PRACTICAL HIGH-THROUGHPUT CROSSBAR SCHEDULING

A PRACTICAL DETERMINISTIC CROSSBAR SCHEDULER ACHIEVES ALMOST FULL THROUGHPUT WITHOUT BEING HEAVILY AFFECTED BY SHORT VIRTUAL OUTPUT QUEUES OR TRAFFIC BURSTINESS. SIMPLE ADDITIONS OFFER DETERMINISTIC SERVICE GUARANTEES AND DISTRIBUTE THE BANDWIDTH OF CONGESTED LINKS IN A WEIGHTED, FAIR MANNER.

Nikos Chrysos
Giorgos Dimitrakopoulos
Foundation for Research and Technology—Hellas

Input-queued crossbars are the common building blocks in Internet routers, datacenter and high-performance computing interconnects, and on-chip networks. These crossbars often contain no buffers, which saves valuable chip area. Arriving packets issue requests to a central scheduler. While waiting for the scheduler to grant their requests, packets wait at input packet buffers in front of the crossbar. To isolate traffic for different outputs, these input buffers are often organized as *virtual output queues* (VOQs).

A VOQ crossbar's trade-off of speed and switching efficiency depends on its crossbar scheduler. Most commercial schedulers rely on per-input and per-output round-robin arbiters that yield maximal matchings after a few rounds of handshaking. The 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. Although iterations might improve the packet delay, they don't improve switch throughput under unfavorable, nonuniform traffic patterns. In these cases, most schedulers use speedup to compensate for the missing throughput. Speedup, however, seriously affects a switching system's energy and effective capacity.

However, deterministic or randomized backlog-aware schedulers—also called *maximum weight matching* (MWM) schedulers—can sustain full throughput under feasible traffic.[2-4] Their main drawback is complexity. Furthermore, their throughput guarantees frequently presume impractically large VOQs. Finally, MWM can be unfair or even starve flows when outputs become congested.

Our deterministic scheduler combines ideas from round-robin and backlog-aware randomized schedulers. It achieves virtually full throughput for realistic VOQ sizes, maintains fairness, and is amenable to fast hardware implementation.

Crossbar scheduler

Figure 1 depicts the scheduler's operation. We use one iteration of the *i*SLIP algorithm (1SLIP), and denote its matching outcome at time t as M_t. We use 1SLIP matchings to configure the crossbar on a cell-time basis, where cell time is the duration of a fixed-size packet (cell). As Figure 1 shows, for cell time $t+1$, 1SLIP receives (in addition to VOQ state) a preferred matching, computed during cell time t. 1SLIP tries to enforce the preferred matching while augmenting it with not-included ports. We compute the preferred matching in parallel with 1SLIP in a pipelined manner. While 1SLIP computes M_t in

Published by the IEEE Computer Society 0272-1732/09/$26.00 © 2009 IEEE

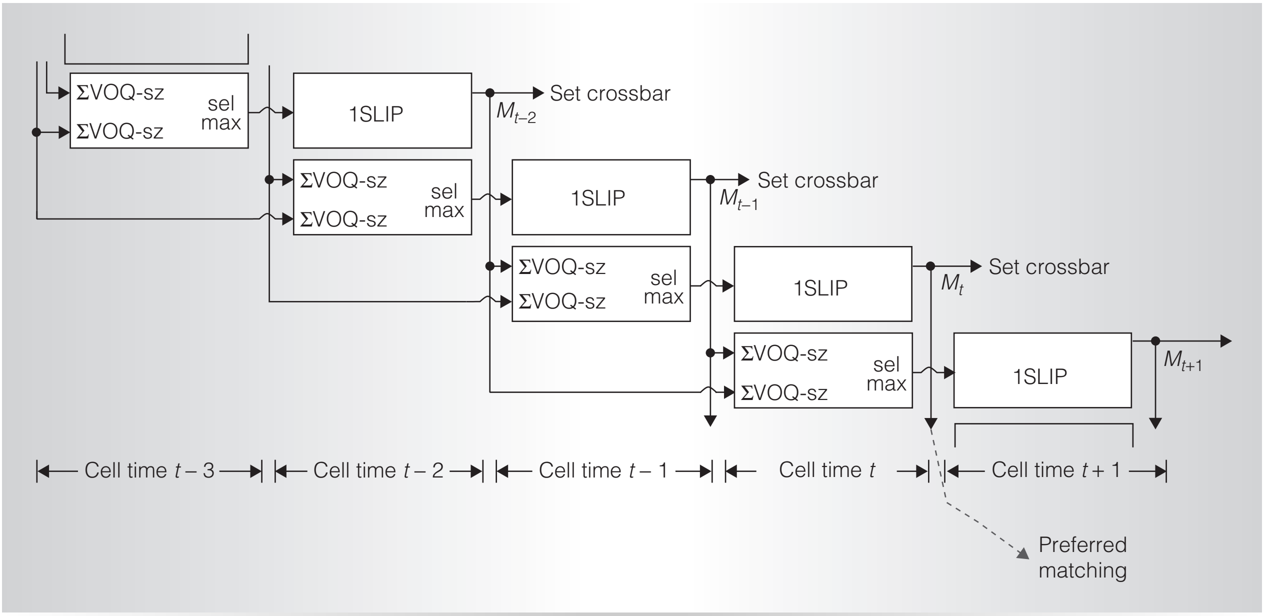

Figure 1. The proposed scheduler's pipelined operation. Matching weight computation and comparison is overlapped in time with schedule computation.

cell time t, we compute and compare the aggregate VOQ length (weight) of the two previous matchings M_{t-1} and M_{t-2}. We use the matching with the greatest weight as the preferred matching for 1SLIP at time $t+1$.

Tassiulas introduced the idea behind this type of scheduling with memory.[2] The concept is quite simple: once you find a matching with a good weight, maintain it until you find a better one. Tassiulas compared the present matching's weight against that of a random matching. Later, Giaccone et al.'s Algo2 used the notion of Hamiltonian walk in the space of $N!$ possible matchings.[3] These methods can approximate MWM, achieving full throughput for admissible traffic.[3] However, this is a blind search in a large space, and results in long delays and poor throughput for realistic VOQ sizes. Unlike these methods, ours doesn't search the matching space randomly, but delegates searching to 1SLIP.

Our algorithm systematically removes edges from the current preferred matching to escape the local maxima (in terms of matching weight) that scheduling with memory might accept. The removal of an edge releases some input and output ports. 1SLIP can combine these ports with others not included in the current preferred matching and find a new, possibly better configuration. Our edge-removal method secures "good" current matchings, while it can increase matching size from one cell time to the next, even when VOQs are nonuniformly loaded. In a word, the proposed algorithm extends *i*SLIP's desynchronization property to nonuniform loads.

Scheduling algorithm

Our 1SLIP scheduling core follows the three-phase regime of input request, output grant, and input accept. In our algorithm, 1SLIP runs in normal mode most of the time, periodically toggling to global-escape mode. In the following descriptions of these two modes, N is the crossbar's size and $F[i]$ denotes the preferred output of input i (Ø if it has none).

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

- *Filter input requests.* If input i has a VOQ cell for output $F[i]$, it sends a preferred request to that output (this pair will definitely be matched), filtering

```
Scheduler-FSM
        if (Cell Time mod $e$ == 0) then
            Global-Escape($F_{t-1}$,$M_{t-1}$);
        else
            if (Cell Time mod $s \neq 0$) then
                $F_{t-1}[q] = \emptyset$;  // local escape
                $q = (q + 1) \bmod N$;
            end if
            Normal($F_{t-1}$);
        end if
```

Figure 2. The scheduler's finite-state machine (FSM). The scheduler executes global escape every e cell times, executing normal 1SLIP at all other cell times. While in normal mode a pointer, q, visits all inputs one by one and nullifies their preferences.

out all other requests. Otherwise, if $F[i] = \emptyset$ 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 subsequent inputs starting from the one selected by $G[j]$ and grants the first request.
- *Input accept*. Input i scans outputs starting from the one selected by $A[i]$ and accepts the first grant.
- *Pointer update*. If a matching (preferred or nonpreferred) is achieved between input i and output j, $G[j] = (i + 1) \bmod N$ and $A[i] = (j + 1) \bmod N$.

Note that matching F might refer to VOQs that are empty when 1SLIP runs. In addition, F might not include some inputs with cells. 1SLIP will try to match the respective inputs with similarly available outputs.

GLOBAL_ESCAPE(F, M). F and M are the preferred matching and the crossbar schedule computed in the previous cell time. We denote the global-escape round-robin next-to-serve pointers of output j and input i as $eG[j]$ and $eA[i]$, respectively.

- *Remove all preferences*. Set $F[i] = \emptyset$, for all i. In this case, there are no preferred requests.
- *Input request*. Input i sends a request from any active VOQ. When there is more than one request, input i won't send a request from the VOQ corresponding to $M[i]$. We crop previously matched pairs to examine new matchings.
- *Output grant*. Output j grants the first request, except now it scans requests starting from the input selected by $eG[j]$.
- *Input accept*. Input i scans outputs, starting from the output selected by $eA[i]$ in this case and accepting the first one.
- *Pointer update*. Pointers are updated when a matching is achieved, except here $eG[j]$ and $eA[i]$ are updated instead of $G[j]$ or $A[i]$.

At time t, the scheduler performs two operations in parallel (also shown schematically in Figure 1):

- It computes the weight of matchings M_{t-2} and M_{t-1}, using the current VOQ lengths, and sets F_t equal to the matching with the maximum weight.
- It runs in normal or global-escape mode as outlined in the scheduler finite-state machine (FSM) in Figure 2. The scheduler executes global escape once every e cell times. At all other cell times, it selects normal 1SLIP. At some time instances before entering normal mode, we remove one edge from the current preferred matching (an operation called *local escape*). Specifically, we maintain a pointer, q, that visits inputs one by one in separate cell times, as controlled by parameter s, nullifying each visited input's preference.

As we discuss later, local escape searches for improved matchings lying near the current preferred matching.

Global-escape properties

Global escape searches for radically new matchings by ignoring current preferences. When it finds a matching with greater weight, normal 1SLIP will start preferring it; otherwise, normal 1SLIP will simply recover to

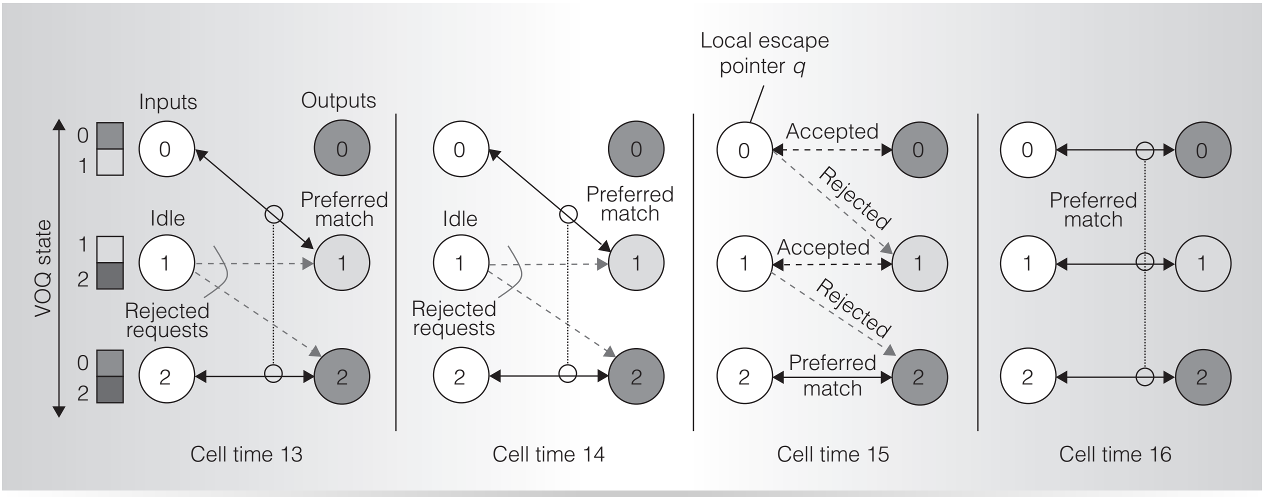

Figure 3. Local escape increases matching size under nonuniform traffic. Matching increases at multiple cell times as local-escape pointer q visits inputs and removes the corresponding edges.

the previous matching. Global escape prevents starvation and allows deterministic service guarantees, as the following theorem describes.

Theorem 1. In the worst case, every nonempty VOQ gets served in $O(N^2)$ time.

Proof. In the worst case, 1SLIP serves any nonempty VOQ in $N^2 + (N - 1)^2$ cell times. Serpanos and Antoniadis compute this as the delay of a tagged output j to grant a tagged input i, plus the delay of input i to accept the grant.[5] Global escape is an independent 1SLIP that runs once every e cell times without being affected by the preferred matchings because it uses private grant and accept pointers.

Therefore, global escape will select a tagged I/O pair within the delay of 1SLIP multiplied by constant e, which is still $O(N^2)$. However, global escape skips pairs that were matched in the previous cell time by normal mode (see the *input request* phase). This fact can't increase the worst-case delay per se. Global escape will serve the same pairs that ordinary 1SLIP would serve, skipping only some intermediate pairs that were matched in the previous cell time by normal 1SLIP. ■

Global escape can yield relatively poor matchings. Assume, for instance, that the average matching size required to sustain some specific traffic pattern with input load ρ is $S(\rho)$. Additionally, assume that normal 1SLIP achieves the targeted average matching size $S(\rho)$, while global escape 1SLIP achieves only 0.1 $S(\rho)$. Then, the net average matching size realized by the proposed algorithm will be $NS(\rho) = e - 1/e\ S(\rho) + 0.1/e\ S(\rho)$, or $e - 0.9/e\ S(\rho)$. Hence, we can adjust e (how frequently global-escape mode takes over) to minimize any possible throughput losses due to global escape. For $e = 100$, $NS(\rho) \geq 99.1/100\ S(\rho)$. For $e = 1$—that is, the algorithm always operates in global-escape mode—the algorithm performs similarly with ordinary 1SLIP. Larger e values allow preferred (good weight) matchings to stay on, improving throughput. But too high an e value might reduce the number of new matchings examined.

Local-escape properties

Local escape is critical to the proposed algorithm's throughput performance. When pointer q visits some input i, input i sets its preferred output o free to try other inputs. Effectively, output o can now match with other inputs that previously couldn't find a free output. Note that input i is also free to request all targeted outputs and has similar pairing possibilities. If both input i and output o find new pairs, the matching size can increase in a nonmaximal way. As with global escape, the new matching will stay on if it has adequately large weight.

Figure 3 demonstrates this behavior for a 3 × 3 switch ($N = 3$) (for simplicity, we ignore pipelining delays). When q visits input 0 at time 15, input 0 and its previously preferred output, 1, are released from their mutual commitment and try to pair with other ports. Due to our pointer update policy, pair $0 \rightarrow 1$ has the lowest priority to match because 1SLIP's output-grant and input-accept pointers in normal mode are $G[1] = (0 + 1) \bmod N = 1$ and $A[0] = (1 + 1) \bmod N = 2$, respectively. Hence, output port 1 matches with the previously unmatched input 1, and input 0 matches with the previously unmatched output 0. No grant comes to input 0 from the matched-by-preference output 2.

Importantly, when local escape removes an edge, it doesn't needlessly harm the size of the current preferred matching. If all ports requested by the visited input 0 are matched by preference, input 0 will pair again to its previously preferred output 1 with no throughput penalty.

Performance evaluation

We used performance simulations to test our algorithm and compare its performance with that of previous proposals. We simulate a 32×32 switch. All VOQs at every input share a buffer space $Q = 16{,}384$ cells. When this buffer is full, it discards arriving cells. We tuned parameters through extensive simulations. In the results presented here, the algorithm enters global escape once every 100 cell times ($e = 100$). While in normal mode, it doesn't perform local escape once every 3 cell times ($s = 3$). Note, however, that we can achieve good performance even if we always perform local escape in normal mode ($s = \infty$). The key is to perform local escape frequently to increase the matching size quickly.

The rationale behind setting $s = 3$ is the following. Suppose local escape finds a large weight matching at time t such that the compare-weight function selects this matching at cell times $t + 1$ and $t + 2$. If we perform local escape in both of these cell times, the final matchings might be worse than the original matching of time t. Then, the good matching of time t wouldn't be visible to the scheduler at time $t + 3$; rather, only the lower-quality matchings of cell times $t + 1$ and $t + 2$ would be visible. But with $s = 3$, either at time $t + 1$ or time $t + 2$, the algorithm won't perform local escape. Thus, the good matching of time t will be repeated and will be available at subsequent cell times. If our algorithm compared the weight of more matchings, thus going deeper in the past, the good matching could stay on for even larger s values.

First, we examine diagonal and power2 nonuniform Bernoulli traffic. In diagonal, input i sends two-thirds of its traffic to output $(2i + 2i/N) \bmod N$ (this is the perfect shuffle pair of i), and the remaining one-third to output $(1 + 2i + 2i/N) \bmod N$. In power2 (sometimes referred to as *logdiagonal*), every input i has traffic for all outputs, but sends twice as much traffic to output k than to $(k + 1) \bmod N$, for all $k \neq i - 1$. Figure 4 shows the results for the proposed system for exhaustive SLIP (E*i*SLIP),[6] 4SLIP (four iterations of *i*SLIP), and derandomized rotating double-static round robin (DRDSRR).[7] As the figure illustrates, our scheme delivers almost full throughput, with its delay under power2 traffic being significantly higher than under diagonal. All other systems saturate at loads ranging from 0.75 to 0.9.

We examined the performance of other randomized algorithms in the literature, and found that only MaxAPSARA[3] achieves similarly good delays under heavy nonuniform traffic. For example, MaxAPSARA's delay at a load close to 0.99 for diagonal traffic is approximately 200 cell times, which is near our algorithm's delay. We don't include additional results because our comparisons consider only equally practical schedulers that can be efficiently implemented in hardware and operate at high frequencies. MaxAPSARA, because of the large search space of good matchings, doesn't belong in this category.

Next, we examined throughput under Zipf Bernoulli traffic controlled by parameter k. In this scenario, input 0 sends to output i with probability

$$\mathrm{Zipf}(i) = i^{-k} / \sum_{j=1}^{N} j^{-k}.$$

We obtain probabilities at other inputs by circularly shifting the probability of input 0. Traffic is uniform when $k = 0$ and completely directed as k approaches infinity.

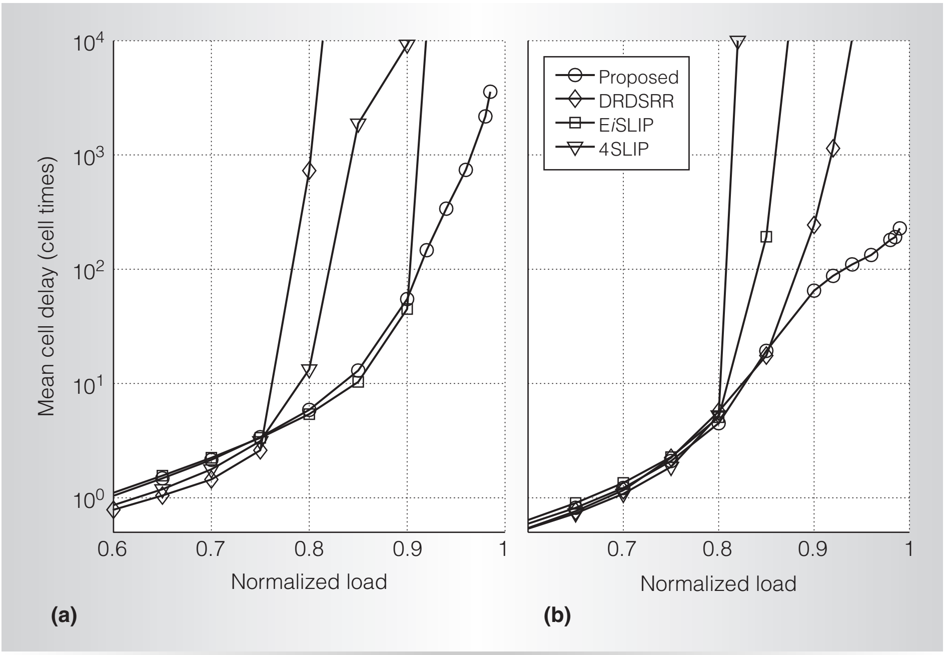

Figure 4. Our algorithm's delay under power2 (a) and diagonal (b) traffic for exhaustive SLIP (E*i*SLIP), 4SLIP (four iterations of *i*SLIP), and DRDSRR.

Figure 5 shows the switch throughput for the same scheduling algorithms. The proposed system achieves full throughput for both uniform and nonuniform traffic, whereas DRDSRR's throughput is below 0.8, and E*i*SLIP's is below 0.85. The no-escape plot that performs almost identically to E*i*SLIP corresponds to the proposed algorithm with neither global nor local escape. Finally, 4SLIP performs similarly to E*i*SLIP but for different values of *k*.

We also tested global only, in which we never perform local escape; and local only, in which we never enter global-escape mode. Under diagonal traffic, local only performs slightly worse than global only. However, for Zipf traffic, local only achieves almost full throughput, whereas throughput for global only is below 0.9. These results indicate that starting the search from scratch, as global only does, is beneficial when the matching space is relatively small (in diagonal, every input has only two VOQ candidates), but performs poorly as space size increases. Searching locally might be slower, but it's more effective in a large matching space as it improves, step-by-step, the current preferred matching.

Matching breakdown

Figure 6 plots the percentage of pairings enforced by preferred matchings as a function of input load. Although preferred matchings are present even at low loads, they only rarely correspond to nonempty VOQs. Thus, most pairings are due to 1SLIP—that is, not enforced by preference. As the load increases, this situation is reversed, with the transition occurring faster for diagonal (nonuniform) traffic. This happens because the VOQs of heavy diagonal flows remain backlogged for a long time, and thus get preferentially matched.

Throughput under bursty size and small input buffers

So far, we've assumed that the VOQs at each input share a space for 16,384 cells. In this experiment, we test the switch throughput for smaller buffer sizes (Q) per input. When Q is small, bursts of cells contending for the same switch output can yield extensive cell drops. To account for this effect, we examined several average burst sizes: 12, 36, and 144, where 144 cells amount to 9 Kbytes for 64-byte cells—that is, close to the size of Ethernet jumbo frames.

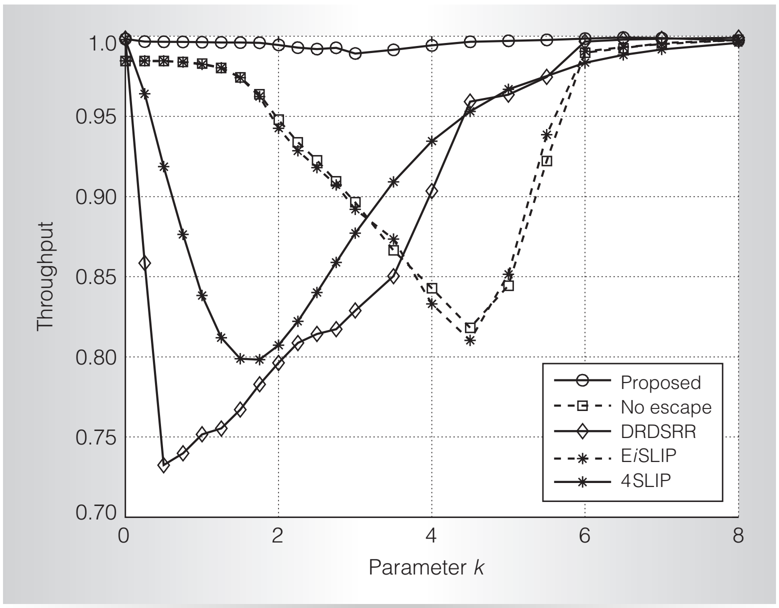

Figure 5. Throughput under Zipf Bernoulli traffic controlled by parameter *k*. As in Figure 4, we show results for E*i*SLIP, 4SLIP, and DRDSRR.

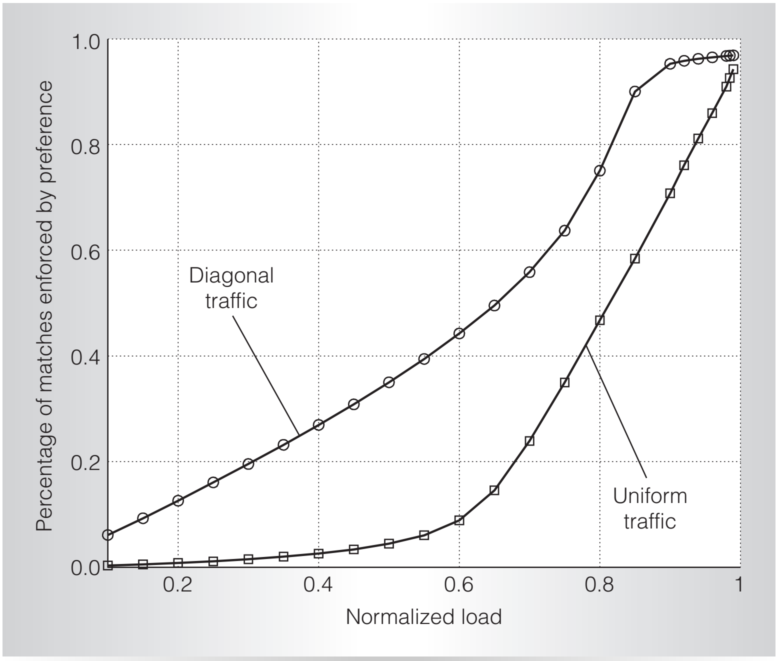

Figure 6. Percentage of preferred matchings as a function of input load. At high loads both uniform and diagonal traffic rely on the preferred large-weight matchings discovered by our algorithm.

Bursty cell arrivals are based on a two-state (on/off) Markov chain, giving exponentially distributed burst lengths. We also plot throughput for burst size equal to 1, which, in this experiment, corresponds to a simple Bernoulli process. Figure 7a shows the proposed system's throughput for $Q = \{16; 256; 4{,}096; 16{,}384\}$. One group of plots is for uniform and the other for nonuniform Zipf traffic with $k = 0.75$. For small buffer sizes, this value of k yields the worst throughput in the systems we examine. As is apparent, nonuniform traffic doesn't affect the proposed algorithm's throughput. All plots manifest that throughput decreases with burst size and increases with buffer size Q. For $Q \geq 4{,}096$, however, burstiness only marginally affects throughput.

For completeness, Figure 7b shows results for the same experiment for pure MWM, which is too complex to implement in reality because it requires $O(N^3)$ iterations per schedule computation. As the figure shows, our algorithm performs similarly to MWM. The only notable discrepancies are for Bernoulli traffic and small buffer sizes, which we can attribute to the multiple cell times required for the proposed algorithm to increase the matching size and weight. For highly bursty traffic, VOQs persist longer, and this discrepancy quickly diminishes.

Fair allocation of congested link bandwidth

Our algorithm prefers large VOQs, and thus, similarly to MWM, it might distribute the bandwidth of overloaded outputs unfairly. We can improve fairness by using an additional arbitration stage in tandem with the proposed scheduler, without requiring any further modifications or harming switch throughput.[8,9] Specifically, we equip the crossbar scheduler with N per-output, fair arbiters, which we call *regulation arbiters* to distinguish them from the 1SLIP per-output arbiters. The requests from the VOQs are first routed to the regulation arbiter of the targeted output, where they are registered in per-flow request counters. Every regulation arbiter serves one nonzero request counter in every cell time. After serving requests, the arbiter registers them in an additional set of per-flow counters, which the proposed scheduler uses to find any nonempty VOQs and to calculate the matchings' weight.

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 1 depicts the proposed algorithm's flow service rates. When regulation arbiters are absent (basic rate), the proposed system favors the two heavy flows and only sporadically serves flow 4 → 1 due to the escape modes. On the contrary, when using regulation arbiters that implement the round-robin discipline, the system distributes equal rates to flows without requiring any additional change to the scheduler's operation. Furthermore, adopting weighted round-robin (WRR) regulation arbiters lets the system serve flows according to their weighted max-min fair shares. We obtain the WRR results after assigning weights of 10, 20, and 30 to flows 1 → 1, 2 → 1, and 4 → 1, respectively.

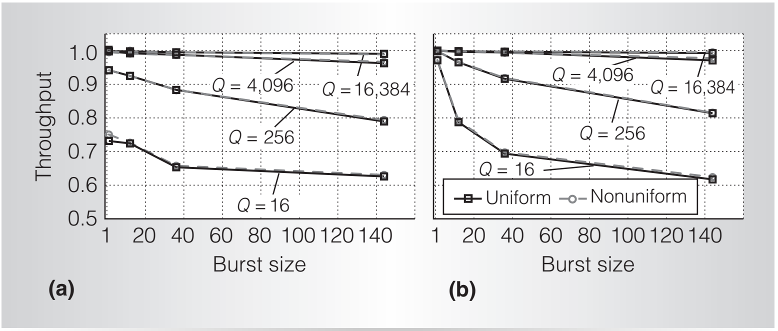

Figure 7. Throughput versus burst size for various input buffer sizes, Q. Results for uniform and nonuniform Zipf traffic with $k = 0.75$ (a), and results for the same experiment for maximum weight matching (MWM) (b).

Table 1. Service rates of three flows that overload output 1 for the proposed scheduler.

		Service rates		
Flow	**Rate**	**Basic**	**Round robin**	**Weighted round robin**
1 → 1	1.0	0.51	0.33	0.16
2 → 1	0.9	0.45	0.33	0.34
4 → 1	0.5	0.04	0.33	0.50

Local-escape extension

As we've shown, our algorithm tends to increase matching size even for nonuniform VOQs. Here, we examine how closely our algorithm approaches MWM. To do so, we configured persistent VOQs with two length distributions.

- In *bimodal*, each VOQ is 10 cells long with probability 0.9, and 900 cells otherwise.
- In *uniform,* the length of each VOQ is uniformly selected in the interval [0, 1000].

Given some VOQs' state, we measure how the ratio of the matching weight achieved by the proposed algorithm to the maximum possible weight, as computed offline by the MWM algorithm, evolves over time. Figure 8 shows the results. Figure 8a corresponds to the time instances that our algorithm runs in normal mode, and Figure 8b reports the ratio of MWM at cell times that are multiples of 100—that is, when global-escape mode is performed. Our algorithm ("basic" in the figure) achieves only a small percentage of the MWM (0.3 for bimodal VOQ lengths and 0.65 for uniform) because, once a maximum-size matching is found, local escape continues to free one input and one output from the preferred matching. However, because all other ports have preferred pairs, the released ports will match together again. In other words, local escape no longer improves the matching's weight.

To improve this behavior, we modify local escape using an idea from the MaxAPSARA algorithm.[3] Up to this point, we've used one pointer q that circularly visits inputs, and we blindly removed the edge selected by this pointer. We now maintain two pointers, q_1 and q_2. Every time we perform a new local escape, these pointers select two different inputs, visiting every possible pair in N^2 rounds.

Suppose, for example, we enter local escape at some cell time, with q_1 pointing at input 1 and q_2 pointing at input 3. If neither input 1 nor input 3 has a preferred output, we go directly to the pointer update phase, where we can examine a new pair of inputs upon re-entering local escape. Assume this isn't the case, and that input 1 is presently paired with output 1, and input 3 with output 3. Now we examine whether the sum of the VOQ lengths 1 → 1 and 3 → 3 is less than or equal to the sum of VOQ lengths 1 → 3 and 3 → 1. If it is, we can increase the present matching's weight if inputs 1 and 3

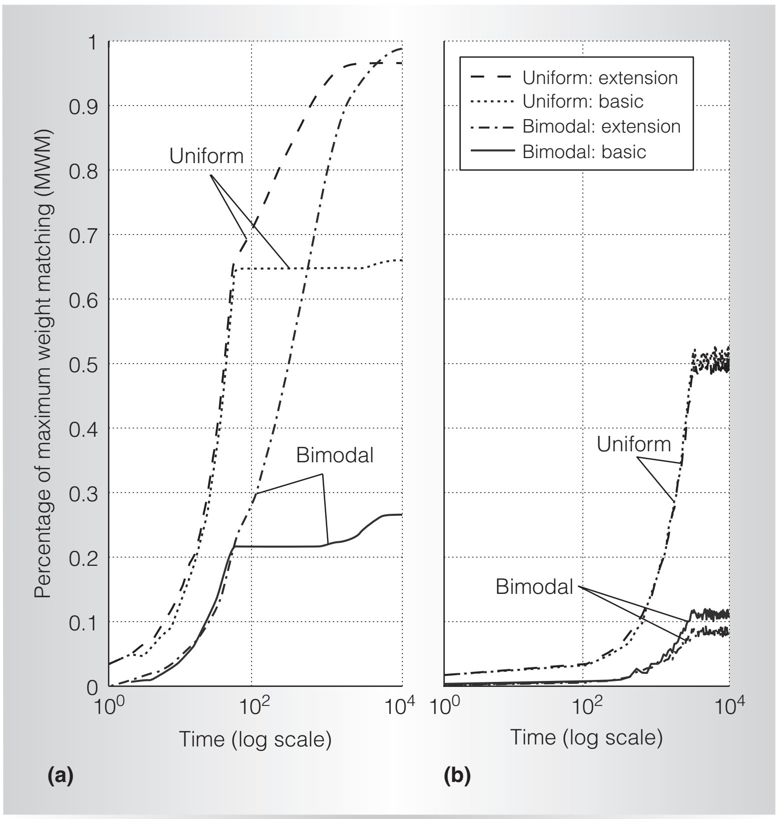

Figure 8. Ratio of the achieved weights relative to MWM for normal mode with local escapes (a) and global escape only (that is, once every 100 cell times) (b).

exchange outputs. In this case, we nullify the preference of both inputs 1 and 3. Otherwise, we simply nullify the preference of the input pointed to by q_2, thus performing the basic local-escape function. Whether we remove an edge or not, we finally update the pointers as follows: If $(q_2 + 1) \bmod N = q_1$, we set $q_1 = (q_1 + 1) \bmod N$, and $q_2 = (q_1 + 1) \bmod N$; otherwise $q_2 = (q_2 + 1) \bmod N$.

Effectively, when 1SLIP receives the preferred matching from local escape, both inputs 1 and 3 will broadcast normal requests. However, the grant pointer of output 1 will point to $(1 + 1) \bmod N$, and the grant pointer of output 3 to $(3 + 1) \bmod N$. Hence, pairs $1 \rightarrow 3$ and $3 \rightarrow 1$ can be enforced, and the scheduler will start preferring them.

This modified local-escape function can reach the full extent of MWM, as Figure 8 shows. Regardless, additional results indicate that the delay-throughput performance is only slightly improved for the traffic patterns we examined compared with the basic scheme. In addition, this modification requires more complex hardware than the basic approach.

Crossbar scheduler implementation

Figure 9 shows our crossbar scheduler's hardware organization. It consists of two pipeline stages—*weight computation* and *schedule computation*. In our implementation, cell time equals the pipeline's clock period.

In the weight-computation stage, we compute the weight corresponding to the current matching using a multioperand adder that follows a Wallace tree structure and a high-speed final adder. The scheduler contains only one weight-computation unit, which all inputs share. Each of the scheduler's input ports has N request counters (one for each output) that store the number of cells present in each VOQ. At every cycle, the inputs transmit to the weight-computation unit the value of the request counters corresponding to the matched outputs at the current and previous cell times.

The schedule-computation stage takes as input the weights computed in the first pipeline stage. The maximum selector decides which of the two weights under comparison is larger. If the weight of M_{t-1} is greater than the weight of M_{t-2}, the scheduler selects M_{t-1} as the preferred matching. In the opposite case, it uses M_{t-2}.

Following the proposed scheduling algorithm, the scheduler filters the active requests according to the preferred matching. To save delay, as Figure 9 shows, we use two filtering units per input running in parallel. M_{t-1} drives the first filtering unit and M_{t-2} the second. Before M_{t-1} and M_{t-2} enter the corresponding filtering units, the scheduler masks these preferred matchings with the state of the current local escape and that of the global escape, thus nullifying some or all the preferences, respectively. The multiplexer at the output of the input preference unit receives the two filtered requests and forwards one of the them to the 1SLIP core. The multiplexer is controlled by the maximum selector that runs in parallel. In this way, the filtering unit's delay and the masking operation are partially hidden

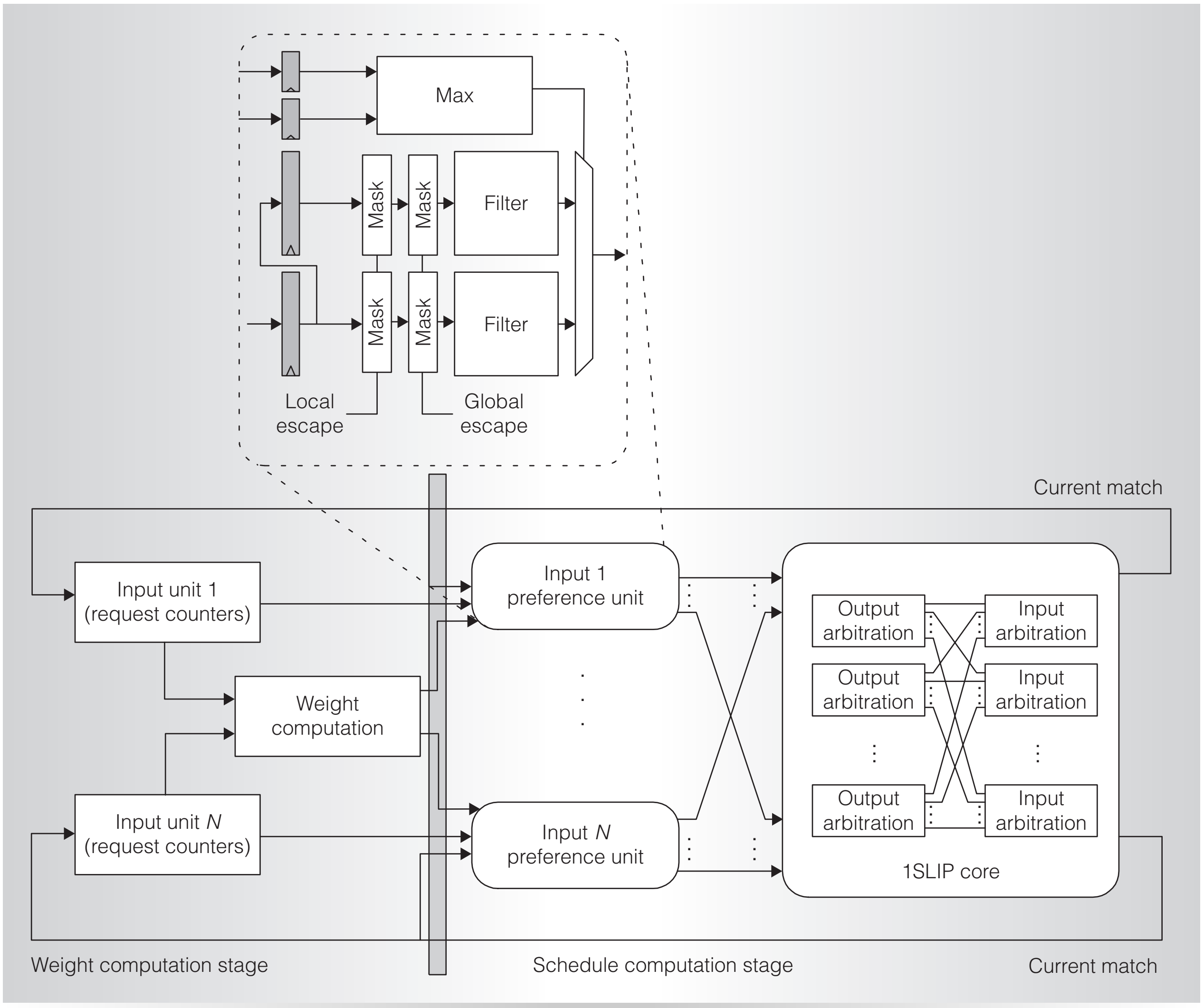

Figure 9. Block diagram of the proposed scheduler. It is organized in two stages that implement matching weight computation, request filtering, and new schedule computation.

because they overlap in time with the maximum selector's operation.

Area-delay breakdown

We implemented the scheduler in 130-nm CMOS technology using a standard-cell-based design flow. We described the functionality of the scheduler in Verilog, and synthesized, placed, and routed the circuit using the Synopsys Design Compiler and Cadence SOC encounter.

Figure 10 shows the final layout of the 32 × 32 scheduler. The scheduler is partitioned in a 6 × 6 grid, with the weight-computation unit partially covering the four blocks in the middle of the circuit. We assigned the remaining 32 blocks to the circuits required for each port of the scheduler. At each block, we implement an input port's request counters and preference unit, as well as that port's 1SLIP input and output arbiters.

The scheduler roughly occupies 3 mm^2 and operates with a clock period of 3.2 ns, satisfying the constraints of line rates above 100 gigabits per second (Gbps) and 40-byte IP packets. The 14-bit-wide request counters are the most area demanding part of the circuit, as Figure 10 shows. In total, the logic cells and memory elements use

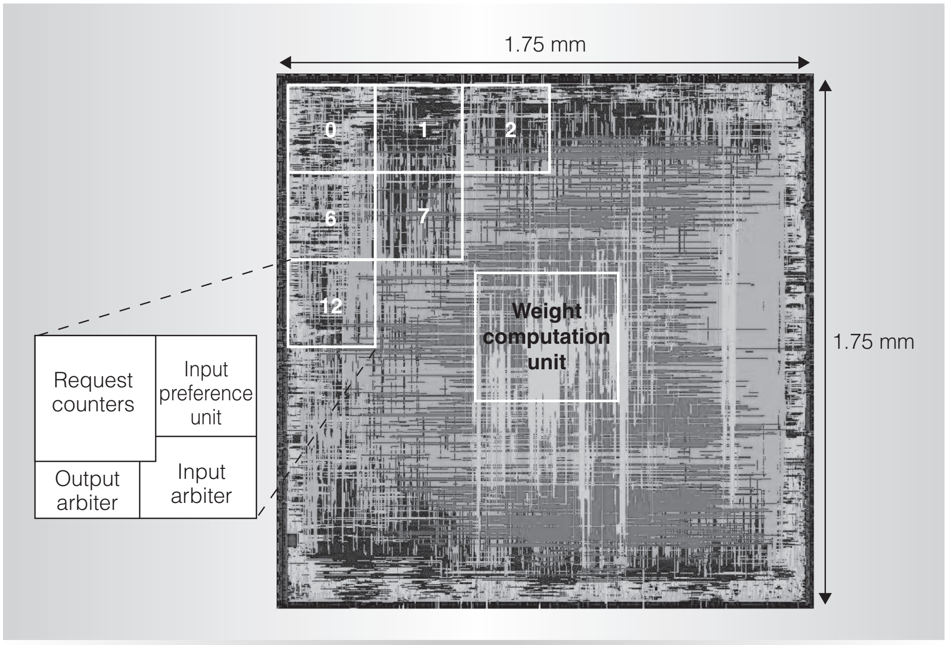

Figure 10. Layout of the proposed scheduler. Each block contains request counters, input preference unit, and input and output arbiters.

70 percent of the scheduler's core area, while the extra space facilitates the scheduler logic's complex wiring patterns.

In terms of delay, schedule computation is the most demanding part of the circuit—in particular, the input and output round-robin arbiters. The input preference unit's logic, and the cascaded output and input arbiters, consume more than 2 ns. The remaining delay corresponds to clocking overhead and wiring delay, including the delay of the buffers used for driving longer wires. When we account for technology scaling, we conclude that we can achieve the same speed for larger switch sizes, such as 64×64.

The 1SLIP core

The 1SLIP implementation consists of per-input and per-output arbiters.[1] Each arbiter should be able to support programmable priorities—that is, to let us change the highest-priority next-to-serve request in every cycle. The generic form of a programmable-priority arbiter (PPA) consists of core arbitration logic that scans requests cyclically, beginning from an arbitrary position set by the priority vector P, and pointer update logic that updates P according to some policy, such as round robin.

The cyclic operation of a PPA's core arbitration logic complicates its design. Gupta and McKeown present several variants with different area-delay characteristics.[1] The most efficient approach, which we call the *dual-path arbiter,* uses two fixed-priority arbiters (FPAs, also called priority encoders) working in parallel. The upper FPA scans requests starting from the highest-priority request h up to position $N - 1$. It doesn't scan the range $[0, h - 1]$. The lower FPA works on all incoming requests. For a request in the range $[h, N - 1]$, the correct output comes from the upper FPA; otherwise, it comes from the lower FPA.

Our 1SLIP core follows the same overall structure, but with the arbiters modified to distinguish between preferred and normal requests. In addition to the request vector, each arbiter accepts a preference vector that marks the preferred request. So, in parallel with the arbitration logic, we need an N-input OR tree to decide whether there's at least one active preference. If there is, we bypass the arbitration logic's output and give the grant directly to the active preference.

We also maintain two priority (next-to-serve) pointers per arbiter, one each for normal and escape modes. This extra circuitry doesn't significantly affect the PPA's area but inserts some nonnegligible delay overhead. To compensate for this increased delay, we designed a new PPA that is significantly faster than dual path and speeds up the entire 1SLIP core.

New programmable-priority arbiters

Our new PPA takes as input an N-bit request vector, R, and produces an N-bit grant vector G. We encode the highest-priority position in one-hot form in variable P.

First, we introduce a new signal for each bit position, named X_i, that serves as priority transfer. Asserting X_i means the ith request has the highest priority to win a grant. Of course, a grant is given to position i only when it has an active request—that is, $R_i = 1$. We therefore set the grant signal G_i as $G_i = R_i \cdot X_i$. The symbol $\cdot$ denotes the logical AND and $+$ the logical OR operation.

Next, we explore what activates X_i. Request R_i initially gets the highest priority when the priority vector's bit P_i is set. The position indexed by the P vector is where the priority is generated. We can transfer this priority to the most significant positions step by step only when the positions in between don't have an active request, and so can't produce a grant. Therefore, position i can get the priority from its neighbor $i - 1$ only when $R_{i-1} = 0$. Merging the two cases, we can say that X_i is set when $P_i = 1$ or $R_{i-1} = 0$ and the incoming priority transfer X_{i-1} is activated. Expressing this result in Boolean algebra leads to the following relation:

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

Observe that the priority transfer signal X_{N-1} should be fed back to position 0—that is, $X_{N-1} = X_{-1}$—to guarantee the diminishing priority's cyclic transfer. For the same reason, $R_{-1} = R_{N-1}$. If the priority transfer X_{N-1} is connected directly back to position 0, a combinational loop is created that commercial static timing-analysis tools can't treat efficiently.

However, we can treat arbitration's cyclic operation differently based on the following observation. The recursive definition of X_i in Equation 1 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 priority signal P_i (*priority generate*), and instead of the carry-propagate bit p_i, we use the inverted request signal $\bar{R}_{i-1}$ (*priority propagate*). In fact, computation of the priority transfer bits is equivalent to an end-around carry operation, in which the carry-out signal should be fed back to the carry-in position. Using the most efficient approach for handling such end-around carry operations,[10] we can compute the priority transfer bits in parallel using a butterfly-like carry-lookahead structure.

Figure 11 shows an implementation of our PPA using four and eight inputs. The four-input arbiter is drawn directly in logic gates, whereas the eight-input arbiter is drawn more abstractly to better show the butterfly-like carry-lookahead structure. The long lines inside the priority transfer unit are the proposed circuit's only drawback. Although the capacitance added by these lines degrades the circuit's delay by a small percentage, the new PPA is still faster than previous implementations.

To quantify the proposed PPA's delay savings separately from the scheduler implementation, we synthesized the new PPA and the dual-path design using the Synopsys Design Compiler driven by the same technology. We set the circuits' available input capacitance equal to that of a drive-strength-2 inverter, and the output loading capacitance four times larger than that. We optimized each design for speed, targeting the minimum possible delay. Table 2 shows our results. They demonstrate that the proposed solution (Proposed I) averages 20 percent faster than dual path.[1] In fact, the proposed arbiter's delay for 64 inputs is less than dual path's delay for 32 inputs. Our solution's delay savings can alternatively be traded for reduced layout area. For the Proposed II column, we sized our circuit's gates for a delay target equal to the delay of the dual-path design. As the table shows, Proposed II saves more than 16 percent in layout area compared to dual path.

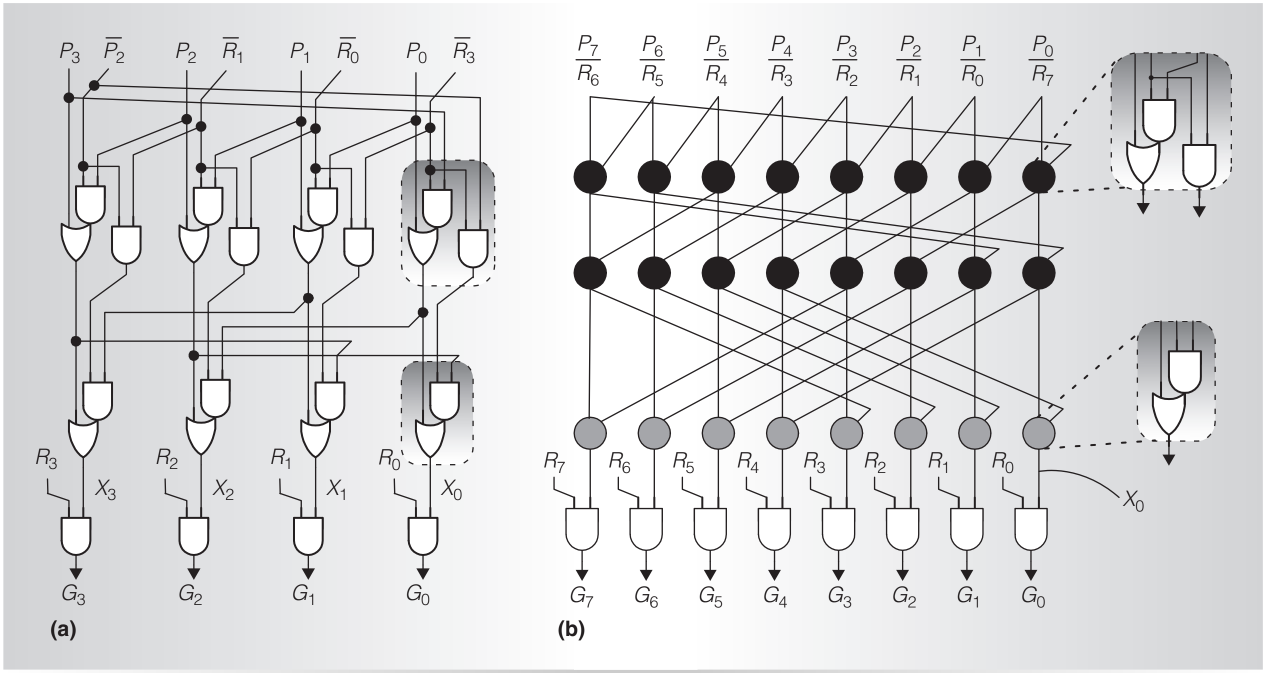
P_3 $\overline{P_2}$ P_2 $\overline{R_1}$ P_1 $\overline{R_0}$ P_0 $\overline{R_3}$
R_3 X_3 R_2 X_2 R_1 X_1 R_0 X_0
G_3 G_2 G_1 G_0
(a)
P_7/R_6 P_6/R_5 P_5/R_4 P_4/R_3 P_3/R_2 P_2/R_1 P_1/R_0 P_0/R_7
R_7 R_6 R_5 R_4 R_3 R_2 R_1 R_0 X_0
G_7 G_6 G_5 G_4 G_3 G_2 G_1 G_0
(b)

Figure 11. Programmable-priority arbiter implementations: proposed four-input (a) and eight-input (b) PPAs that cycle the priority transfer signals inside the priority transfer computation unit, thus avoiding any combinational loops.

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

Number of arbiter inputs	Proposed I Delay	Proposed I Area	Proposed II Delay	Proposed II Area	Dual path Delay	Dual path Area
64	595	10,593	740	8,205	740	8,990
32	510	4,612	640	3,542	640	4,134
16	430	1,961	550	1,473	550	1,948

There have been an overwhelming number of proposals for crossbar schedulers during the last decade or more, but few combine high throughput, low delay, and fairness in a compact, practical design. Our algorithm provides a heuristic that increases crossbar matching size for arbitrarily loaded VOQs. Its efficient pipelined hardware implementations offer new possibilities for the design of high-speed and large-size switches. MICRO

Acknowledgments

The European Commission supported this work in the context of the SARC (Scalable Computer Architectures) integrated project 27648 (FP6), and the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HIPEAC) network of excellence (projects 004408 and 217068) under the Interconnects research cluster.

References

1. P. Gupta and N. McKeown, ''Design and Implementation of a Fast Crossbar Scheduler,'' *IEEE Micro*, vol. 19, no. 1, Jan./Feb. 1999, pp. 20-28.
2. L. Tassiulas, ''Linear Complexity Algorithms for Maximum Throughput in Radio Networks and Input Queues Switches,''

Proc. IEEE Infocom, IEEE Press, 1998, pp. 533-539.
3. P. Giaccone, B. Prabhakar, and D. Shah, ''Randomized Scheduling Algorithms for High-Aggregate Bandwidth Switches,'' *IEEE J. Selected Areas Comm.,* vol. 21, no. 4, May 2003, pp. 546-559.
4. I. Keslassy and N. McKeown, ''Analysis of Scheduling Algorithms that Provide 100% Throughput in Input-Queued Switches,'' *Proc. 39th Allerton Conf. Comm., Control, and Computing,* 2001; http://tiny-tera.stanford.edu/~nickm/papers/allerton2001.pdf.
5. D. Serpanos and P. Antoniadis, ''FIRM: A Class of Distributed Scheduling Algorithms for High-Speed ATM Switches with Multiple Input Queues,'' *Proc. IEEE Infocom,* IEEE Press, 2000, pp. 548-555.
6. Y. Li, S. Panwar, and H.J. Chao, ''Exhaustive Service Matching Algorithms for Input Queued Switches,'' *Proc. IEEE High Performance Switching and Routing* (HPSR), IEEE Press, 2004, pp. 253-258.
7. J. Liu et al., ''Stable Round-Robin Scheduling Algorithms for High-Performance Input-Queued Switches,'' *Proc. IEEE Hot Interconnects,* IEEE CS Press, 2002, p. 43.
8. N. Chrysos and M. Katevenis, ''Scheduling in Non-Blocking Buffered Three-Stage Switching Fabrics,'' *Proc. IEEE Infocom,* IEEE Press, 2006, pp. 1-13.
9. N. Kumar, N.R. Pan, and D. Shah, ''Fair Scheduling in Input-Queued Switches under Inadmissible Traffic,'' *Proc. IEEE Globecom,* IEEE Press, 2004, pp. 1713-1717.
10. L. Kalampoukas et al., ''High-Speed Parallel-Prefix Modulo $2^n - 1$ Adders,'' *IEEE Trans. Computers,* vol. 49, no. 7, July 2000, pp. 673-680.

Nikos Chrysos is a postdoctoral fellow with IBM Research in Zurich. His current research interests include algorithms and hardware for multiprocessor interconnection networks and optical networks. Chrysos has a PhD in computer science from the University of Crete.

Giorgos Dimitrakopoulos is a postdoctoral fellow at the Institute of Computer Science of the Foundation for Research and Technology—Hellas (FORTH) and a visiting assistant professor at the University of Crete. His research interests include VLSI design, computer architecture, and energy optimization of digital systems. Dimitrakopoulos has a PhD in computer engineering from the University of Patras.

Readers can contact Giorgos Dimitrakopoulos, ICS-FORTH, 100 Plastira Ave., Vassilika Vouton, Heraklion, Crete GR-70013, Greece; gdimitrak@gmail.com.