Pi-PIFO: A Scalable Pipelined PIFO Memory Management Architecture

Steven Young, *Student Member, IEEE*, Itamar Arel, *Senior Member, IEEE*, Ortal Arazi, *Member, IEEE*
Networking Research Group
Electrical Engineering and Computer Science Department
University of Tennessee
Knoxville, TN 37996
email: syoung22@utk.edu, itamar@ieee.org, oarazi@utk.edu

***Abstract*— Quality of service (QoS) provisioning is rapidly becoming an assumed attribute of core packet switching systems. Substantial work has been focused on designing algorithms which offer strict QoS guarantees under a broad range of traffic scenarios. The majority of these scheduling algorithms can be realized utilizing Push-in-First-out (PIFO) queues, which are characterized by allowing packets to enter the queue at arbitrary locations while departures occur from the head of line position only. Although the PIFO queueing mechanism has received attention in recent years, it is generally considered impractical from a hardware implementation perspective. This is due primarily to the computational complexity involved in placing arriving packets in a generic PIFO queue. In recent work, the iPIFO memory management scheme has been proposed in which additional data structures are employed to facilitate the realization of PIFO queues. While iPIFO does overcome some of the complexities involved in implementing PIFO queueing, it relies on the existence of a high-speed memory device which supports a large number of concurrent read and write operations. Such assumption substantially limits scalability. This paper introduces Pi-PIFO, a pipelined PIFO queuing memory management architecture, which requires modest memory bandwidth, with sub-linear dependency on the queue size (N), thereby overcoming the limitations previously associated with realizing PIFO queueing. Moreover, the logic complexity of the architecture is $O(logN)$, rendering the approach highly scalable with respect to switch port densities and speeds.**

I. Introduction

Quality of service provisioning (QoS) is a core competency expected from any modern high-end switch or router. There are numerous growing application domains that are associated with strict session quality requirements, such as bounded delay and jitter. To facilitate such need, both input-queued (IQ) and output-queued (OQ) packet switching architectures have evolved to support various forms of QoS guarantees, primarily via realization of packet scheduling algorithms such as ones based on weighted round-robin (WRR) [2] and deficit round robin (DRR) [3]. In input-queued switches, virtual output queueing (VoQ) [4] is the prevailing queueing framework, in which arriving packets at each port are forwarded to a logically unique queue corresponding to the packets' output port. The complexity of such schemes lies in the scheduling algorithm that determines which input port will be connected to which output port at any given time slot [5]. Despite the scheduling complexity, IQ switches continue to dominate the high-end switch fabric architectures. Recent work on scalable output queued switching architectures [6][7], have began to introduce designs that offer the potential to scale to large port densities and data rates, while offering guaranteed QoS performance characteristics. In particular, OQ switching facilitates the employment of per-flow queueing and arbitration algorithms. The latter have been proven to offer strict delay, throughput and jitter guarantees on a per-flow basis, which is critical for many applications.

Support for QoS provisioning inherently implies that some packets may have higher priority than others. As a consequence, if a queueing mechanism is employed to impose service order, these higher priority packets should be placed ahead of other lower-priority packets in the queue. It has been shown that almost all QoS scheduling algorithms can be mapped to the Push-in-First-out (PIFO) queueing mechanism [8], in which newly arriving packets can be placed at an arbitrary location within the queue, while departures occur only from the head of line position. In PIFO queueing, the relative order of packets never changes, such that while newly arriving packets can be placed anywhere in the queue in accordance with their priority, there is a coherent overall structure imposed on the flow of packets. To that end, no explicit scheduling is needed in PIFO queueing.

Although PIFO is an appealing concept, it is generally perceived to be impractical from a hardware realization perspective. The primary hurdle lies in the complexity of the process needed to search for the correct location within the queue into which a packet with a given priority should be placed. Given that packet times are in the tens of nanoseconds scale, it would be impractical to assume that such a placement process can take place within a packet-time.

Recent work has attempted to shatter the prior perception on PIFO queueing, by proposing a different approach to its realization. In this work [9], a memory management method is introduced whereby an additional pointers array is employed to accelerate the rate at which the search for the location on arriving packets can be achieved. This scheme overcomes the inherent complexity of this search process, however it relies on the existence of a high-speed memory device which can perform a large number of read/write operations concurrently. Such a device does not exist, at least not at the scale that is

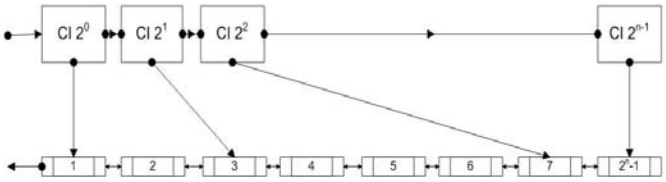

Fig. 1. Critical pointers array pertaining to initial positions in each of the n partitions of the doubly-linked list referencing packet data.

suggested in [9], reflecting existing high-end router sizes.

In this paper, we extend the core framework discussed in [9] by introducing a pipelined memory management architecture that guarantees PIFO queueing can be achieved both at high-speed and while consuming relatively modest hardware resources. The computational complexity of the search process reduced from $O(N \log N)$ to $O(1)$, while the hardware resources are $O(logN)$, where N denotes the maximal size of the PIFO queue assumed. These attributes imply that the proposed memory management architecture scales to very large port densities and data speeds.

The rest of the paper is structured as follows. Section II provides an overview of PIFO queueing formalism as well as the iPIFO memory management scheme. In section III, the proposed architecture is described and analyzed, while in section V the conclusions are drawn.

II. PIFO and Iterative-PIFO Queueing

The PIFO queueing mechanism represents a generic abstraction for many scheduling algorithms that offer QoS guarantees. At its core, that PIFO abstraction assumes that packets can enter a queue at any location/position, while departures occur only from the head-of-line (i.e. first) position of the queue. Another way of interpreting PIFO queueing is that it is one which keeps the relative order of packets. In other words, if packet A was ahead of packet B in the queue, then no packet C can ever change the order in which packet A and B would eventually depart the queue. Such packet C can only distant the time separating packet A and packet B departure times, but never alter their order of departure.

Although PIFO queueing offers a convenient construct for analyzing many scheduling algorithms, in practice it poses substantial difficulties when one attempts to implement it. In particular, given that the location of packets can vary greatly, a dynamic memory management is required to govern the flow of the queueing structure. Such a structure may be a doubly-linked list, where insertion of a new packet is trivial, once the correct position of that packet in the queue is found. However, as the queues grow in size, it becomes challenging to locate the position of a newly arriving packet within a *packet time* interval, which is typically in the order of tens of nanoseconds. In view of this difficulty, PIFO queueing has been considered a theoretical notion rather than a practical methodology used in actual hardware designs.

Recent work [9] has attempted to change this conception, by introducing iPIFO - an architecture that enables accelerated updating of a PIFO queueing structure, including both search for packet placement, as well as processing packet departures. This work introduced an auxiliary data structure called Critical Interval (CI) pointer array, which maintains pointers to positions within the main packet data structure (i.e. the doubly-linked list mentioned above) with exponentially increasing gaps, as illustrated in figure 1. For a PIFO queue with n positions, the corresponding CI pointer array would maintain $N = log_2 N$ pointers, one for each CI. An arriving packet first searches through the CI array to identify the CI to which it belongs, and then linearly searchers through the data packet structure for its precise position. The paper goes to prove that for a packet with a given departure time T, it would take no longer than $T - 1$ time steps to locate the packets precise position within the queue, thereby guaranteeing that every packet will be correctly placed in time. The paper continues to describe the CI array pointer update rules for both arrivals and departures.

Although iPIFO does offer a method for accelerating the search process through the queue, it fails to scale with respect to practical memory bandwidth requirements. Since searching for a packet's position in the queue may take more than a single packet time, several packets may be searching for their position within the queue concurrently. Although this does not impose a logical constraint with respect to the iPIFO memory management, it does suggest that a RAM device which can service a large number of simultaneous read operations is required. When the packets search for their position, they typically spend the majority of the search time linearly going through a given CI to identify their precise designated location. However, since this linear search requires access to a RAM device, a large number of concurrent read operations can not be made to the same device. Moreover, given that the number of parallel searches can be rather large, the iPIFO scheme inherently suffers from limited scalability. This paper attempts to overcome this limitation by means of a pipeline architecture and parallel processing modules.

III. Pipeline Iterative PIFO (Pi-PIFO) Memory Management

A. General Framework

The primary objective of the proposed pipelined PIFO queueing architecture is to overcome the hardware implementation challenges associated with PIFO queueing by means of a framework that is scalable and realizable using current VLSI technology. In particular, retaining low memory bandwidth requirements is addressed as a critical element of consideration. We begin by making the following basic assumptions: first, a packet can arrive/depart at a rate no higher than is needed to perform five memory access (i.e. read/write) operations. Stated differently, within five memory operations at most one packet may arrive and at most one would depart. Second, the time it takes to find the maximal value of n values (where n is $O(\log_2 N)$) is negligible with respect to time it takes to read

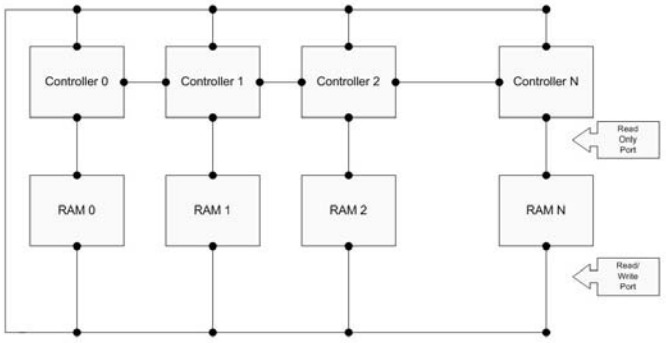

Fig. 2. The proposed scalable hardware architecture for implementing PIFO queueing, comprising of n concurrent memory and controller modules. The RAM modules are assumed to be dual-port memory units with access time in the order of packet duration.

or write a packet to a RAM device. This will assumption will be substantiated later in this section. Also, a QoS algorithm will assign each packet a departure time prior its arrival at the Pi-PIFO framework.

For a PIFO scheme to be practical, it must scale well (with respect to the queue size), exhibit hardware-efficiency, require low-memory bandwidth, and offer low latency attributes. Pi-PIFO accomplishes all these goals using regular, off-the-shelf dual-port SRAM devices. At its core, it is a natural extension to the framework of the iPIFO algorithm. However, unlike iPIFO, Pi-PIFO does not assume a high degree of concurrent read/write operations imposed on a single RAM device. In fact, the latter is assumed to support a single read operation at each of its two ports and single write operation for each packet time.

B. Architecture Description

The Pi-PIFO architecture comprises of an array of two basic components, as illustrated in figure 2. The first, similar to the iPIFO architecture, is a doubly-linked list that contains pointers to the critical intervals (CI). The second is a pool of dual-port RAM devices, one for every CI, each of which is controlled by a unique module responsible for correctly placing arriving packets in their designated positions. The PIFO queue is logically partitioned into the critical intervals, where each is twice as large as the one before it. As with the iPIFO scheme, there are a total of $n-1$ critical intervals, with a possible total of $N = 2^n$ packets in the queue.

Let the packet located last in order (i.e. oldest packet) in a given CI be labeled as the *tail packet* of that CI, and its pointers the *tail pointers*. This information is used in the process of placing newly arriving packets to the queue, as it facilitates the quick identification of the relevant CI in which the packet is to be placed. A detailed search process then takes place, with an aim to precisely locate the appropriate placement position within the CI. When a new packet arrives, it is compared to the tail packet's departure time in the first CI to determine whether it belongs to that interval. If the departure time of the arriving packet is higher than the tail packet in a given CI,

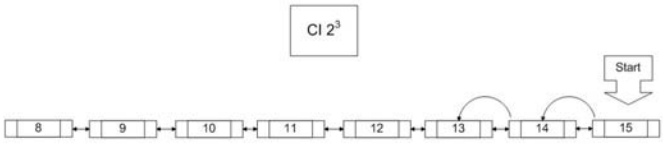

Fig. 3. Search process for a newly arriving packet, illustrated for the fourth critical interval within the doubly-linked list.

the search continues to the subsequent CI. Once the correct CI is found, the pointer to the tail packet in that CI and all CIs following it will be updated accordingly.

Once a packet locates its CI, it enters a FIFO queue at the controller corresponding to its CI, which hosts packets awaiting placement in that CI. Once at the head of the queue, a process of searching for the packet's specific position within that CI takes place. To achieve this goal, a sequential search through the doubly-linked list is required, which entails a read operation from the RAM device for each step. Given that there may be several packets "searching" for their appropriate position within the queue, this imposes a strong constraint on the memory bandwidth. The goal of the proposed architecture is to overcome this constraint by accurately distributing the work across multiple RAM devices in a manner that guarantees in-time packet placements.

As will be established, there is a total of five memory access operations required to take place during each packet time. We first address two of the five cycles, involved in the process of searching for the packets place in the PIFO queue. As such, the memory bandwidth is required to be *at five time* that of the packet arrival rate. In other words, within a single packet time (e.g. ~50nsec for 64-byte packets over a 10Gbps link), five memory access operations should be completed (i.e. access time of less than 10 nsec). The packet searching for its position needs to be able to step through at least two packets in the data structure in order to make up for the time it consumes while searching for its position within the CI, as well as to aid in limiting the number of packets needing to search through any given CI. This accounts for two of the five read/write cycles stated in the assumptions for the operation of Pi-PIFO. If another packet arrives while an earlier packet is searching for its position, it can use the information from the earlier packet's current reads from the RAM device in order to help determine its own position, thus minimizing the amount of time it takes to find its position following the placement of the packet ahead of it. The following proposition proves that a packet is guaranteed to be placed in its CI in time (i.e. prior to its departure time).

Proposition 1: Under the Pi-PIFO memory management scheme, a packet with departure time T can always find its position within its CI prior to time T.

Proof: Let's assume packet P is to be placed in location L within the PIFO queue. L is located in the critical interval described by $[2^j, 2^{j+1}]$. Since it takes one arrival/departure cycle to pass a packet between intervals, and two packets can

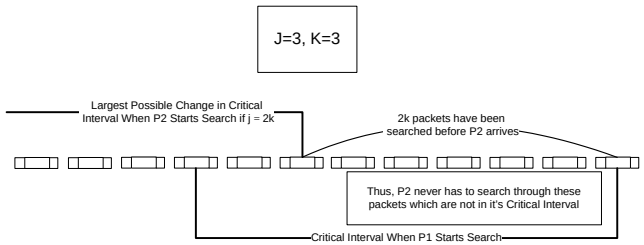

Fig. 4. Illustration of search when j <= 2k, in order to show that a later arriving packet will always find it's position in time.

be searched through during a single arrival/departure cycle, the number of departure/arrival cycles required for a packet to locate its position is always less than the number of packets ahead in its critical interval, since $j+2^{j-1} \leq 2^j$ for $j > 0$. We next show that packet P will be able to locate its position in time even if there is already a packet searching for its position within the same CI, when the latter packet arrives. This is true since P can use information obtained from the earlier arriving packet. Let there be k cycles between the time the first packet is assigned to the critical interval, and the time P is assigned to the same critical interval. Thus, the maximum number of positions that the interval could have shifted earlier in time (due to departures), and correspondingly, the number of positions that the earlier arriving packet could be in is $j+k$. This suggests that P could not possibly be located within these $j+k$ positions. Moreover, the earlier packet will have already searched though $2k$ positions before P arrived. Thus, if $j > 2k$ then the maximal time it would take for the second packet to locate its position is still less than the number of packets ahead in this critical interval, since

$$j + 2^{j-1} + \frac{j+k}{2} - k \leq 2^j. \quad (1)$$

The latter implies that if $j < 2k$ then the earlier arriving packet will not search outside the relevant position boundaries of P and thus the iPIFO proof for a single packet searching for its position holds. This can be seen in figure 4. The maximum number of positions the interval could have shifted later in time, and thus the number of positions that both the earlier arriving packet and P could not both possibly be located, is k. However, since we search at twice the packet arrival rate, all searches can be completed in $k/2$ cycles, therefore if $2^{j-1} > k$ then the first packet will have already found it's position and thus the iPIFO proof for the case of one packet searching for its position holds.

When a packet locates its position it needs to be inserted into the queue, as depicted in figure 3. This is shown to require three memory write cycles, accounting for the remaining three cycles of the five memory access cycles claimed in the assumptions made for the operation of Pi-PIFO. It takes one cycle to write the subject packet and an additional cycle for updating the pointers with respect to its two adjoining packets. Since only one CI can write to the queue at a time, the packet with the earliest departure time will need to be written first in order to guarantee that all packets are written to the queue in time. When implemented in custom silicon, the logic to compare such pointer values would takes approximately 0.5nsec, such that comparing 32 values would take less than 3 nanoseconds. This is a conservative figure using current technology [10]. Thus, we assert that the time required to compare the packets designated to be written from 32 CI's, or from a queue containing over 4 billion packets, would be negligible.

As long as we can write at least one packet to the queue per packet arrival/departure when there is a packet waiting to be written, packets are guaranteed to be written to the queue in time to depart in the proper order. This statement holds since given that a packet is required to be inserted into the queue within M cycles, and a packet is written ahead of it, the first packet gains a cycle in which it can wait and still be written on time. This is reflected by the following proposition. ■

Proposition 2: Under the Pi-PIFO memory management scheme, the critical interval tail pointers will always have time to be updated properly.

Proof: At most one packet can arrive and one packet can depart during any packet time interval. As a result, the largest number of packets that can find their position prior to a particular critical interval, j, is j packets. Let us assume that each controller (associated with each CI) maintains a register of $j+1$ pointers representing the packets prior to its tail packet. Since there are $j+1$ pointers stored in the pointer register, all prior packets to a given CI are accounted for. Moreover, it is important to note that it is not possible for two updates of j positions to occur consecutively. If all the critical intervals found a packet in the previous cycle, only the packet arriving in the current cycle can cause an update event during the current cycle. Thus, as long as the rate at which the registers can be updated is equal to or greater than the rate at which packets arrive, updating the tail pointers is achieved on time (see illustration in figure 5). Thus it is proven that in the proposed Pi-PIFO structure, the rate at which we can update the registers is five pointers per packet arrival/departure, which is established as sufficient to meet the departure time deadlines. ■

Throughout the entire process governing Pi-PIFO, it is important to keep the tail pointers of each CI updated properly in order to keep the intervals correctly sized. If an interval representation becomes larger than it actually is, the guarantee that a packet can find its position in time to be written to the queue in the proper order can be violated. Updating the pointers for each CI before the next packet arrives is critical. Each CI will need to store $x+1$ pointers and departure times in the dedicated register in order to expedite the time it takes to update its tail pointer.

A single pointer in the register is needed for the current tail position, while the other x pointers are required in case all the CI's preceding that CI happen to have one of the arriving packet be located in them. These registers can be updated before the pointers would need to be outside these x packets. Even if all x CI's preceding a given CI are updated during one

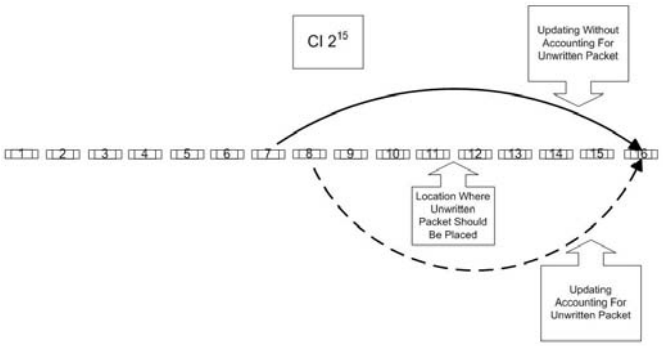

Fig. 5. Illustration of the tail pointer update scheme based on a pointer register locally available at each controller, and on knowledge of a packet waiting to be written.

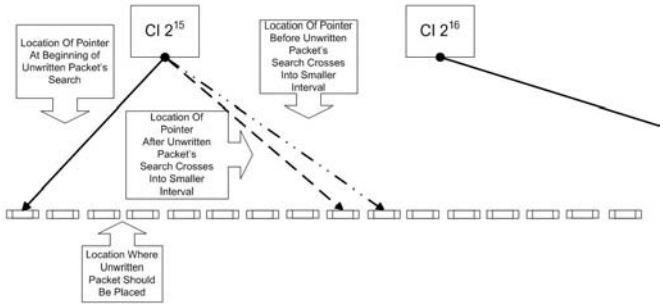

Fig. 6. Illustration of logic governing packet positioning within the doubly-linked list

cycle, it will take x cycles for that scenario to occur again again, hence we can always update the register faster than packets can arrive. As a result, the registers will always be ready when the tail pointers need to be updated. Since the proposed hardware architecture uses DPRAMs, this can be done independent of the read operations involved in the search through the queue as well as the write operations applied to the queue.

In addition to making sure that the tail pointers can be updated on time, it is imperative to guarantee that they are updated accurately. This makes it important to verify that unwritten packets searching for their positions don't get permanently "stepped over" when the tail pointers are being updated. In order to ensure that a packet does not get stepped over when packets are inserted ahead of its CI, the fact that each controller maintains the last x pointers in a dedicated register is exploited. Under the assumption that searching through this small register bank can be performed at high speed, each packet can search through this range before any tail pointer update can skip over the unwritten packet.

It is also crucial to validate that an unwritten packet does not get stepped over as the tail pointers are updated for queue departures. A packet that at one time was determined to be in interval m could actually be written to interval $m-1$ by the time it locates its position. In order to keep the intervals correctly sized, there is a need to track the number of positions an unwritten packet has searched through, and the number of departures and arrivals in front of that packet. This is then used in order to determine when the packet steps into a smaller interval and thus update the smaller interval's tail pointer accordingly (see figure 6). At first sight, achieving this would seem to be a complex task, particularly given that for some amount of time the tail pointer for the smaller interval will be incorrect when compared to what it should be. However, until the unwritten packet crosses into the smaller interval, the fact that it was determined to be in the smaller interval while it is still in the larger interval will be offset by the fact that during the search for its position, that packet does not have to search the unwritten packet.

The Pi-PIFO architecture can be summarized as follows. A packet arrives and its departure time is compared to the tail pointer's departure time for the first Critical Interval. If it is determined to be in this CI, all subsequent tail pointers are notified so they can be updated accordingly and the packet begins its search to find its position. If it is not in this CI, it is passed to the next CI and the process is repeated. Once a packet locates the CI in which it is to be placed, it begins searching from the latest departing packet in that interval towards the beginning of the interval at a rate of at least two packet searches per packet time. Should a packet arrive while an earlier packet is still searching for its position, the later arriving packet can also compare its departure time to the ones the earlier packet is currently comparing itself to in order to avoid searching through these packets positions twice. When a packet locates its position and has the earliest departure time of any packet that has found its position, it is written to the queue. Packets depart from the head of the queue, an event causing all the tail pointers to be notified such that they can be updated appropriately.

C. Hardware Requirements

The hardware required for the implementation of Pi-PIFO are a controller and a dual-port RAM (DPRAM) for each CI, as depicted in figure 1. Each DPRAM will contain identical information, but such replication is needed in order to guarantee that each packet will locate its position in time to be placed into the queue, since we require that only two memory locations can be read from each RAM device at a time. The DPRAMs are arranged such that when a packet is written into the queue, it is written to all DPRAMs concurrently (as facilitated by a dedicated $n-$signal bus). Since the number of RAMs required is only $n = \log_2 N$ (where N denotes the queue length), great degree of scalability is achieved as even with a queue length of four billion packets, this architecture only requires 32 RAM modules. The only other memory requirement imposed by the proposed scheme is that mandated by the registers located in each controller used for the purpose of updating the tail pointers. However, the latter are all $O(n)$, as proven above, such that negligible memory requirements are imposed by these registers.

IV. Conclusions

This paper introduced a novel PIFO memory management architecture, which exploits pipelining and distributed processing to yield a highly-scalable designs which lends itself well to hardware realization, With no packet latency costs associated and an $O(logN)$ hardware resources, the design offers unprecedented feasibility. Although the design requires a complex scheme, it still places reasonable limits on hardware requirements and only requires RAM large enough to hold the queue. The PIFO model and its memory management framework presented here can be directly applied to a broad range QoS scheduling algorithms, such as WRR, DRR and strict priority schemes.

References

[1] R. Cruz and S. Al-Harthi, "A service-curve framework for packet scheduling with switch configuration delays," *Networking, IEEE/ACM Transactions on*, vol. 16, pp. 196–205, Feb. 2008.

[2] Z. Guo and R. Rojas-Cessa, "Framed round-robin arbitration with explicit feedback control for combined input-crosspoint buffered packet switches," in *Communications, 2006 IEEE International Conference on*, vol. 1, (Istanbul), pp. 97–102, June 2006.

[3] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round robin," in *IEEE/ACM Transactions on Networking*, pp. 375–385, 1995.

[4] Y. Tamir and H. C. Chi, "Symmetric crossbar arbiters for VLSI communication switches," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 1, pp. 13–27, 1993.

[5] X. Li and I. Elhanany, "Stability of a frame-based oldest-cell-first maximal weight matching algorithm," *IEEE Transactions on Communications*, vol. 56, pp. 21–26, Jan. 2008.

[6] B. Matthews and I. Elhanany, "A scalable memory-efficient architecture for parallel shared memory switches," in *High Performance Switching and Routing, 2007. HPSR '07. Workshop on*, (Brooklyn, NY,), pp. 1–5, May/June 2007.

[7] I. Elhanany and M. Hamdi, *High Performance Packet Switching Architectures*. Springer-Verlag, September 2006.

[8] S. Iyer, R. Zhang, and N. McKeown, "Routers with a single stage of buffering," in *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 251–264, ACM, 2002.

[9] F. Wang and M. Hamdi, "iPIFO: A network memory architecture for qos routers," in *High Performance Switching and Routing, 2007. HPSR '07. Workshop on*, (Brooklyn, NY,), pp. 1–5, May/June 2007.

[10] R. Zlatanovici, S. Kao, and B. Nikolic, "EnergyŰdelay optimization of 64-bit carry-lookahead adders with a 240 ps 90 nm cmos design example," *Solid-State Circuits, IEEE Journal of*, vol. 44, pp. 569–583, Feb. 2009.