16th IEEE Symposium on High Performance Interconnects

Backlog-Aware Crossbar Schedulers: A New Algorithm and its Efficient Hardware Implementation

Nikos Chrysos and Giorgos Dimitrakopoulos
Inst. of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH) - member of HiPEAC*
FORTH-ICS, Vassilika Vouton, P.O. Box 1385, Heraklion, Crete, GR-711-10 Greece

Abstract

Crossbars switches with input queues are the common building blocks of high-speed networks, while their speed and performance critically depend on their scheduler. In this paper we combine ideas from randomized backlog-aware schedulers, and their round-robin (RR) counterparts, to propose a practical, deterministic crossbar scheduler, that: (i) achieves almost full throughput under the many adverse traffic patterns tested, using just 1 Mbyte buffer memory per input, (ii) provides deterministic delay service guarantees, (iii) yields low delays under both uniform and non-uniform load, and (iv) achieves these performances with a single iteration of an iSLIP-like algorithm. With simple extensions, the proposed crossbar scheduler is shown to distribute the bandwidth of congested links in a fair RR or WRR manner. In order to prove the efficiency of the new scheduling algorithm, we implemented in hardware a 32×32 scheduler, using a novel design for programmable-priority RR arbiters, that is significantly more area-speed efficient than present state-of-the-art. The scheduler's ASIC occupies roughly 3 mm^2, when implemented at 130nm, and gives a new crossbar match every 3.2 ns as needed for above hundred Gb/s line rates, and short packet lengths.

1 Introduction

Crossbar switches are the common building blocks for Internet routers, data-center and HPC interconnects and on-chip networks. The core switching fabric often has no buffers, saving in this way memory area and buffer speed. Arriving packets issue requests to a central scheduler, and get switched upon scheduler grants; meanwhile, packets wait at input packet buffers, in front of the crossbar. In order to isolate traffic flows[1], and provide the basis for proper congestion management, these input buffers must be organized in per-flow queues, forming what is widely known as virtual-output-queuing (VOQ) crossbars, as shown in Fig. 1.

*This work was supported by the European Commission in the context of the SARC integrated project 27648 (FP6), and the HIPEAC network of excellence (project 004408 and 217068) under the Interconnects research cluster.

[1]Flows are identified by distinct input-port/output-port pairs.

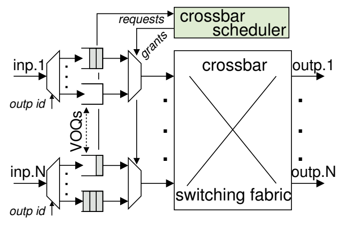


Figure 1. A crossbar switch with VOQs

The speed/switching efficiency tradeoff of a VOQ crossbar critically depends on the design and implementation of its crossbar scheduler. Most commercial crossbars today rely on independent, per-input and per-output round-robin (RR) arbiters, that yield maximal matchings after a few rounds of handshaking. The basic time complexity of these scheduling algorithms is approximately equal to that of two programmable priority arbiters [1], and increases linearly with the number of iterations; hence, iterations normally come along with a port speed penalty. Furthermore, although iterations improve the delay performance –as measured under uniform traffic–, they do not improve switch throughput under unfavorable, non-uniform traffic. In those cases, speedup is usually employed to cover the missing throughput; speedup however seriously affects the energy and the effective capacity of switching systems.

On the other hand, backlog-aware (maximum weight matching, MWM) schedulers of deterministic or randomized flavor can provably sustain full throughput under long-term feasible traffic [2], [5], [3], [4], but their main drawback is hardware complexity, which increases with shrinking scheduling time. Furthermore, many of these guarantees presume impractically large VOQs, and come at the expense of very large delays. A final concern is MWM performance under non-feasible traffic –MWM can be unfair when outputs become congested; MWM may also starve flows.

In this paper, we combine ideas from RR-based and backlog-aware, randomized schedulers, to propose an efficient, fair deterministic crossbar scheduler, that is amenable to fast hardware implementation (Section 2). Our analysis shows that the proposed algorithm guarantees service to non-empty VOQs in O(N^2) time, where N is switch size. Performance simulations in Section 2.1, for realistic VOQ sizes, demonstrate that the new algorithm practically achieves full throughput under adverse, non-uniform traffic. Simple modifications are presented in Section 2.2 that in addition yield fair, RR or weight-RR (WRR) allocation.

An overall description of the hardware design of the crossbar scheduler for the case of 32 ports is presented in Section 3. The new scheduler is implemented in a 130 nm standard-cell ASIC environment, occupies roughly 3 mm^2, and computes a new crossbar matching every 3.2 ns. The scheduler's latency is also 3.2 ns. Along this research, we came up with a new design for fast programmable-priority, RR arbiters that gives significantly faster circuits than state-of-the-art implementations (Section 3.2). These RR arbiters perform the core function of our algorithm, and have a wider applicability in networking and computer systems in general.

1.1. Overview of crossbar scheduling algorithm

We assume slotted time, and fixed-size packets (cells). A time slot, or cell time, is the duration of a cell on a crossbar or external line.

Our scheduler's operation is depicted in Fig. 2(a). We use one iteration of *i*SLIP (1SLIP), and denote as M_t its matching outcome at time t. 1SLIP matchings are used to configure the crossbar on a cell time basis. As shown for cell time $t+1$, besides VOQs state, 1SLIP receives as input a *preferred matching*, computed in the previous pipeline stage, during cell time t. (Please note that this matching is computed using outdated VOQs state, hence it may refer to VOQs that 1SLIP finds empty at $t+1$.) 1SLIP tries to enforce the preferred matching, and at the same time augment it with not included ports.

Preferred matching computation is performed in parallel with 1SLIP, in a pipelined manner[2]: while in cell time t 1SLIP computes a new crossbar matching (M_t), we compute and compare the aggregate VOQ length (*weight*) of M_{t-1} and M_{t-2}; then, we select the one with greater weight among these two as the preferred matching input to 1SLIP for $t+1$.

The underlying idea behind this type of scheduling with memory is due to Tassiulas [2], and is quite simple: once we find a matching with sufficiently good weight we try to maintain it. Tassiulas first proposed to compare the weight of the present match against that of a random one, while later, Algo2, by Giaconne *et al*, used the notion of *Hamiltonian walk (HW)* in the space of $N!$ possible matchings [3]. Using these methods, one approximates MWM and achieves full throughput for admissible traffic [3]. However, this is a blind search, in a large space, that results in large delays, and poor throughput for realistic (finite) VOQ sizes. We differ with those methods in that we do not search the matchings space randomly, but instead we delegate searching to 1SLIP.

[2]VOQ requests undergo just the 1SLIP's delay, i.e. one cell time.

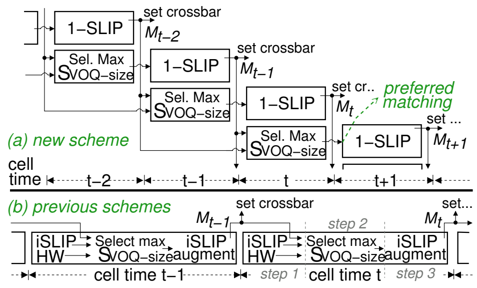


Figure 2. Searching for large-weight matchings.

The performance of our method, as described so far, suffers due to the synergy of the following facts: *(a)* 1SLIP searches for maximal matchings, and is oblivious of matching weights; *(b)* the use of memory improves 1SLIP by preferring larger weight matchings, but may settle with matchings that although locally better are in fact poor.

Our algorithmic novelty is that we systematically *remove edges* from the present preferred matching in order to escape from such local maximums. After the removal of an edge, some input and output ports get released: 1SLIP can combine these ports with other not included in the present preferred match, and find a new, possibly better configuration. Our edge removal method secures "good" present matchings, while it can *increase matching size*, from one cell time to the next, even when VOQs are non-uniformly loaded.

The overall scheduler is simpler, faster, and more efficient than previously proposed schemes that try to speedup the search of Algo2 via "random" maximal matchings, produced by *i*SLIP like circuits [6] [5]. As shown in Fig. 2(b), these systems perform the following operations sequentially in every cell time: **step 1)** compute a new match (e.g. via *i*SLIP), and a random match via HW; **step 2)** compare the weights of these matchings against the weight of the matching selected in the previous cell time; **step 3)** run a new maximal matching round in order to augment the winning match.

One could possibly pipeline these operations, but doing so straightforwardly, two 1SLIP circuits would normally be required. Our pipelined method saves one such circuit: the same 1SLIP circuit augments preferred matchings, and also searches for new ones, via edge removals, in a unified manner. We do not need and do not use the HW.

Performance simulation results, for realistic VOQ sizes, demonstrate that our pipelined solution achieves close to full throughput, outperforming the non-pipelined schemes of Fig. 2(b) that provably sustain full throughput.

To the best of our knowledge, our algorithm is the first to achieve close to full throughput with the time complexity of 1 iteration *i*SLIP. Recently, a new derivative of Exhaustive *i*SLIP (E*i*SLIP) [7] was shown to yield good matchings from the first iteration [8]. Both E*i*SLIP and our system favor "heavy" VOQs, and improve matching size over multiple cell times. But E*i*SLIP improvements are only maximal, whereas ours can additionally be non-maximal, which is reflected in significantly better throughput performance[3]. Moreover [8] does not consider fairness, which, as shown in Section 2.2, is quite poor for E*i*SLIP-like algorithms.

2 Crossbar Scheduling Algorithm

Our scheduling core is 1SLIP that follows the 3-phase regime of input-request, output-grant, input-accept. 1SLIP runs in *normal* mode for most of the time, and periodically toggles to an *escape* mode for one cell time. First we describe these two modes; then, we present the state machine that visits them in a time division multiplexing manner. $F[i]$ denotes the preferred output of input i (= $\varnothing$ if it has none).

NORMAL (F): F is the preferred matching computed in the previous cell time. $G[j]$ and $A[i]$ denote the (normal) RR next-to-serve pointer of output j and input i respectively.

–Filter input requests: if input i has VOQ cell for output $F[i]$, it sends a *preferred request* to that output –this pair will definitely be matched–, and *filters out* the requests for all other outputs. Otherwise, if $F[i]= \varnothing$ or the VOQ for $F[i]$ is empty, input i sends a request from any active VOQ.

–Output grant: if output j receives a preferred request it grants that request. Otherwise, it scans inputs onwards, starting from the one selected by $G[j]$, and grants the first request.

–Input accept: input i scans outputs onwards, starting from the one selected by $A[i]$, and accepts the first grant.

–Pointer update: iff a match, preferred or non-preferred, is achieved between input i and output j in the first scheduling round[4], then $G[j] = (i + 1) \text{mod } N$, $A[i] = (j + 1) \text{mod } N$.

Please note that matching F may refer to VOQs that are empty when 1SLIP runs; also, some input ports that have cells in their VOQs may not be included in F. In these cases, the respective inputs broadcast requests. 1SLIP will try to match those inputs with similarly available outputs.

Similar to E*i*Slip ([7]), the request filtering at inputs that know they will be matched reduces request contention: outputs should better not receive request from (thus not grant to) inputs that will definitely match with their preferred output.

ESCAPE (F, M): F and M are the preferred matching and the crossbar schedule computed in the previous cell time. $eG[j]$ and $eA[i]$ denote the (escape) RR next-to-serve pointer of output j and input i respectively.

First we nullify all preferences: $F[i]=\varnothing$, $\forall\ i$. (There are no preferred requests.) Then:

–Input request: input i sends a request from any active VOQ. When these are more than one, input i will not send a request from the VOQ corresponding to $M[i]$ –we crop previously matched pairs in order to examine "new" matchings.

–Output grant: as in normal mode, except that now output j scans requests starting from the input selected by $eG[j]$.

–Input grant: as in normal mode, except that now input i scans grants starting from the output selected by $eA[i]$.

–Pointer update: as in normal mode, except that now $eG[j]$ and $eA[i]$ are updated, not $G[j]$ or $A[i]$.

SCHEDULER FSM: As shown in Fig. 2(a), at time t we perform two operations in parallel: *(i)* we compute the weight of matchings M_{t-2} and M_{t-1}, using the present VOQ lengths, and then set F_t equal to the matching with maximum weight; *(ii)* we execute 1SLIP in normal or escape mode, as below –e and s are integer constants.

```
if ( $t \mod e == 0$ ) then
    ESCAPE-MODE($F_{t-1}$,$M_{t-1}$);
else
    if ( $t \mod s \neq 0$ ) then
        $F_{t-1}[I] = \varnothing$;              // in local escape
        $I = (I + 1) \mod N$;     // we remove an edge
    end if
    NORMAL-MODE($F_{t-1}$);
end if
```

The escape mode supplies radically new matchings, and for this reason we refer to it as *global escape*. We try a new global escape (nullifying all preferences) once every e cell times. When doing so yields a better matching, the algorithm will start preferring that new matching; otherwise, it will simply recover to the previous one.

Properties: The escape mode prevents starvation, as we have an 1SLIP run once in every e cell times, that provides deterministic service delay to non-empty VOQs. Although we filter previously matched pairs out (see request phase in escape mode), this does not alter the situation.

Theorem **1:** *In the worst-case, every non-empty VOQ gets served in O(N^2) time. Proof:* In the worst-case, 1SLIP serves any non-empty VOQ in $N^2 + (N - 1)^2$ cell times. This is computed in [9] as the delay of a tagged output, j, to grant a tagged input, i, plus the delay it takes input i to accept the grant. While escape-1SLIP "slips" to this tagged pair, making one step every e cell times, it will skip a pair p at time t, only if p was served by normal-1SLIP at $t - 1$. In this case, $eG[j]$ and $eA[i]$ will be one position closer to pair $i \rightarrow j$ (maybe even selecting it) at time $t + e$. Hence, the tagged pair will be served within the 1SLIP delay multiplied by constant e. ■

Escape-1SLIP may yield relatively poor matchings, but its effect on performance can be controlled. Assuming for example that the average matching size required to sustain an input load ρ is $S(\rho)$, and that normal-1SLIP achieves $S(\rho)$, while escape-1SLIP yields at least an average matching size of $0.1 \cdot S(\rho)$. Then the net average matching size, $NS(\rho)$ will be greater than $(e - 0.9) \cdot S(\rho)/e$.

Hence, e can be adjusted so as to minimize any possible throughput losses due to escape-1SLIP. For e= 100, $NS(\rho)$ $\geq \frac{99.1}{100} S(\rho)$. For e= 1, i.e. always in escape mode, the algorithm performs similarly with ordinary 1SLIP. Larger e values allow preferred (good weight) matchings stay on, im-

[3]We compared our results with that in [8]; the latter are very similar with E*i*SLIP results, which are presented in Section 2.1

[4]For generality, we give a multi-iteration description.

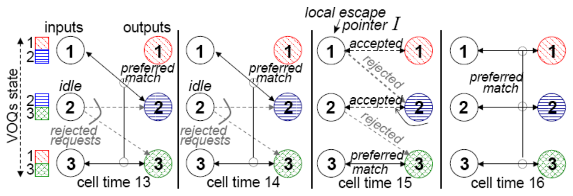


Figure 3. Local escape increases match size & weight.

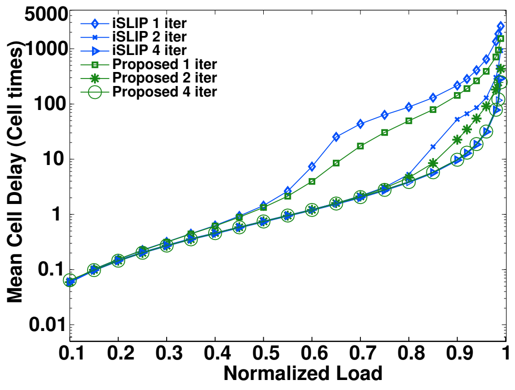


Figure 4. Delay under uniform traffic: multiple iterations.

proving throughput. But too high an e value may reduce the number of new matchings examined.

Removing edges: While in normal mode, we examine whether nullifying some input preference will improve the matching quality –see "remove an edge" comment in scheduler's FSM. Specifically, we maintain a pointer, I, that visits inputs, one by one, in separate cell times, as controlled by parameter s, nullifying the preference of each visited input. These *local escapes* search for improved matchings that lie close to the presently preferred one. As we discuss below, local escape tends to increase, in a non-maximal way, the size, and possibly the weight of the currently preferred matching.

See the example in Fig. 3, which, for simplicity, ignores pipelining delays. When input 1 is visited by I at time 15, input 1 and its previously matched pair, output 2, get released from their mutual commitment, and try to pair with other ports. Thanks to our pointer update policy, after last being matched by preference at cell time 14, pair 1$\rightarrow$2 has the lowest normal-1SLIP priority: $G[2] = 2$ and $A[1] = 3$.

Local escape allows unmatched inputs, which, up to now, have been unable to find a free output[5], like input 2 in Fig. 3, to try match with any available output. On the other hand, the released input 1 issues requests from all non-empty VOQs, and can thus match with any targeted output (1 in our example) that has no preferred pair. As with global escape, the new matching will start being preferred if it is of larger weight than the presently preferred match.

Importantly, when local escape removes an edge, it does not needlessly harm the presently preferred matching: if all ports "requested" by the released input, 1, are matched by preference, then input 1 will pair again to its previously preferred output, 2, with no throughput penalty at all.

Local escape resembles the search of neighbor matchings in APSARA [3]. In every cell time, APSARA computes as many as N^2 neighbor matchings and compares their weight, whereas we just nullify entries of the preference vector.

2.1. Performance Evaluation

We used performance simulations in order to test our scheduling algorithm, and compare its performance with that of previous proposals. All VOQs, at every particular input, share a buffer space of 2^{14} cells, which for 64-byte cells translates to 1 MByte per input. When this buffer is full, arriving cells are discarded[6]. Each experiment was run 3 to 7 times, so as to achieve 5% confidence intervals with 95% confidence.

We ran experiments under uniform traffic, and under the hardest non-uniform traffic patterns that we found in the respective literature: diagonal, power2, and zipf. Cell arrivals were driven by an independent Bernoulli process at each input –our results for bursty traffic show similar trends. The switch size examined in this paper is N= 32.

Unless otherwise noted we use one scheduling iteration, and we set parameters e= 100, and s= 3. We have selected these values after extensive performance simulations for switch sizes $\leq$ 64. For comparison purposes, we also plot the performance of *i*SLIP, of E*i*SLIP, and of *MaxDRDSSR* [6]. *MaxDRDSSR* operates as in Fig. 2(b); it uses the RDSSR algorithm in step 1, and 1SLIP in step 3.

Fig. 4 depicts the delay performance under **uniform traffic** for the proposed system and for *i*SLIP. Thanks to its embedded matching augmentation, under low loads the proposed system delivers delays very close to that of *i*SLIP. For 1 iteration, the proposed system delivers about three (3) times lower delay than ordinary 1SLIP, thanks to the request filtering in normal mode.

Next we examine **diagonal** and **power2** traffic. In diagonal, input i sends 2/3 of its traffic to output $(2 \cdot i + \frac{2 \cdot i}{N}) \mod N$ (this is the perfect shuffle pair of i), and the remaining 1/3 to output $(1 + 2 \cdot i + \frac{2 \cdot i}{N}) \mod N$. Observe that this is a shifted version of the diagonal traffic seen in previous papers, where input i sends its traffic to outputs i and $(i+1) \mod N$. We refer to the unshifted version as **diagonal easy**. In power2, sometimes referred to as *logdiagonal*, every input sends twice more traffic to output i than to $(i+1) \mod N$.

Fig. 5 depicts the results for the proposed system, and for E*i*SLIP, 4-SLIP, and MaxDRDSRR. As can be seen, our scheme delivers almost full throughput for both traffic patterns, with its delay under power2 traffic being significantly

[5]Because all outputs they request have a preferred pair.

[6]Buffer sharing runs the danger of buffer monopolization, but many implementations follow this approach.

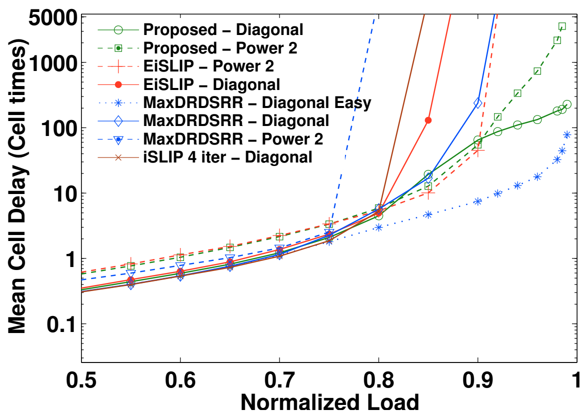


Figure 5. Delay under diagonal and power2 traffic.

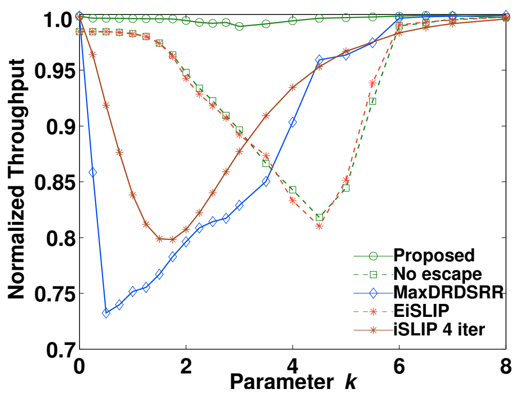


Figure 6. Throughput under Zipf traffic.

higher than under diagonal[7]. All other systems saturate at a load ranging from 0.75 to 0.9.

MaxDRDSSR performs well only under diagonal-easy. Diagonal-easy is easy particularly for MaxDRDSSR, as in every cell time t the RDSSR algorithm assigns highest priority to connections $i \rightarrow (i+t) \mod N, i \in [1, N]$; effectively, MaxDRDSSR finds a large weight configuration in less than N cell times, delivering almost ideal delay. However this is not the case in general, as shown with MaxDRDSSR's performance for the shifted diagonal traffic. We also have results for a modification of MaxDRDSSR, which uses 1SLIP instead of RDSSR at step 1. Essentially, this is the algorithm examined in [5] complemented with the HW in step 1, and the augmentation step 3. The approach presented in [5] performs better than MaxDRDSSR under diagonal traffic, and slightly worse under diagonal-easy.

Finally, Fig. 6 depicts switch throughput under **Zipf** traffic, controlled by parameter k: input 0 sends to output i with probability $Zipf(i) = i^{-k} / \sum_{j=1}^{N} j^{-k}$; probabilities at other inputs are obtained by circularly shifting that of input 0. Then, the output arrival rates at each input are exchanged according to the perfect shuffle permutation. Traffic is uniform when k=0 and completely directed as $k \rightarrow \infty$. We present plots for MaxDRDSRR, E*i*SLIP, and for the proposed system. The proposed system achieves virtually full throughput, while MaxRDRSRR's throughput is below 0.8, and that of E*i*SLIP below 0.85. The *no-escape* plot is the proposed system with no global and no local escape (e=∞, s=0). No-escape performs identically with E*i*SLIP for the reasons described in Section 1.1.

We have also tested *global-only*, were we never perform local escape, and *local-only*, where we never enter the escape mode. Under diagonal traffic, local-only performs slightly worse than global-only, but for Zipf traffic local-only achieves close to full throughput whereas global-only below 0.9. These results indicate that searching from the scratch, as global-only does, is more beneficial when the matchings space is relatively small –in diagonal every input has only two VOQ candidates– but performs poorly as this space increases. On the other hand, searching locally maybe more slow but is more effective in a large matching space as it improves, step-by-step, the presently preferred match.

2.2. Fair allocation of congested link bandwidth

Our algorithm prefers large VOQs, thus, similarly to MWM, may distribute the bandwidth of congested (overloaded) outputs unfairly amongst unequally loaded VOQs[8]. However this becomes problematic only when the requests presented to the scheduler overbook some output port. In [10] [11], per-output RR "credit" arbiters are deployed in order to regulate the rate of aggregate output flows, and prevent congestion expansion in multi-stage fabrics. In such systems, employing a MWM-like algorithm in internal switching elements would not affect port-to-port fairness.

If these arbiters are not in place already, following [12], we propose to explicitly employ them. (Actually, the same scheme can be used in conjunction with any unfair scheduler, not only MWM derivatives.) These RR arbiters are the first to process VOQ requests, each serving one request in every cell time. After serve, requests are registered in per-flow counters. Our crossbar scheduler considers these counters as VOQs demand; it also use them to calculate matching weight. Essentially, the "input" to the crossbar scheduler is now the "output" of a RR output queued switch.

Observe that the RR arbiters operate in a first pipeline stage, not affecting the critical delay of the crossbar scheduler circuit. Additional performance results –not presented due to space limitations– show that these RR arbiters do not harm the throughput performance presented in Section 2.1. However, the cold-start delay under low load increases by one cell time due to the additional pipeline stage.

In order to test fairness, we configured three flows, $1 \rightarrow 1$, $2 \rightarrow 1$, and $4 \rightarrow 1$, with arrival rates 1.0, 0.9, and 0.5, respectively. Table 2 depicts flow service rates under the proposed

[7]Our comparison with with other "randomized" schedulers shows that only MaxAPSARA [3] achieves similarly good delays.

[8]As pointed out in [12], MWM is a dual to max-min fairness in that it "first serves" the highly loaded queues, instead of the lightly loaded ones.

Flow: rate	Proposed			E*i*SLIP		WFA	
	Base	RR	WRR	Base	RR	Base	RR
1-1: 1.0	0.51	0.33	0.16	1.00	0.33	0.25	0.33
2-1: 0.9	0.45	0.33	0.34	0.0	0.33	0.50	0.33
4-1: 0.5	0.04	0.33	0.50	0.0	0.33	0.25	0.33

Table 1. Service rates of 3 flows that overload output 1.

.

algorithm, the E*i*SLIP scheduler, and the WFA scheduler [13], when the RR regulation arbiters are absent (Base column), and when they are present (RR column). The proposed system favors the two heavy flows, and only sporadically serves flow 4→1 –thanks to escape modes–, E*i*SLIP serves only flow 1→1, as its VOQ is always non-empty, and WFA favors flow 2→1 due to diagonals movement. But with the RR arbiters all systems distribute equal rates to flows.

Furthermore, as in [11], one may use *weighted round-robin* instead of plain RR arbiters, hence enabling sophisticated QoS. Based on the previous configuration, we assigned a weight of 10, 20, and 30, to flows 1→1, 2→1, and 4→1, respectively. As shown in Table 2 (column WRR), the proposed system, with WRR regulation arbiters, now serves flows according to their weighted max-min fair shares.

3. Crossbar Implementation

In this section we present the hardware implementation of the proposed crossbar scheduler. The scheduler's organization is shown in Fig. 7. It consists of two parts that are separated in two pipeline stages. The first part of the scheduler is the weight computation stage, while the second part computes the schedule for the crossbar. The weight that corresponds to the current match is computed via a multi-operand adder and a high-speed final adder. The weight computation unit is unique in the scheduler and it is shared between all inputs. Each input port has N request counters (one for each output) that store the number of cells present in each VOQ, as in [14]. Every cycle, each input broadcasts to the weight computation unit the value of the request counters that correspond to the matched output at the current cell time and the previous cell time.

The schedule-compute stage takes as input the weights computed in the previous cycle, by the first pipeline stage[9]. The Maximum selector simply decides which one of the two weights under comparison is larger. If the weight of M_{t-1} is larger than the weight of M_{t-2}, then M_{t-1} is selected as a preferred match, and M_{t-2} otherwise. Instead of adding a multiplexer that selects between M_{t-1} and M_{t-2} and then drive the result to the filtering unit we followed a different approach that offers significantly faster implementations. We use two filtering units per input that run in parallel, one driven by M_{t-1} and the other by M_{t-2}.

[9]For clarity, Fig. 2 presents weight computation and comparison (i.e. Maximum Selector) being performed in the same pipeline stage. Here we place them in separate stages so as to balance the pipeline.

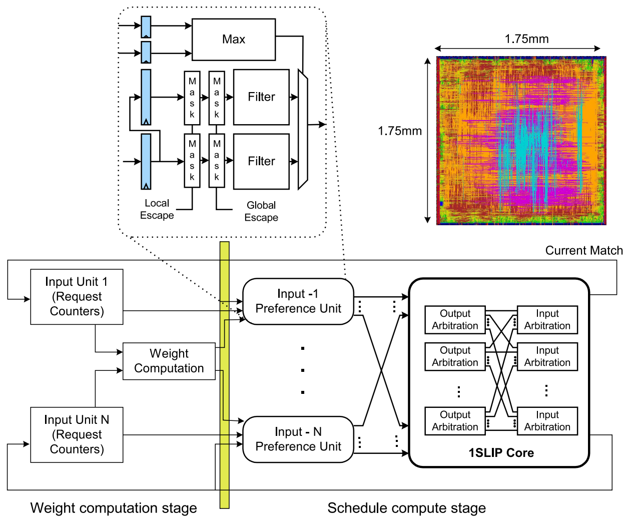


Figure 7. Block diagram & layout of proposed scheduler.

Please note that the preferred matches M_{t-1} and M_{t-2} before entering the corresponding filtering units are at first masked with the state of the current local escape and the global escape that nullify either some or all the requests, respectively. The set of the two filtered requests are given to a multiplexer that forwards one of them to the final scheduling unit, according to the decision made by the Maximum selector that runs in parallel. In this way, the delay of the filtering unit, as well as the masking operation, is partially hidden since it overlaps in time with the computation of the Maximum selector. The schedule-compute stage passes the filtered requests to the 1SLIP core.

The overall scheduler was implemented in 130nm CMOS technology using a standard-cell based design flow. The RTL description was written in Verilog, and the circuit was synthesized and placed & routed using Synopsys Design Compiler and Cadence SOC encounter, respectively. The final layout of the 32× 32 scheduler is also shown in Fig. 7. It occupies roughly 3 mm^2 and operates with a clock period of 3.2 ns. Next, we present the 1SLIP core that was used to derive the aforementioned layout. Our 1SLIP core employs a novel RR implementation that is described in Section 3.2.

3.1. The 1SLIP core

Each arbiter of the 1SLIP core is a programmable priority arbiter (PPA). Each PPA consists of the core arbitration logic that scans requests beginning from an arbitrary position, denoted by vector P, and the pointer update logic that updates P according to some policy (e.g. pure RR, or as described in normal- or escape-1SLIP). The core arbitration logic scans the input requests in a cyclic manner beginning from the position that has the highest priority. For example, if the ith request has the highest priority then the priority is diminishing in a cyclic manner to positions $i+1, i+2, \ldots, N-1, 0, 1, \ldots i-1$, giving to the request $i-1$ the lowest priority to win a grant.

An efficient PPA architecture, which we denote as *Dual Path*, was presented in [1]. It utilizes two fixed priority arbiters (FPAs) that work in parallel. The upper FPA scans requests starting from the highest priority one, up to position $N-1$. It does not scan range $[0, P-1]$. The lower FPA works on all incoming requests. If there is a request in the range $[P, N-1]$, the correct output comes from the upper FPA; otherwise, it comes from the lower FPA.

The output and the input arbiters of 1SLIP need to be slightly modified in order to distinguish between preferred and normal requests. Each arbiter accepts besides the N-bit request vector an additional preference vector that marks the preferred request. So in parallel to the core arbitration logic we need an N-input OR tree to decide if there is an active preference. If this case is true then the output of the core arbitration logic is bypassed and the grant is given directly to the active preference. This extra circuitry does not affect the area of the PPA but inserts some non-negligible delay overhead. In order to alleviate this problem, we designed a new PPA that is significantly faster than Dual Path and speeds up the 1SLIP core.

3.2 New programmable-priority arbiters

In this section we present an efficient PPA design. The PPA takes as input a N-bit request vector, R, and produces a N-bit grant vector G. We encode the highest priority position in one-hot form, in variable P.

As in [1], the "priority transfer" signal, C_i, indicates whether position $i+1$ can produce a grant or not. Position i gives a grant when it has an active request, and either P_i or the incoming priority transfer signal C_{i-1} is asserted. In case of a grant, C_i is set to zero, and G_i is set to one. These actions can be written as:

$$\begin{aligned} G_i &= R_i \cdot (P_i + C_{i-1}) \\ C_i &= \overline{R}_i \cdot (P_i + C_{i-1}) \end{aligned} \qquad (1)$$

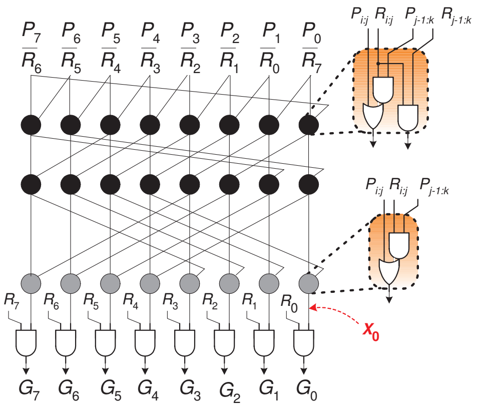
Observe that the priority transfer signal C_{N-1} should be fed back to position 0, i.e., $C_{N-1} = C_{-1}$, in order to guarantee the cyclic transfer of the diminishing priority. In order to break the combinational loop and derive more efficient designs we express the priority transfer differently. First, we combine in one signal, X_i, the two sources of priority transfer. The priority to i either comes from the previous bit position, via C_{i-1}, or is set by P_i:

$$X_i = P_i + C_{i-1} \qquad (2)$$

Next, we derive a recursive equation that connects X_i and X_{i-1}. From (1) and (2) we can write $C_{i-1} = \overline{R}_{i-1} \cdot X_{i-1}$. Replacing in (2), we get:

$$X_i = P_i + \overline{R}_{i-1} \cdot X_{i-1} \qquad (3)$$

Effectively, C terms have disappeared, and X_i depends only on P_i, R_{i-1}, and X_{i-1}. Now the grant signal is computed using $G_i = R_i \cdot X_i$.



P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0
$\overline{R}_6$ $\overline{R}_5$ $\overline{R}_4$ $\overline{R}_3$ $\overline{R}_2$ $\overline{R}_1$ $\overline{R}_0$ $\overline{R}_7$
$P_{i:j}$ $R_{i:j}$ $P_{j-1:k}$ $R_{j-1:k}$
$P_{i:j}$ $R_{i:j}$ $P_{j-1:k}$
R_7 R_6 R_5 R_4 R_3 R_2 R_1 R_0
X_0
G_7 G_6 G_5 G_4 G_3 G_2 G_1 G_0

Figure 8. The proposed 8-input PPA that cycles the priority transfer signals inside the carry computation unit.

The recursive definition of X_i in (3) has exactly the same form as the well known carry lookahead equation $c_i = g_i + p_i \cdot c_{i-1}$, where in place of the carry generate bit g_i we have here the priority signal P_i (called priority generate), and instead of the carry propagate bit p_i we use the inverted request signal $\overline{R}_{i-1}$ (called priority propagate). Following adder design principles, we can define *priority generate groups* and *priority propagate groups*. Therefore, since $g_i = P_i$, $p_i = \overline{R}_{i-1}$ and $p_0 = \overline{R}_{n-1}$, the priority generate group that starts at j and ends at position i, with $i \geq j$ is defined as $P_{i:j} = g_i + \sum_{k=j}^{i-1} (R_{i:k+1} \cdot g_k)$. The term $R_{i:j}$ denotes the group propagate term in the range $i \ldots j$ and is defined as $R_{i:j} = \prod_{k=j}^{i} p_k$. For the degenerated case $P_{i:i} = g_i$ and $R_{i:i} = p_i$.

Using the group terms, X_i can be expressed as $X_i = P_{i:0} + R_{i:0} \cdot X_{in}$, where $X_{in} = X_{-1}$ denotes the incoming priority transfer similar to the carry-in signal of an adder. Due to the cyclical priority transfer $R_{-1} = R_{N-1}$, and $X_{-1} = X_{N-1}$. The priority generate group term $P_{i:0}$ covers the case where the priority is generated for the ith position after having searched all positions $[0, i-1]$. For the case of the most significant bit position $N-1$, the corresponding group generate term $P_{N-1:0}$ searches all the input requests from input 0 to input $N-1$. Therefore, $P_{N-1:0}$ is the desired priority transfer signal for position $N-1$, i.e. X_{N-1}. From this observation and the fact that $X_{N-1} = X_{in}$ the equation for X_i can be transformed as follows:

$$X_i = P_{i:0} + R_{i:0} \cdot P_{N-1:0} \qquad (4)$$

Thus the ith position has the highest priority because either the priority was generated in the range $[0, i]$ or it is coming from a more significant position, as declared by $P_{N-1:0}$, and has been propagated to i via the propagate term $R_{i:0}$. In fact, if the priority is transferred circularly to the ith position, then only the range $[N-1, i+1]$ needs to be examined. By definition, $P_{N-1:0}$ can be derived by any smaller group generate and propagate term as follows

N	Proposed I		Proposed II		Dual Path	
	Delay	Area	Delay	Area	Delay	Area
64	595	10593	740	8205	740	8990
32	510	4612	640	3542	640	4134
16	430	1961	550	1473	550	1948

Table 2. Delay (ps) and area (μm^2) synthesis results, for different number of arbiter inputs.

$P_{N-1:0} = P_{N-1:i+1} + R_{N-1:i+1} \cdot P_{i:0}$. Thus, substituting this relation to (4) and performing simple boolean algebra manipulations we get that $X_i = P_{i:0} + R_{i:0} \cdot P_{N-1:i+1}$.

In this way, the redundant examination of requests $[i, 0]$, in the case of $P_{N-1:0}$, has been removed, and the circular operation of the priority transfer has been embedded inside each relation. In order to better understand the derived relation we write the equations for X_2 for a 4-input arbiter.

$$X_2 = P_2 + \overline{R}_1 \cdot P_1 + \overline{R}_1 \cdot \overline{R}_0 \cdot P_0 + \overline{R}_1 \cdot \overline{R}_0 \cdot \overline{R}_3 \cdot P_3$$

As can be seen, each priority transfer bit X_i is computed in parallel from the input bits without requiring any combinational feedback loop.

An implementation of the proposed PPA, for 8 bits, is shown in Fig. 8. The grant vector is computed in exactly $\log_2 N + 1$ logic levels, as in ordinary FPA, i.e. faster than in Dual Path. Also, no large fanout line is required, since the cyclic nature of the priority transfer is performed inside the carry-lookahead tree. The only drawback of the proposed circuit are the long lines inside the priority transfer computation unit that increase its layout area. Although the extra capacitance added by these lines degrades by a small percentage the delay of the circuit, the overall circuit is significantly faster than the most efficient previous implementation.

In order to quantify the delay savings of the proposed PPA, separately from the complete scheduler implementation, we synthesized the new PPA and the Dual-Path design using Synopsys Design Compiler and the same 130nm CMOS technology. For performance evaluation, we set the available input capacitance of the circuits equal to that of a drive-strength 2 inverter, and the output loading capacitance four times larger than that. Each design was recursively optimized for speed, targeting the minimum possible delay.

Our results, shown in Table 2, demonstrate that the proposed solution (*Proposed I*) is on average 20% faster than Dual Path [1]. In fact, the delay of the proposed arbiter for 64 inputs is less than Dual Path's delay for 32 inputs. The delay savings of our solution can alternatively be traded-off for reduced layout area. In column *Proposed II*, we sized the gates of the our circuit for a delay target equal to the delay Dual Path. As can be seen, Proposed II saves more than 16% of layout area compared to Dual Path.

4 Conclusions

We proposed a new crossbar scheduling algorithm that has the complexity of 1 iteration *i*SLIP. Performance simulation results, for realistic VOQ sizes, demonstrated that this scheduler achieves virtually full throughput under hard non-uniform traffic patterns outperforming algorithms that provably yield full throughput. Analysis showed deterministic time service guarantees to non-empty VOQs, while with simple modifications, the proposed algorithm was also shown to yield fair (RR or WRR) allocation of congested link bandwidth. We currently examine the crossbar throughput for different VOQ sizes and also investigate the output burstiness; Our preliminary results for uniform Bernoulli traffic (100% load) show that the average burst size at the switch outputs is approximately 4 cells.

Due to its simple structure, the new algorithm allows efficient pipelined hardware implementations. In order to further improve the hardware design of the proposed scheduler we described the design of efficient RR arbiters that offer significantly faster implementations than previous solutions. The new scheduler was implemented in 130nm showing that it constitutes an attractive solution for high-speed and large-size switches.

References

[1] P. Gupta, N. McKeown: "Design and implementation of a fast crossbar scheduler", *IEEE Micro*, Jan 1999.

[2] L. Tassiulas: "Linear complexity algorithms for maximum throughput in radio networks and input queues switches", *IEEE INFOCOM*, NY, 1998.

[3] P. Giaccone, B. Prabhakar, D. Shah: "Randomized scheduling algorithms for high-aggregate bandwidth switches", *IEEE JSAC*, May 2003.

[4] A. Dua, N. Bambos, W. Olesinski, H. Eberle, N. Gura: "Backlog aware low complexity schedulers for input queued packet switches", *IEEE Symp. on High-Perf. Interconnects*, 2007.

[5] I. Keslassy, N. McKeown: "Analysis of scheduling algorithms that provide 100% throughput in input-queued switches", *Allerton Conf. on Communication, Control, and Computing*. Monticello, October 2001.

[6] J. Liu, C.K. Hung, M. Hamdi, C.Y. Tsui: "Stable round-robin scheduling algorithms for high-performance input-queued switches", *IEEE Hot Inteconnects*, August 2002.

[7] Y. Li, S. Panwar, H.J. Chao: "Exhaustive service matching algorithms for input queued swithes",*IEEE HPSR*, Phoenix, April 2004.

[8] S. Mneimneh: "Matching from the first iteration: an iterative switching algorithm for an input queued switch", *IEEE/ACM Trans. on Networ.*, February 2008.

[9] D. Serpanos, P. Antoniadis: "FIRM: A Class of distributed scheduling algorithms for high-speed ATM switches with multiple input queues", *IEEE INFOCOM*, Tel Aviv, March 2000.

[10] A. Bianco, P. Giaccone, E.M. Giraudo, e.a.: "Performance analysis of storage area network switches", *IEEE HPSR*, Hong Kong, May 2005.

[11] N. Chrysos, M. Katevenis: "Scheduling in non-blocking buffered three-Stage switching fabrics", *IEEE INFOCOM*, Barcelona, April 2006.

[12] N. Kumar, N. R. Pan, D. Shah: "Fair scheduling in input-queued switches under inadmissible traffic", *IEEE GLOBECOM*, Dallas, November 2004.

[13] Y. Tamir, H.C. Chi: "Symmetric crossbar arbiters for VLSI communication switches", *IEEE Trans. on Parallel and Distributed Systems*, January 1993.

[14] C. Minkenberg: "Performance of i-SLIP scheduling with large round-trip latency", *IEEE HPSR*, Torino, June 2003.