Preventing Buffer-Credit Accumulations in Switches with Small, Shared Output Queues

Nikos Chrysos, Manolis Katevenis[‡]

Inst. of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH) - member of HiPEAC

***Abstract*— We consider a switch with small output queues, shared among the input VOQ linecards. This has been shown to be a useful abstract model for realistic buffered switching fabrics. Cells are being scheduled by a central control unit, comprising independent, single resource schedulers, working in pipeline. This unit allocates output buffer credits to the requesting VOQs. We show how particular unbalanced transient VOQ states, produced by bursty traffic, affect credit reservations: when some input temporarily constitutes a bottleneck, too many credits may get reserved for it at once, leading to poor overall performance. We propose a threshold grant throttling method to control these credit accumulations. Then, we show how, under such grant throttling, typical round-robin credit schedulers can get synchronized, thus deteriorating performance. To avoid scheduler synchronization, we propose modified round-robin disciplines. Simulations under both smooth and bursty traffic demonstrate the effectiveness of the combined method: using only a 12-cell buffers per-output, for any switch size, N, and independently of the number of cells in transit between the linecards and the fabric, the performance achieved is very close to that of pure output queueing. We also discuss the operation of the independent input and output schedulers inside the control unit, their relation with PIM-like schedulers, and their relation with buffered crossbar schedulers.**

1 . Introduction

Networks need fast and low-cost packet switches to keep pace with the increase in communication demand. Switches employ input and output linecards, which usually contain sizable buffer memories, and a core, which is a crossbar or a switching fabric. Packet switch architectures belong to two principal categories, depending on their core: *bufferless* or *buffered*.

Buffered architectures ease scheduling by allowing conflicting packets to enter the fabric. The combined input crosspoint queueing (CICQ) switch (or buffered crossbar) receives considerable research attention because it features simple and efficient scheduling, and also because it uses memories running at the line rate [1] [2] [4] [5]. However, these benefits come at the expense of a memory intensive fabric core. The internal memory of a buffered crossbar is proportional to $N^2 \cdot \lambda \cdot RTT$, where N is the switch valency, RTT is the round-trip time between the input linecards and the crossbar, and λ is the line rate. (The $\lambda \cdot RTT$ factor accommodates the cells in transit between the linecards and the fabric, and is commonly referred to as FC window.) The PRIZMA chipset [6], a commercial product designed by IBM, features the same scheduling architecture with buffered crossbars; although PRIZMA uses a shared-memory, it still has the same memory requirements.

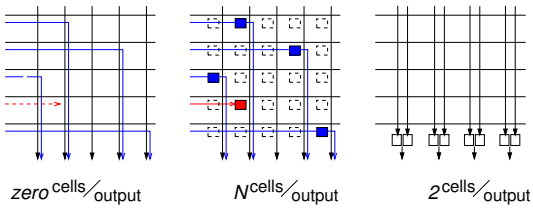

Fig. 1. Placement of buffers in a crossbar. Starting from the left, we have a bufferless crossbar, then a (buffered) crossbar containing N cell buffers per output, and, last, on the right, a system with two cell buffers per output. In this paper we consider the latter type of switches (or fabrics), i.e., buffered, with less than N cell buffers per output.

Our recent work [7] showed that we can do equally well with considerably less than N^2 FC window buffers –see fig. 1. The key understanding point is that with storage for only a few cells inside the fabric, approximate crossbar matchings become feasible: the injected cells may conflict up to the degree that the excess cells fit in the fabric memories. In turn, approximate matchings can easily be produced by independent, pipelined single-resource schedulers, similar to that in buffered crossbars. A 2-cell buffer per-output suffices for this type of scheduling, and yields better performance than 1-SLIP [9]. With a 12-cell buffer per-output, performance is better than that of buffered crossbars, and approaches that of pure output queueing (OQ). Most importantly, the above buffer sizes are *independent* from the FC window size, and, additionally, seem to be independent from the switch size, N.

Besides their theoretical importance, these results are of interest primarily for large, multi-stage fabrics, built out of several smaller switches. Even if each switch is internally organized as a buffered crossbar, the available buffer space on the path to each fabric output will be significantly smaller than the number of input linecards; for such fabrics, the type of buffer scheduling that we evaluate in this paper appears as a viable, new solution [10]. Also, these results may be of future interest for all-optical fabrics: the new architecture saves considerably in buffer space while using a very simple scheduling unit, and achieving excellent performance.

However, the evaluation in [7] was for smooth (Bernoulli) traffic only. In this paper, we augment that study considering *bursty* traffic also. The final result in this paper is that 12-cell

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

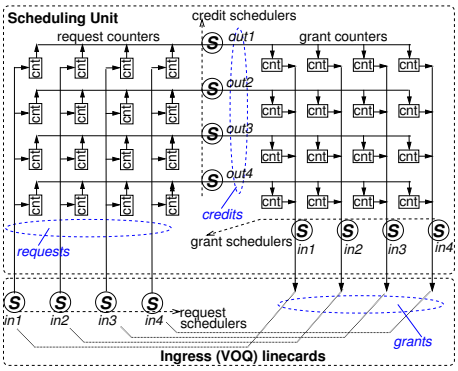

Fig. 2. The scheduling control unit of a VOQ switch with small, shared output queues; request, credit, and grant schedulers form a 3-stage pipeline, with request and grant counters being the pipelining registers.

buffers can yield excellent performance even when the traffic is highly bursty. To achieve this goal, only minor modifications are incorporated in the scheduling subsystem, to prevent and control buffer-credit accumulations.

1.1 Contributions

First, we discuss the operation of the independent, per-output, credit schedulers in this architecture (section 2). We conjecture that performance degrades when these schedulers "overshoot" too many credits to some inputs. To remedy this behavior we need a statistical type of desynchronization among the output schedulers, as opposed to the deterministic desynchronization needed in bufferless crossbars, and as opposed to buffered crossbars, which, thanks to their per flow buffers, work well even under fully synchronized schedulers.

Statistical desynchronization is trivially maintained under smooth traffic, but gets more difficult under heavy, bursty traffic, when VOQs transition between relatively long periods of empty and non-empty states. In section 3, we describe particular unbalanced transient states that deteriorate credit reservations: a bottleneck input tends to accumulate credits, leaving other inputs with no credit at all. We attribute the large delays observed under bursty traffic to the aforementioned transients, and we propose a simple grant throttling scheme that maintains excellent performance.

Next, in section 4, we describe a second-order behavior induced by grant throttling: when *all* VOQs are persistently active (100% Bernoulli load), typical round-robin credit schedulers may synchronize vastly, thus yielding new credit accumulations. We describe this phenomenon in detail, and we present several alternative round-robin disciplines that remain desynchronized. Extensive simulations (sec. 5) are used to demonstrate the effectiveness of our methods. With buffers as small as 12 cells per output, excellent performance is achieved under both bursty or smooth traffic. Even when the average burst length is on the order of one hundred, simulations demonstrate throughputs close to 99%, and delays virtually identical to pure OQ.

2 . Switch Architecture

Storage for cells is provided in the ingress part of the switch, inside large virtual output queues (VOQs) maintained in the linecards. A small, contention resolution buffer, with capacity for B cells, is placed before each output port of the switch. All ports in and out of the switch run at the line rate: we do not use any internal speedup. Hence, we do not need queues in the egress linecards, either.

Before injecting a cell inside the fabric, a VOQ must first reserve a slot (secure a credit) in the buffer of the corresponding output port. For that purpose, a *request scheduler* in each linecard issues one request per cell time towards the scheduling control unit. The control unit comprises N (per-output) *credit* schedulers, and N (per-input) *grant* schedulers. Requests are routed to the credit scheduler that corresponds to the intended output. This scheduler keeps track of the available buffer space in front of that output using a credit counter, and issues one credit per cell time to one of the requesting inputs. Outstanding requests are kept in *request counters*, organized per (input-output pair) flow. Unmatched inputs waiting for credit are allowed to send new requests to the same or to other outputs; thus, multiple credits from different outputs can be issued concurrently to the same input. All credits issued to an input are collected inside the control unit and the *grant scheduler* for that input, selects one among them and sends it to the ingress linecard, as a *grant* that allows cell injection into the switch. (Each grant scheduler sends grants at a rate of one grant per cell time.) The remaining grants are kept in per flow *grant counters*. We consider that credit and grant schedulers reside in the same chip.

As described in [7], using *credit prediction*, once a credit is selected by its grant scheduler, and the grant is issued to the linecard, the credit can safely return to its credit counter, and the corresponding credit scheduler can reuse it: given that no backpressure is exerted from the egress linecards to the output ports of the switch, we know that each output buffer empties by one cell per cell-time, hence we can predict the corresponding credit generation one RTT ahead of time. Effectively, the buffer flow control feedback delay comprises just the delay of a request going through credit and grant scheduling. We denote this delay by SD. We assume that each such scheduler takes a full cell-time to produce a new outcome (thus SD=2 cell-times), therefore a 2-cell buffer per output (B=2 cells) suffices for the credit flow control operation.

Upon receiving a $i \rightarrow j$ grant, linecard i forwards the head cell of VOQ j to the switch. Besides cell forwarding, this grant allows for a new $i \rightarrow j$ request to be sent, if VOQ $i \rightarrow j$ contains enough cells: At start time, each VOQ is allowed to send up to u requests before receiving any grant back. After u requests have been sent, the request scheduler refrains new $i \rightarrow j$ requests until a $i \rightarrow j$ grant arrives. This request control equalizes a VOQ's request rate with its grant rate; it also limits the number of bits that credit and grant counters need to have. Obviously u needs to compensate for the scheduling RTT, which equals two times the propagation delay between the

linecards and the control unit (P), plus SD, i.e., $2 \cdot P + SD$.

2.1 Operation & Throughput of the Independent Schedulers

When B=1 cell, output credit schedulers reduce to link schedulers, and the scheduling control unit reduces to a bufferless crossbar scheduler; actually, it performs identical to 1-SLIP, thanks to desynchronization –see fig. 3(a). As in bufferless crossbars, a complete pipeline operation, comprising both (output) credit and (input) grant scheduling, has to complete within one cell time ($SD \leq 1$ cell time). With slightly larger buffers, the situation changes dramatically.

Independent schedulers in bufferless fabrics: Bufferless crossbar schedulers [11][9], using one iteration of handshaking, achieve acceptable performance either when contention is low, or when their link schedulers fix in some effective matching sequence. For example, desynchronization makes each output scheduler granting a different input in each consecutive cell time, and different than that all other schedulers grant, even though all schedulers work independently; however, this desynchronization is accomplished only under uniform traffic, and only after *all* VOQs have built up, thus introducing large delays. To achieve good performance, schedulers for bufferless crossbars typically need to perform multiple iteration per decision; the reason is that whenever two or more outputs grant to the same input, and the conflict is not resolved in subsequent iterations, throughput is wasted because there remain underutilized output lines.

In bufferless crossbar schedulers, all conflicting output grants except the accepted one are dropped. Consider what would happen if one such grant, g, produced by output o, was kept instead of being dropped. If output o were *not allowed* to issue a new grant before g got accepted by its input scheduler, then output o would stay underutilized for all that time; on the other hand, if output o were *allowed* to issue new grants, then more than one grants for the same output could be accepted concurrently by multiple (independent) input schedulers, thus violating the output constraint of the bufferless crossbar. For this reason, unaccepted grants are normally dropped, and many iterations or speed-up are used to increase match size –essentially to match links with unaccepted grants[1].

Independent, pipelined schedulers for buffered fabrics: When $B \geq 2$, output grants correspond to buffer slots, thus there is no conflict when multiple inputs concurrently receive grants for the same output. In effect, an output can issue new grants to inputs even before being notified that its previous grants have been accepted, and grant buffering at the inputs becomes safe. With grant buffering, a multi-cell-time scheduling pipeline is formed. Each single resource scheduler in this pipeline produces a new valid outcome in every cell-time (SD=2 cell-times), just to keep its line busy, without needing to communicate anything with other schedulers while in the middle of this cell-time operation. Essentially, scheduling operations can run *asynchronously* in different schedulers. Each one picks the state communicated by other schedulers before starting a new operation, and communicates back to them its new outcome when it finishes.

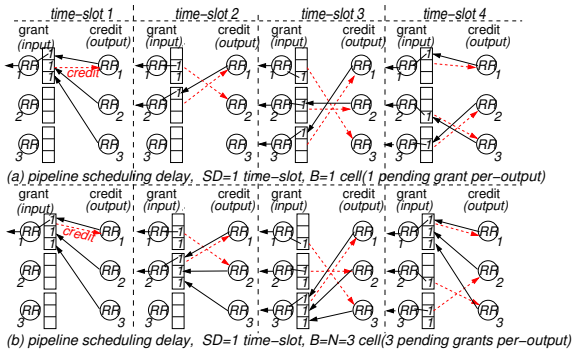

Fig. 3. (a) when B=1, schedulers deterministically desynchronize and achieve 100% throughput. (b) when B=N, 100% thoughput is achieved even under fully synchronized schedulers.

Schedules become feasible thanks to the flow control of the *output* buffers: during any time interval T, the aggregate traffic injected for any given output, from any number of inputs, will always be $\leq \lambda \cdot T + B$ (measured in cells), where λ is the output line rate. Buffers need to be associated with one particular output each so that (a) buffer flow control also controls the congestion of output lines (b) we no longer need exact crossbar matches.

2.2 Statistical Desynchronization

Using simulations, we have observed that when the load approaches 100%, all credits are allocated to inputs, and credit counters are usually null (empty). This happens due to random grant conflicts, which unavoidably delay the return of some credits. Therefore, as credit schedulers are greedy, issuing one new credit per cell time, soon their whole available credit moves to the input grant queues. On average, one new credit is generated per output per cell time, and that is immediately issued to one of the requesting inputs[2].

With large buffers, for $B \geq N$, throughput is not wasted even if output schedulers get completely synchronized –see fig. 3(b). However, for intermediate values of B, we need credit schedulers to desynchronize to some extent. This desynchronization need not be deterministic, but can be of *statistical* nature. Since there are a total of $N \cdot B$ credits available, and only N inputs, each input will normally have an adequate backup of credits to use even if it does not get a new grant in some specific cell time. To this end, output pointers do not have to fix in some particular arrangement, as in bufferless

[1] pipelined schedulers for bufferless crossbars [13] [14] reduce timing constraints, but are cumbersome to build. The intrinsic difficulty of crossbar schedulers compared to schedulers for crossbars with multi-cell output buffers should be tracked down to the underlying observation: it is easier to build an approximate than an exact match.

[2] this explains why increasing the rate at which output schedulers hand out credits does not improve the delay at high loads, as indicated in [7]. Such increments are ineffective as the schedulers rate is limited by the rate that credits are available, i.e. one per cell time, per output. However, when the load is low, (as credits are usually available), a faster credit scheduler can serve faster the "rare" sets of inputs that request its service at about the same time, thus marginally improving the delay.

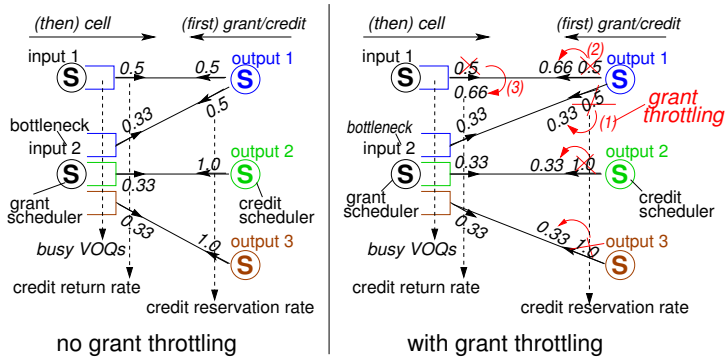

Fig. 4. An unbalanced transient with input 2 being congested. The busy (non-empty) VOQ shown are the ones shown; each busy VOQ corresponds to a busy request queue. The figure shows the credit reservation rates (assuming that credits are always available), and the credit return rates; (a) with no grant throttling, and (b) with grant throttling.

crossbars; they simply need to avoid degenerate, synchronized behavior.

3 . Throttling Grants to a Bottleneck Input

This section discusses credit accumulations under bursty traffic, and proposes a simple method to control them.

3.1 Unbalanced Transients with Congested Inputs

In any switch, there are situations where some flows are "bottlenecked" at their inputs, rather than at their outputs: Packets may first accumulate at inputs due to output contention; then output contention may go away, and multiple packets could depart simultaneously towards multiple outputs if they were not constrained by the limited bandwidth of the input buffer and linecard. Unbalanced transients with bottlenecked inputs appear continously under statistically multiplexed flows (even when the long-term load is feasible), but get more severe under bursty and heavy traffic. When output contention subsides for a while, a congested input may receive multiple grants from different outputs at about the same time.

The problem with bottlenecked inputs is that they tend to accumulate an abnormaly high number of credits, thus limiting service to other inputs. For example, consider figure 4, where both inputs 1 and 2 have requests pending for output 1, and input 2 has requests pending for two additional outputs –an unbalanced VOQ demand with input 2 constituting a bottleneck. Obviously, the credit scheduler for output 1 would do better serving input 2 once every three cell times, giving preference to input 1. However, the round-robin credit schedulers we consider, being oblivious of immediate input contention, steer credits as if all requesting inputs are equally loaded, giving birth to the dynamics we describe next.

Assume that whenever output 1 has credits available, it throws a coin to decide which input to serve. Credits issued to input 1 are recycled after one cell time, whereas credits issued to input 2 are recycled after three cell times, on average. Obviously, all output 1 credits will soon pile up in front of the grant scheduler for input 2. The service rate of input 1, x, will be x=1/2 * y, where y is the rate at which output 1 holds some credit. Obviously, y = 1/3 + x. Solving the above equations yields the suboptimal rates x = 1/3 and y=2/3. We have verified these rates by simulation.

Fortunately, owing to the equalization of VOQ request and grant rates, this inefficient credit allocation cannot last for long, as request queue 2→1 will eventually empty. However, the drain time of a request queue can be quite long, considering that u, its peak size, must be set in proportion to the propagation delay between the linecards and the scheduler, P. While the transient is active, VOQs and delay grow.

One straighforward solution to diminish credit accumulations is to limit the total number of pending requests from any given input, effectively decreasing input contention among the reserved credits. However, an input could then consume its allowable requests sending them to congested outputs, thus not being able afterwards to request other, possibly lightly loaded destinations.

3.2 Grant Throttling Using Thresholds

To control credit accumulations, the key idea is for credit schedulers to stop serving inputs that do not return credits fast enough. The way we want grant throttling to work is presented in fig. 4.

Let $GQ(i)$ denote the cumulative grant queue size for input i. (This is the sum of the current values of all grant counters corresponding to that input.) Our mechanism changes the eligibility of input i for credit schedulers, taking $GQ(i)$ into account: a request from input i is eligible at its output (credit) scheduler, iff *(a)* output buffer credits are available (this was always a requirement), plus additionally *(b)* $GQ(i)$ is less than a threshold, TH. Implementing threshold grant throttling requires that each grant scheduler i circulates a common *On/Off* signal to all credit schedulers, that stays *Off* whenever $GQ(i) \geq TH$.

Certainly, threshold TH must be set below B, since the average GQ is $\leq B$. To make the method more reactive, a low threshold is appropriate. On the other hand, too low a threshold may put at risk schedulers' flexibility: if all inputs block, output schedulers will not be able to produce new grants, and buffer space may be underutilized.

3.3 Effectiveness of Grant Throttling under Bursty Traffic

Figure 5 presents delay performance when B=12 cells, under uniform, bursty traffic. (The simulation environment is presented in section 5, together with additional results.) We can see that, when no grant throttling is employed, or when TH>B (TH-15), the delay of our switch is quite above that of pure output queueing (OQ). But, by using a threhold below B (TH = 3, 5, and 7) delay essentially matches that of OQ.

4 . Rare But Severe Synchronizations

Using threshold grant thottling, the (cumulative) queue size in front of any grant scheduler will always be $\leq TH$+N-1. An input may end up with these credits, if all (N) output schedulers serve it in synchrony when it is just below the threshold, TH. We do not want an input to end up with that

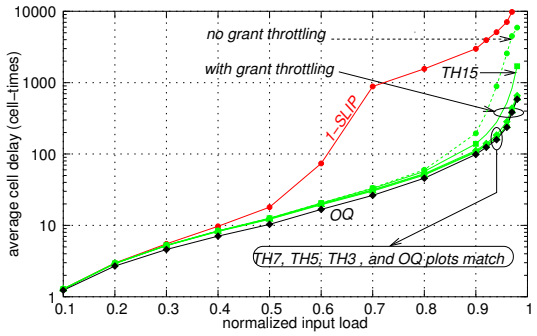

Fig. 5. Performance under bursty traffic, for different grant throttling thresholds, TH; N=32, B=12; average burst length 12 cells.

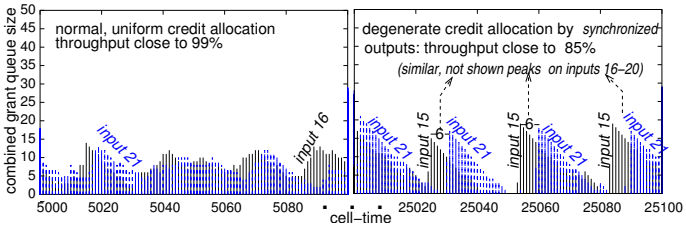

Fig. 6. Evolution of inputs' combined grant queue size (GQ), while simulating a 32×32 switch, with B=12 cells and TH=7, under uniform Bernoulli traffic at 100% input load. For readability, we show only inputs 15 and 21. Before cell time 20000 (left plot), when synchronization has not yet prevailed, $GQ(15)$ and $GQ(21)$ evolve more or less normally, and throughput stays high (above 98%) . But after cell time 20000 (right plot), synchronization is so severe that each input in turn is granted by as many as 20 outputs. The time-axis distance between the peaks in the graphs of inputs 15 and 21 is 5 cell times. In these 5 cell times, inputs 16-20 "collect" credits, one after the other, identically to inputs 15 and 21. The figure on the right also shows that from time to time some input stays with no grant, which explains why throughput can drop as low as 85%.

many credits, because it would then hold roughly $1/B$ of the total credits in the system, which need to be distributed to as many as N inputs, where $B<N$. But why should output credit schedulers get that coordinated?

As we discuss in this section, threshold grant throttling, not only does not prevent synchronizations, but it may indirectly induce them. This section (a) describes this phenomenon, (b) shows that it is due to all credit schedulers using the same round-robin order, while it is also assisted by threshold grant throttling, and (c) proposes simple and efficient ways out of it.

We will use the following terminology. We say that two or more credit schedulers *clash* when they issue credits to (serve) the same input at the same time; we say that two or more credit schedulers are *synchronized*, when their "next-to-serve" pointers point the same input.

Common-Order Round-Robin: The default round-robin discipline that we consider for credit scheduling is as follows. To determine which input to grant to, a scheduler scans among inputs 1 to N, circularly, starting from the input indicated by a *next-to-serve* pointer. The first eligible input found, e, is served, and next-to-serve advances to $(e+1)$ modulo N. If no input is eligible, the next-to-serve pointer stays intact. In the baseline implementation, all (per-output) credit schedulers *share a common ordering of inputs*. Similarly to *i*SLIP, our system achieves deterministic, full desynchronization when B=1, thanks to this common ordering.

4.1 Synchronization Evolution

Using common-order round-robin credit schedulers, we have observed severe synchronizations under uniform, Bernoulli arrivals, at 100% load, when all VOQs become persistent[3]. Under these traffic conditions, the synchronized schedulers clash, thus hurting switch throughput.

The cooperation of several independent behaviors is responsible for bringing the next-to-serve pointers into degenerate situations. When the threshold TH is modest ($<B$) and the load is high, it is reasonable to expect that some inputs will occasionally be *Off* due to random grant conflicts. Assuming that the N next-to-serve pointers of the credit schedulers are (still) in random, uniform positions, it is very likely that the very first *On* input, say input i, whose order is next to a small sequence of consecutive *Off* inputs, will receive more grants than usual. The reason is that all outputs pointing somewhere in this sequence of idling inputs will stop "scanning" on input i. After being granted by these many outputs, input i is likely to turn *Off*. In this way, in the next cell time, input i will participate in a new sequence of idling inputs, therefore reproducing the same phenomenon. But even worse, the output schedulers that served input i in cell time t will be synchronized at input $i+1$, in cell time $t+1$. Through this procedure, more and more output schedulers synchronize –see fig. 6.

(It is interesting to note here that we have not observed severe synchronizations under bursty traffic. This must be due to VOQs fluctuations: at any time instance, each input has requests pending only for a subset of outputs, possibly different than that of other inputs, thus (output) credit schedulers cannot severely synchronize. Using threshold grant throttling, the throughput of the switch under uniform bursty traffic is close to 99%, whereas under uniform Bernoulli traffic throughput may drop as low as 85%.)

4.2 Preventing & Breaking Synchronizations

Since we want to preserve grant throttling in order to deal with unbalanced transients, we next look for alternative scheduling disciplines that avoid synchronizations. We found that *fair queueing (FQ)* and *random* credit scheduling do not suffer from synchronizations of this kind. However, these schemes increase complexity, and we do not study them further in this paper.

Synchronization emerges because, after a random grant conflict on input i in cell time t, the clashing outputs synchronize again on input $i+1$ in cell time $t+1$. In other words the source of synchronizations is the *common ordering* in which all credit schedulers visit and serve the input ports. But round-

[3]these are the very same conditions that produce beneficial desynchronization in *i*SLIP

robin operation does not presume a common or fixed order of service!

Random-Shuffle Round-Robin Orders: Our first method tackles "common ordering". Each output credit scheduler is preprogrammed with an ordering of inputs produced by some random shuffle of the N inputs to the N positions in the round-robin "frame". In this way, even if pointers happen to clash in one cell time on some input, say on input a, they will not necessarily synchronize in the next cell time, for each output sees a different physical input as next to input a. We name this method *random-shuffle*.

Inert Round-Robin Pointers: Another simple method maintains common ordering of inputs, but modifies the update policy of next-to-serve pointers. Assume that this pointer had the value p and that, searching from p onwards ended up serving input i. Then, instead of updating p to (i+1)mod N, we update p to (p+1)mod N. We name this scheme *inert round-robin*.

Essentially, inert round-robin is a heuristic method. On the negative side, if the distance between p and i is large, the same input i is likely to be serviced repeatedly while p gets incremented by 1 every cell time, until it reaches i; however notice that all inputs between p and i are ineligible, hence these either have not issued requests or have already received many credits, thus it may be OK for i to be serviced multiple times. On the positive side, schedulers will only synchronize in cell time $t+1$ if they were already synchronized in cell time t. Assuming an one-to-one initial mapping of the N output pointers to the N inputs, two outputs will synchronize when one of them had been busy $k \cdot N + d$ more times than the other, where $k \in Z$, and d is the distance between the initial values of the two pointers.

Desynchronized Clocks: A nice modification to the previous scheme that *deterministically* desynchronizes outputs is the following. As with inert round-robin schedulers, at start time each output points to a distinct input. From that point on, in each cell time, either all output pointers advance by one, or all stay still. It is trivial to see that if the N "next-to-serve" pointers go like this, hand-by-hand, they will never synchronize. It should be made clear though that clashes can still occur when some connections are inactive.

The particular distributed policy that we propose advances *all* pointers in *every* cell time, irrespective of whether each particular scheduler served a flow or not. We call this discipline *desynchronized clocks*.

As we will see in the simulation results, the aforementioned methods avoid severe scheduler synchronizations, and achieve 100% throughput under uniform load. Among them, only the *random-shuffle* method keeps unchanged the underlying round-robin discipline. The other two methods, *inert* and *desynchronized clocks*, may be unfair under particular circumstances. Consider for instance that an output is oversubscribed by two persistent flows, one from input 0 and one from input 1. It is easy to verify that *inert* and *desynchronized clocks* will allocate a bandwidth of $1/N$ to input 1, and a bandwidth of $(N-1)/N$ to input 0.

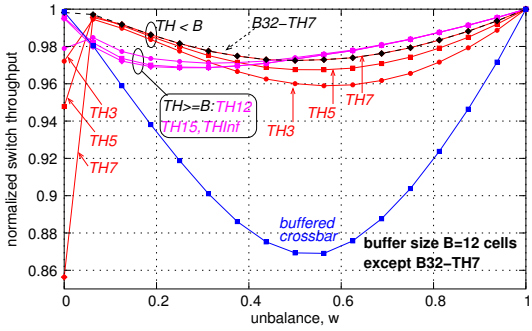

Fig. 7. Throughput under unbalanced traffic for varying buffer size, B, and varying threshold values, TH, using common-order round-robin credit scheduling; N=32; Bernoulli arrivals.

5. Simulation Results

The performance of the switch was evaluated under uniform and unbalanced traffic patterns by means of simulations, and was compared to that of pure OQ, buffered crossbar, and *i*SLIP. No internal speedup was used in any switch. In the unbalanced traffic experiments, we used Bernoulli arrivals, because these produce the synchronization behavior we described in section 4. Simulations were run long enough to eliminate the effects of initial transients –depending on switch size, input load, and burstiness factor[4], a few thousands up to tens of millions of "start" cells were discarded before gathering statistics–, and the confidence intervals achieved were better than 10% around the reported values with confidence 95% for delay and 1% for throughput. We measure average delay of cells in cell times. Unless otherwise noted, we allow for a very large number of pending requests per VOQ, u= 10000, in order to study the behavior of credit and grant schedulers in isolation. The propagation delay (P) and the scheduling delay (SD) are both set to zero, thus the minimum recorded cell delay equals zero (0). Our previous results [7] demonstrate that non-zero P and SD values merely add some constant offset to the reported delay.

5.1 Throughput under UnBalanced Traffic

Figure 7 depicts switch throughput for Bernoulli cell arrivals at 100% load. As in [2], traffic unbalance is controlled by w: when w=0 traffic is uniform, whereas when w=1 traffic consists of persistent, input-output, non-conflicting connections. In this experiment, we use common-order round-robin schedulers, and we vary the grant throttling threshold, TH; all plots are for B=12 cells, except for one, $B32$-$TH7$.

When the load is uniform (w=0), the normalized throughput of configurations with $TH<B$ drops quite below 1, due to the synchronizations discussed in section 4. (Only $B32$-$TH7$ keeps performing well, as it contains sufficient buffer space – N cells– per output to cope with synchronized schedulers.)

[4]bursty traffic is based on a two-state (ON/OFF) Markov chain; ON periods (consecutive, back-to-back cells arriving at an input for a given output) last for at least one cell-time, whereas OFF periods may last zero cell-times, in order to achieve 100% loading of the switch. The state probabilities are calibrated so as to achieve the desirable load, giving exponentially distributed burst lengths.

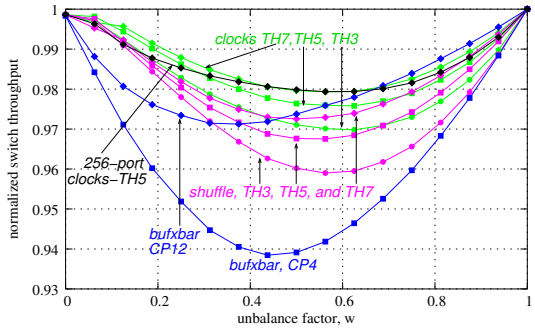

Fig. 8. Throughput under unbalanced traffic for varying threshold, TH, under different credit scheduling disciplines; B=12 cells; all switches have 32 ports, except one, 256-port $clocks$-$TH5$; Bernoulli arrivals.

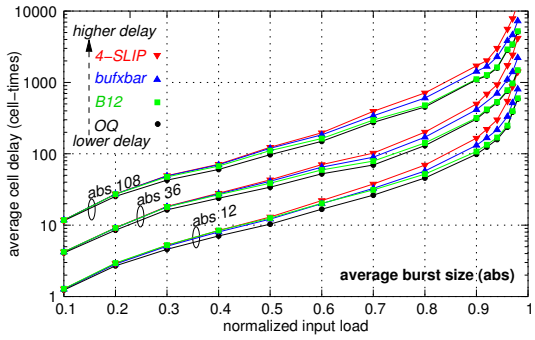

Fig. 9. Performance under varying average burst size (abs), for OQ, *i*SLIP with 4 iterations, buffered crossbar with 1-cell crosspoint buffers, and for the switch we propose in this paper with B=12, TH=5, and desynchronized clock credit schedulers; N=32.

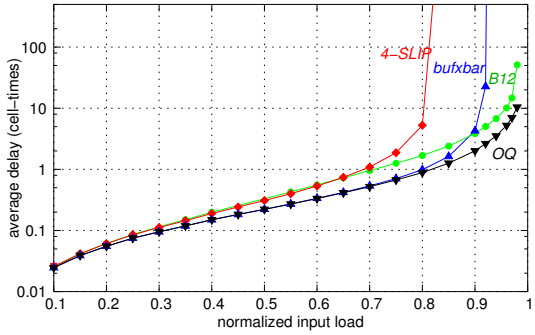

Fig. 10. Performance under diagonal traffic for OQ, *i*SLIP with 4 iterations, buffered crossbar with 1-cell crosspoint buffers, and for the switch we propose in this paper with B=12, TH=5, and desynchronized clock credit schedulers; N=32; Bernoulli arrivals.

Once w goes above 0, throughput improves considerably, since flows alternate between active and inactive, and the synchronization behavior vanishes. With $TH \geq B$, throughput is high even when w=0, as grant throttling only rarely gets activated, and schedulers do not synchronize vastly.

At low w values, between 0 and 0.3, lower thresholds yield higher throughput, because credit accumulations are more effectively controlled when the throttling threshold is low. At higher unbalance factors, this trend is reversed: higher thresholds yield higher throughput. When unbalance is high, flows can be categorized in two distinct groups: one flow per input (or per output) is heavy, while the remaining (light) flows are almost always inactive, as they get served by the time they become active. Under this load, a low TH value may prevent a heavy flow from receiving the excess service available when its output neighbors are inactive, by turning *Off* its input.

Figure 8 shows the throughput of our switch using random-shuffle (*shuffle*) and desynchronized-clocks (*clocks*) credit schedulers, for varying TH values. All plots are for B=12 cells. A first noteworthy point is that with these credit scheduling disciplines, severe synchronizations do not occur under uniform traffic (w=0), even though grant throttling is employed; in this way, uniform traffic throughput is well above 99%. Both *shuffle* and *clocks* maintain a throughput above 0.96, even when the traffic is unbalanced, with $clocks$ performing slightly better. Simulations, not presented in this paper, demonstrate that *inert* round-robin performs very close to $clocks$.

All plots in figure 8 are for 32-port switches, except 256-port $clocks$-$TH5$. The plot for the 256-port switch shows that performance stays high even when the valency of the switch grows. Finally, we see that a buffered crossbar requires at least a 12-cell buffer per crosspoint (CP=12), i.e. 384 cells per output (N=32) or 6114 cells per output (N=256), in order to achieve throughput comparable to that of the present system, when the latter only uses 12 cells per output.

5.2 Tolerance to Burst Size

We have already seen the effectiveness of threshold grant throttling under bursty traffic in section 3.3. The results there were for average burst size (abs) equal to 12 cells. Figure 9 depicts the delay performance under uniform bursty traffic, with varying abs from 12 up to 108 cells. It compares our switch with B=12, TH=5, and desynchronized-clocks credit schedulers[5], to the 4-SLIP switch, to buffered crossbars, and to the OQ switch. As the figure shows, our architecture achieves performance almost identical to pure OQ; it exhibits strictly lower delays than 4-SLIP and buffered crossbar. These results are for 32-port switches; we have additional simulations exhibiting similar performance for 256-port switches, *without* increasing buffer space per output.

5.3 Diagonal Traffic

Last, we use *diagonal* traffic. Each input i hosts two active flows, flow $i \rightarrow i$, and $i \rightarrow (i+1)\%N$. The former flow consumes two thirds (2/3) of the incoming load, and the latter flow consumes the remaining one third (1/3).

Figure 10 depicts the performance of our switch with TH=5, and desynchronized-clocks credit schedulers, comparing it with that of 4 iterations *i*SLIP, buffered crossbar, and OQ. All switches have 32 ports. We see that our switch, $B12$, attains a throughput close to that of OQ, whereas the 4-SLIP switch saturates near 0.82 load, and the buffered crossbar

[5]method *random-shuffle* (not presented in this figure) achieves essentially the same performance with desynchronized-clocks

switch saturates near 0.92 load. Simulation results, not presented in this paper, indicate that random-shuffle achieves the same or better performance than *desynchronized-clocks* under this diagonal traffic pattern.

6 . Conclusions

We studied the performance of a new switch architecture under bursty traffic. We showed that VOQ fluctuations under bursty traffic may bring credit accumulations that deteriorate performance, and we proposed a grant throttling mechanism using thresholds to control accumulations. We then observed that threshold grant throttling may synchronize the round-robin credit schedulers, and we proposed modified round-robin disciplines that resolve this problem. Our simulation results show that using these methods the switch first described in [7] performs very close to ideal output queueing, with buffers as small as 12 cells per output, under both smooth and bursty traffic, independent of the number of cells in transit between the linecards and the fabric. The throughput achieved under unbalanced traffic exceeds 96%.

7 . Acknowledgments

This work has been supported by an IBM Ph.D. Fellowship, and the "SARC" FET IP funded by the European Commission.

References

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

[2] R. Rojas-Cessa, E. Oki, H. Jonathan Chao: "CIXOB-k: Combined Input-Crosspoint-Output Buffered Switch", *Proc. IEEE GLOBECOM'01*, vol. 4, pp. 2654-2660.

[3] K. Yoshigoe, K. Christensen: "An Evolution to Crossbar Switches with Virtual Output Queueing and Buffered Cross Points" *IEEE Network*, Vol. 17, No. 5, pp.48-56, September-October 2003.

[4] G. Georgakopoulos: "Nash Equilibria as a Fundamental Issue Concerning Network-Switches Design", *Proc. IEEE ICC 2004*, Paris, France, 20-24 June 2004, vol. 2, pp. 1080-1084.

[5] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, N. Chrysos: "Variable Packet Size Buffered Crossbar (CICQ) Switches", *Proc. IEEE ICC'04*, Paris, France, vol. 2, pp. 1090-1096. http://archvlsi.ics.forth.gr/bufxbar

[6] A. Engbersen, C. Minkenberg,"A combined input and output queued packet-switched system based on a Prizma switch-on-a-chip technology", *IEEE Comm. Mag.* pp. 70-77, Dec. 2000.

[7] N. Chrysos, Manolis Katevenis: "Scheduling in Switches with Small Internal Buffers", *Proc. IEEE Globecom'05*, St. Louis, MO, USA, 28 Nov. - 2 Dec. 2005; http://archvlsi.ics.forth.gr/bpbenes

[8] 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.

[9] N. McKeown: "The *i*SLIP Scheduling Algorithm for Input-Queued Switches",*IEEE/ACM Trans. on Networking*, vol. 7, no. 2, April 1999.

[10] N. Chrysos, M. Katevenis: "Scheduling in Non-Blocking Buffered Three-Stage Switching Fabrics", *Proc. IEEE INFOCOM Conf.*, Barcelona, Spain, 23-29 Apr. 2006; http://archvlsi.ics.forth.gr/bpbenes/

[11] 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.

[12] 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.

[13] E. Oki, R. Rojas-Cessa, H. J. Chao: "A Pipeline-Based Approach for a Maximal-Sized Matching Scheduling in Input-Buffered Switches", *IEEE Comm. Letters*, vol. 5, no. 6, pp. 263-265, June 2001.

[14] C. Minkenberg, I. Iliadis, F. Abel: "Low-latency pipelined crossbar arbitration", *IEEE GLOBECOM'04*, Dallas, Tex., CR-ROM paper ID "GE15-2".

