iPIFO: A Network Memory Architecture for QoS Routers

Feng Wang and Mounir Hamdi
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
{fwang, hamdi}@cs.ust.hk

Abstract[1] – Routers need memories to store and forward packets. More than that, routers use memories to schedule flows according to their quality-of-service (QoS) requirements. The simple first-in-first-out (FIFO) queue memory is insufficient to provide QoS guarantees. Most current routers are based on the virtual-output-queue (VOQ) memory management and use heuristic algorithms, such as iSLIP or DRRM, to schedule packets. On the other hand, push-in-first-out (PIFO) queue memory has also been proposed as a model for routers to meet the QoS requirements. The PIFO queue does not need a scheduler since packets are always *first-out* from the queue head. However, due to the sorting-related problems of the *push-in* operation, it is normally supposed impractical to build the PIFO queues in real hardware. We try to touch this problem in this paper and propose an *indexed* PIFO queue (iPIFO) architecture and a memory management algorithm on it. We believe it is a feasible solution to bring the PIFO queue to practice.

I. INTRODUCTION

One of the fundamental differences between circuit switches and packet routers lies in that routers need memories to place packets. Memories in routers are not only used in case of network congestion [1], they are also indispensable to provide quality-of-service (QoS). In a simple router, it may treat incoming packets first-come-first-served using one single FIFO queue memory. However, in a QoS enabled router, some packets may give way to other packets with higher priorities. Packets got delayed should find a place to stay, or get dropped in short of memories. How to schedule packets according to their QoS requirements rather than just relying on their arriving orders needs the help of more sophisticated memory systems.

The virtual-output-queue (VOQ) memory architecture has been prevailing for rather a long time in literature. Most QoS fair queuing algorithms assume the per-flow queues as basic architecture, e.g. weighted round-robin (WRR) and deficit round-robin (DRR) [2]. (*We note here that we may be ambiguous in talking about the per-flow queues and the VOQ, since we only focus on the multi-queued memory systems in this paper, regardless of per-flow based or per-output based.*) Based on the VOQ technique, many scheduling algorithms are designed, such as iSLIP [3], FIRM [4], and DRRM [1]. They are proved practical in terms of packet throughputs and delays.

The basic idea behind the VOQ memory management is to separate packets belonging to different outputs/flows in the input side. In this way, ill-behaving flows will have little control over other friendly flows. Therefore, it is possible to provide qualities of service to individual flows. However, we can see that the implementation complexity scales with the number of queues. It is a challenging task to design a finely granulated VOQ memory in today's core routers where the number of queues is normally at the order of millions.

Another dimension to solve the QoS problem is to use the push-in-first-out (PIFO) queue. This is a more *generalized* model to provide QoS than the VOQ based models. It is obvious to see that even with the VOQ memory packets are still switched from the input port sequentially, which they actually form a logical departure queue and their departure orders are calculated by the fair queuing algorithms. The most well-known theoretical works in QoS scheduling are the weighted fair queuing (WFQ) [5] and its packetized version GPS [6]. According to the WFQ algorithm, the incoming packet first gets a departure order issued by the scheduler and then inserts into the departure queue. Packets depart from the queue one by one from the head. Conceptually, this departure queue can be viewed as an instance of the *push-in-first-out* (PIFO) queue.

However, the PIFO queue has long been assumed impractical since the inserting/pushing operation can be easily reduced to a sorting problem. The fastest sorting algorithm we know has a time complexity of $O(N\log N)$, where N is the number of present packets in memory in our problem here. Using heap sort [7] as an example, inserting or extracting a packet takes $O(\log N)$ time complexity. This is too slow and not practical for hardware implementation, especially in the core routers where packets come in every several nanoseconds.

We try to touch this problem and propose an *indexed* PIFO queue (iPIFO) architecture in this paper. We believe we are among the first to think about real implementations of the PIFO queue, rather than just referring to it as a theoretical model for proofs. The iPIFO solution only requires the memory supports dynamic allocation and multiple concurrent

[1] This work is supported under RGC HKUST6260/04E.

reads. We believe it is of both practical and theoretical value and can be ready in hardware implementations.

The rest of this paper is organized as follows. We introduce the PIFO queue background and formulate the model in section II. The iPIFO memory architecture and detailed operations on it are shown in section III. In section IV, we prove the properties of the iPIFO memory architecture. We talk about some practical considerations in section V. Then we have a conclusion.

II. THE iPIFO QUEUE ARCHITECTURE

A. Problem statement: the PIFO queue

The push-in-first-out (PIFO) queue can be defined according to the following three rules.

1. Arriving packets are placed at (or, *pushed in*) an arbitrary position in the queue;
2. The relative order of packets in the queue does not change once packets are in the queue, i.e., cells in the queue *cannot* switch positions; and
3. Packets may be selected to depart from the queue *only* from the head.

The PIFO queue is logically a single queue in the memory. Incoming packets will be inserted/pushed into the queue to a position which is calculated by the queuing algorithm, such as WFQ [5] and GPS [6]. Their positions may change due to later incoming packets. For example, a packet is inserted in the position 4 at its arrival. It may be pushed back to position 5 if there is another packet with higher priority and should be inserted into a position less than 4. It may be further pushed back if more packets with higher priorities arrive subsequently. However, once two packets have been inserted into the queue, they cannot switch their positions.

The main difficulty in building the PIFO queue comes from the first rule. This dictates a *dynamic* allocation memory system. However, dynamic memory is normally implemented with *linked lists*, which make the insertion operation costly to implement since elements cannot be *statically* addressed. The most efficient *insert* operation in linked date structures comes from the heap-sort [7] algorithm that has an $O(\log N)$ time complexity[2]. However, there are two main problems in applying this heap-sort algorithm. *First*, the insert operation of $O(\log N)$ time is still too large for urgent-to-leave packets. *Secondly*, to *dequeue* a packet from the heap also needs $O(\log N)$ time to perform, which is definitely impractical.

Fortunately, the third rule helps us circumvent this problem. We observe one important property of the PIFO queue. *Packets far from the head will wait for some time to depart.* Put another way, the farther is the packet from the head, the later it departs. We utilize this property and propose an *indexed* PIFO (iPIFO) queue architecture. Then we devise an efficient algorithm to perform the two basic operations on the iPIFO: *insert* and *dequeue*. The basic intuition behind the algorithm is that we distribute weighted complexities to different packets according to their final positions. The earlier will the packet depart, the less complex is the *insert* operation. Although the overall complexity may be larger than $O(N\log N)$, we prove that every packet has sufficient time to finish the inserting operation before its time to depart.

B. The indexed PIFO (iPIFO) queue architecture

We use a *doubly linked list* to implement the PIFO queue, as shown in Figure 1. Every packet in the queue has two pointers. One points to the *next* packet and the other points to the *previous* one. Packets are inserted into the positions calculated by schedulers. Packets depart from the list head.

We use another small memory to index the doubly linked list, which is named index memory (IM). The IM not only accelerates the inserting process, but can distribute weighted complexities to packets according to their inserting positions as well. Since we use an additional index memory to enhance the performance of the doubly linked list, we call this architecture *indexed* PIFO (iPIFO) queue memory system.

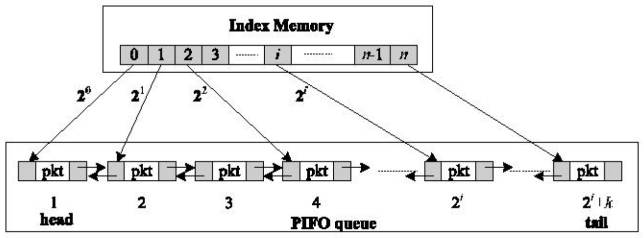

Figure 1: The iPIFO memory system

The size of the IM is $n+1$. It is *statically* allocated and can be directly addressed. The IM stores the selected $n+1$ packets addresses in the doubly linked list. We call the static addresses $(0,1,2...,n)$ in IM *indices*. We use i to represent the index of the IM and I to represent the mapped position in the PIFO queue. The mapping function is the following:

$$I = 2^i \; (0 \le i < n)$$

The index n in the IM always points to the *tail* of the PIFO queue which remembers the last packet address in the queue, as shown in Figure 1. We note that index 0 points to the *head* of the PIFO queue.

To get a feel of the size for real hardware implementation, consider a normal 10 *Gb* storage of the network memory. It may contain at most 32 million packets, assuming the minimum packet size of 40 bytes. The number of indices maintained by the IM is less than $\log_2 32\times10^6 < 25$. Therefore, the index memory is small enough to be built on-chip into the

[2] It can be proven that $O(\log N)$ is the lowest bound for the inserting operation. Otherwise, the sorting algorithm based on the heap data structure will have a time complexity less than $O(N\log N)$, which is false.

scheduler chip.

We define an index is *valid* if it points to a packet in the PIFO queue. Otherwise, the index value is assigned NULL. For example, in Figure 1, indices $(i+1, i+2, ...n-1)$ are all assigned Null. The total number of packets in the PIFO queue is $2^i + k (0 \le k < 2^i)$.

III. Operating on the iPIFO Queue

The iPIFO memory system mainly provides two basic operations to the scheduler:

1. *dequeue* a packet from the head of the PIFO queue
2. *insert* a packet into an arbitrary position in the PIFO queue, as the scheduler requires

In the following presentation, we use 'PIFO queue' to refer to the doubly linked list in Figure 1 and use 'iPIFO queue' to refer to the whole memory system, including the IM.

A. Dequeue a packet from the iPIFO queue

It is very simple for the iPIFO queue to dequeue the head packet. It just gives out the packet pointed by the 0-th index and updates all indices in the IM. All valid indices except the *n*-th index change their pointers to the *next* packets of their original ones.

For the last valid index except *n*, e.g. *i* in Figure 1, a little more work should be carried out. It becomes NULL if the index *i* points to the same packet as the tail does. Otherwise, it changes its pointer to the next packet of its original one.

B. Inserting a packet into the iPIFO queue

Here comes the most essential and difficult part of the iPIFO queue operation, inserting a packet into an arbitrary position. The position where the incoming packet should insert is calculated by the queuing algorithms, such as the WFQ [5] or GPS [6].

The remainder of this paper mainly talks about how to insert a packet into the position *p* correctly and efficiently. We assume that before the packet's arriving, there are $2^i + k (0 \le k < 2^i)$ packets presenting in the PIFO queue. That is to say, in the IM, indices $(0, 1, ...i)$ are really pointing packets in the PIFO queue, while indices $(i+1, ...n-1)$ are NULL. The *n*-th index points to the tail packet in the PIFO queue. We denote the position of the last packet $tail = 2^i + k$.

The insert operation can be elaborated in three phases.

Phase 1: find the critical interval containing p

We define the interval $[2^j, 2^{j+1})$ as the *critical interval* for *p*, if $2^j \le p < 2^{j+1} (j < i)$. The critical interval is unique for a particular *p*, as shown in Figure 2.

If $p \ge 2^i$, the critical interval of *p* is defined as $[2^i, 2^i + k + 1]$.

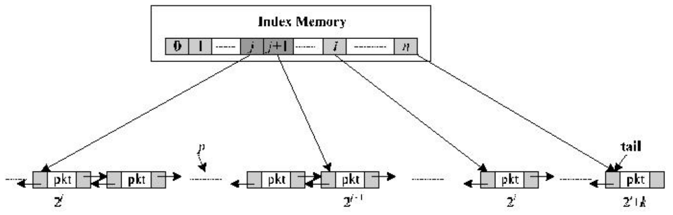

Figure 2: Critical interval of *p*

Phase 2: update indices in the IM

We use the pseudo-code below to explain how to update the indices in the IM. The algorithm is very simple since the affected indices are just those whose mapped positions are larger than *p* and they just have to point *one packet before* their current pointed packets. When the total number of packets reaches 2^{i+1}, the index $i+1$ should be validated and points to the last packet in the PIFO queue. For simplicity of the presentation, we assume *p* falls into the critical interval of $[2^j, 2^{j+1}), (j < i)$. It is an easier job if *p* falls into $[2^i, 2^i + k + 1]$.

```
Update_Indices ( )
  IF $p = 2^j$
      Insert the packet to position $2^j$
      Change_indices_forward $(j, j+1, ...i)$
      EXIT
  ELSE
      Change_indices_forward $(j+1, j+2, ..., i)$
      GOTO phase 3
  END IF
Change_Indices_Forward $(s, ...l)$
Every index in $(s, ...l)$ changes its pointer to one packet previous to their original one
```

Pseudo-code 1: update affected indices

Phase 3: insert the packet into the critical interval

This is the most time-consuming part in the iPIFO memory system. From phase 2, we can see that if *p* happens to be the boundary of the critical interval, the packet is inserted to the doubly linked list directly and all indices update correspondingly. However, if *p* falls into somewhere middle in the critical interval, we have no fast way to locate the position other than just searching from the boundary packet since the data structure is a dynamically allocated linked list. We state the detailed searching algorithm below and prove in the next section that an incoming packet always has sufficient time to find its position, no matter where it is.

Intuitively, if *p* is in the first half part of the critical interval, we search p from the left boundary of the interval. If p is in the second half part of the critical interval, we search p from the right boundary. Searching in this way is not only a consideration of efficiency, but very important to maintain proper packet orders as well, which we will prove later.

```
Find_Location (p)
If $p \in (2^j, 2^j + 2^{j-1}]$, the incoming packet moves in tail direction along the linked list from position $2^j$ until it has passed $p - 2^j$ packets. The packet inserts there and maintains proper forward and backward pointers in the doubly linked list.
If $p \in (2^j + 2^{j-1}, 2^{j+1})$, the incoming packet moves in head direction along the linked list from position $2^{j+1}$ until it has passed $2^{j+1} - p$ packets. The packet inserts there and maintains proper forward and backward pointers in the doubly linked list.
```

Pseudo-code 2: find position p

IV. Analysis of the iPIFO Algorithm

For dequeuing a packet out of the iPIFO queue, it only involves updating at most n indices (n is normally less than 25 for a large network memory of 10 Gb size) and every index just changes its pointer to one packet next.

For inserting a packet into the iPIFO queue, phase 1 and 2 are trivial tasks since they only involve updating several indices in IM. Phase 3 is the most time-consuming part.

We define a *time slot* to be the smallest time gap between two consecutive incoming packets to the iPIFO queue. It is obvious to see that packets can endure some time slots delay in finding their positions if it is not urgent to leave. We prove that under a trivial assumption, our insert algorithm guarantees packets be *ready* in their *proper* positions when it is time for them to depart.

Assumption: In phase 3 of the insert operation, when searching its position, the incoming packet advances forward/backward *more than* one packet in one time slot along the doubly linked list.

This assumption is very trivial. One time slot is at least the time for a packet to be written in the memory and it is for sure larger than the time to read an address from the memory.

Proposition 1: *A packet is always ready in its final location of the PIFO queue before the time for it to departure.*

Proof: Consider a packet A to find the location p, and the critical interval of p is $[2^j, 2^{j+1})$. According to phase 3, the number of packets A will search is less than the length of the critical interval $2^{j+1} - 2^j = 2^j$.

Since $2^j \le p < 2^{j+1}$, the number of packets before the critical interval of p is 2^j, and the queue takes at least 2^j time slots to switch those packets out. Therefore, packet A has sufficient time to find its location p before the queue starts to switch out any packet from its critical interval, including itself.

A similar proof can be obtained if the critical interval of p is $[2^i, tail]$.

■

Proposition 2: *If several packets fall into the same critical interval and search their positions simultaneously, they can still find their proper positions in the final PIFO queue.*

Proof: Take a packet A as an example. Without loss of generality, we assume A is searching position p from the left boundary of the critical interval – position 2^j.

It suffices to prove that no successive incoming packets can jump into the interval between A's current position and its final position p while A advances to p, so that passing $p - 2^j$ packets is correct for A to find its final position.

The successive incoming packets felling into the same critical interval as A can be divided into two parts. The first part of packets search from the left boundary – position 2^j, and the second part of packets search from the right boundary – position 2^{j+1}.

We first prove that the packets searching from the *left* boundary *cannot* catch up with A. It suffices to prove that the left boundary will not catch up with A while A is advancing to its final position. Since the left boundary can only move one packet backward in the tail direction due to a head packet's departure, it can move in the tail direction at most one position in one time slot. According to the assumption, the left boundary cannot catch up with packet A, thus, neither do the successive arriving packets searching from the left boundary.

We then prove the successive packets searching from the *right* boundary of the critical interval *cannot* jump into A's way to its final position. According to our algorithm, it is easy to see that the number of packets A moves over is less than

$$\frac{2^{j+1} - 2^j}{2} = 2^{j-1}$$

While the distance from p to the right boundary is larger than

$$2^{j+1} - (2^j + 2^{j-1}) = 2^{j-1}$$

In addition, the right boundary can move at most one packet forward in the head direction due to an incoming packet in one time slot. Therefore, successive arriving packets searching from the right boundary of the critical interval of p cannot jump into A's way to its final position. Please note here that after A has inserted into its position, packets from the right boundary may jump over A.

Thus, we prove that once packet A starts searching from the left boundary, no successive incoming packets can jump into its way to the final position p. Therefore, passing over $p-2^j$ packets guarantees packet A to find its proper position.

Similar proof can be obtained for packets searching from the right boundary of the critical interval.

■

In summary, we have proven that in the iPIFO memory system, every incoming packet has *sufficient* time to find its memory position before it is scheduled to depart. All incoming packets search in the iPIFO memory system *independently* and they are proven not to interfere with each other under our memory management algorithm.

V. Practical Considerations

To implement the iPIFO memory system, it is required that we employ memories supporting dynamic allocations. This is dictated by the inserting operation in the PIFO rules. It is common practice to build large capacity dynamic memory systems with current technology.

In addition, as we can see from the proofs above, the iPIFO memory system should support one write and multiple read operations simultaneously. In every one time slot, there is at most one packet being written into the memory, while there may be multiple address reads from packets searching for their correct positions. For this purpose, a Concurrent Read Exclusive Write (CREW) PRAM[7], where multiple processors can read from a cell, but only one write to it, will suffice.

VI. Conclusions

In this paper, we try to build a practical push-in-first-out (PIFO) memory system, which is normally only a theoretical model in analyzing QoS schedulers and assumed impractical to implement.

We first point out an interesting property that is inherent in the PIFO queue memory. That is, packets far from the head can wait for some time to depart. This allows us to distribute weighted complexities to packets according to their final positions. Then we build an indexed PIFO (iPIFO) memory system and design an practical memory management system on it.

The iPIFO algorithm only requires the memory support dynamic allocation and multiple concurrent reads. With current memory technologies, we believe that our solution is feasible to implement in common hardware for high performance routers supporting quality-of-service guarantees.

References

[1] H. J. Chao and J. S. Park, "Centralized contention resolution schemes for a large-capacity optical ATM switch," in *Proceedings of IEEE ATM workshop*, 1998.

[2] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round robin," in *Proceedings of ACM SIGCOMM*, 1995.

[3] N. McKeown, "The iSLIP scheduling algorithm for input-queued switches," *IEEE/ACM Transactions on Networking,* vol. 7, pp. 188-201, 1999.

[4] D. N. Serpanos and P. I. Antoniadis, "FIRM: a class of distributed scheduling algorithms for high speed ATM switches with multiple input queues," in *Proceedings of IEEE INFOCOM*, 2000.

[5] Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queuing algorithm," *Journal of Internetworking: Research and Experience,* 1990.

[6] Parekh and R. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single node case," *IEEE/ACM Transactions on Networking,* vol. 1, pp. 344-357, 1993.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (2nd edition)*: The MIT Press, 2002.