned in section 1, our queues can*not* dynamically share buffer space with each other, because that would require packet segmentation, hence speedup. Our solution is to statically allocate space to each crosspoint queue, thus solving the buffer hogging problem too.

2.2. *High Pushes Low Out of the Way*

In our system with two lanes (two queues per crosspoint) – like in any system with less lanes than priority levels– there are cases when a queue must be shared among packets of different priorities. To resolve the HOL blocking problem that may arise in such cases, we employ a *push-out* discipline: the "effective priority" of a crosspoint queue, for output scheduling purposes, is the highest of the priorities of the packets currently queued in it. Figure 2 illustrates this in the simple case of one lane: a high-priority packet arrives in the top queue, but it is blocked behind a low-priority packet; rather than wait, the high-priority packet "pushes" the low-priority one out of the queue, by declaring to the scheduler that this queue now has high priority. This case is analogous to an ambulance behind a normal car in a single-lane road: as long as there is no space for the car to pull out of the way, that car has to rush ahead at the "effective priority" of the ambulance. The disadvantage of the push-out discipline is that some low-priority packets –e.g. "low" in figure 2– depart before other, higher-priority packets, thus

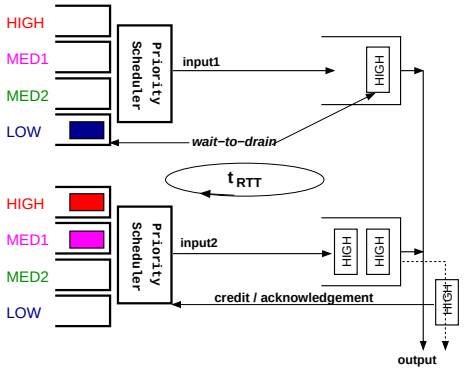

Fig. 3. The wait-drain-discipline: Congestion at higher-priority levels can be speculated at the input by means of unacknowledged higher-priority packets

increasing their delay, and this happens *not just* for packets in the same queue, but also for higher-priority packets in other crosspoints of the same column –e.g. "med1" and "med2" in figure 2.

The same problem can appear in the buffer hogging form, as illustrated in the middle row of figure 2: that queue is filled up, so when the high-priority packet arrives at the line card it cannot even reach the crosspoint buffer so as to raise its effective priority, due to lack of credits. To resolve this, we assume that the line card sends a special *control packet* to signal the necessary priority upgrade to the crosspoint, rather than sending the actual packet itself. Control packets are not stored in crosspoint queues –they merely change the effective priority of the queue– but they are subject to scheduling at the input, and they are transmitted through the same link as normal packets. We assume their size to be equal to a minimum-size packet, so that scheduling rate is not affected.

2.3. *Low Waits for High to Drain*

When a queue must be shared among packets of different priorities, the converse of the above situation occurs when the next packet to be sent from the line card to the switch has a priority level below the previous packet that was sent.

In this case, we observed by simulation that the following heuristic pays off, on the average. The line card adopts a *non-work-conserving* discipline: first wait for all higher-priority packets to drain out of the crosspoint queue, and only then send the low-priority packet, provided of course that it still is the "top-of-the-list" packet in the line card. We attribute this gain to the following effect, illustrated in figure 3: the presence of high-priority packets destined to a specific output in the part of the system that is visible by a line card (the upper line card in figure 3) –i.e. in the line card itself or at the corresponding crosspoint– is an indication of current network activity at this high-priority level, destined to this specific output: more packets at this level for this output may exist in other crosspoints (from the lower line cards in figure 3), or the arrival of more of them, belonging to a same burst, through the same input, may be imminent. Under these circumstances, if the line card refrains, for a while, from forwarding background

(low-priority) traffic to the crosspoint queue, it will give the newly-arriving high-priority packets a chance to bypass the low-priority ones on the way to the shared queue.

The precise policy that we implemented in our simulated system, under the name *wait-to-drain*, is as follows: as long as the top candidate priority, from an input line card to a specific crosspoint queue, is below the highest *unacknowledged-packet* priority level in that queue, the scheduler in the line card does *not* send any packet to that queue. A packet remains *unacknowledged* until the line card receives the credit that corresponds to that packet, indicating that the packet has started departing from the crosspoint [1]. Note that if congestion is not actually present, then the method can cause output under-utilization, if the unacknowledged high-priority packet, that causes the scheduler at the input to remain idle, has size smaller than $R \cdot t_{RTT}$. Our simulation results indicate not significant throughput reduction under the wait-to-drain discipline. More importantly, this reduction affects only the lower-priority levels, while the higher levels benefit considerably.

2.4. Two Queues with Adaptive Mapping

Having established methods to react to HOL blocking and buffer hogging, we next consider a system with two lanes (queues) per crosspoint –see figure 1. Call *UP* one of these lanes, and *DOWN* the other. Input scheduling is as follows: at each input line card, for each output, consider the highest-priority non-empty VOQ, and decide whether that VOQ is candidate for transmission to the UP lane, or to the DOWN lane, or to none of the two. These lane assignments are performed on a per packet basis, i.e., priority levels are not statically assigned to specific crosspoint lanes. The input scheduler selects one of the eligible VOQs, as described in section 2.

Our goal is for high-priority packets to use the "upper" queue to bypass lower priority packets that may be blocked in the "lower" queue. Hence, the lower queue is the default lane, and the upper lane is used as auxiliary. To keep the upper lane uncongested, for all flows except top-most priority ones, we are conservative when mapping their packets UP, to account for future packet arrivals of even higher priority. In addition to this simple strategy, our method uses the *push-out* and the *wait-to-drain* disciplines, described in the previous sections.

The output scheduler considers all the non-empty UP and DOWN queues in its column, and serves the one with the highest effective priority; when both the UP and DOWN queues in a crosspoint have equal effective priority, it chooses UP. At any time, the input considers that the effective priority of a queue equals to the priority of the highest priority packet that is currently unacknowledged inside it. By EffPr[{UP,DOWN}] we denote the effective priority of a crosspoint queue, as seen at the input.

[1] A positive by-product of the wait-to-drain policy is to simplify the circuit that keeps track of the effective-priority of a crosspoint queue: that priority rises upon arrival of higher-priority packets, and only drops (to the bottom-most level) when the last packet departs from the queue. Without wait-to-drain, a mechanism would be needed to find the highest priority level among all remaining packets after departure of a packet at the previously-highest priority.

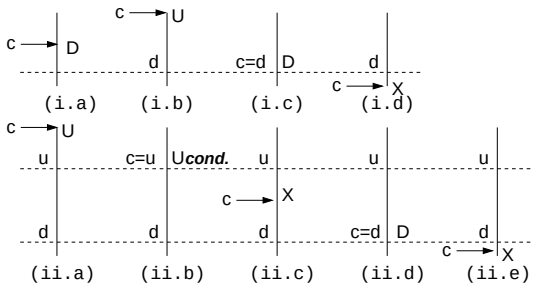

Fig. 4. The candidate priority level, c, is mapped either UP (U), or DOWN (D), or nowhere (X), depending on its priority and on the effective priority of the two lanes, as estimated at the input.

Figure 4, illustrates our 2Q discipline. Call c the priority level of the *candidate* VOQ, at the input line card, for a specific crosspoint, as defined earlier in this section 2.4. If both lanes are empty, (i.a), then VOQ c is mapped DOWN. Next (cases (i.b), (i.c), (i.d)), assume that the upper lane is empty, while EffPr[DOWN] = d (as seen on the line card). If $c > d$, (i.b), then c is mapped UP to bypass the lower-priority packets that are pending DOWN. If $c = d$, (i.c), then c keeps using DOWN, as it has already been doing. If $c < d$, (i.d), then c is ineligible because "low waits for high to drain" (section 2.3).

In the second row of figure 4, both lanes have packets inside them, and EffPr[UP] $= u$, EffPr[DOWN] $= d$. Normally, $u > d$, except for rare situations not considered here –see [12], fig. 18. In case (ii.a), c has higher priority than both lanes and is mapped UP. If not enough credits exist for c to use UP, a special *control* packet is sent to force the upper lane to drain with effective priority c, as discussed in section 2.2. In case (ii.b), $c = u$ and thus c will continue using UP, but only if an additional condition is satisfied, aimed at keeping the upper lane uncongested, as discussed shortly, in section 2.5; if the condition fails, we consider VOQ c ineligible for UP, and possibly for DOWN as well. In (ii.c), when $d < c$ and $c < u$, c is ineligible for UP, and may also be ineligible for DOWN –see section 2.5. In case (ii.d), $c = d$, then c keeps using DOWN, as it has already been doing. Finally, in (ii.e), $c < d$, and VOQ c is marked as ineligible.

2.5. Keeping the upper lane uncongested

A crucial point of our 2Q discipline is to keep the upper lane empty most of the time [2]; to achieve this, we "hesitate" to route consecutive packets of a flow through that upper lane, as noted above and as marked by *Ucond* in figure 4(ii.b). A simple discipline to do that is to allow only a limited number, say K, of *unacknowledged* packets of a flow to be pending UP (see section 2.3 for the term "unacknowledged"). We simulated this discipline for $K = 1$, called *noRTT*, and found that it reduces the delay of the top priority levels. However, for small K or for short packets –relative to t_{RTT}– this discipline will place a quite low bound on the throughput that its flow can achieve.

[2] except for packets of the top-most priority level

To address this issue, our 2Q discipline uses the round-trip time, t_{RTT}, to define how many consecutive packets a flow is allowed to send UP: record the time, t_u, when a flow sent its first packet UP; allow this flow to send UP until $t_u + t_{RTT}$; after that, wait for lane UP to drain before a new t_u is recorded.

In case (ii.b) when the above condition fails, and in case (ii.c) (figure 4), instead of considering c ineligible, we can send c packet(s) through the DOWN lane; we call this policy *try-down (TD)*. Through simulations we observed that *TD* deteriorates the performance for almost all of the intermediate priority levels. We attribute this behavior to the following. In the cases under discussion, the upper lane contains unacknowledged packets; this is an indication of heavy traffic at that priority level, and hence even heavier traffic at the DOWN level. TD sends c DOWN, but it is likely that c could have gone through the UP lane earlier that the time it will take to reach the HOL of the DOWN queue, given that, in the second case, c will have to "push-out" lower-priority packets that currently reside in the DOWN lane.

2.6. In-Order Delivery

Our two-lane system always forwards the packets of a flow in-order, because we never send a packet of a flow UP when another packet of the same flow is pending unacknowledged DOWN. Since the UP queue is always selected first by the output scheduler when both queues have the same effective priority, if a packet is sent UP and a subsequent packet is sent DOWN, the oldest one will depart first. Note that the effective priority of the DOWN queue cannot increase after the second packet is sent DOWN, e.g. because of push-out: any packet with priority high enough to push-out the second packet would be mapped UP.

3 . SIMULATIONS

3.1. Simulation Enviroment

We used the same simulator as in [9] to experiment with our methods. In this paper we simulate a 32×32 switch with port speed 10 Gbps. For simplicity we assume no internal packet-header and consequently no speedup. The VOQs and the crosspoint queues implement cut-through. Finally, the t_{RTT} is set equal to 400*ns* (or 500 byte time).

We explicitly model variable-size packet arrivals. Packets' destinations are uniformly distributed and all priorities arrive with equal probability. The uniform priorities distribution, that has also been used in [4], stands for a worst-case benchmark: when the "higher" priorities arrive less frequently than the "lower" ones, buffer hogging and HOL blocking become less pervasive.

Regarding packet arrivals, we use the **Poisson** process, and a bursty one, **Bursts60**, which exhibits high temporal and spatial locality. In **Bursts60**, a burst consists of sixty (**60**) back-to-back packets and the length of idle periods is exponentially distributed. Packets within a burst have the same destination and the same priority. The size of each packet is always selected independently using a Pareto distribution: S_{ave} 400 bytes, S_{min} 40 bytes and S_{max} 1500 bytes. This gives us an average burst size equal to 23 KBytes. The **Bursts60** pattern models worst-case real traffic scenarios, and additionally reveals the vulnerability of a system to HOL blocking and buffer hogging.

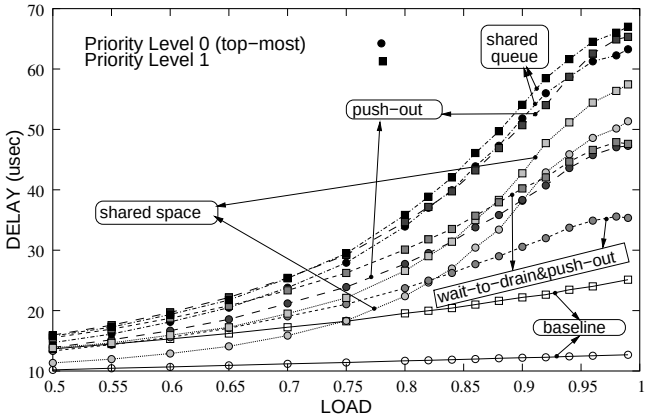

Fig. 5. Four (**4**) priority levels; **Bursts60** arrivals; Delay of levels p0 and p1 shown.

We plot packets' average delay just-before-output-service, for each priority in separate, versus the aggregate input load. By "p0" we denote the top-most priority level, and by "p3" or "p7" we denote the bottom-most one, depending on the number of priority levels that we simulate (4 or 8 respectively).

3.2. One Queue Per Crosspoint

We start by verifying the known negative effects of HOL blocking and buffer hogging: we compare the ideal *baseline* system, which has a separate queue/lane for each priority level at each crosspoint, against: (*shared-queue*) a system with a single queue per crosspoint; (*push-out*, *wait-to-drain&push-out*) shared-queue systems augmented with the disciplines discussed in section 2.2 and 2.3; and (*shared-space*), a system with multiple logical queues per crosspoint, one for each priority level, sharing a common crosspoint buffer space. Each separate crosspoint queue has dedicated space equal to 2 KBytes, i.e. slightly larger than the minimum required (see section 2). To model *shared-space*, we use a single credit counter at each input for each corresponding crosspoint, that keeps track of the available buffer space of *all* logical queues in that crosspoint; this buffer space is set equal to 2 KBytes.

In this set of experiments, we use **Bursts60** arrivals and four (**4**) priority levels. Figure 5 presents the average delay of priority levels p0 and p1, and figure 6 presents the average delay of p2 and p3. As figure 5 demostrates, as the input load increases, the performance degradation in all shared-queue (and shared-space) systems becomes dramatic [3].

At the one extreme, with *baseline*, the average delay of the top-most priority level (p0) is not affected by the increase of the aggregate input load, while at the other extreme, *shared-queue* cannot discriminate low from high priorities, and as the

[3]We should note that considerable performance degradation is present even under less irregular traffic patterns [12], but certainly to a smaller extent.

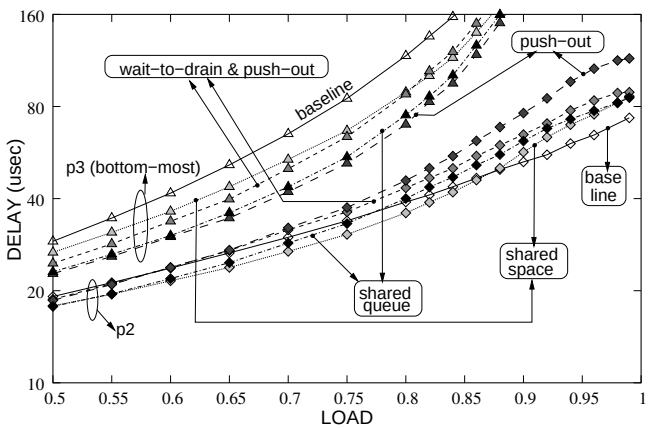

Fig. 6. Four (**4**) priority levels; **Bursts60** arrivals; Delay of levels p2 and p3 shown; Y axis in logscale.

input load increases all priority levels experience more or less the same quality of service. At low input load (i.e., 0.5 to 0.7), *shared-space* performs better than *push-out* and *wait-to-drain&push-out*, owing to that buffer hogging does not appear frequently when the crosspoint buffers are relative empty. It performs much worse though, at higher input load, when the crosspoint buffers fill more frequently. Finally observe that *wait-to-service&push-out* performs much better than *push-out* alone: it slightly increases the average delay of the bottom-most priority level (p3), in benefit of the higher priority levels.

3.3. Two Queues Per Crosspoint

Under four (**4**) priority levels and **Poisson** arrivals, 2Q performs *identically* to the *baseline*, while under **Bursts60** arrivals, small discrepancies occur when the input load is higher than 0.8; these discrepancies grow with increasing load, but in our results [12], these never exceed 50%, for any priority level. The results that we present in this section are for eight (**8**) priority levels that are even more difficult to handle. In figure 7 we used the **Poisson** arrivals model. As we can see from the figure, 2Q, performs very close to the *baseline*; the larger discrepancies appear at the top-most priority level and are bellow 15% under any input load.

Next, we experiment with the **Bursts60** arrivals model. We again compare 2Q with the expensive *baseline* system; we also examine the noRTT alternative of 2Q, that we describe in section 2.5. The results of this set of experiments are presented in two figures: figure 8 contains plots for the even priority levels, while figure 9 contains the plots for the odd priority levels. Two things are worth noticing in these figures. First, the maximum discrepancy of our method, 2Q, compared to the *baseline* appears at the top-most priority level, and it is close to 75% under 0.99 input load (19 vs 11 *usec* delay, see fig. 8); concerning lower priority levels the two system perform very close. Second, under the noRTT policy, the respective maximum discrepancy drops down to 35%. This happens because the noRTT policy is even more conservative when it uses the upper queue, and thus, high-priority packets

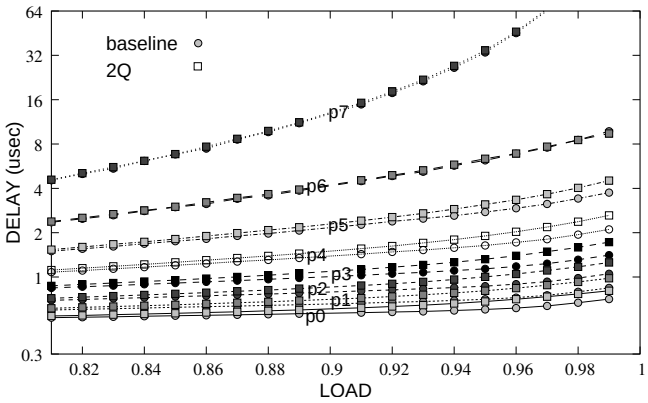

Fig. 7. Eight (**8**) priority levels; **Poisson** arrivals; Y axis in logscale.

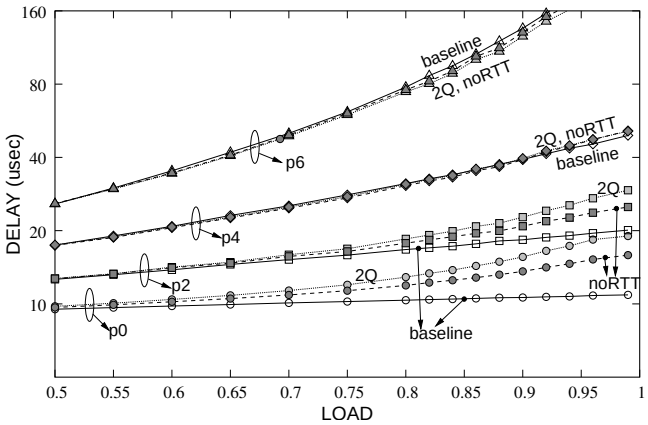

Fig. 8. Eight (**8**) priority levels; **Bursts60** arrivals; Delay of even levels shown; Y axis in logscale.

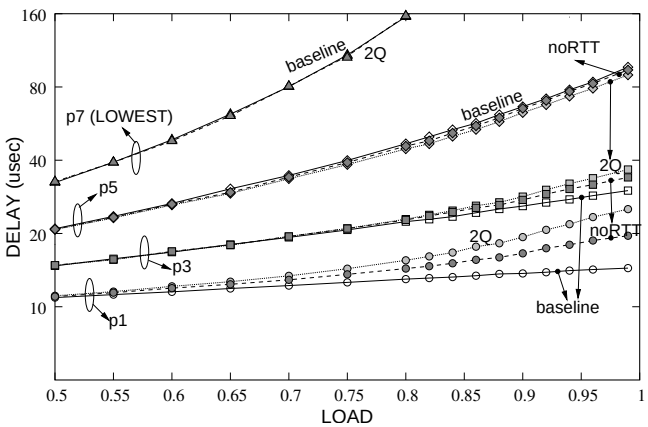

Fig. 9. Eight (**8**) priority levels; **Bursts60** arrivals; Delay of odd levels shown; Y axis in logscale.

more frequently find a free way to bypass the low-priority ones.

Finally, in figure 10, we use a synthetic traffic pattern, **SynthBackb**, that tries to emulate as much as possible backbone, realistic IP traffic. In synopsy, under the **SynthBackb** pattern, the packet arrivals at an input line card are generated by multiplexing thousands of interactive (IC) and bulk (BC)

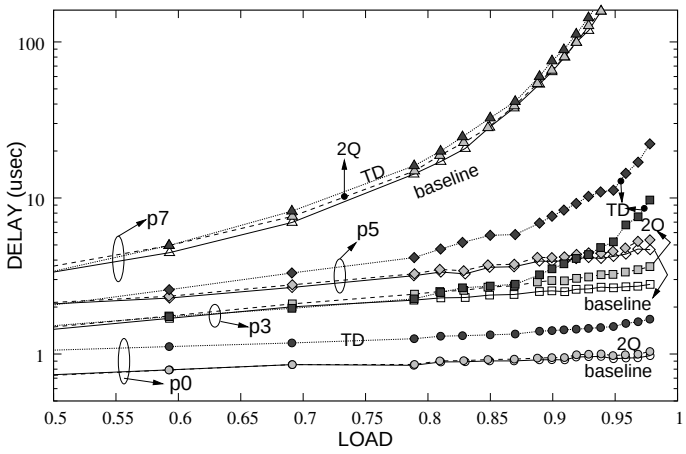

Fig. 10. Eight (**8**) priority levels; **SynthBackb** arrivals; Delay of levels p0, p3, p5 and p7 shown. Y axis in logscale.

"conversations" in a FIFO queue; an IC is modeled as a Poisson process that sends 125 packets with size 40 to 44 bytes, while a BC is modeled as a burst with average size 8 KB [9]. A uniform distribution is used to map each generated conversation to one out of eight priority levels. Figure 10, contains delay plots for the baseline system, for 2Q and for TD, an alternative of 2Q which is described in section 2.5. To make the figure readable, we present delay plots only for the more interesting priority levels. As the figure demostrates, 2Q performs very close to the ideal *baseline* system, while the TD alternative policy increases the delay of all priority levels, except for the bottom-most one (p7). Section 2.5 explained the believed reasons for this behavior.

4 . HARDWARE COST

Other than the two queues per crosspoint and the priority-aware output schedulers, the only additional circuits required by our method within the crossbar chip is a register for each crosspoint queue, holding the current effective priority of that queue. Regarding the input line cards, we need a mechanism to determine the current candidate priority level, and whether the candidancy is valid or not, per crossbar output. This mechanism must run at most three times per S_{min} interval, i.e., at packet or credit arrival, and at packet departure. The critical path of this mechanism (see [12], fig. 18) consists of three to four comparisons between small values.

5 . CONCLUSIONS

We presented a novel, multiple priority, variable-packet-size CICQ switch with only two queues per crosspoint. Our system, is capable to provide fine-grained QoS support with no speed-up and no output buffers. The simulation results presented indicate, that more than two queues per crosspoint are not worth the associated cost, especially given the large size of queues required for variable-packet-size operation. Our "two-lane" system performs close to a system with separate, per-priority queues: under a worst-case scenario the maximum discrepancy is 75%, whereas in the typical case it is only 15%.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

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

[6] G. Georgakopoulos: "Few buffers suffice: Explaining why and how crossbars with weighted fair queuing converge to weighted max-min fairness", http://archvlsi.ics.forth.gr/bufxbar/

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

[8] K. Yoshigoe, K. Christensen: "A Parallel-Polled Virtual Output Queued Switch with a Buffered Crossbar", *Proc. IEEE Workshop High Perf. Switching & Routing 2001*, Dallas, TX, USA, May 2001, pp. 271-275;

[9] Manolis Katevenis, Giorgos Passas, Dimitris Simos, Ioannis Papaefstathiou and Nikos Chrysos "Variable Packet Size Buffered Crossbar (CICQ) Switches", to appear in Proc. of ICC 2004. http://archvlsi.ics.forth.gr/bufxbar

[10] V. Fineberg: "A Practical Architecture for Implementing End-to-End QoS in an IP Network", IEEE Communications Magazine, vol. 40, no. 1, Jan. 2002, pp. 122-130.

[11] Wu-chang Feng et. al. "Adaptive Packet Marking for Providing Differentiated Services in the Internet", *Proc. of Int. Conf. on Network Protocols* Oct. 1998;

[12] Nikos Chrysos: "Design Issues of a Multiple-Priority, Variable-Size-Packet Buffered Crossbar" *Technical Report, ICS FORTH Hellas, Department of Computer Science, University of Crete, October 2004*; http://archvlsi.ics.forth.gr/bufxbar