

# Distributed Crossbar Schedulers

Cyriel Minkenbergh, François Abel  
IBM Research, Zurich Research Laboratory  
 Säumerstrasse 4, 8803 Rüschlikon, Switzerland  
sil@zurich.ibm.com

Enrico Schiattarella\*  
Dipartimento di Elettronica, Politecnico di Torino  
Corso Duca degli Abruzzi 24, 10129 Torino, Italy  
enrico.schiattarella@polito.it

**Abstract**—The goal of this work is to enable distributed (multi-chip) implementations of iterative matching algorithms for crossbar-based packet switches, as opposed to the traditional monolithic (single-chip) ones. The practical motivation for this effort is the design and implementation in FPGAs of a scheduler for a 64-port optical crossbar switch. Sizing experiments show that the scheduler logic must be distributed over multiple devices, which introduces a number of new challenges. Most importantly, the physical distances introduce latencies that exceed the timing requirements, and the separation of logical units prevents shared access to status information. We consider three levels of distribution, and present techniques to mitigate the consequences of specific distribution levels. The performance results obtained via simulation show that, using these methods, a distributed scheduler can achieve a performance close to that of a monolithic scheduler, even with large internal latencies.

## I. INTRODUCTION

Input-queued (IQ), crossbar-based switching architectures have emerged as a popular choice for the development of high-performance network switches and routers, largely because they offer a combination of good performance and practicality. These features also make them an attractive choice for other application domains, such as storage area networks (SANs) [1] and high-performance computing (HPC) interconnects [2], [3]. These applications call for a large number of ports, high line rates, and low latency.

IQ switches usually operate on fixed-size data units called cells. Input buffers are organized as virtual output queues (VOQs) to avoid Head-of-Line (HOL) blocking. A critical component of the switch is the scheduler, which at every time slot computes a matching between inputs and outputs and configures the crossbar. As the known optimum matching algorithms are not practically feasible, heuristic, parallel, iterative matching algorithms are typically employed. These algorithms employ  $2N$  independent selectors (also referred to as arbiters in [4]–[6]) at the inputs and the outputs. These selectors perform one-out-of- $N$  selections using, e.g., random or round-robin policies.

Despite extremely high-density CMOS technologies, the scheduler complexity quickly exceeds single-chip implementation limits as the number of ports  $N$  grows. The implementation is limited by gate count, pin count, I/O bandwidth, and wiring, owing to the high degree of connectivity between

the input and output selectors. In addition to managing complexity, a distributed implementation spreads the overall power dissipation over more devices, thus eliminating hot spots.

The main contribution of this work is a set of techniques that allows multi-chip implementations of such matching algorithms, enabling the construction of schedulers for large switches, while achieving a level of performance that is close to that of a monolithic implementation.

Section II recalls the most important iterative algorithms and their properties. Section III discusses the challenges that must be mastered to enable a distributed implementation, and Section IV presents a novel technique to achieve this goal. Section V discusses support for multiple iterations. Section VI shows simulation results that demonstrate the level of performance attainable with such a distributed implementation. Finally, we conclude in Section VII.

## II. ITERATIVE MATCHING ALGORITHMS

Sequential schedulers such as RPA [7] and RRGs [8] are inherently suitable for distribution, because they let inputs independently add edges to a circulating matching. These schemes can be pipelined to produce a matching at every time slot, but they suffer from a major drawback: they impose an average pipeline latency of  $N/2$  time slots, which increases further if inputs schedulers are physically distributed. Thus, they are not appropriate for the applications we are targeting, which demand both low latency and high port count. Therefore, only parallel iterative matching algorithms such as *i*-SLIP [4], FIRM [5], and DRRM [6] are considered here. These are widely used owing to the key advantages they offer:

- 1) High performance: More precisely, they guarantee 100% throughput under uniform uncorrelated traffic with a single iteration. Additional iterations significantly reduce the mean latency.
- 2) Fairness: They ensure that under any traffic pattern any nonempty VOQ receives service within finite time.
- 3) Practicality: Although a total of  $2N$  selectors (one per input and one per output) is required, these selectors operate independently and in parallel. Thus, high matching rates can be achieved. Moreover, the selectors are readily implemented in fast hardware [9].

### A. Two- vs. three-phase algorithms

Iterative matching algorithms can be classified into two- and three-phase algorithms according to the number of steps per iteration. In three-phase algorithms, there are request, grant,

\*This work was performed while the author was at the IBM Zurich Research Laboratory, Rüschlikon, Switzerland.

This research is supported in part by the University of California under subcontract number B527064.

TABLE I  
SIZING IN XILINX VIRTEX-II-PRO (SPEED GRADE -6), FROM [2].

$N$	input + output selectors							only output selectors
	4	8	16	32	48	52	64	64
slices	266	1K	4.5K	15K	35K	41K	—	31K
%	0.60	2.43	10.3	34.1	78.6	93.8	—	69
#nets	2K	8K	34K	115K	264K	317K	—	249K

and accept steps in every iteration. In the request phase, every input sends a request to every output it has at least one cell for. In the grant phase, every output independently selects one request to grant. As these decisions are independent, multiple outputs may grant the same input. Therefore, in the third phase, every input selects one grant to accept. Two-phase algorithms, on the other hand, comprise only a request and a grant phase. In the request phase, every input sends a request to one output for which it has at least one cell. In the grant phase, every output independently selects a request to grant. Because every input can receive at most one grant, there is no need for an accept phase, i.e., every grant is automatically accepted. *i*-SLIP and FIRM are three-phase algorithms, whereas DRRM is a two-phase one.

These algorithms are pointer-based, meaning that input and output selections are based on a prioritized round-robin mechanism, i.e., the input (output) selector chooses the first eligible output (input) starting from the position indicated by a pointer. The pointer-update policy is a crucial characteristic of each algorithm and must be chosen carefully to guarantee performance and fairness. The update policies employed by these algorithms share a common trait: once a connection becomes highest priority, it will be given precedence over the other competing ones until it is established. In *i*-SLIP this is achieved by having an output grant the same input (in the first iteration) until the grant is accepted. In DRRM, on the contrary, an input will keep requesting the same output (in the first iteration) until it receives a grant. This feature guarantees fairness and leads to *pointer desynchronization* [10], i.e., it assures that under heavy traffic (when all VOQs are nonempty) each output grants a different input (*i*-SLIP) or each input requests a different output (DRRM). When this happens, there are no conflicts and a maximum-size matching is produced in every time slot, leading to 100% throughput.

### B. Sizing experiments

This study is motivated by the implementation of a crossbar scheduler for a 64-port optical switch demonstrator (called OSMOSIS) with 40-Gb/s ports for high-performance computing applications [2], [11]. One of the challenges in this project is to implement a scheduler of this size and speed in FPGA technology, which is used mainly for reasons of cost and flexibility.

Our sizing results, shown in Table I (also previously reported in [2]), demonstrate that a monolithic implementation does not fit in the targeted FPGA device, which is the biggest and fastest FPGA available from Xilinx at the time of implementation, namely, the “xc2vp100-6ff1704,” a Virtex-

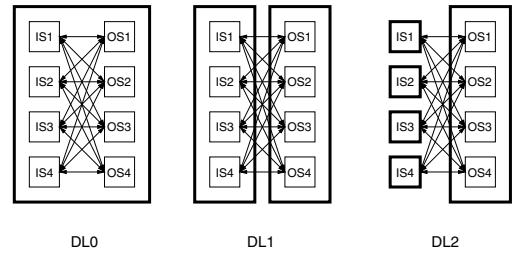


Fig. 1. Schematic representations of the three distribution levels. Bold lines represent the device boundaries. IS = input selector, OS = output selector.

II-Pro series FPGA providing 8 M system gates (100 K logic cells)<sup>1</sup> and 1040 users I/Os.

Table I presents the sizing measurements for the unconstrained placement and routing of the request-grant-accept phases of *i*-SLIP, based on the pipelined implementation described in [9]. These sizing numbers only represent the core of the algorithm without the I/O interfaces required to convey the external requests and grants to/from the scheduler device. The table lists the device utilization in number of slices, percentage, and number of nets.

The device utilization results of Table I show that the largest monolithic *i*-SLIP scheduler feasible in a single Virtex-II-Pro xc2vp100 is somewhere in the range of  $52 \times 52$ . Therefore, we cannot use a monolithic matching algorithm for our centralized crossbar scheduler. Clearly, this holds for three- as well as two-phase algorithms, as both require  $2N$  selectors. However, the rightmost column of Table I shows that it is feasible to implement 64 output selectors in a single device. These results indicate that without distribution the full scheduler cannot be implemented.

### III. DISTRIBUTION CHALLENGES

The above sizing results call for a distributed implementation, which entails partitioning the selectors over multiple physical devices. In a monolithic implementation, the selectors are tightly coupled and decisions taken at the inputs are known to the outputs (and vice versa) within the same time slot. In a distributed implementation, in contrast, the communication latency between physical devices starts to play a significant role because of wire and I/O pin delays. This inter-chip latency can amount to multiple time slots when pin sharing or (de)serialization techniques are required to overcome the I/O pin count limitation of current packages. This latency implies that critical information needed, e.g. to update pointers or issue new requests, is not available in a timely fashion. Consequently, distribution will cause the performance and the fairness of an iterative algorithm to deteriorate. For ease of reference, we introduce three levels of distribution DL0 through DL2, as illustrated in Fig. 1:

DL0 Monolithic implementation: All input and output selectors are implemented in a single device. The

<sup>1</sup>Virtex logic cell = (1) 4-input LUT + (1) flip-flop + carry logic. Virtex slice = 2 logic cells.

implicit assumption is that the result of every iteration is known globally before the next iteration is being executed.

- DL1 Separates the input from the output selectors, creating two groups of  $N$  selectors each, enabling distribution over two devices.
- DL2 Additionally separates the input selectors from each other, enabling distribution over  $N + 1$  devices.

### A. Monolithic DRRM implementation

To clarify the issues that arise when selectors are distributed and to explain our solutions, we will refer to an implementation of the DRRM algorithm. However, the techniques we present are applicable to other two- and three-phase, pointer-based algorithms as well. Before proceeding with the distribution techniques, we review DRRM in greater detail.

Listing 1. C++ implementation of the DRRM matching algorithm.

```

1 void DRRM::schedule() {
2   int i, x, inp, outp, inpReq[N];
3   for (i = 0; i < I; i++) {
4     // request
5     for (inp = 0; inp < N; inp++) {
6       inpReq[inp] = -1;
7       if (imatch[inp] == -1) {
8         for (x = 0; x < N; x++) {
9           outp = (reqPtr[inp]+x) % N;
10          if (omatch[outp] == -1
11              && requests[inp][outp] > 0) {
12            inpReq[inp] = outp;
13            if (i == 0)
14              reqPtr[inp] = outp;
15            break;
16          } } } }
17     // grant
18     for (outp = 0; outp < N; outp++) {
19       if (omatch[outp] == -1) {
20         for (x = 0; x < N; x++) {
21           inp = (grtPtr[outp]+x) % N;
22           if (imatch[inp] == -1
23               && inpReq[inp] == outp) {
24             imatch[inp] = outp;
25             omatch[outp] = inp;
26             if (i == 0) {
27               reqPtr[inp] = (outp+1) % N;
28               grtPtr[outp] = (inp+1) % N;
29             }
30             break;
31           } } } } } }

```

DRRM computes a matching in every time slot in a sequence of  $I$  iterations. We consider the enhanced version of DRRM [6], which achieves lower mean latency via a modified pointer update rule similar to that used in FIRM. The following steps are performed in every iteration (initially all inputs and outputs are unmatched) [6], [12]:

- Step 1: Request. Each unmatched input sends at most one request, selecting the first unmatched, backlogged output in the round-robin order starting from the current position of the request pointer. In the first iteration, the pointer is updated to point to the output just selected. The pointer is further updated to one position beyond the output selected (modulo  $N$ ) if and only if the request is granted in Step 2 of the first iteration.

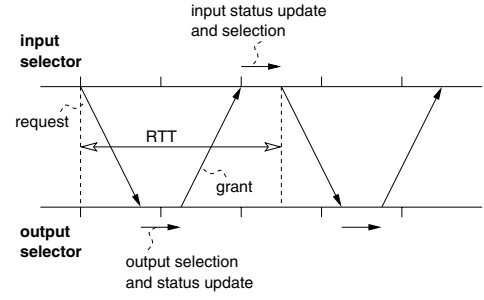


Fig. 2. Communication round-trip time between input and output selectors.

- Step 2: Grant. If an output receives one or more requests, it grants the one that appears first in the round-robin order starting from the current position of the grant pointer. The output notifies each requesting input whether its request was granted. The pointer is updated to one position (modulo  $N$ ) beyond the input granted in the first iteration. If there are no requests, the pointer remains where it is.

To facilitate the discussions that follow, Listing 1 shows a piece of C++ code that implements the DRRM matching algorithm in a monolithic fashion.  $N$  represents the number of ports and  $I$  the number of iterations. The arrays `imatch` and `omatch` store the port numbers that each input and output, respectively, are matched to. They are initialized to the value  $-1$  (i.e., unmatched) at the start of every time slot. `reqPtr` and `grtPtr` are the round-robin request and grant pointers, respectively. The two-dimensional `requests` array stores the number of requests per VOQ. The input selection (request) is performed in lines 5–16, whereas the output selection (grant) takes place in lines 21–31. Lines 13–14 of Listing 1 implement the enhanced DRRM request pointer update.

### B. Separating Input Selectors from Output Selectors

Physically separating the input and output selectors introduces a nonnegligible round-trip time (RTT) in the communication between them, as illustrated in Fig. 2. Assuming that this RTT is larger than the time-slot duration, there are two major implications, which we explain with the help of Listing 1. We assume that the RTT is symmetric, i.e., the up- and downstream latencies are equal. Although DL2 has been proposed before in [13], the RTT that results from this distribution is not considered there at all.

The request decision of a given input  $i$  depends on the position of the request pointer `reqPtr[i]` and is stored temporarily in `inpReq[i]` (line 12). The requests made are then considered in the grant loop (line 23). In a distributed implementation, things are different. First, the request information is delayed by  $RTT/2$ . Moreover, as the request pointers are physically located at the input side, the request pointer update (line 27) cannot be performed immediately after the grant; this update occurs after another  $RTT/2$ , i.e., when the grant arrives at the input selector. This has a further important consequence: Unlike in a monolithic implementation, the requests to be issued in the next time slot are based on pointer positions that

have not been updated according to the most recent grants. This breaks the round-robin desynchronization mechanism, leading to throughput limitations well below 100%.

Another consequence of the delayed availability of grant information is that the request selector is not able to accurately know the number of grants that are already underway. This affects its request decisions: Clearly, issuing requests for VOQs which are soon going to be empty is a waste of resources. Sections IV-A and IV-B address the pointer-desynchronization and latency issues, respectively.

#### IV. DISTRIBUTED IMPLEMENTATION

For the moment we assume that during each time slot only one iteration is performed. Performing multiple iterations poses additional challenges and will be discussed in Sec. V.

##### A. Pointer Update Approach

Our objective is to enable physical separation of the input and output selectors with an arbitrary RTT latency between them, while preserving performance and fairness. The key to achieving this is to conserve the pointer desynchronization property. This can be accomplished by ensuring that every pointer is updated at most once in every RTT time slots.

To this end, each input and output selector maintains a distinct pointer for every time slot of the RTT. These pointers are labelled  $R_i(t)$  and  $G_j(t)$  for the request and grant pointers, respectively, with  $t$  being the temporal index. By *pointer set* we denote the set of all pointers  $R_i(t)$  and  $G_j(t)$  corresponding to a specific index  $t$ , so there are RTT sets in total.

The traditional pointer update rules are used: Request pointers are only updated at the time a grant arrives (which happens one RTT after issuing the corresponding request), whereas grant pointers are updated immediately after issuing a grant, because issued grants are automatically accepted.

In a given time slot  $t$ , each input  $i$  issues requests using pointer  $R_i(t \bmod \text{RTT})$ . When a request issued using a pointer with temporal index  $t$  is granted, the corresponding grant pointer  $G_j(t)$  with the same index  $t$  is updated. At time slot  $t_0 + \text{RTT} - 1$ , the grant decision for requests submitted at time slot  $t_0$  will arrive, so that the pointers  $R_i(t_0)$  can be updated and used again in time slot  $t_0 + \text{RTT}$ . The output selectors use a different pointer at every time slot in the same way.

This pointer update policy implies that all pointer sets evolve independently and that no pointer is ever reused before being updated according to the result of its previous request. Therefore, it preserves the important features of the matching algorithm regardless of the value of RTT. In particular, every pointer set will eventually desynchronize, resulting in 100% throughput. Fairness is also preserved, as each input will request the same output at least once every RTT time slots, until it is granted.

This scheme requires RTT pointer registers (each  $\log_2 N$  bits wide) per selector. However, the combinatorial selection logic does not have to be duplicated. Instead, every selector employs a multiplexer to select one of the registers depending

on the temporal index. Also needed is a counter (modulo RTT) indicating the current pointer set to be used.

##### B. Pending Request Counters

The VOQ status registers reside close to the input selectors. The variable `requests[i][j]` is incremented whenever a new cell arrives for VOQ( $i, j$ ) and decremented whenever a grant for VOQ( $i, j$ ) is received. The RTT introduced by the distribution implies that when an input selector submits a request, it has to wait RTT time slots before knowing whether it was granted. In the meantime, the input selector can submit further requests. If the number of submitted requests exceeds the number of enqueued cells, it may happen that a slot is reserved for a VOQ that is currently empty. In general, this is undesirable because grants that arrive for an empty VOQ are wasted, while there may be another cell that could have used this time slot.

To avoid the problem of issuing too many requests for a given VOQ, we introduce *pending request counters* (PRCs, labelled  $P_{ij}$ ) per VOQ plus a request history per input selector. The pending request counter  $P_{ij}$  is incremented when input  $i$  issues a request for output  $j$ . The request history  $H_i(t)$  for input selector  $i$  is an array with RTT entries, where entry  $H_i(t)$  indicates the output that was requested  $t$  time slots ago. In every time slot, input selector  $i$  decrements  $P_{ij}$  for which  $j = H_i(\text{RTT} - 1)$ . As a result,  $P_{ij}$  keeps track of the number of requests per VOQ for which the results are still pending.

The input selectors use these counters to filter their requests. Any VOQ( $i, j$ ) for which  $P_{ij}$  exceeds or equals the current VOQ occupancy is not eligible for issuing a new request, which improves performance by preventing grants from being wasted. The use of PRCs, while not strictly necessary for the distribution solution proposed, is beneficial with large RTTs and light loads or when traffic is heavily unbalanced and different VOQs have significantly varying occupancy. Section VI shows the performance improvement obtained by the PRCs.

#### V. PERFORMING MULTIPLE ITERATIONS

In a monolithic implementation, performing multiple iterations per time slot significantly improves performance by allowing more edges to be added if multiple inputs requested the same output (or multiple outputs granted the same input in the case of  $i$ -SLIP). In our distributed implementation the effectiveness of subsequent iterations will be much lower, as explained below.

In Listing 1, the matched ports are indicated by the `imatch` and `omatch` arrays. These are updated in lines 24–25 when a new edge is added. In the next iteration, these updated values are taken into account in lines 7 and 10 to produce requests for that iteration. In our distributed implementation, these updates occur at the output side, so the input side does not learn of them for another RTT/2 time slots; as a consequence, the input selectors do not know which outputs to disregard. Accordingly, requests can turn out to be useless, as the requested output is already matched. These are *wasted requests*. In contrast, the approach of [13] assumes global knowledge of outputs requested in previous iterations.

However, the output notification signals (free/busy) have to propagate across all input ports before the start of the next iteration. These output signals experience propagation delays associated with the pin driver of the output selector, the PCB trace and the input pin driver of the input selector. When considering fast port rates, these propagation delays amount to a significant part of the iteration time. Next, there are cases where the propagation delays even exceed the cell duration, e.g. when serialization/deserialization and re-synchronization techniques are used to achieve a practical implementation of the distributed system with a large number of ports.

We address this issue by adding a separate pointer `flywheel[inp]` to every input selector. In the first iteration, a selection is made using the round-robin pointer `reqPtr[inp]`. `flywheel[inp]` is updated to one beyond the output just requested, modulo  $N$ . In subsequent iterations, the input selector is operated using `flywheel[inp]` rather than `reqPtr[inp]`. After every selection, `flywheel[inp]` is updated to one beyond the output just requested, modulo  $N$ . In this way, we make sure that the input selector at least requests as many different outputs as possible across the iterations, although there is no guarantee that the outputs requested are still available. Each input selector also keeps track of which outputs it has already requested in the current time slot and avoids requesting the same output more than once, as this would be useless.

PRC-based request filtering, as described above, ensures that the number of wasted grants is minimized. On the other hand, overly conservative filtering can be detrimental: Once the filtering condition is met, a new request can only be submitted when the result for the oldest in-flight one is received. This can introduce gaps in the request pipeline and therefore cause unnecessary delays. Furthermore, requests for subsequent iterations are increasingly less likely to be successful. Indeed, our findings show that it is counterproductive to include requests beyond the first iteration in the PRCs and request history. Therefore, the PRC and request history operations (updating, filtering) apply *only* to requests in the *first* iteration.

In the case of multiple iterations, the input selectors also base their choice on previously matched edges (`omatch[outp]` in line 10 of Listing 1). Note that this information is lost in moving from DL0 to DL1. Therefore, no additional limitation is introduced when moving from DL1 to DL2, i.e., when separating the input selectors. As their decisions are based on local information only, they can operate independently.

## VI. SIMULATION RESULTS

We built a software model of the proposed architecture with the OMNeT++ simulation environment. Using Akaroa2, we simulated this model to obtain its performance characteristics. Specifically, we are interested in the mean throughput (measured at the egress across all ports) and the mean packet latency (measured from source to sink). In our experiments, we study a switch with  $N = 16$  ports using the distributed DRRM architecture according to DL2. We vary RTT and the number of iterations per time slot.

Figures 3(a,b) show the performance results with uniform Bernoulli traffic for  $RTT = 4$  and 20 time slots, respectively.  $I$  is varied from 1 to 16 iterations per time slot. Note that the minimum latency at very light load equals RTT. For reference, results using a monolithic DRRM implementation are also included, adjusted to take into account the constant latency component of the distributed implementation. These results lead to the following observations:

- The achievable maximum throughput exceeds 98% in all simulations, i.e., for all values of RTT and  $I$ .
- The mean latency decreases significantly as the number of iterations increases. When  $I = N = 16$ , the performance of the distributed implementation is almost identical to that of the monolithic implementation with four iterations. Using as many iterations as there are ports overcomes the issue of wasted requests, as there is an opportunity to request every output in every time slot. However, it does not overcome the issue of uncertainty due to pending requests.
- For large RTT, there is a load region in which the mean latency decreases as the load increases (Fig. 3b). This effect is caused by excess grants that, instead of being wasted on an empty VOQ, find a new arrival in their VOQ; these cells experience a latency smaller than RTT.

Figure 3(d) shows the performance *without* PRCs with  $N = 16$  and  $RTT = 4$ . These graphs clearly show that the use of PRCs drastically reduces latency throughout the load range. Considering the case  $I = 1$ , the main difference is in the load range from 10 to 70%; beyond 70% there is no noticeable latency difference vs. Fig. 3(a). The reason is that, with heavy loads, the rate of wasted grants will be low, as most VOQs will be backlogged; therefore, the negative effect of excess requests is not noticed. At low to medium loads, on the other hand, many of the excess requests will result in wasted grants; every wasted grant potentially is a wasted opportunity to transmit another cell, which therefore incurs a longer latency. As a result, the mean latency increases.

To study the performance under nonuniform traffic, we adopt a destination distribution characterized by a nonuniformity parameter  $w$ , where  $w = 0$  corresponds to uniform traffic and  $w = 1$  to fully unbalanced, contention-free traffic:  $\lambda_{ij} = \lambda \left( w + \frac{1-w}{N} \right)$  if  $i = j$ ,  $\lambda \frac{1-w}{N}$  otherwise. Here,  $\lambda_{ij}$  represents the traffic intensity from input  $i$  to output  $j$ ,  $0 \leq i, j < N$ ;  $\lambda$  is the aggregate offered load, and  $w$  the nonuniformity factor. No input or output is oversubscribed and traffic is admissible as long as  $\lambda \leq 1$ . We vary the value of  $w$  from 0 to 1 and measure the throughput achieved at an offered load of 100%.

Figure 3(e) shows the results for  $N = 16$ ,  $RTT = 4$ , and Bernoulli arrivals for  $I$  ranging from 1 to 16. Also included for reference is a curve for monolithic DRRM with  $I = 4$ . All curves dip to significantly less than 100% throughput as  $w$  moves away from the extremes. However, increasing  $I$  increases the throughput. The method proposed is able to reduce the gap with the reference to less than four percentage points when  $I \geq 8$ .

We also evaluate the performance using bursty traffic with

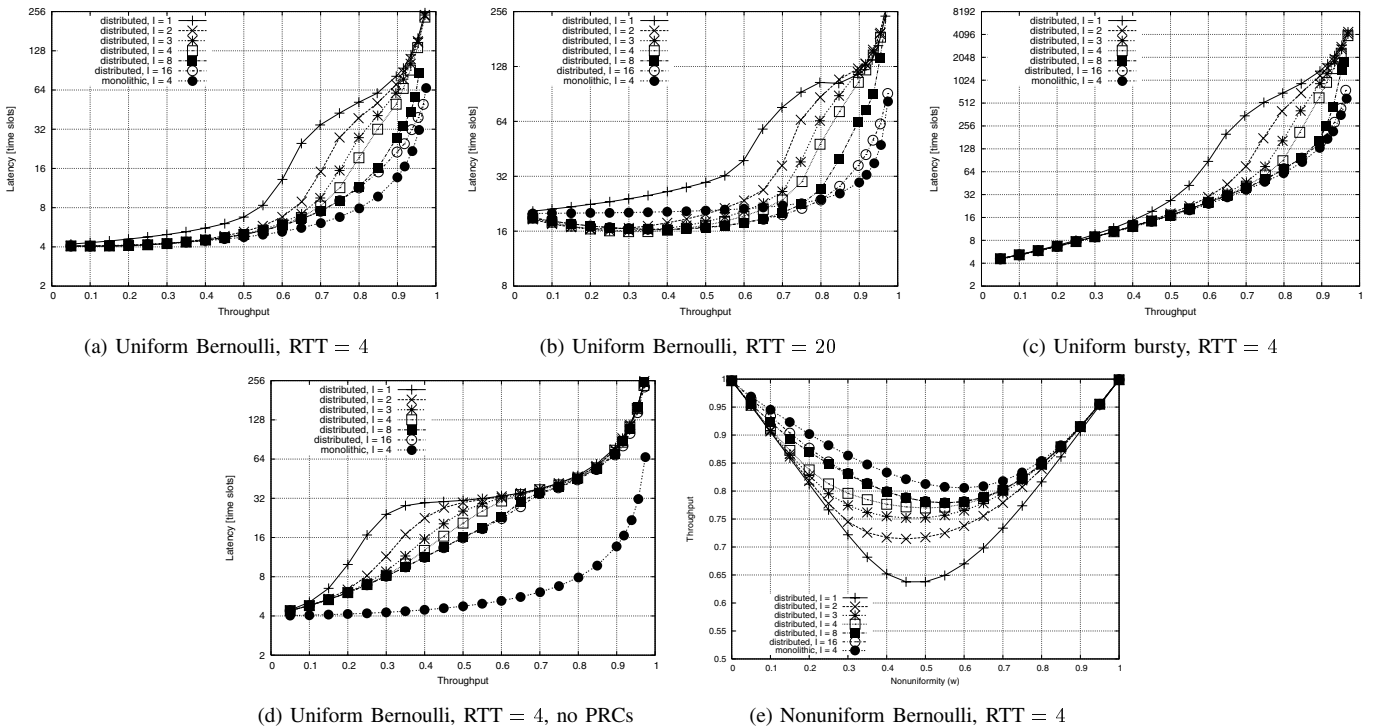


Fig. 3. Delay vs. throughput curves (a–e),  $N = 16$ .

geometrically distributed burst sizes with an average burst size of 10 cells. Figure 3(c) shows the results. Here, we first observe that the maximum throughput again exceeds 98% in all cases. Moreover, the latency differences to the reference curves are even smaller than with Bernoulli traffic. Hence, the proposed distributed implementation is able to closely approximate a monolithic implementation in terms of performance for correlated as well as uncorrelated arrivals.

## VII. CONCLUSIONS

The main contribution of this paper is a set of techniques that enable the construction of crossbar schedulers in a highly distributed fashion, while maintaining a high level of performance with uncorrelated and correlated arrivals as well with uniform and nonuniform destination distributions.

A particular constraint that results from distribution is that requests for subsequent iterations are “blind,” which reduces their effectiveness. This drawback can be compensated by performing more iterations; specifically, the optimal number of iterations changes from  $\log_2(N)$  in a monolithic implementation to  $N$  in a distributed one. This creates a new challenge in the timing constraints to complete enough iterations. This issue can be addressed with pipelining techniques as introduced in [14], [15]. We are currently working towards combining these approaches into a single architecture that is both highly distributed as well as deeply pipelined.

## REFERENCES

- [1] A. Bianco, P. Giaccone, E. Giraudo, F. Neri, and E. Schiattarella, “Performance analysis of storage area network switches,” in *Proc. IEEE HPSR 2005*, Hong-Kong, May 2005.
- [2] C. Minkenberg, F. Abel, P. Müller, R. Krishnamurthy, and M. Gusat, “Control path implementation of a low-latency optical HPC switch,” in *Proc. Hot Interconnects 13*, Stanford, CA, Aug. 17–19 2005, pp. 29–35.
- [3] J. Kim, W. J. Dally, B. Towles, and A. K. Gupta, “Microarchitecture of a high-radix router,” in *Proc. ISCA 2005*, Madison, WI, June 2005.
- [4] N. McKeown, “The iSLIP scheduling algorithm for input-queued switches,” *IEEE/ACM Trans. Networking*, vol. 7, no. 2, pp. 188–201, Apr. 1999.
- [5] D. Serpanos and P. Antoniadis, “FIRM: A class of distributed scheduling algorithms for high-speed ATM switches with multiple input queues,” in *Proc. IEEE INFOCOM 2000*, vol. 2, Tel Aviv, Israel, Mar. 2000, pp. 548–555.
- [6] Y. Li, S. Panwar, and H. Chao, “On the performance of a dual round-robin switch,” in *Proc. IEEE INFOCOM 2001*, vol. 3, Anchorage, AK, Apr. 2001, pp. 1688–1697.
- [7] M. Ajmone Marsan, A. Bianco, and E. Leonardi, “RPA: A simple, efficient, and flexible policy for input buffered ATM switches,” *IEEE Commun. Lett.*, vol. 1, no. 3, pp. 83–86, May 1997.
- [8] A. Smiljanić, R. Fan, and G. Ramamurthy, “RRGS-round-robin greedy scheduling for electronic/optical terabit switches,” in *Proc. IEEE GLOBECOM 1999*, Rio de Janeiro, Brazil, Dec. 1999, pp. 584–555.
- [9] P. Gupta and N. McKeown, “Designing and implementing a fast crossbar scheduler,” *IEEE Micro*, vol. 19, no. 1, pp. 20–28, Jan./Feb. 1999.
- [10] N. McKeown, “Scheduling algorithms for input-queued switches,” Ph.D. dissertation, University of California at Berkeley, 1995.
- [11] R. Hemenway, R. Grzybowski, C. Minkenberg, and R. Luijten, “Optical-packet-switched interconnect for supercomputer applications,” *OSA J. Opt. Netw.*, vol. 3, no. 12, pp. 900–913, Dec. 2004.
- [12] H. Chao and J. Park, “Centralized contention resolution schemes for a large-capacity optical ATM switch,” in *Proc. IEEE ATM Workshop*, Fairfax, VA, May 1998, pp. 11–16.
- [13] L. Peng, C. Tian, and S. Zheng, “iRGR: A fast scheduling scheme with less control messages for scalable crossbar switches,” in *Proc. IEEE HSNMC 2004*, Toulouse, France, June 30–July 2 2004, pp. 191–202.
- [14] E. Oki, R. Rojas-Cessa, and H. Chao, “A pipeline-based approach for maximal-sized matching scheduling in input-buffered switches,” *IEEE Commun. Lett.*, vol. 5, no. 6, pp. 263–265, June 2001.
- [15] C. Minkenberg, I. Iliadis, and F. Abel, “Low-latency pipelined crossbar arbitration,” in *Proc. IEEE GLOBECOM 2004*, Dallas, TX, Dec. 2004, paper no. GE15-2.