EFFICIENT PRIORITY-QUEUE DATA STRUCTURE FOR HARDWARE IMPLEMENTATION

Andrew Morton, Jeffrey Liu

Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
email: arrmorton,jc2liu@uwaterloo.ca

Insop Song

Dalsa, Inc.
Waterloo, Canada
email: insop.song@gmail.com

ABSTRACT

Priority queues are data structures that maintain a list of data sorted first by priority and second by order of insertion (first in first out). These data structures are used in network routers to schedule outgoing packets from streams requiring various quality of service. Priority queues have also been used in hybrid operating systems that employ hardware to accelerate task scheduling.

A novel hardware data structure is proposed here to implement priority queues. By using a system of indices, the need for comparators is eliminated, reducing size and improving performance. After describing the hardware architecture, synthesis results for realistic size systems are presented. The results are promising, improving on other recent studies.

1. INTRODUCTION

A priority queue is a data structure that consists of multiple queues, each with a priority. Data is inserted into the appropriate queue, based on its priority. Data is dequeued (extracted) from the highest priority queue that is non-empty. It is a data structure that can be used in software, for example discrete event simulation, but also has application in hardware.

Network routers use priority queues implemented in hardware to sort outgoing packets. Each packet is assigned a priority based on its quality of service (QoS) requirement. The priority queue ensures that the packets with highest QoS are sent first (and in FIFO order for that priority level).

Embedded systems also make use of priority queues implemented in hardware. For example, the HybridThreads project [1] implements a fixed-priority scheduler in hardware. It schedules software tasks and coordinates resource sharing between software and hardware threads of execution. The scheduler uses a hardware priority queue. Hardware scheduling of software tasks offers: 1) fast scheduling time, 2) reduced CPU load from periodic clock tick, and 3) low jitter. The third benefit can be very significant in control applications.

A review of hardware architectures for priority queues is presented in Section 2. This is followed by a description of the new architecture in Section 3 and the hardware implementation in Section 4. Finally the implementation is analyzed in Section 5 and conclusions drawn in Section 6.

Equipment support from Canadian Microelectronics Corporation.

2. RELEVANT WORK

Moon *et al* [2], identify four main types of hardware architectures for priority queues: binary tree, multi-FIFO, shift register and systolic array.

In the binary tree structure, there is an n entry storage block feeding n leaf nodes in the tree. The tree is of depth $\log_2 n$ and has $2n - 1$ nodes. Each node consists of a comparator and multiplexer. It can support multiple data sets by connecting different storage blocks. The binary tree extracts the storage entry with the highest/lowest priority. However it does not maintain FIFO order among entries of the same priority. It also has the disadvantage that the dequeue time increases with $\log_2 n$.

The multi-FIFO structure implements one FIFO (queue) per priority. Entries are enqueued in the appropriate FIFO. For dequeue operations, a priority encoder identifies the highest priority non-empty FIFO. The enqueue operations are fast but the time for the dequeue operation is dependent on the priority encoder which is affected by the number of priorities. As well as slowing the priority encoder, increasing the number of priorities P also requires more FIFOs. A solution to this size problem is to use logically linked lists, as is done by [3]. Agron [3] uses two memories, one is a list of head and tail pointers for each priority queue and the second memory contains the entries (task data) which have pointers to the next and previous entries in their linked lists. This solution still has the problem of the priority encoder time overhead.

The shift register solution uses an array of blocks capable of shifting their contents to their left or right neighbour. On an enqueue, all blocks compare the new value with their

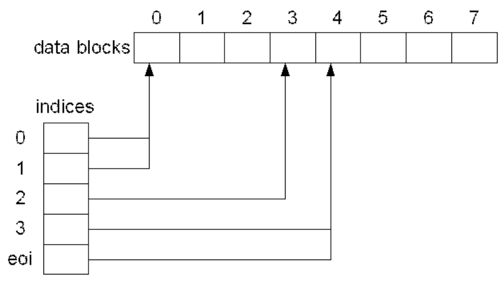

Fig. 1. mQ Example

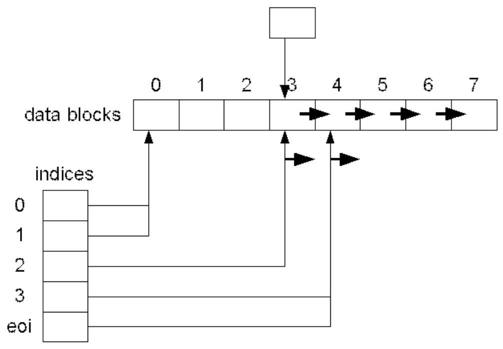

Fig. 2. Insert into Queue 1

value. One block will accept the new value and shift its contents to its neighbour block. All blocks containing lower priority entries will shift their values, making room for the new block. A dequeue just requires reading from the block at the high-priority end of the array and all the other blocks shifting their values one position so that the highest priority block position remains filled. Each block requires a register, comparator, multiplexer and control logic. There is also the issue of bus loading as all blocks will read the new value so that they can compare it with their contents. Shift register arrays are used for scheduling in [4].

Systolic arrays are similar to the shift register implementation except that new values are inserted at the high-priority end of the array. On each clock cycle the value is shifted one position until it comes to rest in the correct position in the array. Each block requires two registers, a comparator, two multiplexers and control logic. Despite the fact that it can take several cycles for a low priority entry to arrive in the correct position, the block at the high-priority end of the array always contains the entry of highest priority. This structure is somewhat faster than the shift register array since there is not the same bus loading issue.

Of these four data types, all have enqueue/dequeue times independent of n except the binary tree with is $O(\log n)$. Of the remaining three which all scale well with n, the multi-FIFO dequeue time which is dependent on the priority encoder is dependent on P. This leaves the shift register and systolic arrays to scale well to large n and P. The shift register array is more compact than the systolic array but it's maximum clock frequency is lower due to bus loading.

3. ARCHITECTURE

The new priority queue architecture proposed here is called an "mQ" since it implements multiple queues in a single array. As with the shift register and systolic array, an array of n blocks is used to contain a maximum n entries. The is logically sub-divided into multiple queues and a flexible

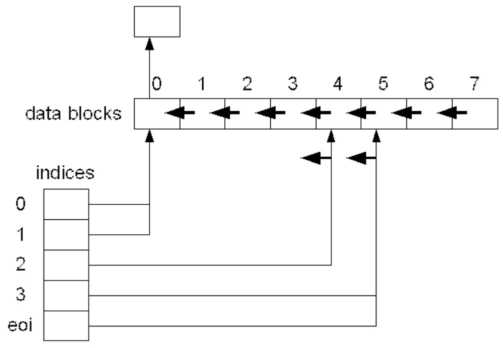

Fig. 3. Extract from Queue 1

system of indices is used to manage their locations within the array. The example in Figure 1 is used to describe how it works.

In Figure 1, the array at the top holds the data. Each index $0, \ldots, 3$ points to the start of a queue. Queue 0 has highest priority. Indices 0 and 1 point to the same element (0), indicating that Queue 0 is empty. Queue 1 has three elements $(0, 1, 2)$. Queue 2 has one element (3). Indices 3 and eoi (end of indices) point to the same element indicating that Queue 3 is also empty. In this case the first four elements in the array hold meaningful data and the last four are considered empty (or more accurately, undefined). The start of each queue is determined by its index value and the end of each queue is determined by the value of the next index.

To insert into a queue, a value is added at the end. For example to insert into Queue 1 (Figure 2), the data is inserted in element 3 of the array (pointed to by Index 2). All array elements from position 3 to 7 shift to the right and Indices 2 to eoi increment by one.

To extract from a queue (Figure 3), values are removed at the head, causing following array elements to shift left and following indices to decrement by one.

The mQ architecture implements a priority queue with-

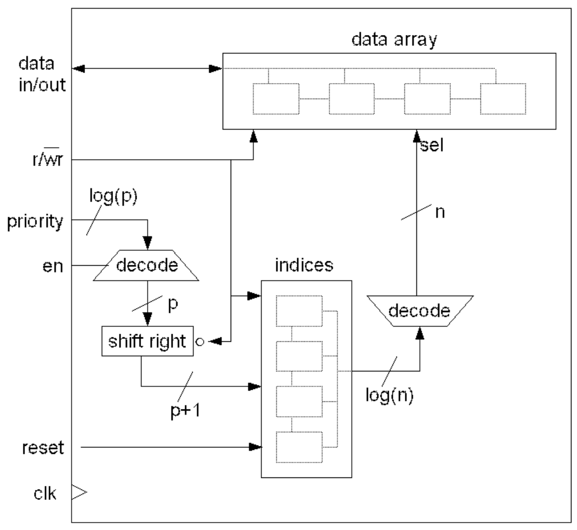

Fig. 4. mQ Implementation

out using comparators. Instead the indices point to the queues. This offers size and speed advantages over other implementations that require a comparator for each element of the array. Extracting from the highest priority queue is done by dequeuing at Queue 0. Due to the indexing scheme, if Queue 0 is empty, this operation will dequeue from the first non-empty queue. Hence no priority encoder is required for dequeue operations.

4. HARDWARE IMPLEMENTATION

A block diagram of the mQ implementation is presented in Figure 4. The hardware implementation consists primarily of an array of blocks holding data and an array of indices that point into the data array.

The detail of a block of the array is shown in Figure 5. When an block is selected for a read, it outputs its data on the *data out* lines and reads in data from its neighbour on the right. It also asserts *shift left out* which is propagated to all neighbours to the right. These neighbour blocks will shift their data to their left neighbours. When an block is selected for a write, it inputs the *data* on the *data in* lines. It also asserts *shift right out* which propagates to all neighbours to the right. These neighbours will shift their data to their right neighbour.

The diagram of an index block is shown in Figure 6. When an index is selected for a read, it outputs its position which is used to select the correct data array element. Since the right side neighbours of the element will shift left, the index asserts *count down out* which is propagated to indices below the index. If a write is performed, the index of the next queue is selected. It outputs its position and increments

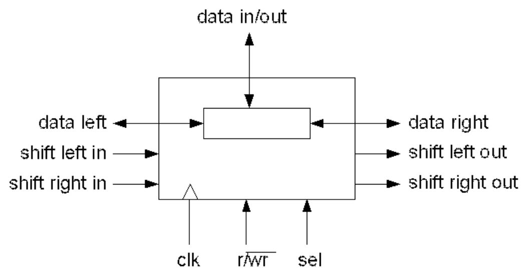

Fig. 5. array block

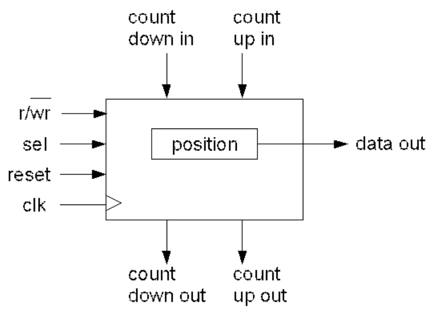

Fig. 6. index block

its value. It also asserts *count up out* which is propagated to indices below the index.

5. ANALYSIS

The mQ was synthesized for the Xilinx Virtex II Pro 70, speed grade -6. The number of elements in the data array was varied over $[16, 32, 64, 256]$ and the number of queue (indices) was varied over $[4, 16, 32, 64]$. The results are reported in Table 1. Number of FPGA slices is reported in Table 1(a) and minimum clock period in nanoseconds is reported in Table 1(b). There are 4 slices in each CLB on the Virtex II Pro. The minimum clock period is reported instead of the maximum clock frequency to get meaningful linear regression results. The sizes reported (215 – 7032 slices) represent between 0.65% – 21.25% of the targeted FPGA's logic resources. The periods reported (8.684 – 50.32 ns) represent maximum clock frequencies between 115MHz – 20MHz.

These results can be compared against those in [3, 2]. Agron [3] implemented multiple logically linked lists with a priority encoder for dequeue operations. Agron reports a maximum clock frequency of 143.8 MHz on a Xilinx Virtex II Pro 30, smaller but equal FPGA. This is for 256 items and 128 priority levels. The design uses 1034 slices and 2 block

Table 1. Synthesis Results

(a) FPGA Slices (4 Slices = 1 CLB)

	Array Size				
Queues	16	32	64	256	R^2
4	215	498	1194	5748	0.999
16	341	669	1257	5806	0.999
32		858	1500	5797	
64			1751	7032	
R^2			0.978	0.828	

(b) Minimum Clock Period (ns)

	Array Size				
Queues	16	32	64	256	R^2
4	8.684	11.37	17.68	33.69	0.974
16	11.38	14.01	19.18	40.36	0.995
32		14.14	18.34	38.90	
64			22.28	50.32	
R^2			0.816	0.888	

RAMs. A dequeue operation takes 24 clock cycles, allowing a maximum of 5.99 million dequeue operations per second. Synthesizing the mQ data for 256 items and 128 priority levels results in a maximum frequency of 19.95 million operations per second (enqueue/dequeue operations are single cycle), and 7334 slices. This is more than three times as fast but significantly larger than [3]. The speed increase is likely because the priority encoding problem is eliminated with the mQ. The linked list data structure of [3] however is smaller since it does not require the shifting mechanism associated with the mQ.

In [2], a systolic array was synthesized using a 120nm process. For a configuration with 256 entries and 16 priorities, $\sim$ 500000 transistors and 50 million operations per second was reported. The reported gate equivalent for the 256/16 mQ was 95602 gates. Assuming 4 transistors per gate gives $\sim$ 380000 transistors for the equivalent mQ design. This mQ design could perform 24.8 million operations per second. The case study in [5] found that ASICs provide 2-3x performance improvements over FPGAs given the same fabrication process. Considering that the 120nm process is close to the 130nm process used for the Virtex II Pro, this indicates that the mQ performance is probably similar to or better than the systolic array, while using less hardware.

In summary, the mQ outperforms the linked list with priority encoder of [3] while requiring more hardware, and has possibly better performance than the systolic array [2] while using less hardware. The performance advantage over the linked lists comes from not requiring a priority encoder: the dequeue operation uses Index 0 (highest priority). The size advantage over the systolic array comes from not requiring any comparators. In fact, a more significant difference is expected and this result needs to be investigated.

Linear regression analyses were performed on the synthesis results in Table 1. R^2 values are only reported where there were four points to analyze. From the regression analysis results on FPGA slices, it appears that there is a strong linear relationship between array size and number of slices. The relationship does not appear to be as strong for number of indices. These results indicate that the mQ scales well with n. From the regression analysis on minimum period, it also appears that there is a strong linear relationship between array size and minimum period. This is not as expected. The only signal propagations in the data array that are related to n are the *shift left* and *shift right* signals. It was attempted to overcome this by adding hierarchical look-ahead logic, similar to but simpler than carry look-ahead propagation in adders. This issue merits further investigation.

6. CONCLUSION

The mQ hardware architecture for priority queues has been described. It is a hybrid between an array of logically linked lists and shift register array structures. Instead of pointers, indices are used that can increment/decrement in parallel. This avoids the need for comparators in the array elements and the bus loading problem. Synthesis results compare favourably with other types of data structures in terms of speed and size. The reason for the linear relationship between n and minimum clock period needs to be investigated, as it is not expected. The mQ architecture is currently being integrated with a real-time operating system, to provide scheduling services.

7. REFERENCES

[1] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden, "Programming models for hybrid FPGA-CPU computational components: A missing link," *IEEE Micro*, vol. 24, no. 4, pp. 42–53, 2004.

[2] S.-W. Moon, J. Rexford, and K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Transactions on Computers*, vol. 49, no. 11, pp. 1215–1227, 2000.

[3] J. Agron, "Run-time scheduling support for hybrid CPU/FPGA SoCs," Master's thesis, University of Kansas, 2006.

[4] P. Kuacharoen, M. Shalan, and V. Mooney III, "A configurable hardware scheduler for real-time systems," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2003, pp. 96–101.

[5] D. Wentzlaff and A. Agarwal, "A quantitative comparison of reconfigurable, tiled, and conventional architectures on bit-level computation," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, pp. 289–290.