Scalable Schedulers for High-Performance Switches

Chuanjun Li and S. Q. Zheng
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083
{chuanjun, sizheng}@utdallas.edu

Mei Yang
Department of Computer Science
Columbus State University
Columbus, GA 31907
yang_mei@colstate.edu

***Abstract*— Scheduler and switching fabric are two major hardware components of a cell switch. For a switch using a non-blocking switching fabric, the performance of the switch depends on the performance of its cell scheduler. We introduce the concepts of relative and universal scheduler scalabilities. Informally, a scheduler is relatively scalable with respect to a switching fabric if its structure is not more complex than the structure of its associated non-blocking switching fabric. A scheduler is universally scalable if its structural complexity is not larger than the structural complexity of any non-blocking switching fabric. Based on algorithm-hardware co-design, we present a universally scalable scheduler with $O(N \log N)$ interconnection complexity. We show by simulation that the performance of the proposed scheduler is almost the same as non-scalable schedulers.**

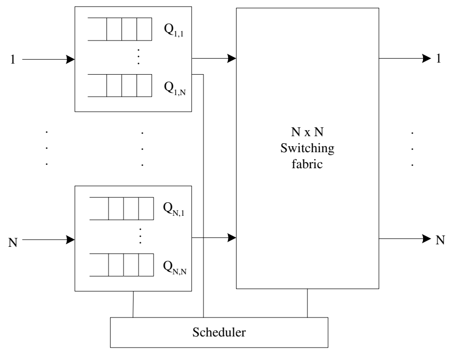


Fig. 1. An IQ switch with VOQs.

I. Introduction

Most high-speed packet switches employ cell switches as their cores. In such a switch, variable-length packets are segmented into fixed-size cells as they arrive, transferred across the switching fabric, and reassembled back into original packets before they depart. Different queuing schemes have been proposed for such a switch. Due to their capabilities of achieving 100% throughput and providing quality of service (QoS) guarantee, output queueing (OQ) switches are commonly employed for many commercial switches and routers. However, OQ is impractical for switches with line rates and/or large numbers of ports since it requires that the switching fabric and memory run as fast as N times the line rate for an $N \times N$ switch. With the switching fabric and memory running at the line rate, input queuing (IQ) switches are scalable for high line rate and/or large number of ports. However, due to the problem of head-of-line (HOL) blocking, the maximum throughput of an IQ switch with first-in-first-out (FIFO) queues is limited to 58.6% under uniform traffic [1]. Virtual output queues (VOQs) have been proposed for IQ switches to remove HOL blocking and to achieve the scalability of IQ switches. Fig. 1 shows an $N \times N$ IQ switch with VOQs, in which each input port maintains N virtual output queues (VOQs) with $Q_{i,j}$ buffering cells from input port I_i destined for output port O_j.

We assume that time is slotted and a cell slot equals to the transmission time of a cell on an input/output line. For an IQ switch with VOQs, its performance highly depends on the scheduling algorithm, which decides which N cells out of N^2 HOL cells to be sent across the switching fabric in each cell slot. The cell scheduling problem for a VOQ-based IQ switch can be modelled as a bipartite matching problem on the bipartite graph composed of nodes of input ports and output ports and edges of connection requests from input ports to output ports [2]. A matching is a subset of edges such that no two edges are incident to one node in the graph. A maximum size matching is one with the maximum number of edges. A maximal size matching is one that is not contained in any other matching. Although it has been shown that maximum size matching algorithms can achieve 100% throughput under uniform traffic, they are too complex for hardware implementation and can cause unfairness [3].

Instead, most practical scheduling algorithms proposed in the literature are iterative maximal size matching algorithms, such as parallel iterative matching (PIM) [4], iSLIP [2], dual round-robin matching (DRRM) [5], first-come-first-serve in round-robin matching (FIRM) [6], and static round-robin (SRR) [7]. These algorithms operate either in one or multiple iterations with each iteration composed of either three steps, Request-Grant-Accept (RGA), or two steps, Request-Grant (RG). All these algorithms can be implemented by the hardware scheduler architecture shown in Fig. 2 [2], which consists of N request/grant arbiters, each associated with an input port, and N grant/accept arbiters, each associated with an output port.

A high-performance switch tends to have a large number of input/output ports. It is well-known that wiring takes the most chip area in a large digital circuit/system. Thus, the implementability, cost and performance of such a large size switch depend on the interconnection complexities of its switching fabric and scheduler. We define the interconnection complexity (also called wiring density) of a circuit as the number of wires used in the circuit. We use C_{sf} and C_{sch} to denote the interconnection complexity of the non-blocking switching fabric and the scheduler of a switch, respectively. We say that

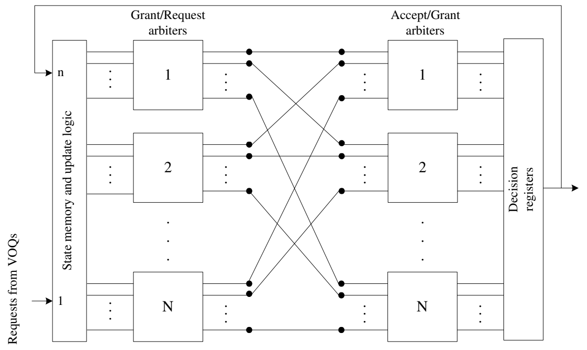


Fig. 2. Block diagram of a scheduler based on an RGA/RG maximal size matching algorithm.

a scheduler architecture A is *relatively scalable* with respect to an $N \times N$ switching fabric S if $C_{sch}(N, A) \leq C_{sf}(N, S)$ and each iteration of A takes poly-logarithmic time. We say that a scheduler architecture A is *universally scalable* if it is relatively scalable with respect to any switching fabric S of a given size. As we can see, the scheduler A shown in Fig. 2 has $C_{sch}(N, A) = O(N^2)$, which is as complex as a crossbar switching fabric S which has $C_{sf}(N, S) = O(N^2)$. The interconnection complexity of such a scheduler is even larger than that of some existing non-blocking and rearrangeable non-blocking switching fabrics. For instance, for an $N \times N$ non-blocking Clos network C [8], [9] and rearrangeable non-blocking Benes network B [8], $C_{sf}(N, C) = O(N^{1.5})$ and $C_{sf}(N, B) = O(N \log N)$ [1], respectively. For such a switching fabric, the scheduler shown in Fig. 2 is non-scalable, and its use cannot be justified.

Clearly, $\Omega(N \log N)$ is the lower bound for the interconnection complexity of any $N \times N$ switching fabric. Thus, any scheduler A with interconnection complexity $C_{sch}(N, A) = O(N \log N)$ is universally scalable. This motivates our study on universal scalable schedulers. In this paper, we propose a universal scalable scheduler architecture based on the proposed round-robin priority matching (RRPM) algorithm and a multiprocessor system. RRPM is an iterative algorithm with each iteration consisting of two steps, Request and Grant. Combining the features of DRRM and PIM, in RRPM, each input port selects one request according to the round-robin discipline and each output port grants one request randomly. We show that using a hypercube and a linear array, each iteration of RRPM can be implemented with $O(N \log N)$ interconnection complexity and in $O(\log^2 N)$ time. By simulation, we show that RRPM achieves better performance than DRRM under uniform Bernoulli and bursty traffic. Noticeably, RRPM with $\log N$ requests can achieve almost the same performance as iSLIP.

The rest of the paper is organized as follows. Section II presents the RRPM algorithm and the hardware scheduler used to implement it. Section III presents the simulation results of RRPM and comparison with other scheduling algorithms. Section IV concludes the paper.

[1]In this paper, all logarithms are in base 2.

II. The RRPM Algorithm and Its Implementation

In this section, we first present the RRPM algorithm and then discuss its hardware implementation architecture.

A. The RRPM Algorithm

Designing a scalable scheduler is an algorithm-architecture co-design problem. Algorithm-structured scheduler is designed to fully explore the parallelism existing in the cell scheduling problem. By combining the features of DRRM and PIM, we propose the round-robin priority matching (RRPM) algorithm which uses the round-robin discipline to select a request at each input (port) and random selection to decide a grant at each output (port). As we will discuss in Section II-B, the random selection is implemented by selecting the smallest one among a subset of random numbers (or priorities), generated by inputs. Assuming that each input port I_i is associated with a request pointer r_i, which indicates the request starting point, RRPM operates iteratively with each iteration consisting of the following two steps.

Step 1: **Request**. If an unmatched I_i has at least one request, it selects one request starting from the VOQ that r_i points to in a round-robin manner, and sends the request to its corresponding output. r_i is updated to one beyond the requested output if and only if the request is granted in Step 2 of *the first iteration*.

Step 2: **Grant**. If an unmatched O_j receives at least one request, it grants a request randomly.

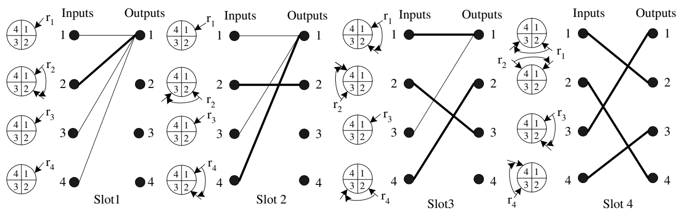


Fig. 3. An example of RRPM for a 4×4 switch.

RRPM stops either after a predetermined number of iterations or until no more matching can be found, which means a maximal size matching is found. Fig. 3 shows an example of RRPM with one iteration for a 4×4 switch assuming that each input has no empty VOQ. Initially, we assume all the request pointers are pointing at 1. In the first cell slot, all inputs send requests to output 1, which randomly grants one request (shown as dark edge in Fig. 3), say from input 2. Then only the request pointer at input 2 will be updated to 2. In the second (cell) slot, input 2 will request output 2 and get granted, while other inputs will continue requesting output 1, which randomly grants one request, say from input 4. Then the request pointer at input 2 will be updated to 3 and the request pointer at input 4 will be updated to 2. In the third cell slot, inputs 2 and 4 will request to outputs 3 and 2 and get granted respectively, while inputs 1 and 3 continue requesting output 1, which randomly grants one request, say from input 1. Then the request pointers at inputs 1, 2, and 4 will be updated to 2, 4, and 3 respectively. In the fourth slot, request pointers are fully desynchronized, each request is coming from a different input and will be granted. A maximal size matching of size 4 is found in this slot.

We can run RRPM for multiple iterations to enlarge the matching size found in each cell slot. For an $N \times N$ switch, it takes up to N iterations to find a maximal size matching. However, in practice, due to the desynchronization effect of the request pointers, it takes much less iterations for RRPM to find a maximal size matching. By simulations, we show that $\log N$ iterations are adequate to achieve satisfying performance.

An important objective of designing a scheduling algorithm is the simplicity in implementation. In the following, we discuss a unique hardware implementation architecture for RRPM.

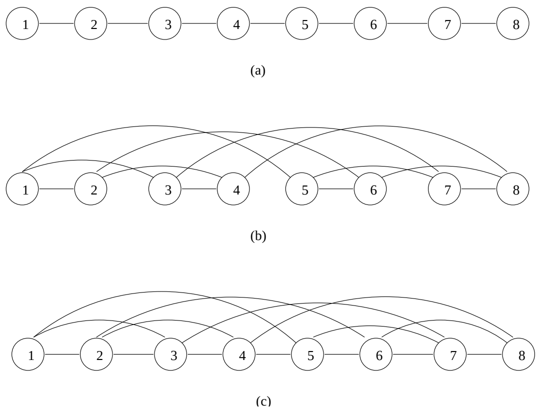


Fig. 4. (a) A linear array. (b) A hypercube. (c) A modified hypercube.

B. Scalable Hardware Implementation

Our hardware implementation of RRPM is based on a modified hypercube (MH) with N simple processing elements (PE's). The inter-processor interconnection of MH is a combination of a linear array and a hypercube, as shown in Fig. 4. We define a *scheduling cycle* as the process of finding a matching, which is not necessarily a maximal size matching. Each PE_i, $1 \leq i \leq N$, is associated with input I_i and output O_i. PE_i maintains two Boolean variables, $SI(i)$ and $SO(i)$, where $SI(i) = 1$ (resp. $SO(i) = 1$) iff I_i (resp. O_i) has been included in the partial matching found in the current scheduling cycle. An additional local variable $P(i)$ is needed to indicate the priority value. At the beginning of each scheduling cycle, both variables $SI(i)$ and $SO(i)$ are initialized as 0, and each PE_i generates a random number as $P(i)$ and constructs a *request word* $W(i)$ with four fields $f_1(i) \mid f_2(i) \mid f_3(i) \mid f_4(i) = IN(i) \mid OUT(i) \mid P(i) \mid S(i)$, where $IN(i) = i$ if $SI(i) = 0$, and $IN(i) = \infty$ otherwise; $OUT(i) = j$ if a request $Q_{i,j}$ is selected for scheduling according to the round-robin scheme, and $OUT(i) = \infty$ otherwise; and $S(i)$ indicates whether I_i is selected in an iteration; "|" stands for concatenation operation. Each iteration of RRPM is implemented on an MH by the following steps.

Step 1: **Sort.** Sort $W(i)$'s using keys $f_2 \mid f_3 = OUT(i) \mid P(i)$ in non-decreasing order. The sorted $W(i)$'s, one in each PE, consists of a sequence of segments such that the $W(i)$'s in each segment have the same f_2 value and the f_3 values are in non-decreasing order.

Step 2: **Select.** Each PE_i compares the f_2 values of its neighboring PEs (using the linear array connections according to linear order) to check if it is the first request word in its segment. If it is the first request word in its segment, and its f_2 value is not ∞, set its $f_4 = 1$ (which was initialized to 0).

Step 3: **Pack.** Assuming that k requests are selected in Step 2, pack all request words with $f_4 = 1$ to the first k PEs, with one request word in each PE and their relative order maintained.

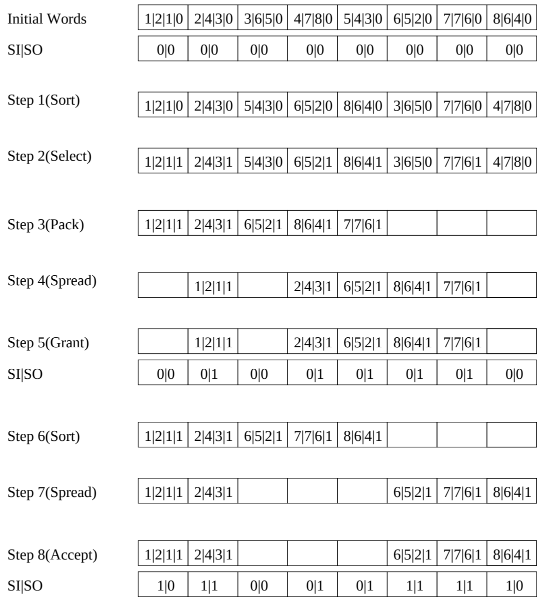
Step 4: **Spread.** Send each request word with $f_4 = 1$ to the PE whose index is equal to its f_2 value.

Step 5: **Grant.** If PE_i receives a request word in Step 4, do the following. If $SO(i) = 0$ then set $SO(i) = 1$ else set $f_4 = 0$ in its received request word.

Step 6: **Sort.** Sort the request words with $f_4 = 1$ using key f_1 in non-increasing order.

Step 7: **Spread.** Send each request word with $f_4 = 1$ to the PE with index equal to its f_1.

Step 8: **Accept.** If PE_i receives a request word in step 7, set $SI(i) = 1$.



Initial Words	1|2|1|0	2|4|3|0	3|6|5|0	4|7|8|0	5|4|3|0	6|5|2|0	7|7|6|0	8|6|4|0
SI|SO	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0
Step 1(Sort)	1|2|1|0	2|4|3|0	5|4|3|0	6|5|2|0	8|6|4|0	3|6|5|0	7|7|6|0	4|7|8|0
Step 2(Select)	1|2|1|1	2|4|3|1	5|4|3|0	6|5|2|1	8|6|4|1	3|6|5|0	7|7|6|1	4|7|8|0
Step 3(Pack)	1|2|1|1	2|4|3|1	6|5|2|1	8|6|4|1	7|7|6|1			
Step 4(Spread)		1|2|1|1		2|4|3|1	6|5|2|1	8|6|4|1	7|7|6|1	
Step 5(Grant)		1|2|1|1		2|4|3|1	6|5|2|1	8|6|4|1	7|7|6|1	
SI|SO	0|0	0|1	0|0	0|1	0|1	0|1	0|1	0|0
Step 6(Sort)	1|2|1|1	2|4|3|1	6|5|2|1	7|7|6|1	8|6|4|1			
Step 7(Spread)	1|2|1|1	2|4|3|1				6|5|2|1	7|7|6|1	8|6|4|1
Step 8(Accept)	1|2|1|1	2|4|3|1				6|5|2|1	7|7|6|1	8|6|4|1
SI|SO	1|0	1|1	0|0	0|1	0|1	1|1	1|1	1|0

Fig. 5. An example of how an MH implements RRPM.

One can verify that this implementation of algorithm RRPM is correct. After sorting in Step 1, at most one request word is selected for each output in Step 2. Steps 3 and 4 are used to check if the selected outputs have been matched in previous iterations. Step 5 updates the status of matched outputs. Steps 6 and 7 inform the inputs whether or not their requests are granted. Step 8 updates the status of matched inputs.

Fig. 5 illustrates how the MH implements RRPM using an example for an 8×8 switch. Each rectangle represents a PE and the four numbers inside each rectangle representing the four fields of the request word in each PE. Assume that at the beginning of a schedule cycle, we have the initial request words in each PE shown in the first row. All SI's and SO's

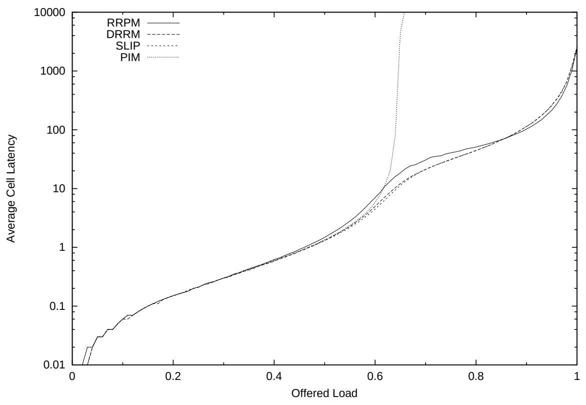


Fig. 6. Delay performance of RRPM, PIM, iSLIP, and DRRM, all with one iteration, under uniform i.i.d. Bernoulli arrivals.

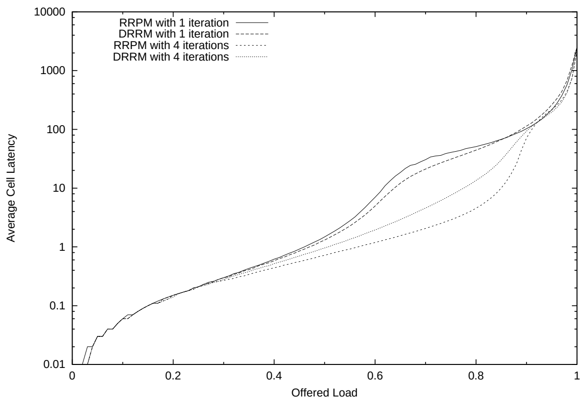


Fig. 7. Delay performance of RRPM and DRRM with 1 and 4 iterations under uniform i.i.d. Bernoulli arrivals.

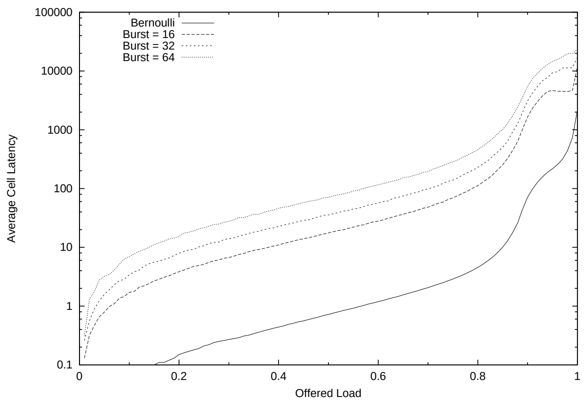


Fig. 8. Delay performance of RRPM with 4 iterations under Bernoulli and busty arrivals.

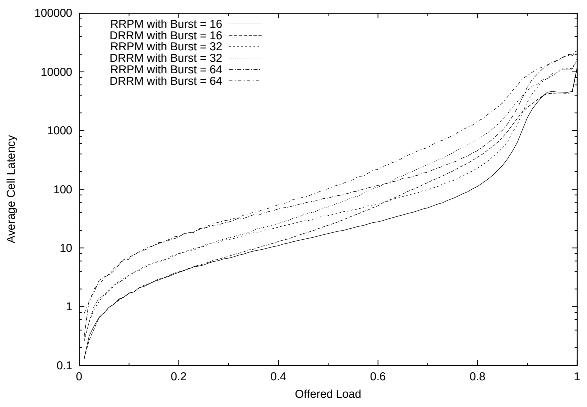


Fig. 9. Delay performance of RRPM and DRRM with four iterations under bursty arrivals.

are initialized as 0. After Step 1, we have the list sorted by fields f_2 and f_3 (i.e., $OUT(i)$ and $P(i)$). We have in the sorted list five segments, which have $f_2 = 1$, $f_2 = 4$, $f_2 = 5$, $f_2 = 6$, and $f_2 = 8, respectively$. The first request word of each segment will be selected, as indicated by the change in field f_4 after Step 2. In Step 3, these request words are packed to the first five PEs such that they can be spread to PEs associated with their f_2 fields after Step 4. In Step 5, those PEs that receive request words in Step 4 will change their SO variables to 1. These request words are sorted again according to their f_1 fields in Step 6 and spread to PEs associated with their f_1 fields in Step 7. In Step 8, those PEs receiving request words will change their SI variables to 1. Finally we grant the requests $1-2$ (i.e., input 1 to output 2), $2-4$, $6-5$, $7-7$, and $8-6$.

Steps 1 and 6 can be done in $O(\log^2 N)$ time [10]. Steps 3, 4 and 7 can be done in $O(\log N)$ time [10]. The remaining steps can be done in $O(1)$ time. Therefore, each iteration of RRPM takes $O(\log^2 N)$ time. The interconnection complexity of the proposed structure is $O(N \log N)$.

The proposed scheduler can be generalized to allow each input port to issue up to k requests, say $k = \log N$. This extension makes it possible for RRPM to approximate PIM and iSLIP. The implementation can be based on a modified hypercube of kN PE's. The performance of RRPM with multiple requests per input is expected to be better than that of RRPM with one request per input as shown in the next section.

III. Performance Evaluation

In this section, we evaluate the performance of RRPM in terms of the average cell latency under both Bernoulli and bursty arrivals. The cell latency is the time that a cell spends in a switch measured in number of cell slots. We consider a 16×16 switch assuming first that the arrival at each input is independent and identically distributed (i.i.d.).

We first show the performance of RRPM under Bernoulli arrivals. Fig. 6 compares the performance of RRPM, DRRM, iSLIP, and PIM, all with one iteration, under uniform Beroulli arrivals. As shown in the figure, the average cell latency of PIM increases rapidly when the offered load exceeds 0.64. The average cell latency of RRPM is identical to that of iSLIP and DRRM when the offered load is lower than 0.5, slightly higher than that of iSLIP and DRRM when offered load is between 0.5 and 0.85, and slightly lower than that of iSLIP and DRRM when the offered load is larger than 0.85. This indicates that when rates become greatly high and there is no enough time for multiple iterations, RRPM is a better choice than DRRM and iSLIP for a large load range.

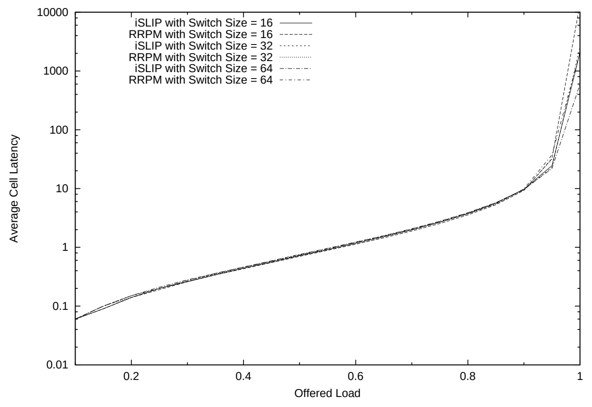


Fig. 10. Delay performance of RRPM with $\log N$ requests per input and iSLIP with switch sizes of 16×16 to 64×64.

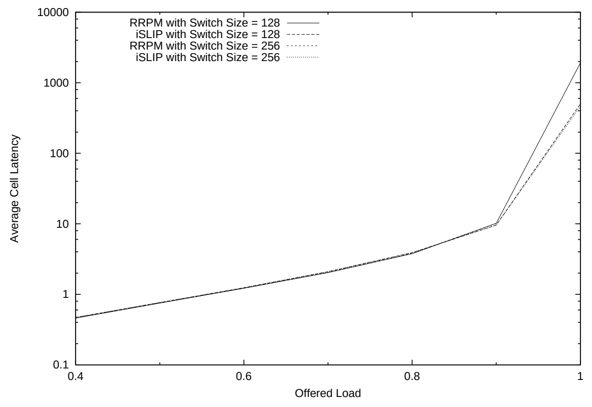


Fig. 11. Delay performance of RRPM with $\log N$ requests per input and iSLIP with switch sizes of 128×128 and 256×256.

Fig. 7 illustrates the impact of multiple iterations on the average cell latency. We extend DRRM to run for multiple iterations and update request pointers in the same way as RRPM. When four iterations are used, the average latency of RRPM decreases significantly compared with that of RRPM with one iteration. RRPM with four iterations outperforms DRRM with four iterations, especially when the offered load is between 0.4 and 0.9.

Next, we study the performance of RRPM under bursty arrivals using 2-state modulated Markov-chain sources [2]. Each source alternately generates a burst of full cells with the same destination followed by an idle period of empty cells. The number of cells in each burst or idle period is geometrically distributed. Let $E(B)$ and $E(D)$ be the average burst length and the average idle length in terms of the number of cells respectively. Then, we have $E(D) = E(B)(1-\rho)/\rho$, where ρ is the offered load of each input source. We assume that the destination of each burst is uniformly distributed.

Fig. 8 illustrates the performance of RRPM with four iterations under Beroulli arrivals and bursty arrivals with average burst lengths of 16, 32, and 64. As shown in the figure, with the average burst length increasing, the average cell latency increases correspondingly as expected. As shown in Fig. 9, under bursty traffic, RRPM with four iterations achieves better performance than DRRM with four iterations.

As discussed in Section II-B, we can extend RRPM by allowing each input to issue more than one request per iteration. We show below that by allowing each input to issue only $\log N$ requests, RRPM achieves comparable performance as that of iSLIP. Fig. 10 and Fig. 11 compares the average cell latencies of the extended RRPM with $\log N$ requests per input with those of iSLIP for switch sizes of 16×16 to 256×256. As shown in the two figures, the extended RRPM achieves almost the same performance as iSLIP when the offered load is below 0.9. We expect that for larger switch sizes, the extended RRPM with $\log N$ requests per input has almost the same performance as iSLIP.

IV. Conclusion

We introduced interconnection complexity to measure the cost and implementability of switching fabrics and cell schedulers. Based on this complexity, we introduced the concepts of relative and universal scalability of schedulers. These concepts are important in guiding the design and implementation of cell switches. Using algorithm-architecture co-design, we proposed a universally scalable scheduler, which is based on a new scheduling algorithm RRPM and a multiprocessor system. We showed that it has scheduling performance comparable to that of non-scalable schedulers. It remains a great challenge to design faster universally scalable schedulers that have good performance.

References

[1] M. Karol, M. Hluchyj, and S. Morgan, “Input versus output queueing on a space division switch,” *IEEE Trans. Commun.*, vol. 35, no. 12, pp. 1347–1356, Dec. 1987.

[2] N. McKeown, “The islip scheduling algorithm for input-output switches,” *IEEE/ACM Trans. Networking*, vol. 7, no. 2, pp. 188–201, Apr. 1999.

[3] N. Mckeown, A. Mekkittikul, V. Anantharam, and J. Walrand, “Achieving 100% throughput in an input-queued switch,” *IEEE Trans. Commun.*, vol. 7, no. 2, pp. 188 –201, Apr. 1999.

[4] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker, “High speed switch scheduling for local area networks,” *ACM Trans. on Computer Systems*, vol. 11, no. 4, pp. 319–352, Nov. 1993.

[5] J. Chao, “Saturn: A terabit packet switch using dual round robin,” *IEEE Commun. Mag.*, vol. 38, no. 12, pp. 78–84, Dec. 2000.

[6] D. N. Serpanos and P. I. Antoniadis, “Firm: a class of distributed scheduling algorithms for high-speed atm switches with multiple input queues,” in *Proc. IEEE INFOCOM*, 5 2000, pp. 548 –555.

[7] Y. Jiang and M. Hamdi, “A fully desynchronized round-robin matching scheduler for a voq packet switch architecture,” in *Proc. IEEE HPSR*, 12 2001, pp. 407 –412.

[8] V. Benes, “Optimal rearrangeable multi-stage connecting networks,” *Bell System Technical Journal*, vol. 43, pp. 1641–1656, 1964.

[9] C. Clos, “A study of non-blocking switching networks,” *Bell System Technical Journal*, vol. 32, pp. 406–424, 1953.

[10] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays. Trees. Hypercubes.* San Mateo, California: Morgan Kaufmann Publishers, Inc., 1992.