Architecture of an Embedded Queue Management Engine for High-Speed Network Devices

M. Alisafaee, S. M. Fakhraie
Department of Electrical and Computer Engineering,
University of Tehran,
Tehran, Iran
alisafaee@cad.ece.ut.ac.ir, fakhraie@ut.ac.ir

M. Tehranipoor
Department of Computer Science and Electrical Engineering,
University of Maryland Baltimore County,
Baltimore, MD
tehrani@umbc.edu

***Abstract*—Network buffers used in network devices, have to allow line-speed buffering of packets while they maintain a large number of queues. Due to the growing speed of network links and the increasing number of data queues, the design of network embedded systems operating at high speeds is often restricted by their memory subsystem performance. In this paper, we show architecture of a network buffer subsystem which efficiently manages storing and retrieving data packets of multi gigabit network lines among 16 K different queues. The proposed architecture is implemented in FPGA and uses available memory technologies. It is ideal to be used as a queue management component in network processors, switches, stream processors, or any other application which require high-performance queue management engines.**

I. INTRODUCTION

Network buffers (packet buffers) are used in almost all types of network switches and routers to buffer incoming flow of packets. Speed and capacity of packet buffers must be increased as the line rate of the network communication links increases. Due to the growing speed of network links, design of the network embedded systems operating at high speeds is often restricted by their memory subsystem performance. Besides, advanced network applications demand the provision of the Quality of Service (QoS) which one of its fundamental requirements is packet buffers maintaining a large number of queues [1]. This demands the packet buffers to have complex data structures for implementing such queues and their related operations.

Most of existing packet processing engines do not have a dedicated component for packet memory management and use software-based solutions. With the rapid growth in network line rates and the desire for advanced QoS techniques, such software-based methods fail to fulfill the required performance.

In this paper, we propose architecture of an embedded queue management engine that is implemented using FPGA technology. It uses available DRAM technologies to handle high line rates. This architecture comes along our research for enhancing buffering capabilities of network devices and realizes a novel method for increasing memory bandwidth. This method along with a method we use for accessing the different banks within a multi-bank DRAM device, will simplify free linked-list management, memory access arbitration, and bank conflict avoidance while a high portion of memory bandwidth is utilized. The proposed architecture concurrently provides two elementary interfaces, one for read and one for write. It supports different operations on queues and operates on both fixed-size and variable size data packets and provides a 512 MB buffering space which could be shared among 16 K different queues. All of its functionalities are available through a rich instruction set designed for maximum flexibility.

The rest of the paper is organized as follows: Section II describes our previous proposed method to increase the memory bandwidth. Section III describes the proposed architecture and decisions made in the design of the proposed architecture. Hardware implementation results and performance analysis are shown in Section IV, and Section V contains our concluding remarks.

II. INCREASING THE MEMORY BANDWIDTH

In [2], we proposed a methodology for designing high-performance packet buffers using slower lower-cost single-port memory building blocks. To increase the memory bandwidth, we used parallel memory devices along a scheduling policy for accessing them. Fig. 1 shows an example of this method when we use two separate memories to construct a double bandwidth packet buffer.

In this figure, two separate memory devices are used in parallel and each memory device is coupled to an input first-in first-out (FIFO) memory. Using this method, two memory accesses (one read and one write) can be performed concurrently, so, the entire buffer operates as a dual-port memory. Incoming packets are distributed evenly among two memories such that for any arrival and departure traffic pattern, the occupancies of the two memories will remain almost equal. For architectures which use multiple parallel

0-7803-9197-7/05/$20.00 © 2005 IEEE.

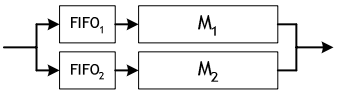

Figure 1. Schematic of a packet buffer with double bandwidth

memory devices, this is a requirement to ensure that the entire buffering space will be utilized for any traffic pattern [3].

The architecture we proposed in the current paper is a realization of the described method which uses two separate memory devices to operate as a dual-port network buffer. For implementing this method, some decision about memory types, data structures, and memory access pattern are made which are discussed in the following section.

III. Architecture of the Queue Management Engine

The block diagram of the proposed queue management engine is shown in Fig. 2. In this architecture, packets are arriving the system from the input port and are departing form the output port. The system is controlled through the command interface from which different packet commands are received. This architecture uses two memory devices to store the packet data and also uses an internal data structure to implement the queues. Dedicated to each memory, there is a memory controller which is responsible for writing (reading) data packets to (from) the memory. Memory controllers get the required instructions about the operation they must do from the system controller. System controller receives commands from the command interface. After processing these commands, it issues some micro commands to either of memory controllers. Inside the queue management system, we work with fixed-size segments of data, also known as cells. Therefore at the input port, there is a segmentation unit which splits incoming variable size packet, into fixed-size cells. At the output, a reassembly unit will merge the departing cells and make a complete packet.

A. Data Structures

The proposed architecture manages buffering the data packets among 16 K different queues. Therefore, we need to use some data structures to implement the queues. To decrease complexity of the entire system, the buffering space is virtually divided into fixed-size memory chunks known as segments. Using segments, the operation of allocating and releasing memory is simple and queue implementation is also easier. To implement the queues inside the memories, we used the linked-lists. For each of two memory devices there is a pointer memory (PT1 and PT2), and for each segment in either of two memories, there is a corresponding pointer in the respective pointer memory. These pointers are used to relate the packets of each queue together. To determine the boundary of packets (which are stored as a series of consequent linked segments within a queue), an end segment flag is provided in the pointer memory which determines if the corresponding segment is the last one of the packet or not.

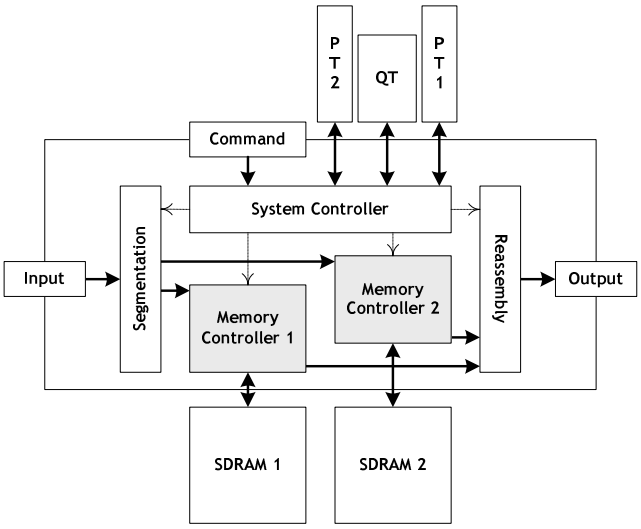

Figure 2. Block diagram of the queue management engine

As well, there is a queue table (QT) which any queue has an entry within it. Each entry of the queue table, keeps two pointers. One of them points to the first respective segment of the queue and the other one points to the last segment.

In our design, we used 128-byte segments and each of two memories is 256 MB. Therefore, each pointer is 21 bits long. Considering one extra bit for end of packet flag, the segment table width is 22 bits.

B. Memory Technology

As noted in [4], bandwidth and capacity of the network buffers grow linearly with the line rate. Therefore, memory devices which provide high capacity and bandwidth are required. From commercially available memories, DRAM devices provide high capacity at a low cost but, suffer from the high random access time. SRAM device are fast, but also expensive and consume a higher power. In our proposed architecture, since large buffering space is required, DRAM is used as the main storage device. However, segment pointers and queue table which are accessed more frequently are implemented using SRAM devices.

We have used DRAM modules with 64-bit wide data bus and have considered their specific timing characteristics [5]. Noteworthy, these characteristics are so general that many DRAM vendors have similar widely available devices. Therefore, we claim that our solution is applicable to a wide range of DRAM devices.

C. Memory Access Scheduling

When using DRAMs as the packet memory, the most important problem which might causes some performance loss is the bank conflict. To cope with the problem of bank conflict, we split a data segment across all banks instead of storing it entirely in a single bank. Using this method, all banks of the DRAM are accessed to transfer a data segment. So, we can guaranty that all or most of the DRAM access latencies are hidden. For example, consider Fig. 4 which

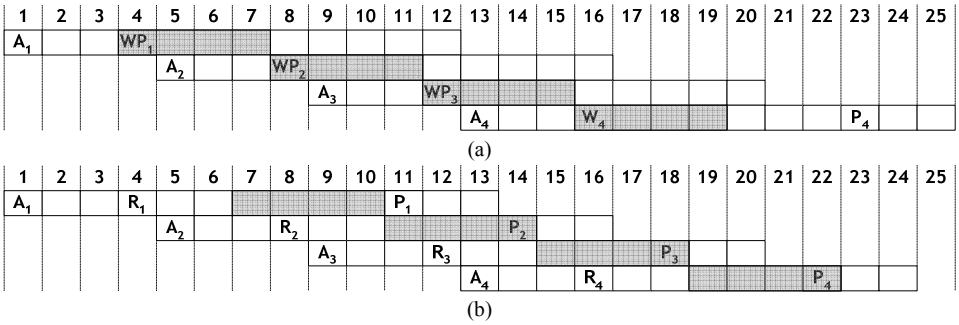

Figure 3. Sequence of commands issued to a DRAM device in our architecture for (a) writing a 128-byte segment, (b) reading a 128-byte segment

shows a sequence of commands issued to the DRAM that will write (read) an entire segment to (from) a single DRAM device with four internal banks. In this figure, the abbreviations A_i, R_i, W_i, P_i, and WP_i respectively correspond to DRAM commands Active, Read, Write, Precharge, and Write with Auto Precharge, all operating on bank i. Data transfers for each bank occur during the shaded clock cycles.

As shown, a 128-byte segment is split into four 32-byte sub-segments, each of which is stored in one of the four banks. For writing (reading) an entire segment to (from) the memory, the shown command sequence is applied to the DRAM. We refer to this sequence of commands as command sequence diagram.

The proposed method has considerable enhancements comparing to the case when we store the whole segment in one bank [6, 7] which will remove bank conflicts efficiently. To see why, consider the case in which a whole segment is written to a single bank and suppose that we want to buffer a group of segments. When different segments are written into the buffer, we can avoid bank conflict completely, since we can choose into which bank each segment is written (using a simple round-robin access method will avoid bank conflicts). The problem arises when the segments leave the memory. Since, the order in which read requests arrive for the segments is not in our control, it is possible that a large number of requests receive for one bank which will imply consequent bank conflicts. At worst, all read requests in a period of time could be made to a single bank which gives the worst performance.

Now, consider the method discussed here. Referring to Fig. 4(a), when writing the first segment into memory, the next write sequence can begin in time slot 17, and the one after 17 slots later and so on. So, the overall bandwidth of data pins is utilized which is the maximum bandwidth that can be exploited from the memory. The same is true when segments are read (Fig. 4(b)). After reading the first segment, the next read sequence can begin on time slot 17 without causing any bank conflicts. Accessing the DRAM in this manner is similar to pipelining techniques used in computer architecture. The access arbitration we shown here fits well in the field of packet buffers since size of the data packet transferred from/to memory is large enough to be split among different banks.

The above discussion is for cases when the size of packets is a multiple of segment size. This will not be the case under real network traffic conditions. When the packet size is not a multiple of segment size, then the last segment of the packet is not completely filled. However, our controller can detect such empty sub-segments and avoid occupation of memory transfer bandwidth for writing/reading empty sub-segments.

We have designed a complete set of command sequence diagrams for all possible combinations of segment utilized portions for read or write commands, which are not shown here. We note that granularity of these utilization diagrams is 32 bytes.

D. Free List Management

One main problem when using linked-list data structure is keeping track of free segments. In our architecture using the described memory access pattern simplifies the task of free list management. Hence, only one free list is required for each DRAM device, implying only two separate lists. In other methods which use a single bank for writing an entire segment, if the free list is not managed efficiently, it may increase the number of bank conflicts [8]. Therefore, those techniques may use up to one free list per bank despite the fact that the operation of allocating and releasing free segments are getting harder.

IV. HARDWARE IMPLEMENTATION AND PERFORMANCE ANALYSIS RESULTS

The described architecture was implemented using VHDL hardware description language by RT Level modeling and the code was synthesized on an Altera Stratix II EP1S10 FPGA device [9]. Table I shows the results of hardware implementation cost in the term of FPGA consumed resources and the operating clock frequency as reported by the synthesis tool.

To measure the maximum achievable bandwidth using our architecture, we performed a set of clock cycle accurate performance analyses for different sizes of transferred data and different sequences of read and write operations. These analyses are performed considering the DRAM timings and the command sequence diagrams previously described. The

TABLE I. HARDWARE IMPLEMENTATION RESULTS OF THE QUEUE MANAGEMENT ENGINE

Hardware Implementation Cost		Clock Frequency
Total Logic Elements	*Total Memory Bits*	
796	80 Kb	170 MHz

TABLE II. PERFORMANCE ANALYSIS RESULTS

Size of the Transferred Data (bytes)	Operations								
	Write			*Read*			*Interleaved Write and Read*		
	Used Clock Cycles	*Unused Clock Cycles*	*Utilization (percent)*	*Used Clock Cycles*	*Unused Clock Cycles*	*Utilization (percent)*	*Used Clock Cycles*	*Unused Clock Cycles*	*Utilization (percent)*
64	8	4	67	8	3	73	16	7	70
96	12	0	100	12	0	100	24	3	88
128	16	0	100	16	0	100	32	0	100
160	20	4	83	20	2	90	40	8	83
192	24	0	100	24	0	100	48	3	94
224	28	0	100	28	0	100	56	0	100

results of our analysis are shown in the Table II.

In this table, the first column shows the size of data transferred from or to the memory in bytes which is a multiple of 32. For each data size, three different operations are analyzed considering when all commands are write (enqueue), when all commands are read (dequeue), and when write and read commands are interleaved. For each of these operations, three values are shown. The first value is the number of clock cycles used for data transfers. The second value shows the number of clock cycles which are empty and no data transfers occur within them. The third value shows the percentage of memory bandwidth utilized. As shown, for these packet sizes, at least 67 percent and in average 91.5 percent of the memory bandwidth is utilized. Based on the information provided in Table II, someone can calculate the utilized memory bandwidth for any other packet sizes.

To have a quantitative measurement, we measure the bandwidth of the proposed architecture when we use two 64-bit 133 MHz SDRAM modules (Based on the delay we obtained by the synthesis, our architecture could handle 133 MHz SDRAM devices). The maximum bandwidth of these memories is about 17 Gbps. Table III shows the performance results of our architecture for packet sizes and memory operations. As seen, for large packet sizes, the efficiency approaches the maximum possible. Also, the average utilized bandwidth for different packet sizes is 14.53 Gbps.

V. CONCLUSION

This paper proposed architecture of a queue management engine which could be employed as an embedded queue manager in networking devices. To design this architecture, several performance optimization methods were used, among which a method for providing a dual-port memory out of single-port devices, and the proposed memory accessed policy. Our performance analysis results showed that from the total memory bandwidth of 17 Gbps, our architecture can utilize an average bandwidth of 14.53 Gbps among different packet sizes.

TABLE III. QUANTITATIVE PERFORMANCE RESULTS

Packet Size (bytes)	Size of Data Actually Transferred (bytes)	Utilized Bandwidth for Write Commands (Gbps)	Utilized Bandwidth for Read Commands (Gbps)
53	64	9.43	10.28
64	64	11.39	12.41
100	128	13.28	13.28
128	128	17	17
500	504	16.87	16.87
900	928	15.99	16.16
1512	1536	16.73	16.73

REFERENCES

[1] B. Suter, T. V. Lakshman, D. Stiliadis, and A. K. Choudhury, "Buffer management schemes for supporting TCP in gigabit routers with per-flow queueing," IEEE Jour. in Selected Areas in Communications, 1999.

[2] M. Alisafaee, Sh. Ataee, S. M. Fakhraie, "Bandwidth-enhanced waste-free control technique for multi-queue network buffers," 3rd Int. Symp. Telecommunications, in press.

[3] M. Alisafaee, Z. Navabi and S. M. Fakhraie, "Doubling memory bandwidth for single-queue network buffers using uniform distribution methods," 10th Conf. Computer Society of Iran, Tehran, Iran, pp. 172-179, 2005.

[4] S. Iyer, R. R. Kompella, and N. McKeown, "Analysis of a memory architecture for fast packet buffers," Proc. IEEE Workshop on High Performance Switching and Routing, Dallas, Texas, 2001.

[5] Micron. http://www.micron.com/sdram

[6] A. Nikologiannis, I. Papaefstathiou, G.Kornaros, C. Kachris, "An FPGA-based queue management system for high speed networking devices," Elsevier Jour. Microprocessors and Microsystems, Vol. 28, Issues 5-6 , pp. 223-236, 2004.

[7] S. Rixner, W. Dally, U. Kapasi, P. Mattson, J. Owens, "Memory access scheduling," Int. Symp. on Computer Architecture (ISCA), Vancouver, Canada, pp. 128-138, 2000.

[8] Ch. YkmanCouvreur, J. Lambrecht, D. Verkest, F. Catthoor, A. Nikologiannis, "System-level performance optimization of the data queueing memory management in high-speed network processors," Proc. 39th Conf. on Design automation, New Orleans, USA, pp. 518-523, 2002.

[9] Altera. http://www.altera.com/