Efficient Per-Flow Queueing in DRAM at OC-192 Line Rate using Out-of-Order Execution Techniques

Aristides Nikologiannis† and Manolis Katevenis†

Institute of Computer Science, Foundation for Research and Technology (FORTH),
Science and Technology Park of Crete, Vassilika Vouton, P.O. Box 1385, Heraklion, Crete, GR 711 10 Greece
http://archvlsi.ics.forth.gr/muqpro/queueMgt.html anikol@ics.forth.gr kateveni@ics.forth.gr
Tel: +30 (81) 391660, Fax: +30 (81) 391661

Abstract-Modern switches and routers often use dynamic RAM (DRAM) in order to provide large buffer space. For advanced quality of service (QoS), per-flow queueing is desirable. We study the architecture of a queue manager for many thousands of queues at OC-192 (10 Gbps) line rate. It forms the core of the "datapath" chip in an efficient chip partitioning for the line cards of switches and routers that we propose. To effectively deal with bank conflicts in the DRAM buffer, we use pipelining and out-of-order execution techniques, like the ones originating in the supercomputers of the 60's. To avoid off-chip SRAM, we maintain the pointers in the DRAM, using free buffer preallocation and free list bypassing. We have described our architecture using behavioral Verilog (a Hardware Description Language), at the clock-cycle accuracy level, assuming Rambus DRAM. We estimate the complexity of the queue manager at roughly 60 thousand gates, 80 thousand flip-flops, and 4180 Kbits of on-chip SRAM, for 64 K flows.

I. INTRODUCTION

The explosive growth of Internet traffic has created an acute demand for integrated-service networks of ever increasing bandwidth. Networking companies are called upon to design and manufacture the fastest possible switches and routers. Line (port) speed is one parameter that must grow, and valence (number of ports) is the other such parameter. Port speed is in the OC-12 to OC-48 (622 Mbps to 2.5 Gbps) range today, and will grow to OC-192 (10 Gbps) very soon. The number of ports is in the tens to hundreds range today, and will need to grow to thousands.

High valence, high-speed switches/routers usually consist of a switching fabric, an ingress module for each input link and an egress module for each output link. The implementation of the switching fabric is challenging; however, it is not a topic of this paper. This paper concentrates on the architecture of ingress and egress modules. Both of the ingress and egress modules provide buffering and scheduling. In addition, the ingress module provides header processing and routing decisions. Since the ingress functions are usually a superset of the egress ones, we will focus mostly on the former module architecture.

Besides raw throughput, the customers of modern networks also demand Quality of Service (QoS) guarantees. In our opinion, the provision of advanced QoS guarantees requires true flow isolation that can only be achieved using per-flow queueing [3], [4] in connection with a good scheduler [2]. Per-flow queueing, for many thousands of flows, was considered an excessively expensive architecture up to a few years ago. Modern technology, however, provides the means to implement such architectures within a fraction of an integrated chip (IC) [1].

This paper studies the implementation of such architectures at OC-192 (10 Gbps) line rates. We show that, although challenging, this implementation is feasible, using the advanced hardware techniques that were developed for supercomputers in the 60's and are used in high-end microprocessors now-a-days. We present our precise model of such an implementation; it was written using the "Verilog" Hardware Description Language, at the accuracy level of individual clock cycles.

Section II presents a chip partitioning for the ingress module that economizes on chip-to-chip communication, so that pin count and power consumption are reduced. Section III describes the buffer memory technology (Rambus). In order to effectively use DRAM buffer memory, accesses have to be scheduled in the presence of bank conflicts. We use multiple, pipelined control processes to achieve out-of-order scheduling of DRAM accesses. The data dependencies among successive operations are handled using Tomasulo's dynamic scheduling techniques (operand renaming) [5], [6]. These sophisticated techniques also handle variable header processing time in an efficient manner. Section IV reports on our method of economizing off-chip memories and chip pins by locating a fraction of the queue manager data structures in the buffer memory itself, and using free list bypassing [7] and buffer preallocation. Section V describes the structure of the advanced queue manager pipeline. Finally, section VI estimates the implementation cost based on our cycle-accurate Verilog hardware description.

II. INGRESS MODULE CHIP PARTITIONING

The ingress module has considerable complexity, and thus its implementation as a single chip would be problematic, even using modern VLSI technology. This complexity is due to both the number of functions to be performed (routing, buffering, scheduling), and to the number and size of memories required (routing/classification tables, buffer memory, queueing data structures, scheduling parameters and state).

† Also with the Dept. of Computer Science, University of Crete, Heraklion, Crete, Greece

We assumed a partitioning of the ingress module into three chips plus the off-chip memories, as shown in figure 1. This partitioning reduces the chip-to-chip communication throughput, so as to reduce both pin count and power consumption. Packet bodies account for the majority of bits under manipulation, when compared to packet headers. Thus, packet bodies are kept in single chip boundaries when being buffered in (necessarily off-chip) memory and when entering or leaving the ingress module. Header processing represents a considerable amount of work that only communicates with the rest of the module though packet headers and flow identifiers; thus we assumed that it is positioned in a separate chip. The same is true for scheduling, which only communicates with the other chips through narrow words: flow identifiers.

When packets enter the datapath chip, their bodies have to wait until header processing has identified the flow to which they belong; subsequently, queue management must identify a buffer address; then, DRAM memory has to become available (see section III-A). During this waiting period, packet bodies are kept in a memory, which we call "transit buffer" (figure 4) and are not moved from processing stage to processing stage, so as to avoid additional power consumption.

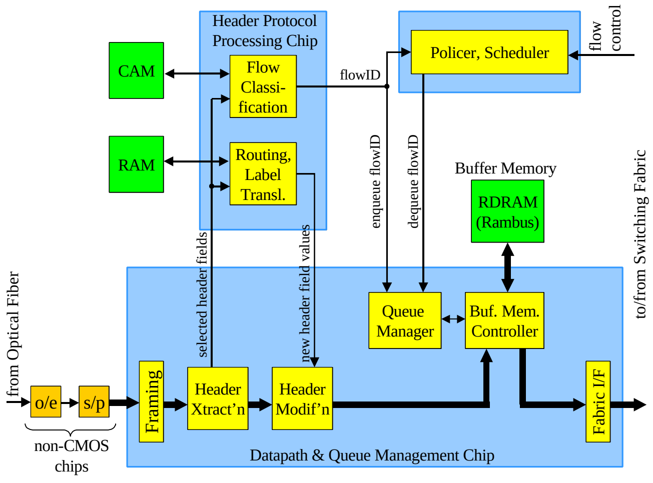

Fig. 1. Ingress Module Chip Partitioning

III. Interleaved DRAM Accesses and Out-of-Order Scheduling

A crucial design decision at such high rates is the choice of the buffer memory technology. SRAM provides high throughput but limited capacity, while DRAM offers comparable throughput[1] and significantly higher capacity per unit cost. Thus, we chose DRAM [8]. Among DRAM technologies, we chose Rambus [9] over DDR-SDRAM, because Rambus offers higher throughput at lower pin count.

[1] Memory chip throughput is a matter of I/O interface rather than storage core; SRAM and DRAM both use similar I/O interface techniques, today.

A. Rambus DRAM Technology

Rambus technology [9] provides 12.8 Gbps peak throughput per memory chip (RDRAM) by using a 2-byte data bus at 400 MHz with double clocking (i.e. 800 Mbps/pin). A RIMM module contains up to 16 RDRAM chips and provides 128 MBytes total capacity. Each RDRAM chip is partitioned into 16 banks (a RIMM module contains 256 banks) in order to provide interleaving (i.e. to allow multiple parallel accesses). Up to four accesses to different banks can be in progress, simultaneously. The memory access latency is about 60ns, while successive accesses to the same or adjacent banks may be performed every 100ns due to the bank-precharging interval.

B. Out-of-Order DRAM Accesses

When a memory transaction tries to access a currently busy bank (a bank that has not yet been precharged), as opposed to an available bank, we say that a bank conflict has occurred. This conflict causes the new transaction to be delayed until the bank becomes available, thus reducing memory utilization. When random accesses are made to an interleaved DRAM, some bank conflict will inevitable occur, as illustrated in figure 2, left, where a memory consisting of 4 banks (A, B, C, D) is assumed. If the bank cycle time is 3 time units, we can access the same bank every 3 time units. Therefore, the second and the successive transactions would be delayed two time units.

To reduce the number of bank conflicts, thus increasing memory utilization, we can rearrange the order of memory accesses, as in the simple example of figure 2, right. This implies out-of-order execution [5], which requires some control hardware complexity (section V-B), but is quite beneficial to performance.

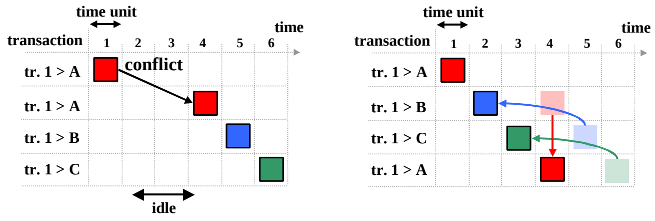

Fig. 2. Non-Interleaved versus Interleaved Transactions

IV. Queue Pointer Management

Modern switching fabrics operate on fixed size segments, because that simplifies the hardware, reduces the cost, and increases system speed. Moreover, implementing multiple queues inside a shared memory is almost impossible unless all memory allocation is done in multiples of a fixed-size unit. We assumed a 60-byte-size data segment since this size is relatively small, so as to reduce the delay for high priority traffic, and it is close to the ATM cell size.

In order to organize the segments in multiple queues, two pointers per queue are required: one pointing to the head of queue and the other to its tail. The head/tail pairs of all the system queues are kept in the Queue Table. Additionally, a pointer is needed per memory segment, pointing to the next segment in a queue, as shown in figure 3. The memory that maintains these pointers is called Next Pointer Memory.

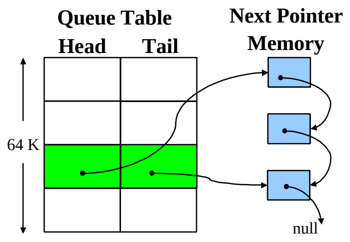

Fig. 3. Queue management data structures.

A. *Next-Pointers in Buffer Memory*

In order to economize on off-chip memories and minimize the number of chip pins, we locate the next pointers in the DRAM buffer memory instead of a separate off-chip SRAM. This choice adds an overhead of 4 bytes per memory segment (4 bytes next pointers + 60 bytes data = 64 bytes memory segments). It also grows the latency of accessing next pointers, because the access time for DRAM is about half an order of magnitude longer than for SRAM. Additionally, locating the next pointers in the buffer memory would normally double the number of buffer memory accesses per enqueue or dequeue operation, because two accesses at different addresses would have to be performed: an access to the data field and an access to the next pointer field[2].

B. *Buffer Preallocation Technique*

To avoid the increase of memory accesses, we used *buffer preallocation*: one free buffer is preallocated for every queue, and linked as the last buffer in the queue. When a new segment is to be enqueued, it is always written into the preallocated buffer; at the same time, we write into the *same buffer* the pointer to a new free block, which is thus preallocated and linked into the queue.

C. *Free List Bypassing Technique*

Another drawback of placing the next pointers in the buffer memory is the need for additional accesses to this memory when manipulating the free list. In the case of enqueueing and dequeueing a segment in the same time slot, four memory accesses would have to be performed: read the departing and write the incoming segment plus a read and a write operations to update the free list head and tail, correspondingly. To avoid this, we use *free list bypassing* [7]: rather than dequeueing a departing buffer from an output queue and enqueueing that buffer into the free list, and rather than extracting a buffer from the free list and enqueueing it into another queue upon arrival, we combine the two operations: the buffer into which an arriving segment is placed, is precisely the buffer from which a segment is departing during the same time slot; thus, there is no free list operation. This technique requires only two memory accesses: write the data field of the arriving segment and read the data field of the departing segment. Note that, for this technique to be applicable, multicast segments have to depart at once –not multiple times: the buffer of every departing segment has to be freed right away. During time slots when there is no incoming traffic, a free list enqueue operation is performed instead. When there is no outgoing traffic, free list dequeues are performed instead.

[2] When the next pointers are located in a separate memory, the next pointer accesses are performed in parallel with the data accesses, thus keeping the rate of buffer memory accesses down to one per enqueue or dequeue operation.

V. MULTI-PIPELINE STRUCTURE

At least two Rambus channels are needed in order to support incoming and outgoing link rates of 10 Gbps. Each Rambus channel provides a throughput of one 64-byte block every 40 ns, i.e. 12.8 Gbps. Thus, two channels just suffice for 10 Gbps incoming traffic, 10 Gbps outgoing traffic, 4 Gbps of fragmentation loss[3], plus 1.6 Gbps for a small speedup, for a total of 25.6 Gbps. We assumed two Rambus channels in our design; the architecture would be similar if one decided to use more channels in order to support a higher speedup factor and/or shorter packet sizes (higher fragmentation loss).

Because of the long and variable delay of header processing and the long latency of buffer memory accesses, an enqueue or a dequeue operation require many time slots to complete. Without operation overlap, the accomplished operation rate would be inferior to the required operation rate. Thus, in order to achieve the expected rate of 25.6 Gbps, the queue manager must be designed in a pipelined fashion.

A. *Control Processes*

Due to the interleaved management of DRAM accesses and the placement of the next pointers in buffer memory, advanced pipelining techniques are required. In our design, the queue manager architecture consists of five parallel and pipelined processes: three for enqueue, and two for dequeue.

The first enqueue process receives the incoming packets and places them in the transit buffer (figure 4). Additionally, it extracts the packet header fields and transmits them to the header processor for routing and classification. As soon as the header processor identifies the flow that an incoming packet belongs to, the second enqueue process issues an enqueue operation for that packet. More precisely, this process must issue multiple enqueue operations corresponding to the number of segments that the packet is fragmented in; this will be discussed thoroughly in section V-C. The issue of an enqueue operation means the acquisition of its operands. An enqueue operation has two operands: the pointer to the preallocated buffer for writing the incoming segment in, and the pointer of a new free buffer for linking it in the queue (pre-allocation). Finally the third enqueue process executes

[3] Computed for an average packet size of 270 bytes, using 60-byte segments.

an enqueue operation and updates the queue manager data structures. The third enqueue process is separate and asynchronous relative to the second because of the variable delay of out-of-order DRAM accesses, while waiting for bank conflict resolution.

The first and second dequeue process have corresponding functionality with the second and third enqueue processes. When the scheduler of packet departures decides to dequeue a segment from a queue, the first dequeue process receives this information and issues a dequeue operation in the system. Next, it tries to acquire the dequeue operand: the pointer to the head of queue. Finally, the second dequeue process executes the dequeue operation and updates the queue manager data structures.

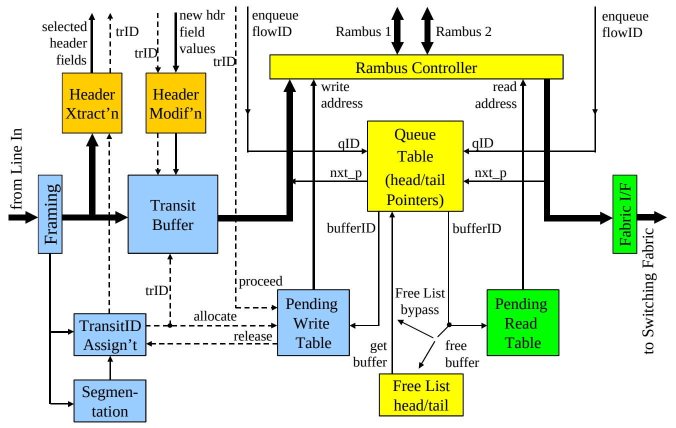

Fig. 4. Queue Manager Architecture Block Diagram

B. *Pending Operation Tables*

Rearranging enqueue and dequeue operations in a different order than the order they are issued is a suitable solution to maximize memory throughput utilization in the presence of memory bank conflicts. To achieve it, we used the Pending Write Table (PWT) and the Pending Read Table (PRT), as shown in figure 4. The former holds pending write transactions generated by enqueue operations, while the latter holds pending read transactions generated by dequeue operations. An operation is characterized as pending in two cases: when it cannot be executed due to a memory bank conflict, or when it has not yet received its operands because they are not yet available in the queue management data structures. PWT is the means of communication between the second and third enqueue processes; the second enqueue process places pending write tasks into the PWT, while the third enqueue process consumes them from the PWT. Similarly, the PRT is the means of communication between the first and second dequeue processes. Note that the PWT and the transit buffer are related and have the same number of entries. A transit buffer entry keeps the body of a segment that is waiting to be written into the buffer memory, while the corresponding entry in the PWT keeps the control information for this write transaction. A transit_id is assigned to each PWT entry in order to identify the pending write operations in the system. We considered that a PWT with 128 entries is large enough. Respectively, we assign a transit_id to each PRT entry in order to identify the pending read operations. The PRT contains 128 entries, but this choice is independent of the PWT capacity.

C. *Data Dependencies Handling (Operand Renaming)*

If two successive enqueue or dequeue operations are directed to the same flow, the second operation may find the state of the queue tail/head pointer as pending because the first has not updated it yet. In this case a data hazard occurs. The tail/head pointer may remain in the pending state for many time slots due to the memory access rearrangement and the high memory access latency. We handle this situation by using the operand renaming technique; this technique originates in Tomasulo's dynamic scheduling [5], [6]. For simplicity, we will only examine the enqueue operation; the dequeue has identical logic. Whenever an enqueue operation acquires the queue tail operand, it sets the queue tail state as pending and stores its transit_id in the corresponding tail field of the Queue Table. A successive enqueue operation that finds the state of the queue tail as pending, acquires the transit_id of the enqueue operation, which will create the expected operand. Then, the newly issued enqueue operation is kept in a PWT entry and waits the updated value of the queue tail. As soon as the expected queue tail is available, it must be communicated to the proper pending enqueue operation.

In our implementation, operand communication is done by organizing all the pending enqueue operations for the same flow in a single linked list. Each enqueue operation in the list updates the operands of its next pending enqueue operation during its execution period. As soon as a pending enqueue operation acquires its operands, it is characterized as "ready for execution". The data structures for the pending lists are kept in the PWT. Similarly, dequeue operations for the same flow are organized in a pending list, which is kept in the PRT.

Multiple enqueue operations per packet arrival (section V-A) are handled in the same way as the operand communication, above. The first enqueue process (section V-A) organizes the incoming segments of a packet in a linked list. The data structure for such a list is kept in the PWT. The list of successive enqueue operations for a flow and the list of enqueue operations for the packet segments in the same flow are merged in the same list; for more details see [10].

VI. HARDWARE IMPLEMENTATION COST

We described our design using the Verilog hardware description language, in a behavioral style, at the accuracy level of individual clock cycles; some modules were described in structural Verilog form. We assumed a clock frequency of 100 MHz; each Rambus channel is carried over a 128-bit wide datapath, inside the chip, at that clock frequency. This clock frequency is conservative for, e.g., a 0.18-micron technology; this simplifies logic-partitioning and pipelining tasks; one access to an on-chip memory plus several levels of combinational logic will normally fit within a clock cycle.

Table I shows the on-chip memory requirements for the datapath chip when 64 thousand queues are supported and the transit buffer, the PWT, and the PRT have 128 entries each. The PWT, PRT, and Queue Table are composed of multiple memory blocks each, in order to allow independent parallel accesses to various of their fields. Table II shows the number of Verilog code lines used to describe the queue manager and Rambus memory controller architecture. Finally, we estimate very roughly the hardware complexity of our architecture to be in the range of 60 thousand gates, 80 thousand flip-flops, and 4180 Kbits of on-chip SRAM, for 64 K flows; for more details see [10].

TABLE I
DATAPATH CHIP MEMORY REQUIREMENTS

Memory Block	Organization	Capacity (Kbits)	Ports	ASIC area (0.18μm)
Queue Table	64K x 64	4096	1 port	50 mm^2
Pending Write Table	128 x 96	12	2 ports	0.36 mm^2
Pending Read Table	128 x 64	8	2 ports	0.24 mm^2
Transit Buffer	128 x 512	64	2 ports	1.96 mm^2

TABLE II
VERILOG CODE LINES

Processes	Code Lines
Enqueue	1900
Dequeue	2300
Rambus Memory Controller	3800

VII. CONCLUSION

We presented the design and evaluation of a high-performance queue manager architecture that operates at OC-192 line rate. It uses modern DRAM (Rambus) technology for buffer memory, which provides high throughput and adequate buffer space. In order to fully utilize the DRAM throughput, in the presence of bank conflicts, we perform out-of-order transactions. By placing the next pointers in the buffer memory we economize on off-chip memories; when combined with buffer pre-allocation and free list bypass, access performance does not suffer. We showed how the dynamic scheduling techniques, originating in the supercomputers, are applied in our architecture. Finally, we wrote a clock-cycle accurate model, in Verilog, and estimated the hardware complexity of our architecture.

ACKNOWLEDGMENT

We would like to thank Europractice and the Univ. of Crete for providing many of the CAD tools used, and the Greek General Secretariat for Research & Technology for the funding provided.

REFERENCES

[1] G. Kornaros, C. Kozyrakis, P. Vatsolaki, M. Katevenis: “Pipelined Multi-Queue Management in a VLSI ATM Switch Chip with Credit-Based Flow Control”, in Proc. ARVLSI'97 (17th Conf. on Adv. Research in VLSI), Univ. of Michigan, Ann Arbor, MI, Sept. 1997, IEEE Comp. Soc. Press, ISBN 0-8186-7913-1, pp.127-144; http://archvlsi.ics.forth.gr/atlasI/atlasI_arvlsi97.ps.gz

[2] S. Keshav: “An Engineering Approach to Computer Networking”, Addision-Wesley, 1997, chapter 9.

[3] V. Kumar, T. Lakshman, D. Stiliadis: “Beyond Best Effort: Router Architectures for the Differentiated Services of Tomorrow's Internet”, IEEE Communications Magazine, May 1998, pp152-164.

[4] B. Suter, T.V. Lakshman, D. Stiliadis, A.K. Choudhury: “Buffer Management Schemes for Supporting TCP in Gigabit Routers with Per-Flow Queueing”, IEEE Journal in Selected Areas in Communications, August 1999.

[5] J. Hennessy, D. Patterson: “Computer Architecture: A Quantitative Approach”, Second Edition, Morgan Kaufmann Publishers, 1996, ISBN 1-55860-329-8, chapter 4.

[6] D. Krning, S. Mller, P. Wolfgang: “A Rigorous Correctness Proof of the Tomasulo Scheduling Algorithm with Precise Interrupts”, Proc. of the SCI'99/ISAS'99 International Conference, 1999.

[7] P. Andersson, C. Svensson (Lund Univ. of Sweden): “A VLSI Architecture for an 80 Gb/s ATM Switch Core”, IEEE Innovative Systems in Silicon Conference, Oct. 1996.

[8] Tzi-cker Chiueh, Varadarajan: “Design and evaluation of a DRAM-based shared memory ATM”, 1997 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 97) Seattle, WA, USA, 1997.

[9] http://www.rambus.com

[10] A. Nikologiannis: “Efficient Per-Flow Queueing in DRAM at OC-192 Line Rate using Out-of-Order Execution Techniques”, Master of Science Thesis, University of Crete, Greece; Technical Report FORTH-ICS/TR-279, Institute of Computer Science, FORTH, Heraklion, Crete, Greece, November 2000; http://archvlsi.ics.forth.gr/muqpro/queueMgt.html