Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches

Sung-Whan Moon, *Student Member, IEEE*, Jennifer Rexford, *Member, IEEE*, and Kang G. Shin, *Fellow, IEEE*

Abstract - With effective packet-scheduling mechanisms, modern integrated networks can support the diverse quality-of-service requirements of emerging applications. However, arbitrating between a large number of small packets on a high-speed link requires an efficient hardware implementation of a priority queue. To highlight the challenges of building scalable priority queue architectures, this paper includes a detailed comparison of four existing approaches: a binary tree of comparators, priority encoder with multiple first-in-first-out lists, shift register, and systolic array. Based on these comparison results, we propose two new architectures that scale to the large number of packets (N) and large number of priority levels (P) necessary in modern switch designs. The first architecture combines the faster clock speed of a systolic array with the lower memory requirements of a shift register, resulting in a hybrid design; a tunable parameter allows switch designers to carefully balance the trade-off between bus loading and chip area. We then extend this architecture to serve multiple output ports in a shared-memory switch. This significantly decreases complexity over the traditional approach of dedicating a separate priority queue to each outgoing link. Using the Verilog hardware description language and the Epoch silicon compiler, we have designed and simulated these two new architectures, as well as the four existing approaches. The simulation experiments compare the designs across a range of priority queue sizes and performance metrics, including enqueue/dequeue speed, chip area, and number of transistors.

Index Terms - priority queue, packet switch, link scheduling, VLSI, real-time communications.

1 INTRODUCTION

Applications with real-time traffic, such as video and audio, need more than just good average performance from the network. Such real-time communication [1][13] requires quality-of-service (QoS) guarantees, such as bounded end-to-end delay, bounded cell-loss rates, and guaranteed bandwidth from the network. Emerging packet-switched networks employ a variety of methods to provide the QoS guarantees for each connection. At each node of the network, an admission control algorithm grants a request for a new connection when the QoS requirements can be met. Once established, traffic shaping and link scheduling algorithms [1][13][14] ensure that the QoS requirements are satisfied for all of the connections that pass through the node. Traffic shapers monitor and control connections so that they abide by their connection traffic parameters (e.g., maximum packet rate). Link schedulers coordinate the transmission of packets between several connections on a given link. Since a link can only send one packet at a time, other packets trying to use that link must be queued. The link scheduler typically assigns some priority number to each packet (or group of packets) in the queue to determine which one gets access to the link once it becomes available.

The simplest link-scheduling algorithm is first-in-first-out (FIFO). The problem with this approach is that it is characterized by poor utilization of resources and poor performance. In particular, a FIFO scheduler cannot admit many new connections, especially when the link services connections with a wide range of traffic parameters and QoS requirements. Other link-scheduling algorithms achieve better performance by assigning a priority number to connections or individual packets. This priority field can represent a traffic class, a deadline, a virtual finishing time, or a sequence number, depending on the link-scheduling algorithm. Once the priority number is determined, a priority queue ranks packets based on the priority assignment. The net effect of the link-scheduling algorithm and the priority queue is to interleave the packet transmission from the various connections such that each connection's QoS requirements are satisfied.

The priority queue is essential in implementing the link-

S.-W. Moon and K. G. Shin are with Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122. E-mail: {swmoon, kgshin}@eecs.umich.edu

Jennifer Rexford is with AT&T Labs Research, 180 Park Avenue, Room A169, Florham Park, NJ 07932. E-mail: jrex@research.att.com

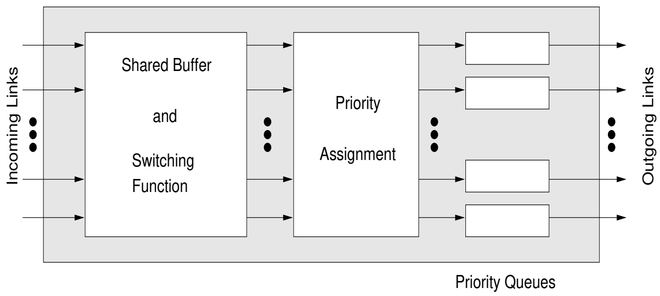

Fig. 1. Simplified block diagram of a single shared buffer switch architecture with link scheduling

scheduling algorithm. Due to the high-speed at which the networks operate, a hardware priority queue [10] is needed to transmit packets at link rates. For example, in a 155 Mbps (2.5 Gbps) Asynchronous Transfer Mode (ATM) network, an ATM cell can be transmitted every 2.7 µsecs (0.17 µsecs). In a worst-case scenario the priority queue must determine the next highest priority cell (dequeue operation) every 2.7 µsecs (0.17 µsecs), while being able to accept new cells (enqueue operation) from all incoming links within the same 2.7 µsecs (0.17 µsecs). Software solutions, which are logarithmic in time complexity, are typically not fast enough to keep up with the packet transmission rate due to the associated overhead (i.e., in requesting service from the processor, sending and retrieving data from the processor). On the other hand, a hardware solution can operate close to the operating speeds of the link. Also, a hardware solution can overlap enqueue and dequeue operations with packet transmission to avoid wasting link bandwidth.

Each node in the network provides a switching function by forwarding incoming packets to their correct outgoing links. For all priority queue architectures discussed in this paper, we consider a common switch model, as shown in Figure 1. The switch is characterized by a shared buffer space and output queueing [5], with a separate priority queue servicing each output link. Although there are other possible memory configurations [11], output buffering offers better performance than input buffering while a shared buffer configuration has better memory utilization. When a packet's output link is busy, the packet is queued and an entry for that packet is created. The entry is inserted into the priority queue corresponding to the correct output link. This entry consists of a valid/invalid bit, an address ($\log_2 N$ bits), and a priority ($\log_2 P$ bits). Here N is the total storage capacity of the shared buffer measured in number of packets, while P is the number of priority levels supported in the link-scheduling algorithm. The address can be interpreted as a page number, indicating where the packet resides in the shared memory. The page numbers are obtained from an idle address pool, which holds page numbers corresponding to idle spaces in the memory. Each arriving packet obtains a page number before being written into the shared memory; the page number is then returned to the pool after the corresponding packet has been transmitted. The priority queue is responsible for storing the entries and calculating the highest-priority entry when the output link is ready to transmit another packet. So, regardless of internal architecture the priority queue must provide for the storage of packet tags, initialization (clear contents of the priority queue), enqueue of new tags, and dequeue of the highest priority tag.

Since a switch's buffer size (N) and the number of priority levels (P) needed by the link scheduler can be both very large, the priority queue must be easily scalable to these two parameters. That is, the total entry capacity of the priority queue must match the total packet capacity of the shared buffer, and it must support a large number of priority levels. At the same time, the priority queue's performance must not fall behind link rates as it is scaled to N and P. If this were to happen, then a link will remain idle even though there are packets to be transmitted. Finally, the priority queue design should scale well with the number of output ports in the shared-memory switch, instead of requiring completely separate logic for each outgoing link, as in existing architectures.

We present two new priority queue architectures which were designed to minimize the effects of scaling (with respect to N and P). The first new architecture reduces and controls the performance loss due to increasing the queue capacity without adding a large amount of extra hardware. This was done by combining the salient features of two existing priority queue architectures, the shift register [3][4][12] and systolic array [6][7]. We then extend this architecture to service multiple links instead of just one. Both of the new architectures perform well enough to support very high-speed links, and both provide constant-time (in terms of number of clock cycles) enqueue and dequeue operations. But before describing our new architectures, we first describe four priority queue architectures – binary tree, FIFO, shift register, systolic array – from the current literature in Section 2. A brief description of each architecture and operation is given, followed by a discussion on limitations to their scalability. Two new architectures are then proposed and evaluated in Section 3 (hybrid systolic/shift) and in Section 4 (multiple link). Each of these two sections also gives a detailed explanation of the new architecture's

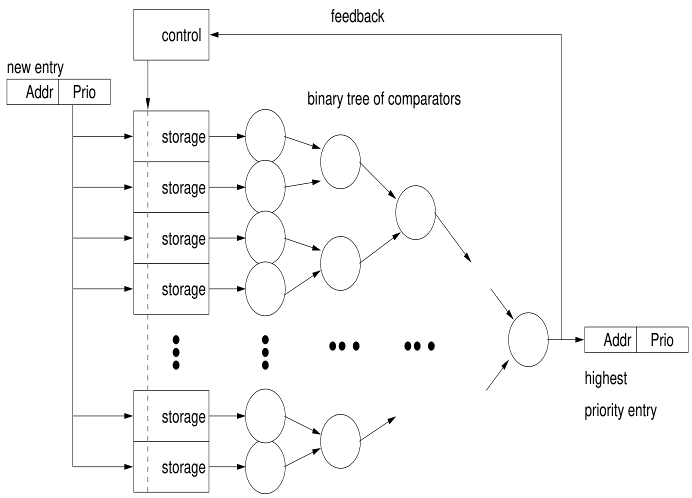

Fig. 2. Binary tree of comparators priority queue.

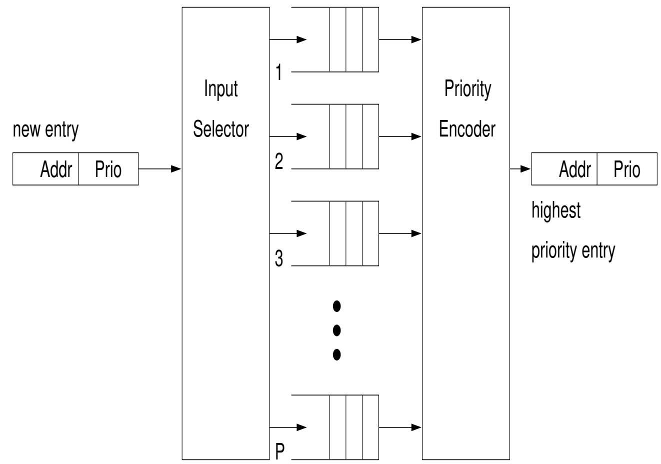

Fig. 3. FIFO priority queue.

operations. Section 5 presents the results of some implementations of the various priority queues for several switch parameters. The implementations were done using the Verilog hardware description language and the Epoch silicon compiler for several combinations of P (up to 256) and N (up to 1024). These results show limitations of the existing architectures when scaled to large N and P, and are compared to implementations of the new architectures. Section 6 concludes with a summary of our contributions and a brief list of future directions.

2 PRIORITY QUEUE ARCHITECTURES

This section presents four priority queue (PQ) architectures from the current literature. The FIFO and the binary tree architectures are the more intuitive approaches to the priority queue problem. However, these two architectures do not scale well with increasing N and P. The shift register and the systolic array architectures take a different approach and scale much better than the FIFO and binary tree. The following subsections describe each of these architectures and discuss the effects of scaling on architectural complexity and implementation.

2.1 Binary Tree of Comparators

A binary tree comparator architecture [8][9], shown in Figure 2, consists of an N-entry storage block and a comparator tree of depth $\log_2 N$, whose output is the highest-priority entry among those in storage. A feedback mechanism is used to remove the output of the tree from storage. An advantage of this architecture is that the comparator tree logic can be shared among several storage blocks, reducing hardware costs. A disadvantage is that FIFO ordering is not maintained among entries with the same priority. Such FIFO ordering is important when applications assume that packets at the same priority level will arrive in the same order in which they were sent. Increasing N results in more leaf nodes (i.e., comparators) being added to the tree and increasing the capacity of the storage block. Problems with such scaling include bus loading problems with distributing the new entry to each storage element in the storage block, and increased dequeue time resulting from an increase in depth ($\log_2 N$) of the comparator tree. A possible solution to the increased dequeue time is to pipeline the comparator tree operation to reduce the clock period and increase performance; this can be useful if the comparator tree is shared among several outgoing links [9]. Another solution is to initiate the comparator tree only after dequeue operations and use extra logic to handle entries that arrive during and in between dequeue operations. This takes advantage of the fact that packet transmission time is longer than the comparator tree operation [8].

2.2 FIFO Priority

Like the bucket sorting algorithm, the FIFO PQ architecture [2][3], shown in Figure 3, inserts entries into one of the P FIFOs based on the entry's priority. Since each FIFO corresponds to a particular priority level, the queue does not need to store a priority field with each entry. During a dequeue operation, a priority encoder scans the head of the FIFOs in decreasing priority order and removes an entry from the first non-empty FIFO. Increasing P requires adding more FIFOs, which results in added hardware costs and increased complexity of the priority encoder. Using logi-

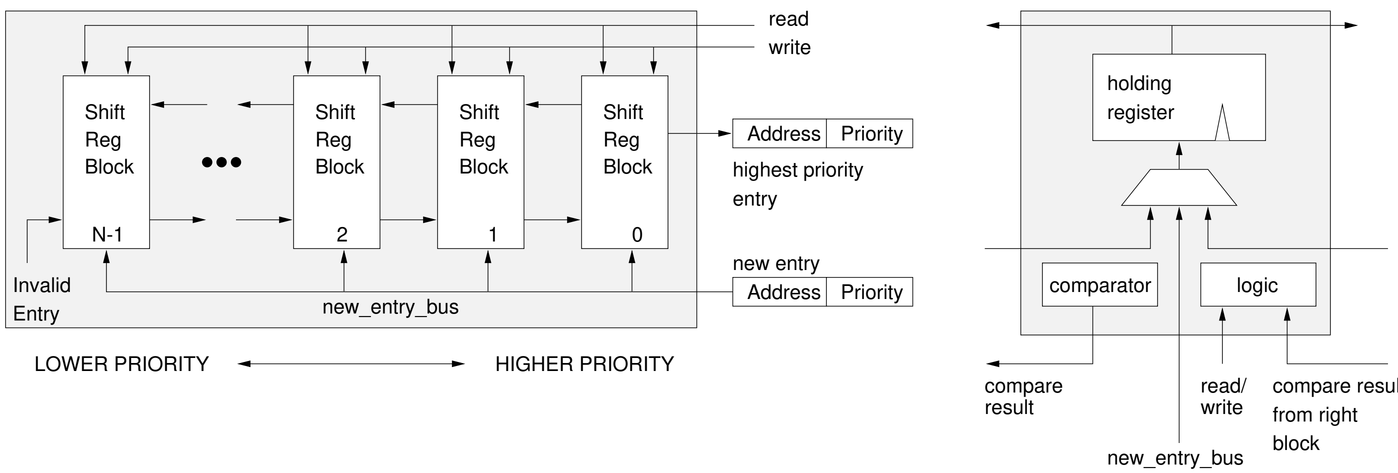

Fig. 4. Shift register priority queue and shift register block

cally linked lists [3][15] instead of physical FIFOs can reduce hardware costs. But this approach still suffers from the complexity problem of the priority encoder for large P.

2.3 Shift Register

The shift register PQ [3][4][12], as shown in Figure 4, consists of an array of blocks that store the entries in sorted order. Each block stores a single entry and communicates with the blocks immediately to its right and left. Higher-priority entries are stored to the right of lower-priority entries, with the 0^{th} block containing the current highest-priority entry. On an enqueue operation, the new entry is broadcast to all the blocks via the new_entry_bus. Each block makes a local decision as to what action to take, with only one of the blocks latching the new entry. The others will either keep their current entry or latch the right neighbor's entry. The net effect is to have the new entry force all entries with lower priority to shift one block to the left, while the new entry places itself to the left of the entries with higher and equal priority. The lowest priority entry is discarded during an enqueue if the queue is full. A dequeue operation in the shift register simply reads the 0^{th} block's entry while all other entries shift one block to the right.

As shown in Figure 4, each block consists of a holding register which stores the entry, a comparator which compares the priorities of the entry on the new_entry_bus and the holding register, a multiplexor (to choose from the left, right or new entry) and decision logic [3][4]. Since each block stores one entry, the queue's capacity can be increased by adding more blocks to the existing queue. Because each block makes decisions based on just local information, increasing queue capacity does not require modifications to the block's decision logic nor any central control logic for the queue. This makes scaling for large N very simple. As P increases, additional bits are added to the priority field in the entry's tag. This simply requires modifying each block's storage requirement and its comparator.

Unfortunately, implementation problems limit the scalability of this architecture. As seen in Figure 4, before any decision can be made by each block during an enqueue operation, the new entry must be present at the inputs of all the blocks. At the VLSI level, the new_entry_bus must be routed to the inputs of all the blocks in the array. As we saw with the binary tree architecture, this creates a bus loading problem, which adds to the hardware costs (buffers), and decreases the maximum operating speed of the queue. Thus, the shift register architecture's scalability with respect to N is limited by performance, not by architectural complexity. Performance also decreases as P increases due to the added delay in the comparator logic. This is because the comparator's time complexity grows linearly (for a serial comparator) with the number of bits in the priority field.

2.4 Systolic Array

The systolic array PQ [6][7] is shown in Figure 5. Similar to the shift register architecture, the systolic array architecture consists of an array of identical blocks, with each block holding a single entry. On an enqueue operation, only the 0^{th} block compares priorities of its entry and that of the new entry. On the next cycle the lower-priority entry is inserted into the left neighbor's block which repeats the same process of comparing and sending the lower-priority entry to the next block. So the systolic array does not

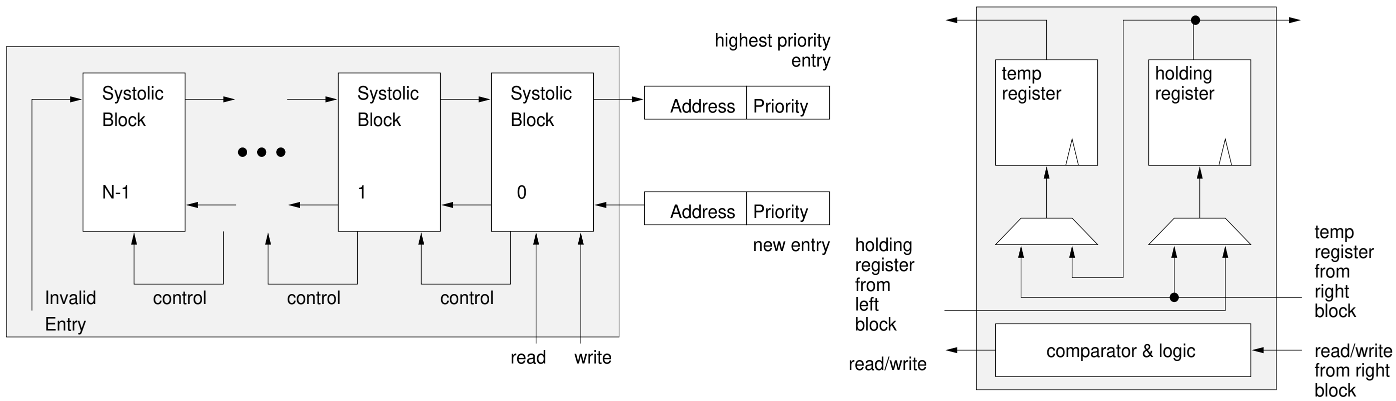

Fig. 5. Systolic array priority queue and systolic array block

become fully sorted until several cycles after the new insertion. Despite this feature, both insertion and removal still remain constant-time operations from the outgoing link's point of view. Because each block passes the lower-priority entry to the next block, the 0^{th} block always holds the highest-priority entry in the queue. Once an entry is removed from a block, it gets the entry from its left neighboring block, creating a right shift operation on the entire queue.

Each systolic array block consists of a holding register, which stores the entries in sorted order, as well as a temporary register, that holds passing entries enroute to the next block to the left. The passing entry is the lower-priority entry in a block during an enqueue operation. Multiplexors, a comparator, and decision logic also make up the rest of the block. A block diagram is shown in Figure 5. Queue capacity is increased by adding more blocks to the end of the queue without worrying about a central controller. Also, there is no bus loading problem as was the case with the shift register PQ. Increasing P requires extra storage and a wider comparator, as in the shift register priority queue. Unfortunately, the one main drawback is that the systolic array PQ requires twice more storage than the shift register architecture. Considering the simplicity of each block, the temporary register adds a considerable hardware cost to each block, compared to the shift register block. Also, the cost and delay of the comparator increases linearly with each extra bit in the priority field, which decreases the maximum operating clock frequency.

3 MODIFIED SYSTOLIC ARRAY PRIORITY QUEUE

This section presents a new priority-queue architecture that combines the salient features of the systolic array and the shift register. The architecture has a tunable parameter which enables us to balance the trade-off between bus loading and hardware costs. We then extend this hybrid architecture to guarantee FIFO transmission of packets within the same priority level.

3.1 Hybrid Shift/Systolic

Of the four PQ architectures discussed in Section 2, the shift register architecture and the systolic array architecture have better properties in terms of supporting very large N and P. The FIFO architecture is limited to a small number of priority levels, while the binary tree comparator's complexity makes it difficult to scale with increasing N. On the other hand, the shift register and systolic array are more favorable because they have no centralized logic, and each block can be replicated as many times as necessary without any modifications. Also, a large number of priority levels can be easily supported by simply using more bits in the priority encoding. Unfortunately, the shift register's bus loading problem limits the maximum clock frequency, while the systolic array block's double storage requirement makes it considerably more hardware-intensive than the shift register.

The systolic array architecture scales well with N and its maximum operating clock frequency does not decrease as N increases. But, because 50% of all the registers are used as temporary registers, the systolic array uses much more hardware than the shift register. To reduce this overhead, we propose a modified systolic architecture where each block consists of a length **c** shift register. So instead of one temporary register for every holding register in each block, the ratio decreases by a factor of 1/c. Also, because scaling the modified systolic architecture for larger N does not

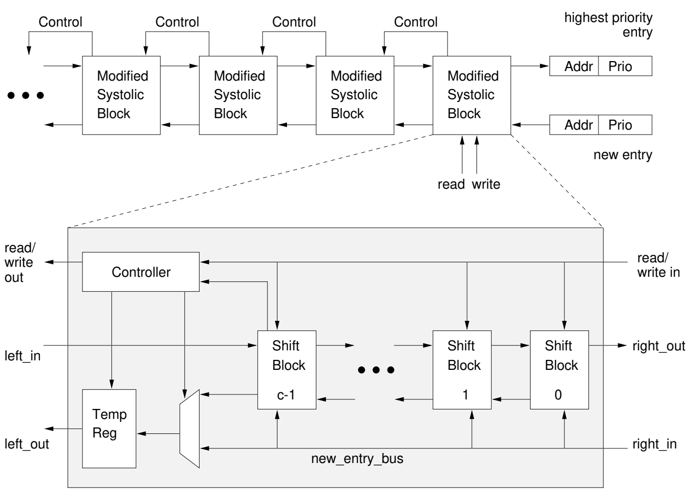

Fig. 6. Modified systolic array priority queue

involve changing c, the bus loading problem associated with the shift register stays constants as N grows.

Each modified systolic block holds **c** entries by replacing the single holding register with a length **c** shift register PQ, as shown in Figure 6. The interface of the modified systolic block is the same as that of the systolic block. Enqueue and dequeue requests are received from the right neighboring block and the results of those requests are sent to the right neighboring block. The right-most block receives requests and sends results to the link. During a new entry insertion into the modified systolic block, the new entry is placed in one of the blocks of the shift register PQ. If there is an overflow of the shift register PQ, either the new entry or the entry in the c^{th} shift block (whichever has lower priority) is placed into the temporary register and inserted into the left neighboring modified systolic block during the next cycle. Since the shift register PQ stores all the entries in sorted order with the highest-priority entry in the first block, the removal request is satisfied by moving all the entries one block to the right. The entry in the right-most block is sent to the neighboring right modified systolic block. During the next cycle, a removal request is made to the neighboring left modified block and the resulting entry is stored in the shift register PQ.

3.2 FIFO Ordering

Without any further modifications, the modified systolic array PQ will not maintain FIFO ordering among entries of equal priority, as illustrated in Figure 7. Here the number represents the priority and the subindex (not part of the entry) represents the arrival order among entries with the same priority. Insertion of a new entry with priority 9

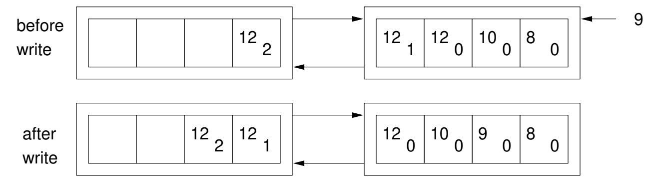

Fig. 7. Data movement in the modified systolic array PQ showing a potential ordering problem

pushes the 12_1 entry to the next modified systolic block and is placed behind the 12_2 entry. The problem here is that the second shift block cannot determine if the 12_1 entry corresponds to a new entry (which should go after 12_2) or an old entry (which should stay ahead of 12_2). This is solved by adding a one bit field (new/old) to the end (least significant bit) of the priority field and is included as part of the priority number when priority comparisons are done. The new/old bit is added as the entry enters the priority queue, and is stripped off when the entry leaves the queue. New entries that are inserted into the queue have this bit set. Likewise, all entries that are stored in a shift block have this bit set. The bit is cleared when an entry that was already in a shift block is pushed into the temporary register and sent to the neighboring left modified systolic block.

The modified systolic architecture improves on the systolic architecture by lowering the percentage of total registers used for temporary storage. This reduction in hardware is accomplished without losing any of the advantages of the systolic architecture - simple block architecture (easily scaled for increasing N by adding new blocks to the end of the existing queue), no performance loss as more blocks are added to the queue, and constant-time (cycles) enqueue and dequeue operations. Also, because the bus driving the shift register blocks is broken up into small length-c parts, the bus load within each modified systolic block is not affected by the additional modified systolic blocks. So, once a value for c is determined, only one modified systolic block must be designed and optimized for performance and area. This block is then replicated as many times as necessary without any modifications.

4 MULTIPLE OUTPUT LINK PRIORITY QUEUE

To further reduce the hardware complexity of packet switches, we extend the architecture from Section 3 to service multiple output links. For simplicity, we first present a multiple-link priority queue based on the shift-register architecture before generalizing the technique to the hybrid

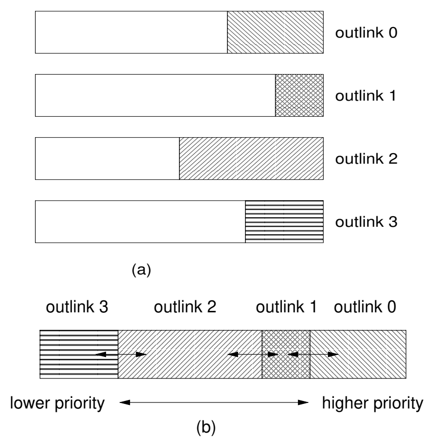

Fig. 8. (a) Sorted entries in separate priority queue showing wasted resources; (b) Same entries in the multiple shift register priority queue

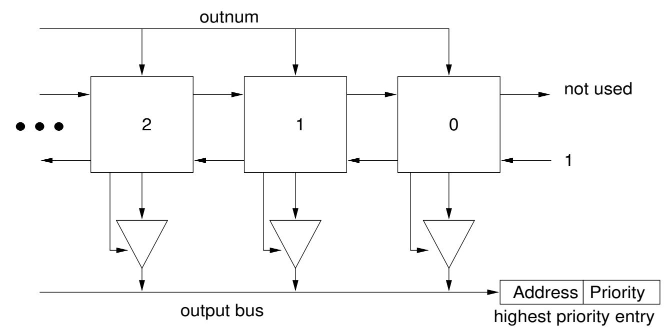

Fig. 9. Multiple shift register priority queue

systolic/shift design. We then discuss how the architecture can provide a constant-time dequeue operation, while still differentiating between packets destined for different output links. To avoid overlapping multiple operations in a single systolic array block, the design includes a small, constant number of "wait states."

4.1 Multiple Shift Register Priority Queue

Given that a switch has a separate priority queue for each of its M (>1) output links, the total queue capacity is MN entries. Since the shared buffer can only hold N packets, most of the blocks in the priority are unused at any given moment, as shown in Figure 8(a). An N-entry priority queue which services M (<N) output links can potentially save a maximum of 50% in hardware for M=2, and up to 75% for M=4. Here we present a multiple output link priority queue architecture, which has good scaling properties and constant-time enqueue and dequeue operations which are independent of M and N.

We first extend the shift register architecture to support multiple links. This requires modifications to the entries and shift register block. The packet's entry is augmented such that the priority field consists of the output link number, priority number, and new/old bit. The shift register stores entries such that those corresponding to higher output link numbers come after those corresponding to lower output link numbers, as shown in Figure 8(b).

The blocks in the shift register architecture require several modifications to support multiple output links. First, each block receives another control signal (outnum) which indicates the requested output link number. The value on outnum is latched along with the new entry during an enqueue operation, while it is used to determine which entry to output during a dequeue operation. Second, each block has a tristate buffer, which drives an output bus. This tristate buffer is needed because the highest-priority entry for a given output link can be in any of the blocks in the shift register. On a dequeue operation, a block will drive the output bus with the value in its holding register if the block decides it has the highest-priority entry for the requested output link. Figure 9 shows the block diagram of the multiple shift register queue with just the added control signals.

Within each multiple shift register block, no extra control logic is required for the enqueue operation. But during the dequeue operation each block needs to decide if it must drive the output bus. As seen from Figure 8(b), the highest-priority entry of any output link is always to the right of all other entries with the same output link number. Once the output bus has been read, all entries to the left of the one just read move one block to the right. The decision-making process for this dequeue operation is shown in Figure 10(a). A similar operation can also remove the lowest-priority entry for an outgoing link, as shown in Figure 10(b). This operation is useful when extending the modified systolic architecture to support multiple outgoing links, as explained in the next subsection.

4.2 Multiple Systolic Array Priority Queue

Due to the bus loading problem in the shift register architecture, the PQ described in Section 4.1 does not scale well with respect to N. Besides the new entry bus, shown in Figure 4, the multiple shift register architecture also has the problem of each shift register block driving the output bus, and the associated delay and hardware costs of having to drive a very large bus. Despite this problem, the multiple

```
if read {
  if ((holding_reg_outnum == outnum) &&
      (rght_blk_holdng_reg_outnum < outnum)) {
    drive output bus;
    load left shift blk's entry on next cycle;
  }
  else if ((holding_reg_outnum >= outnum) &&
            (rght_blk_holdng_reg_outnum >= outnum)) {
    load left shift blk's entry on next cycle;
  }
  else {/*holding_reg_outnum < outnum*/
    do nothing;
  }
}
```

(a) Remove highest-priority entry

```
if read_low {
  if ((holding_reg_outnum == outnum) &&
      (left_blk_holdng_reg_outnum > outnum)) {
    drive output bus;
    load left shift blk's entry on next cycle;
  }
  else if (holding_reg_outnum > outnum) {
    load left shift blk's entry on next cycle;
  }
  else { /*holding_reg_outnum < outnum*/
    do nothing;
  }
}
```

(b) Remove lowest-priority entry

Fig. 10. Pseudo code for read and write operations in multiple shift register block

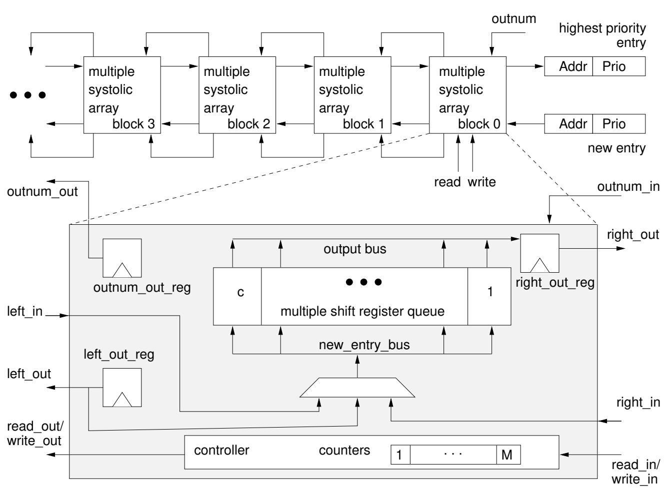

Fig. 11. Multiple systolic array priority queue and block

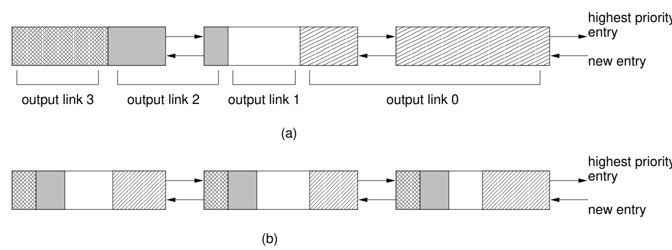

Fig. 12. Storage of entries in the multiple systolic array queue (a) before and (b) after "atleast-1-entry-per-outputlink" property

shift register can be used as a building block to support multiple outgoing links in the modified systolic architecture. By using the same ideas as in Section 3, the multiple systolic array architecture replaces the single holding register with the multiple shift register. By choosing a value for c which minimizes the total number of temporary registers without introducing significant bus loading problems, a single c-entry multiple shift register can be designed and used in the multiple systolic array architecture.

As seen in Figure 11, the external interface to the multiple systolic array block remains the same, with the addition of the outnum control signals. Besides the temporary register (left_out_reg), there is also another register (onum_out_reg) which indicates the link number of the entry in the temporary register. The right out register (right_out_reg) stores the output from the output bus, while a multiplexor chooses among three sources to drive the new entry bus. Also, instead of the read and write control signals directly feeding the shift register, the controller uses them to generate its own internal read and write control signals which are then fed to the shift register.

4.3 Constant-Time Dequeue/Enqueue

Without further modifications to the architecture described in Section 4.2, a situation as shown in Figure 12(a) can occur. If there are more than c entries in the queue for any output link, a dequeue request can result in extra remove requests being sent from the one systolic block to the next systolic block. In the worst case, the requests can propagate to the last block, in which case the result will need to propagate all the way back up. In order to avoid this problem and have a constant-time dequeue operation, each systolic block uses counter to maintain a "atleast-1-entry-per-output-link" property, whenever possible. This assumes that c ≥ M. A counter is used for each output link to keep track of the number of packets queued at that link. The counter is incremented (decremented) whenever an entry corresponding to the counter's output link is inserted (removed) from the systolic block.

After an enqueue operation, entries start to propagate through the array of systolic blocks in the left direction. Writing another entry into a systolic block that is full will result in either the right_in entry or the entry in the shift register queue's c block (which ever has lower priority) to be written into the left_out register. But before the entry in the left_out register is sent to the left systolic block, the controller makes sure that doing so does not violate the "atleast-1-entry-per-output-link" property. If it does, another entry is chosen to be sent to the left systolic block while the entry from the left_out register is reinserted into the shift register queue. Here the other entry that is chosen is the lowest-priority entry corresponding to an output link with more than one entry in the systolic block. The controller obtains this replacement entry by checking all the counters, and then issues a read_low command to the shift register queue. Also, following a dequeue operation, the 0th systolic block requests an entry with the same output link number from the 1st systolic block which, after sending the result, requests an entry with the same output link number from the 2nd systolic block, and so on. This is done to maintain the property for all systolic blocks. Details of these systolic block operations are shown in Figure 13. Note that the state transitions are not shown there.

4.4 Non-Overlapping Operations

Since the systolic block must finish one request before processing another request, wait states are needed to prevent overlapping of operations. The insertion operation takes 5 clock cycles, while the remove operation takes 4 clock cycles (these are independent of c and M). Without any wait states, a block can make an insertion request to its left neighbor on the 5th cycle while servicing an insertion request. Similarly, a remove request can be made to the left block on the 2nd cycle while servicing a remove request. Doing this will result in the overlapping of request service, as shown in Figure 14. This overlap is avoided by delaying the remove request till the 5th cycle, instead of the 2nd

```
counter[outnum_in}++;
left_out=lower priority between right_in and
        shift-register-queue[c];
outnum_out=outnum of lower priority between
        right_in and shit-register-queue[c];
if (left_out is valid) do
  if (counter[outnum_dout]==1) do
    var outnum_temp = outnum such that
        counter[outnum_temp] > 1;
    right_out=lowest priority entry in shift
            register queue with outnum_temp;
    insert left_out into shift register queue;
    left_out=right_out;
    outnum_out=outnum of entry in right_out;
    invalidate right_out;
    coiunter{outnum_out}--;
    write left_out and outnum_out into left
        systolic block;
  enddo
  else do
    counter[out_out]--;
    write left_out and outnum_out into left
      systolic block;
  enddo
enddo
```

(a) write

```
if (counter[outnum_in]==0) do
  invalidate right_out;
enddo
else do
  counter[outnum_in]--;
  right_out=highest priority entry in shift
     register queue with outnum_in;
  outnum_out=outnum_in;
  request read from left systolic block;
  if (left_in is valid) do
    insert left_in into shift register queue;
    counter[outnum_in]++;
  enddo
enddo
```

(b) read

Fig. 13. Pseudo code for write and read operations in systolic block

cycle. Since 2 cycles are need to get the result, a total of 7 cycles are needed for the remove request. Although the insert operation still takes just 5 cycles, consecutive

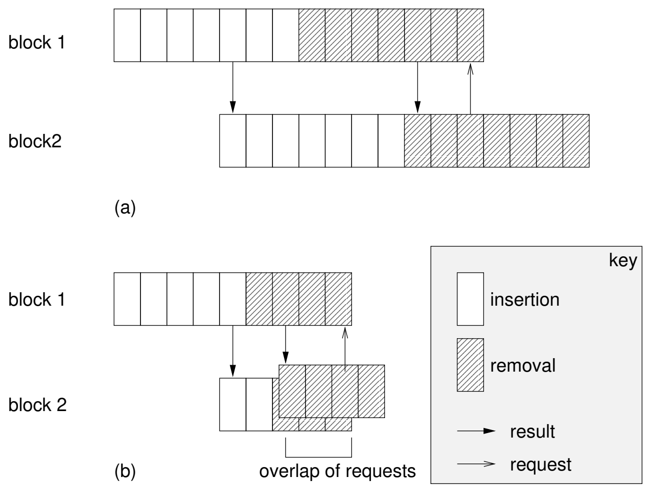

Fig. 14. Time line of operation for multiple systolic block (a) with wait states (b) without wait states

requests can only be made every 7 cycles to the multiple systolic block.

Despite the added complexity of the state machine and extra hardware needed to support multiple output links, the multiple systolic array is still much cheaper to implement than individual priority queues for each output link. Also, the time (cycles) required to service the dequeue and enqueue operations is constant for any output link, and remains unchanged regardless of how large N becomes. Like the modified systolic architecture, each block is self-contained and no outside controller is required. As N increases, more blocks are added to the existing chain without modifications to the existing blocks. Also, since the priority number is encoded within each entry, a large number of priority levels can be supported without requiring a large amount of hardware. Thus, scaling does not involve modifying the architecture, implementation for large N is simplified since only one systolic block needs to be designed, and there is no loss in performance due to scaling.

5 PERFORMANCE AND IMPLEMENTATION

To compare the various priority queue architectures discussed thus far, each architecture was implemented using the Verilog hardware description language and the Epoch silicon compiler, an automatic layout generator. This provides a common framework which makes the cost and performance comparisons more meaningful. The implementation results showed that both new architectures had better scaling properties in terms of performance and hardware costs than the four existing architectures. Also, the multiple-link architecture was shown to scale well with M, provide good performance, and offered considerable hardware savings in comparison to using a priority queue per-link approach.

5.1 Evaluation Methodology

Costs were measured in terms of amount of silicon area and the number of transistors used by the design, while performance was measured by the maximum clock speed and throughput (number of enqueue/dequeue operations completed per second). Throughput can be easily calculated by using the maximum clock speed and number of clock cycles needed by each operation. Maximum clock speed was calculated by doing a critical path analysis of the design, and determining the delay through these critical paths using Epoch's timing analyzer. All designs were structurally specified using parts from Epoch's Verilog library, while state machines and control logic were described in behavioral Verilog. Each of the layouts was compiled by Epoch, which uses standard cells to generate a layout, using a 1.2 µm CMOS technology. Although custom layout would give better results, we are more interested in comparing the scaling effects than in raw numbers. In other words, we want to look at the relative costs and performance of the various architectures as N and P increase. Also, note that our implementations were limited to a maximum of 1024 for N. This was due to insufficient workstation memory for performing the various simulations.

One final note regarding the implementations concerns the use of registers for storage of priority queue tags. Although other storage devices could have been used in the designs, we chose to use registers because they allowed us to quickly implement, scale and compare the various designs. Since the main goal of this paper was to study the relative scaling effects of the various designs, other storage alternatives were not studied. For example, in the shift and systolic architectures, the single holding register per block can be replaced with a SRAM module with the capacity to hold several priority queue tags. This will reduce hardware costs since the comparison and other logic is shared among several tags instead of one. But doing so increases the time required for dequeue and enqueue operations since these need to be serialized due to accesses to SRAM. So, choosing the size of the SRAM becomes a problem of sacrificing performance for smaller hardware costs. Although this paper does not consider the trade-off between serial SRAMs and parallel registers, we have evaluated the associated cost and performance implications in the priority queue architectures in [9].

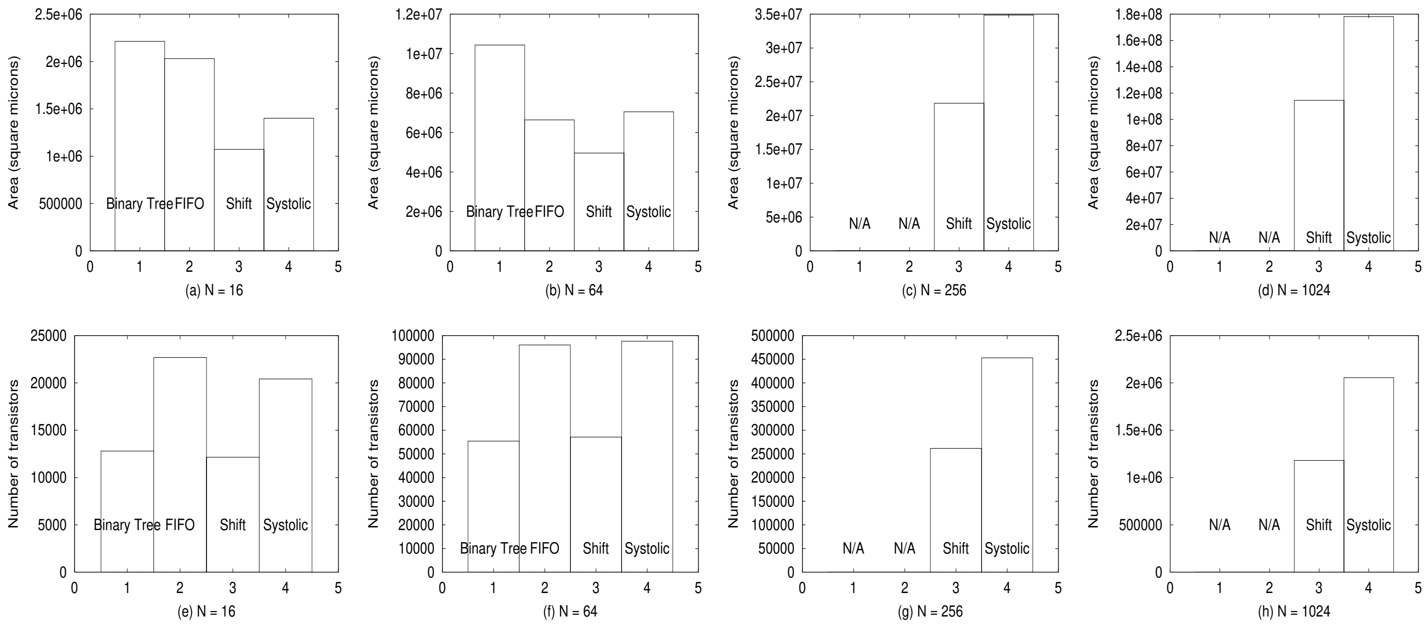

Fig. 15. Implementation results for existing priority queue architectures (P=16)

5.2 Existing Architectures

Figure 15 compares the four existing priority queue architectures in terms of VLSI hardware costs as a function of N, with P fixed at 16. Here we chose a small value of P for two reasons. It allowed for implementations with large N, and made the scaling effects associated with large N more pronounced. As expected, we see the systolic array architecture's hardware cost is much larger than that of the shift register due to the extra register used for temporary storage. Also, despite having similar transistor counts, the binary tree architecture occupies more area than the shift register architecture. This is mainly because of the routing required from the storage to the priority comparator tree, and routing within the comparator tree.

As expected, performance degrades with increasing N, as shown in Figure 16. Here we see the throughput is highest for the shift register architecture. But as N increases, the performance degradation is much steeper for the shift and binary tree architectures than that of the systolic and FIFO architectures. This is due to the bus loading problem in the shift register and binary tree architecture, and the increase in depth of the comparator tree in the binary tree architecture. The gradual decrease in performance in the systolic and FIFO architectures can be attributed mainly to the extra bits in the registers and multiplexors, which add delay to the control signals which must drive these components.

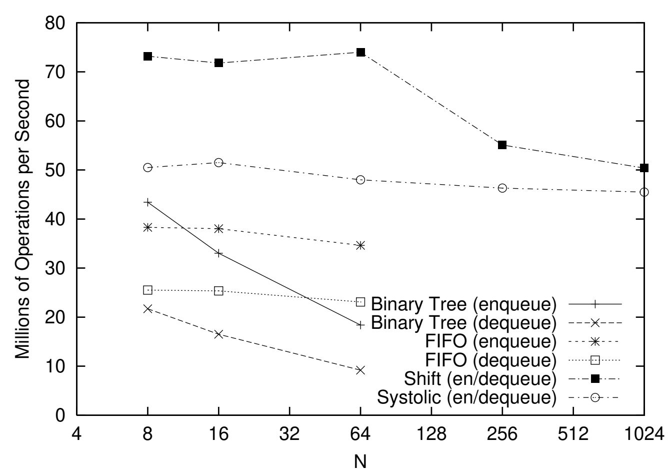

Fig. 16. Scaling effects on performance as N increases (P=16)

Although Figure 16 shows the shift register architecture with better throughput than the systolic array architecture, for larger values of N, we can predict the throughput of the systolic to be higher than the shift. Due to insufficient workstation memory we could not obtain data for larger values of N other than the ones shown in Figure 16. But by extrapolating the curves for the shift and systolic in Figure 16, we can see the two curves should cross at a point somewhere between N=1024 and N=2048. At this point,

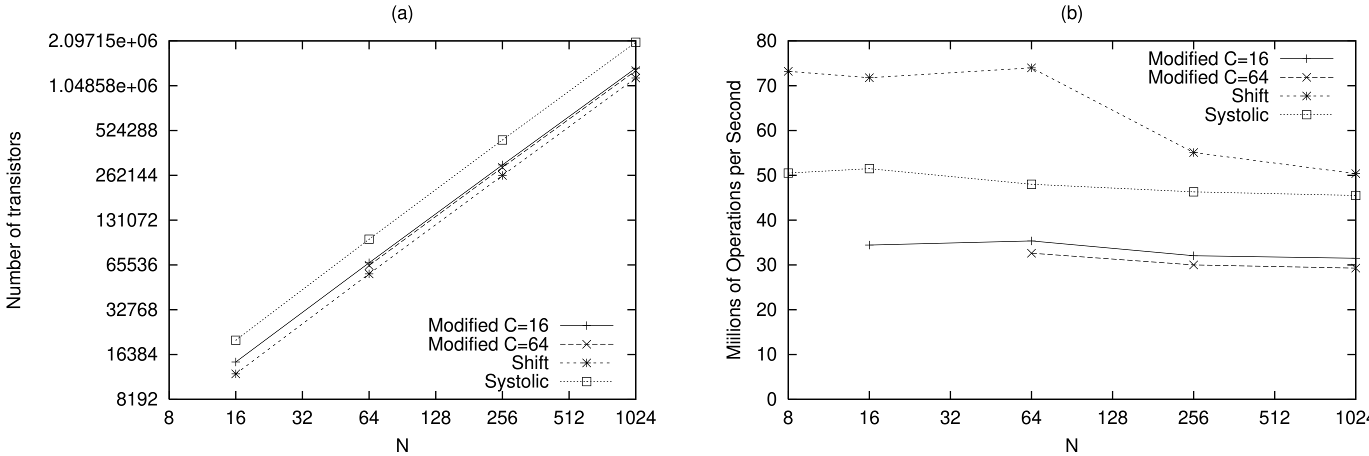

Fig. 18. Modified systolic architecture results compared to that of the shift and systolic (P=16)

throughput of the systolic should be higher, while the performance of the shift architecture should continue to degrade at a much faster rate than that of the systolic due to the dominating effect of the bus loading problem. For much larger values of N, this bus problem should make the shift register architecture an ineffective solution due to the associated hardware costs and performance loss.

Each bit added to the priority field adds delay to the priority comparator, which in turn slows down the operation of the priority queue for the shift register, systolic array, and binary tree architectures. Since a large number of priority levels can be supported with relatively few bits, and because the delay associated with the extra bit is small compared to the total delay, scaling for large P is feasible and the resulting implementations can be effective. With a non-pipelined binary tree though, the delay is multiplied by the depth of the tree. In the FIFO case, the bottleneck is in the priority encoder, which must scan each FIFO to select the next highest priority entry. This can be seen in Figure 17. Note also that the depth of the physical FIFO (due to increasing N) does not affect performance, but adds to the FIFO fall-through time. So, it is possible that an entry might not be available immediately after it is inserted into the queue. The logical FIFO architecture avoids this problem by using linked lists instead.

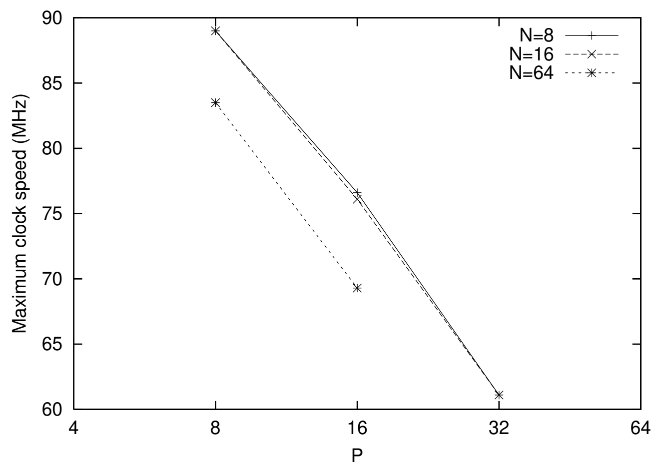

Fig. 17. Scaling effects on FIFO architecture as P increases.

5.3 Modified Systolic Array Architecture

The motivation for the modified systolic array architecture was to take advantage of the shift register and systolic array architecture's features and, at the same time, reduce the negative side-effects due to scaling with respect to N. The shift register architecture suffered from the bus loading problem, while the systolic array architecture used a significant amount of extra hardware for the extra register. The solution that was proposed was to use a separate shift register queue inside each systolic array block. Each shift register queue stores c entries, where c is determined by hardware and performance requirements. When c=1, this is the same as the original systolic architecture, whereas if c=N, then we get the original shift register queue. So for small c, hardware costs and performance are close to those of the systolic architecture, and as c increases the hardware costs steadily approach those of the shift register architecture. Also, as c increases, the performance of the modified decreases due to extra bus loading, as is the case with the shift architecture. This point is shown in Figure 18. Here

P=16, and two values of c are used. Initially for small values of N, the shift architecture has the best performance, mainly because of negligible bus loading and also because the operations in the shift architecture require one cycle, as opposed to two in the systolic and modified. But as N increases, the rate at which performance decreases is much sharper in the shift register case due to the bus loading problem. With the systolic and modified systolic, the performance curve is relatively horizontal. For larger N, performance for the modified systolic should be higher than that of the shift register due to this very gradual decrease in performance in the modified systolic architecture. Considering the amount of hardware used in the modified systolic is only slightly more than that of the shift, and considering the aggressive buffering strategies required in the shift architecture to get these performance numbers, the modified systolic is a much more effective solution despite the performance difference. So once a value for c is determined based on the performance requirements, the design can be scaled to very large N, without worrying about severe performance degradation from bus loading, or buffering strategies.

5.4 Multiple Systolic Array Architecture

Despite added hardware costs (due to extra registers, added complexity of control logic, tristate buffers, and counters), we see that there is still a substantial amount of hardware saved by using the multiple queue. Based on implementations with a 16-entry multiple systolic array block, we observed the following. For M=4, the multiple architecture occupied 32% less area and used 55% less transistors versus the shift, and 46% less area and 72% less transistors versus the systolic. For M=8, the multiple architecture occupied 67% less area and used 75% less transistors versus the shift, and 73% less area and 85% less transistors versus the systolic. Here we multiplied the costs for a single shift or systolic queue by M to account for one queue per output link. Some results are shown in Figure 19. We also observed that adding support for more output links in the multiple systolic block increased the costs only slightly. This is because most of the multiple link support already exists, and all that is needed are extra counters and minor additions in the controller. This can be seen by the extra line inside the bars for the multiple systolic architecture in the M=8 graphs. The top line indicates the value for M=8, while the lower line shows the value for M=4. As seen, the difference in the two lines is very small indicating a small increase in cost for extra output link support.

For c=16, N=64, and P=256, the maximum clock speeds for the multiple systolic architecture are 40 MHz (M=4) and 38 MHz (M=8). This drop in speed is due to the extra bits in the priority field used to encode the output link number. Considering each enqueue and dequeue operation requires 7 cycles, this translates into 5.71 mops (millions of operations per second) for M=4, and 5.43 mops for M=8. If we consider each switch as having M inputs and M outputs, with all input and output links getting round-robin access to the queue, the queue can support link speeds up to 303 Mbps for M=4, and 144 Mbps for M=8 (assuming 53 byte packets). At current ATM standards of 155 Mbps, a multiple systolic priority queue can be designed and implemented to support such switches. For switches with a larger number of links, by grouping 4 to 8 outgoing links together, hardware costs can still be significantly reduced while being able to support very high-speed links.

6 CONCLUSION

In this paper we proposed and evaluated two new hardware priority queue architectures for link scheduling in high-speed switches. Based on Verilog and Epoch designs and simulations, we showed that the four existing architectures were limited by scalability (with respect to either N or P or both). For small N and P, all four existing architectures had comparable hardware costs and performance. But as they were scaled to support large N and P, each architecture's limitations became more pronounced. Of the four architectures, the shift register architecture and the systolic array architecture had better scalability. By combining the two architectures, the modified systolic architecture reduced the negative effects of scaling suffered by the two architectures. In particular, hardware costs were significantly reduced by decreasing the number of total temporary storage registers; performance loss due to the bus loading problem in the shift register could be controlled and isolated from N by using several length-c shift register queues. Here c was chosen by considering hardware and performance requirements. The multiple systolic architecture added multiple link support to the modified systolic architecture, without sacrificing scalability. Although extra cycles were added to the dequeue and enqueue operations, both these operations could be done in constant time (cycles), regardless of N or M, the number of output links supported by the architecture. We have also observed that scaling with respect to M was possible with very little added hardware. Verilog and Epoch simulations have confirmed the salient features of the new architectures.

We showed that the two new hardware priority queue architectures scale well to increasing N and P. Both offer

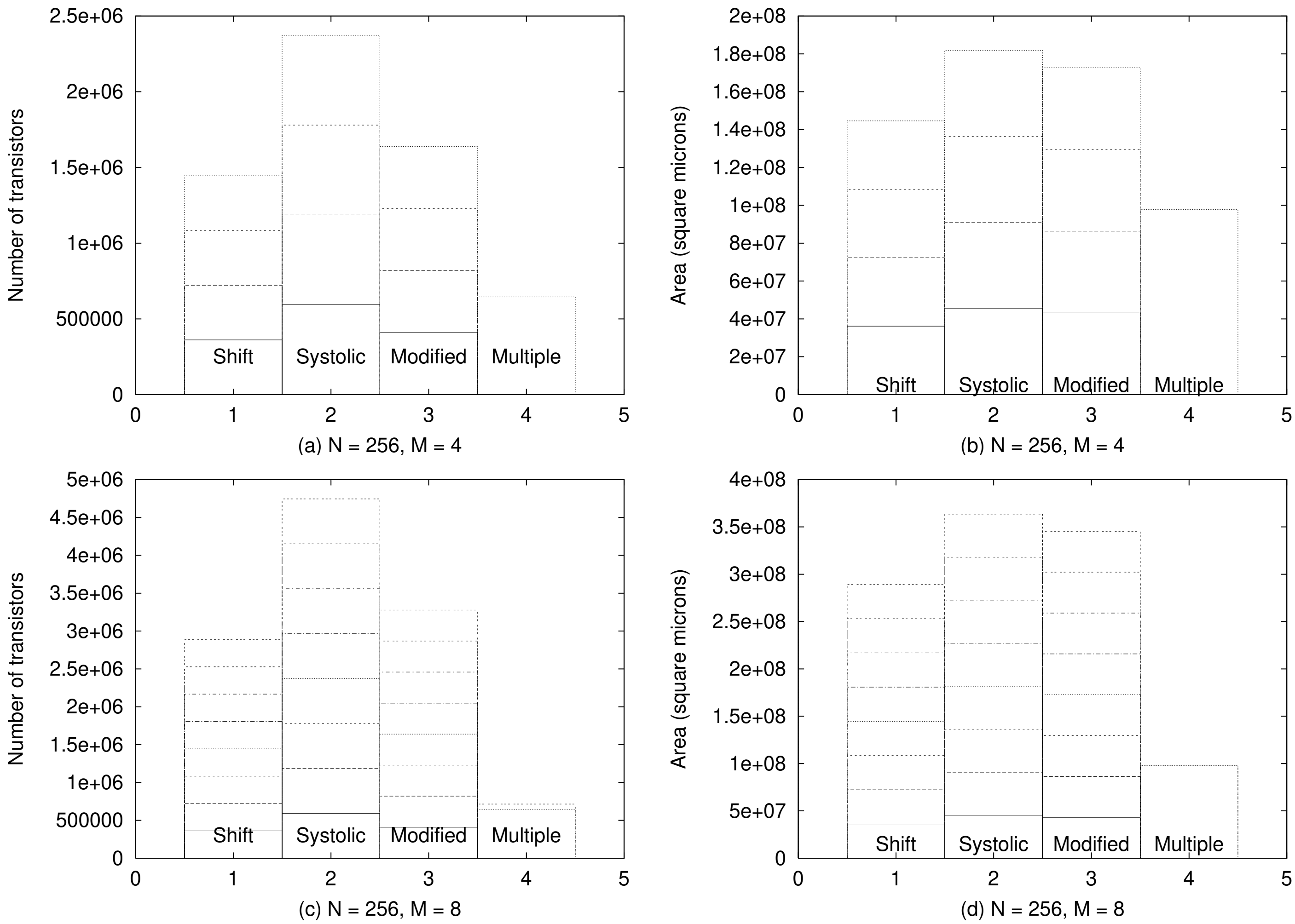

Fig. 19. Implementation comparison with multiple systolic architecture (P=256)

good performance and are easy to implement, and hence can be used in guaranteeing QoS requirements in high-speed networks. Such effective priority queue implementations allow switches to use more aggressive link-scheduling algorithms that can admit more connections with diverse traffic patterns and QoS requirements. A possible future investigation in this area can involve implementing various link scheduling algorithms and using the hardware priority queue to compare implementation complexity and performance. Also, since all the priority queue architectures have a common interface, this facilitates their use in other applications which require priority queueing. It would be interesting to look at the priority queue in such applications as a linear-time sorting engine, or even task scheduling in a uniprocessor or multiprocessor environment.

ACKNOWLEDGEMENTS

The work reported in this paper was supported in part by the National Science Foundation under Grant MIP-9203895. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the NSF.

REFERENCES

[1] C. M. Aras, J. F. Kurose, D. S. Reeves, and Henning Schulzrinne, "Real-Time Communication in Packet-Switched Networks," *Proceedings of IEEE*, vol. 82, no. 1, pp. 122-139, Jan 1994.

[2] R. Brown, "Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem," *Communications of the ACM*, vol. 31, no. 10, pp. 1220-1227, Oct 1988.

[3] J. Chao, "A Novel Architecture for Queue Management in the ATM Network," *IEEE Journal on Selected Areas in Communications*," vol. 9, no. 7, pp. 1110-1118, Sept 1991.

[4] J. Chao, and N. Uzun, "A VLSI Sequencer Chip for ATM Traffic

Shaper and Queue Management," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 11, pp. 1634-1643, Nov 1992.

[5] M. G. Hluchyj and M. J. Karol, "Queueing in High-Performance Packet Switching," *IEEE Journal on Selected Areas in Communications*, vol. 6, no 9, pp. 1587-1597, Dec 1988.

[6] P. Lavoie, and Y. Savaria, "A Systolic Architecture for Fast Stack Sequential Decoders," *IEEE Transactions on Communications*, vol. 42, no. 2/3/4, pp. 324-334, Feb/Mar/April 1994.

[7] C. E. Leiserson, "Systolic Priority Queues," *Caltech Conference on VLSI*, pp. 200-214, Jan 1979.

[8] D. Picker and R. Fellman, "A VLSI Priority Packet Queue with Inheritance and Overwrite," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 3, no. 2, pp. 245-252, Jun 1995.

[9] J. Rexford, J. Hall, and K. G. Shin, "A Router Architecture for Real-Time Point-to-Point Networks," *IEEE Transactions on Computers*, vol. 47, no. 10, pp. 1088-1101, October 1998.

[10] J. Rexford, A. G. Greenberg, and F. G. Bonomi, "Hardware-Efficient Fair Queueing Architectures for High-Speed Networks," *Proceedings of IEEE INFOCOM*, pp. 638-646, Mar 1996.

[11] F. A. Tobagi, "Fast Packet Switch Architectures for Broadband Integrated Services Digital Network," *Proceedings of the IEEE*, vol. 78, no. 1, pp 133-167, Jan 1990.

[12] K. Toda, K. Nishida, E. Takahashi, N. Michell, and Y. Yamaguchi, "Design and Implementation of a Priority Forwarding Router Chip for Real-Time Interconnection Networks," *International Journal of Mini and Microcomputers*, vol. 17, no. 1, pp 42-51, 1995.

[13] D. Towsley, "Providing Quality of Service in Packet Switched Networks," *Performance Evaluation of Computer and Communication Systems*, L. Donatiello and R. Nelson, editors, pp. 560-586, Springer Verlag, 1993.

[14] H. Zhang, "Service Disciplines For Guaranteed Performance Service in Packet-Switching Networks," *Proceedings of IEEE*, vol. 83, no. 10, pp 1374-1396, Oct 1995.

[15] H. Zhang and D. Ferrari, "Rate-Controlled Service Disciplines," *Journal of High Speed Networks*, vol. 3, no. 4, pp. 389-412, 1994.

Sung-Whan Moon received his BS degree in electrical engineering from Cornell University, Ithaca, New York in 1994 and the MS degree in Computer Science and Engineering from the University of Michigan, Ann Arbor, Michigan in 1996. He is currently working towards his PhD degree at the University of Michigan.

Jennifer Rexford (jrex@research.att.com) received her BSE degree in electrical engineering from Princeton University in 1991, and her MSE and PhD degrees in computer science and electrical engineering from the University of Michigan in 1993 and 1996, respectively. She is now a member of the Internet and Networking Systems group at AT&T Labs -- Research, in Florham Park, NJ. Her research interests include routing protocols, Internet traffic characterization, and multimedia proxy services.

Kang G. Shin Kang G. Shin is Professor and Founding Director of the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan.

He has supervised the completion of 40 PhD theses, and authored/coauthored over 600 technical papers and numerous book chapters in the areas of distributed real-time computing and control, computer networking, fault-tolerant computing, and intelligent manufacturing. He has co-authored (jointly with C. M. Krishna) a textbook "Real-Time Systems," McGraw Hill, 1997. In 1987, he received the Outstanding IEEE Transactions on Automatic Control Paper Award, and Research Excellence Award in 1989, Outstanding Achievement Award in 1999, and Service Excellence Award in 2000 from The University of Michigan.In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are investigating various issues related to real-time and fault-tolerant computing.

His current research focuses on Quality of Service (QoS) sensitive computing and networking with emphases on timeliness and dependability. He has also been applying the basic research results to telecommunication and multimedia systems, embedded systems, and manufacturing applications.

He received the B.S. degree in Electronics Engineering from Seoul National University, Seoul, Korea in 1970, and both the M.S. and Ph.D degrees in Electrical Engineering from Cornell University, Ithaca, New York in 1976 and 1978, respectively. From 1978 to 1982 he was on the faculty of Rensselaer Polytechnic Institute, Troy, New York. He has held visiting positions at the U.S. Airforce Flight Dynamics Laboratory, AT&T Bell Laboratories, Computer Science Division within the Department of Electrical Engineering and Computer Science at UC Berkeley, and International Computer Science Institute, Berkeley, CA, IBM T. J. Watson Research Center, and Software Engineering Institute at Carnegie Mellon University. He also chaired the Computer Science and Engineering Division, EECS Department, The University of Michigan for three years beginning January 1991.

He is an IEEE fellow and member of the Korean Academy of Engineering, was the General Chair of the 2000 IEEE Real-Time Technlogy and Applications Symposium, the Program Chairman of the 1986 IEEE Real-Time Systems Symposium (RTSS), the General Chairman of the 1987 RTSS, the Guest Editor of the 1987 August special issue of *IEEE Transactions on Computers* on Real-Time Systems, a Program Co-Chair for the 1992 *International Conference on Parallel Processing*, and served numerous technical program committees. He also chaired the IEEE Technical Committee on Real-Time Systems during 1991-93, was a Distinguished Visitor of the Computer Society of the IEEE, an Editor of *IEEE Trans. on Parallel and Distributed Computing*, and an Area Editor of *International Journal of Time-Critical Computing Systems* and *Computer Networks*.