Design of a High-Speed Packet Switch with Fine-Grained Quality-of-Service Guarantees

Ranjita Bhagwan, Bill Lin
Center for Wireless Communications
University of California, San Diego
rbhagwan@ucsd.edu, billlin@cwc.ucsd.edu

Abstract— **We present a new input-queued switch architecture designed to support deadline-ordered scheduling at extremely high-speeds. In particular, deadline-ordered scheduling is enabled through a combination of hardware-based sorted priority queues called P-heaps and a round-robin crossbar scheduler. The priority queues are implemented using a novel scalable pipelined heap-based architecture. Using a 0.35 micron CMOS standard-cell technology, we demonstrate a 32-port switch capable of sustaining 10 Gb/s line rates.**

I. Introduction

Recently, a number of service disciplines have been proposed for providing QoS guarantees in packet-switched networks, including [5], [17], [6], [18], [3], [15], [10], [4]. Many of these methodologies are based on a *deadline-ordered* service discipline. In a deadline-ordered service discipline, packets are assigned transmission deadlines and are transmitted in an *increasing order of deadlines* using a *deadline-ordered scheduling policy*.

In this paper, we present a new *deadline-ordered input-queued crossbar architecture* designed to support deadline-ordered scheduling at extremely high-speeds. The goal is to design a 32-port single-stage crossbar switch capable of sustaining port speeds of 10 Gb/s while at the same time capable of providing fine-grained QoS guarantees. Our current switch design is targeted to support a 32-bit deadline field for ordering the packets, but it is efficiently scalable to shorter or longer deadline fields. The switch is designed to switch fixed-size cells, but variable length packets can be handled by first segmenting them into fixed-size cells upon arrival. Once transferred across the switch, they can be reassembled. While a number of high-performance input-queued crossbar switch designs have been proposed, e.g. [13], [7], [16], [8], they generally do not support fine-grained QoS guarantees.

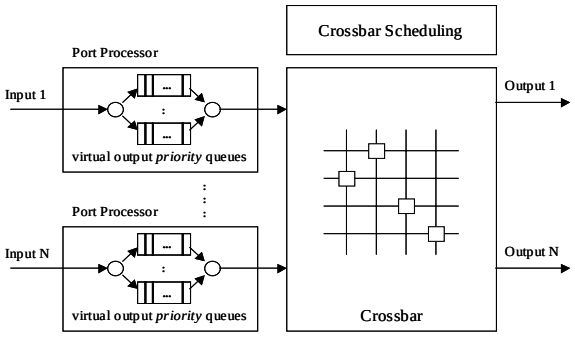

Fig. 1. Deadline-ordered input-queued crossbar architecture.

Figure 1 depicts the top-level view of our input-queued crossbar architecture. We employ a well-known technique called virtual output queueing in which each input maintains a separate queue for each output. This technique is used to avoid the so-called head-of-line blocking problem. However, in contrast to conventional input-queued crossbar switch designs, we employ a novel technique to ensure deadline-ordered scheduling of cells. Instead of conventional first-in-first-out queues, we use *sorted priority queues* to implement the virtual output queues so that the queued cells are ordered *earliest-deadline-first*. In particular, we present an efficient hardware-based, constant time priority queue architecture based on the *P-heap data structure* [9]. We have extended the P-heap data structure to support the efficient implementation of *virtual output queues*. These virtual output queues are referred to as *virtual output priority queues* (VOPQs). When a new cell arrives at an input port, it is stored in the corresponding VOPQ in accordance to its output destination. To achieve high-performance, the crossbar scheduling stage is *pipelined* together with the sorted priority queue operations. That is, the scheduling of current requests is overlapped with the enqueueing of new cells and the dequeueing of last scheduled cells.

II. Scalable P-heap Based Priority Queues

In the literature, several hardware-based sorted priority queue architectures have been described: binary-tree-of-comparators-based priority queues [19], [20], shift-register-based priority queues [1], and systolic-array-based priority queues [11], [12], [14]. All these architectures are generally difficult to scale because the hardware size is dependent on the worst-case queue size: each queue element requires a separate comparator datapath and separate register storage. In addition, the binary-tree-of-comparators-based architectures and the shift-register-based architectures are difficult to scale because they suffer from large bus loading problems: each new entry must be distributed to all storage elements in the priority queue. These scalability problems are particularly acute for input-queued switch architectures where there are N^2 number of virtual output queues. For a 32-port switch, 1024 queues are needed.

In our approach, the sorted priority queues are implemented using a novel *heap-based* pipelined hardware priority queue architecture, which we call the *P-heap*. While software implementations of algorithms based on heaps are well-known [2], they are much too slow for high-performance switches that are intended to operate at multi-gigabit per second port speeds. In contrast to previous hardware-based priority queue architectures that are based on a separate comparator datapath for each separate queue storage element, for all queues, the P-heap processor engine requires a smaller number of comparator datapaths, the number logarithmically scaling with the size of the queue. Also, it provides constant time enqueue and dequeue operations similar to the more expensive solutions.

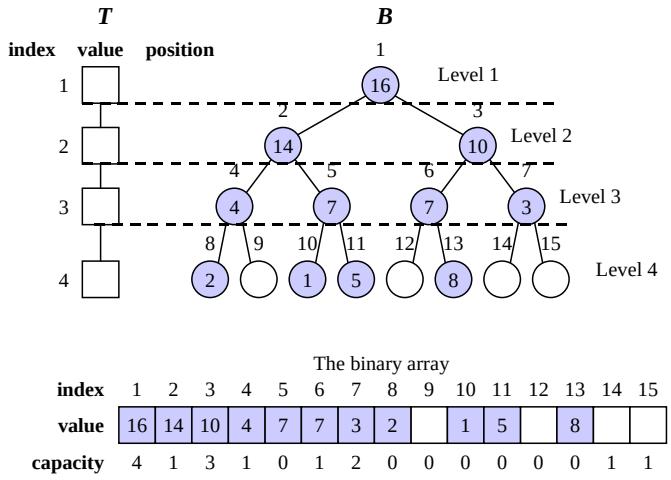

Fig. 2. The P-heap data structure

A. The P-heap Data Structure

The Pipelined heap or the P-heap data structure P can be defined as a tuple $\langle B, T \rangle$ where both B and T are array objects. B, which we refer to as the *P-heap binary array*, is the structure which stores the sorted priority values. It can be viewed as a complete binary tree, as shown in Figure 2. The length of B is given by

$$length(B) = 2^l - 1 \quad l \; \epsilon \; Z^+$$

where l is the *number of levels* in the tree represented by B. This data structure is in fact very similar to the conventional binary heap. The difference lies in the fact that a binary heap's size may vary as long as it stays an *almost complete binary tree*. In contrast, the size of B is fixed.

The root of B is $B[1]$, and given the index i of any node in B, the indices of its parent and children can be determined in the following way:

$$parent(i) = \lfloor i/2 \rfloor$$
$$left(i) = 2i$$
$$right(i) = 2i + 1$$

A node of B, say $B[i]$, contains three fields as given below:
- $B[i].active$: this is a boolean field which is set to `true` if the node $B[i]$ is filled with a valid priority value (the node is $active$). It is set to `false` if the node is empty, or *inactive*.
- $B[i].value$: if the node is active, this field holds the actual priority value.
- $B[i].capacity$: this field contains the number of inactive nodes in the sub-tree rooted at $B[i]$.

In the Figure, the active nodes are shown shaded while the inactive nodes are left unshaded. The following property is satisfied by all nodes in B.

If $B[i]$ is an ancestor of $B[j]$, then

$$B[i].capacity \geq B[j].capacity.$$

A conventional binary heap satisfies the *heap property*, that is, for every node apart from the root, the value of the parent of the node will always be greater than or equal to the value of the node itself. A similar property holds for the binary array of a P-heap which we refer to as the *P-heap Property*. The P-heap property has to be satisfied *by every node* $B[i]$ of B except the leaves. It can be summarized as follows:

P-heap Property : *Let $B[i]$ be a node in B and $B[j]$ be an immediate (left or right) child of $B[i]$. Then,*

1. $B[i].active \wedge B[j].active \Rightarrow B[i].value \geq B[j].value$
2. $B[j].active \Rightarrow B[i].active$

The P-heap property 1 makes sure that the highest priority value is always in the root of B, *i.e.*, in $B[1].value$.

The array object T, called the *token array*, is also shown in Figure 2. The length of the token array is exactly equal to l.

B. Priority Queue Operations on the P-heap

Our purpose is to design a modified heap data structure and associate algorithms with it, which while being simple and easy to implement, can be easily pipelined to provide constant time priority queue operation at low hardware costs. The following algorithms have been designed keeping this in mind.

B.1 The `enqueue` operation

To enqueue a new value into B, we need to find an inactive node in B. We do this by traversing a *valid path* from the root to a leaf of B, where a valid path is defined as follows:

A valid path $B[i_1] \rightarrow B[i_2] \rightarrow \cdots \rightarrow B[i_l]$ where $B[i_j] \; \epsilon \; L_j$ and $i_j = parent(i_{j+1})$, is a path in B where

$$\forall B[i_j], \; B[i_j].capacity > 0.$$

Since the capacities of all the nodes in the valid path are greater than 0, there exists *at least* one inactive node in it. The `enqueue` operation first writes the new value into $T[1]$. It then travels through a valid path, maintaining the P-heap properties throughout till an inactive node is reached and the new value has been successfully accommodated in the queue.

An example of the enqueue operation is shown in Figure 3 and explained below.

Cycle 1: The new value 9 is stored in $T[1]$(Figure 3 (a)). Since 9 is smaller than $B[1].value$,*i.e.* 16, there is no swap. The new value 9 is moved down to $T[2]$. The capacity of the left child, *i.e* $B[2]$ is examined. If this value is non-zero, the left branch of *B* is taken and the next comparisons are made with the left child of $B[1]$. If this capacity is zero, then we assume that the right subtree of $B[1]$ has at least one inactive node and the right branch is taken. In this case, the capacity of $B[2]$, the left child, is found to by equal to 1 since $B[10]$ is inactive. So the left branch is taken.

Cycle 2: The value of $B[2]$ is read and is found to be 14(Figure 3 (b)). $T[2]$, which is 9, is smaller than 14, and so the two values are not swapped. The value 9 is moved down further to $T[3]$. The capacity of the left child, *i.e* $B[4]$ is examined and is found to be 0. This implies that there are no inactive nodes in the sub-tree rooted at $B[4]$. Therefore, the right branch is taken in this cycle.

Cycle 3: The value 9 is compared with $B[5].value$, *i.e.* 7(Figure 3 (c)). Since 9 is larger, the two are swapped. $T[3]$ now holds the value 7 and it is moved down to $T[4]$. The capacity of $B[10]$, the left child of $B[5]$, is examined. It is found to be 1 and so the left branch is taken.

Cycle 4: It is found that $B[10]$ is inactive(Figure 3 (d)), and so the value of $T[4]$ which is 5, is written directly into the node $B[10]$ and its capacity is reduced to 0. The P-heap, at the end of the `enqueue`, is shown in Figure 3 (e).

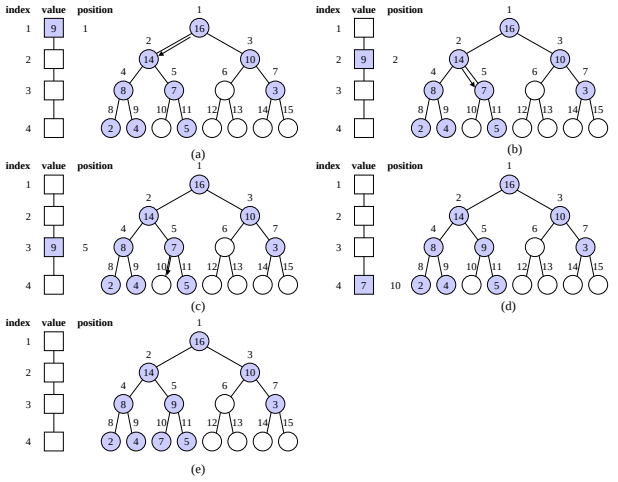

Fig. 3. An example of the enqueue procedure

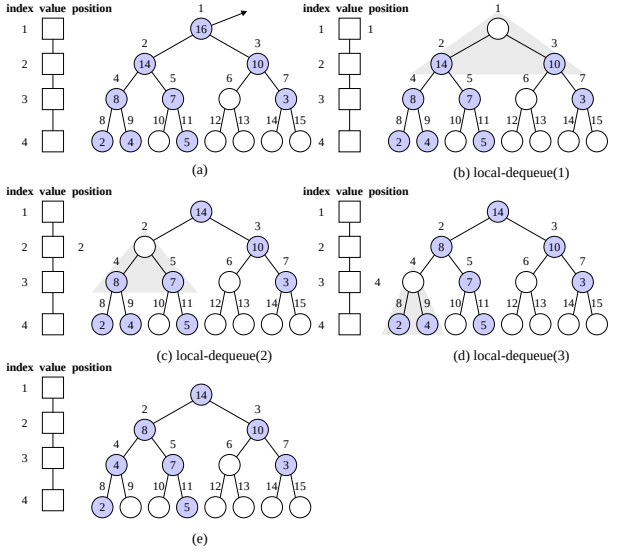

Fig. 4. The P-heap dequeue operation

B.2 The dequeue Operation

The dequeue operation extracts $B[1].value$ from the P-heap, since it has the highest priority value, making $B[1]$ inactive and hence increasing the capacity of B[1] by one. The children of $B[1]$, however, may be active, thus violating the P-heap property 2. We need to push down the inactive node to the lower levels till the P-heap property is maintained throughout B.

An example is shown in Figure 4 and described below.

Cycle 1: The top-most value, 16 is first removed (Figure 4 (a)) and $B[1]$ is made inactive, while its capacity is increased by one. The operation moves the inactive node from $B[1]$ down to $B[2]$ (Figure 4 (b)), since it has a larger value than $B[3]$. This comparison between $B[2]$ and $B[3]$ is necessary to maintain the P-heap property. The value 14 is moved from $B[2]$ to $B[1]$, making $B[1]$ active.

Cycle 2: The value of $B[4]$, $i.e.$ 8, which is larger than $B[5]$, is moved up to $B[2]$, while the inactive node moves further down to $B[4]$(Figure 4 (c)).

Cycle 3: Finally, as shown in Figure 4 (d), the inactive node is moved down to $B[9]$, while $B[4]$ is filled up with the value 4. Since $B[9]$ is a leaf node, its being inactive does not violate the P-heap properties. Figure 4 (e) shows what B looks like at the end of the dequeue.

B.3 The enqueue-dequeue operation

The P-heap data structure is built to accommodate a new priority queue operation, the enqueue-dequeue, which allows simultaneous enqueue and dequeue to occur in the P-heap. The process is similar to the dequeue. The difference is that in the first step, instead of removing the value of the top node and making it inactive, we remove the value and $replace$ it by the new value to be inserted into the queue. This may violate P-heap property 1 within Δ_1. So the new value needs to be trickled down in B till all properties hold. All these operations are explained in more detail in [9].

C. Pipelining the P-heap Operations

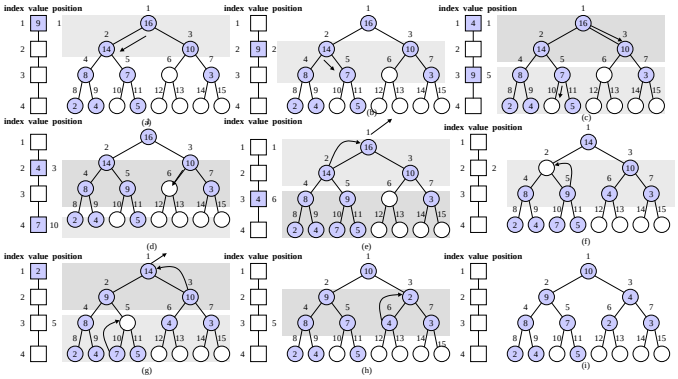

Fig. 5. The pipelined operation of a 4-level P-heap

The three operations on the P-heap all work top-down, and each cycle involves only two consecutive levels of the P-heap. Also, it should be noted that when an operation is working on levels l and $l+1$, then all the levels of the heap less than l appear to be completely sorted and consistent. This means that while an operation is executing on levels l and $l+1$, another consequent operation can execute on the P-heap at the levels less than l. This enables us to pipeline the P-heap operations.

We now give an example of the P-heap pipeline in action. Figure 5 shows a four-level P-heap with the following operations executed on it.

enqueue(9);
enqueue(4);
dequeue;

Cycle 1: The new value to be enqueued, 9, is written into $T[1]$ (Figure 5 (a)). The value 9, which is smaller than 16, is pushed down to $T[2]$. The left branch of B is taken, since the left sub-tree has an inactive node.

Cycle 2: In Figure 5 (b)), since 9 is smaller than $B[2].value$, which is 14, it is moved down to the next level, $i.e.$ $T[3]$. $B[2].capacity$, which was originally 1, is decremented to 0. The right branch is taken in this case, since the left sub-tree does not have an inactive node. $T[3].operation$ is set to enq.

Cycle 3: Note that levels 1 and 2 are completely sorted and ready to receive another operation. So, while enqueue(9) continues at levels 3 and 4 of B (Figure 5 (c)), enqueue(4) is also started on levels 1 and 2.

Cycle 4: The enqueue(9) completes(Figure 5 (d)), having found an inactive node in $B[10]$ to insert the value 7. At the same time, enqueue(4) operates on levels 2 and 3 of the P-heap.

Cycle 5: enqueue(4) (Figure 5 (e)) continues on levels 3 and 4. It finds an inactive node, $B[6]$, and writes the value 4

into it, thus ending this enqueue operation. At the same time, the dequeue operation starts working on levels 1 and 2.The highest value, 16, is removed from $B[1]$ and it is made inactive. Since the larger of the values $B[2]$ and $B[3]$ is 14, it is moved up to $B[1]$, while $B[2]$ is made inactive.

Cycle 6: The dequeue executes on levels 2 and 3(Figure 5 (f)), in which since $B[2]$ is inactive, the value of $B[5]$, 9, is moved up to $B[2]$ while $B[5]$ is rendered inactive.

Cycle 7: The dequeue continues its execution and the value of $B[10]$, 7, is moved up to $B[5]$(Figure 5 (g)). The dequeue can be stopped at this stage, since no P-heap property is violated any longer.

Figure 5 (h) shows the P-heap after the completion of these pipelined operations.

From the example, it is clear that we can start a new operation on the P-heap every two cycles. Effectively, we achieve constant time enqueue and dequeue operations using the P-heap pipelined priority queue.

III. Crossbar Scheduling with Deadline-Ordered P-heap based VOPQs

In a crossbar switch, all cells arriving at an input port are buffered in the same external memory module associated with the port. To avoid memory access conflicts, the crossbar scheduler must ensure that each input delivers at most one cell into the crossbar fabric at each time slot. Similarly, each port is only capable of transmitting one cell out of its outgoing link. Thus, the crossbar scheduler must also ensure that at most one cell is scheduled to each output.

The conventional crossbar scheduling problem is defined as follows. For each input port i, there are N requests going to the scheduler, $R_{i,1}, R_{i,2}, \ldots, R_{i,N}$, each corresponding to a virtual output queue. $R_{i,j} = 1$ indicates that there is a request from input i to switch a cell to output port j. Correspondingly, there are N grant signals produced by the scheduler, $G_{i,1}, G_{i,2}, \ldots, G_{i,N}$, where $G_{i,j} = 1$ indicates that the scheduler has granted the request to transfer a cell from input port i to output port j. To ensure that each input delivers at most one cell into the crossbar fabric, and that each output receives at most one cell, the conditions

$$\sum_{j=1}^{N} G_{i,j} \leq 1, \text{ for } 1 \leq i \leq N$$

and

$$\sum_{i=1}^{N} G_{i,j} \leq 1, \text{ for } 1 \leq j \leq N$$

must hold, respectively. Several effective scheduling algorithms and their hardware implementations have been reported in the literature [13], [7], [16], [8].

In our deadline-ordered switch design, we ensure that the cells with the most urgent deadline in a particular VOPQ are scheduled ahead of all other cells *in the same VOPQ*. However, this property does not hold across VOPQs. That is, the cell with the highest priority among all VOPQs for the same input port may not be scheduled first. This depends on the scheduling decisions made by the arbiter. For the arbitration step, we use the round-robin-based symmetric arbiter design described in [8] known as DPA. This strategy ensures that if the bandwidth supported by the switch is B, and the number of ports is N, then each VOPQ gets its bandwidth share of B/N^2. Our strategy thus provides fair, though not always ensuring exact priority scheduling over the entire input port, but with respect to each individual VOPQ, exact priority scheduling is achieved.

The entire switch architecture is pipelined to ensure the promised OC-192 speeds. The P-heap operations are pipelined with the functioning of the arbiter as well. Thus, in a single pipeline stage, apart from the enqueue and/or a dequeue, the arbiter also makes the scheduling decisions, which are fed back to the P-heap controller unit.

IV. Hardware Requirements

In this section, the hardware necessary to implement virtual output priority queues(VOPQs) as P-heaps in a high-speed packet switch is described.

In order to pipeline the P-heap operations, simultaneous memory accesses need be performed to the different levels of the VOPQs. Each level of the binary array therefore requires a separate SRAM for storage. To avoid memory access conflicts in the crossbar switch, at any given clock cycle, no more than one enqueue and/or one dequeue can be started at a particular input port. Thus, in a given clock cycle, if an operation is working on level i of a particular VOPQ, no other operations can be active on level i of any other VOPQ belonging to the same input port. This is because only one operation can be operating at a particular level of the input port at any given time. This implies that there can be no more than two simultaneous memory accesses to the same levels of two different VOPQs belonging to the same input port[1]. So, level i of all the binary arrays of a port's N VOPQs can be stored in the same dual ported SRAM. Thus for each input port, the number of SRAMs used is l, where l is the number of levels in the VOPQs, as shown in Figure 6.

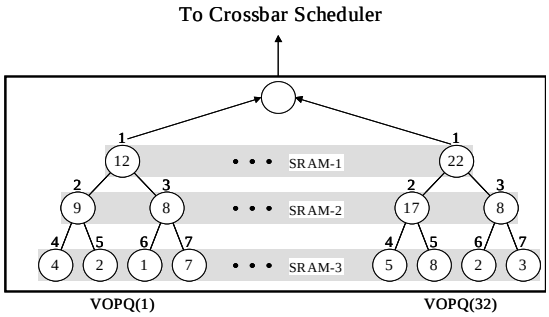

Fig. 6. Organization of memories for the binary arrays of the VOPQs

For the same reasons as stated above, we do not require separate token arrays for each and every VOPQ. It suffices to have two token arrays per input port; one for the enqueue operations and one for the dequeue and enqueue-dequeue operations. We call these two arrays $T_{enqueue}$ and $T_{dequeue}$ respectively. We need two separate datapaths per input port, as we could need to enqueue a cell on a particular VOPQ while simultaneously dequeue a cell from another VOPQ of the same input port. Hence each input port requires $2l$ registers for the two token arrays.

Each entry in the SRAMs, apart from maintaining the $active$, $value$ and $capacity$ fields of B, also holds a *pointer* to the external memory module where the cell data are stored. The priority management function datapath and the SRAMs required for implementing the P-heap are integrated into a single module called the *P-heap Manager* (PHM) shown in Figure 7.

[1] In the worst case, we might need to perform enqueue and dequeue on two different VOPQs at the same time, requiring two simultaneous memory accesses to the same levels of the VOPQs.

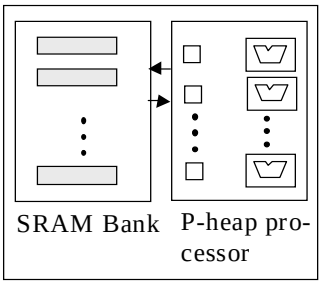

Fig. 7. The P-heap Manager (PHM) that works with the external DRAM module and the Priority Assignment Unit.

The PHM holds two sub-modules: the SRAM Bank and the P-heap Processor Engine. While the SRAM Bank is a collection of the l SRAMs, the P-heap Processor Engine consists of the datapath and controller used to implement the three P-heap operations. The two token arrays $T_{enqueue}$ and $T_{dequeue}$ are a part of the datapath. The controller holds $2l$ comparators, two for each level of the P-heap, so that different local operations may occur at different levels of the P-heap without any resource hazards. Each input port therefore requires l SRAMs, $2l$ registers and $2l$ comparators for constant-time operation. This is in contrast to the systolic array, which, for the same queue length, requires 2^{l+1} registers and 2^l comparators. For example, a 1024 packet-long systolic array requires 2048 registers and 1024 comparators. The hardware requirements increase linearly with the size of the queue. The P-heap implementation of the input queues of the same size requires only 20 comparators. The hardware required for the P-heap increases logarithmically with the size of the queue.

Currently available on-chip SRAMs can have sizes up to 256 KB. Using these memory modules along with 32 bit-wide priority values, each P-heap entry requires about 8 bytes of space, which would mean that the P-heap size can vary up to 32,000 packet entries. This translates to an external memory buffer space of 1.7 MB for an ATM cell switch.

V. Implementation Results

For implementation analysis, we used the TSMC 0.35 micron CMOS standard-cell technology. All delay results were measured in nanoseconds under a nominal supply voltage of 3.3V. As mentioned in the introduction, crossbar scheduling is overlapped with the enqueueing and dequeueing of cells. The enqueueing and dequeueing of cells is handled by the P-heap processor engine, and the scheduling of requests is handled by the crossbar scheduler. Our current implementation is based on a 32-port switch.

We implemented the P-heap processor engine using the memory access times for a 256 KB synchronous SRAM. Table I shows the P-heap pipeline stage time, which is the effective time required to execute a single enqueue, dequeue or an enqueue-dequeue operation, for different sizes of the priority field. These values were obtained by calculating th size of a P-heap Pipeline cycle (referred to in the enqueue, dequeue and enqueue-dequeue examples) and multiplying it by 2, since this is the time required between any two consecutive P-heap operations.

We know that the DPA arbiter works at a speed of 12.61 ns for a 32x32 port switch [8]. So, for a 32x32 switch with 32 bit priority fields, the bottleneck is the P=heap cycle time, which is 28.12 ns. From these figures, we conclude that with 32 bit priorities, the proposed switch architecture can schedule one packet every 28.12 ns, $i.e.$, at the rate of 35.56 Mpps. For an ATM cell switch, with cell sizes of 424 bits, this rate would translate to allowing link speeds of 15.08 Gb/s, which is quite a bit higher than our objective of meeting the 10 Gb/s OC-192 rates.

TABLE I
Variation of P-heap pipeline stage time with number of bits in the priority value field

Priority (bits)	P-heap Pipeline Stage Time (ns)
4	13.94
8	15.10
12	16.52
16	18.84
20	21.16
24	23.48
28	25.80
32	28.12

References

[1] J. Chao, "A novel architecture for queue management in the ATM network", In IEEE Journal on Selected Areas in Communications, 9(7):1110-1118, September 1991.
[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to algorithms", McGraw-Hill Book Company, ISBN 0-07-013143-0.
[3] R. L. Cruz, "Quality of service guarantees in virtual circuit switched networks", In IEEE Journal on Selected Areas in Communications, vol. 13, no. 6, August 1995.
[4] A. Demers, S. Keshav, S. Shenker, "Analysis and simulation of a fair queueing algorithm", In Proc. of ACM SIGCOMM'89, pp. 1-12, 1989.
[5] D. Ferrari and D. Verma, "A scheme for real-time channel establishment in wide-area networks", In IEEE Journal on Selected Areas in Communications, vol. 8, no. 4, pp. 368-379, April 1990.
[6] N. R. Figueira and J. Pasquale, "Leave-in-time: a new service discipline for control of real-time communications in a packet-switching network", In Proc. of ACM SIGCOMM'95, August 1995.
[7] P. Gupta and N. McKeown, "Design and Implmentation of a Fast Crossbar Scheduler", In Hot Interconnects VI, Stanford University, August 1998.
[8] J. Hurt, A. May, X. Zhu, and B. Lin, "Design and implementation of high-speed symmetric crossbar schedulars", In IEEE Int. Conf. on Communications, June 1999.
[9] R. Bhagwan and B. Lin, "Fast and Scalable Priority Queue Architecture for High-Speed Netwok Switches", In Proc. of IEEE Infocom'00, vol. 2, pp. 12-20, 2000.
[10] C. Kalmanek, H. Kanakia, and S. Keshav, "Rate controlled servers for very high-speed networks", In Proc. of IEEE GLOBECOM'90, vol. 1, pp. 12-20, 1990.
[11] P. Lavoie and Y. Savaria, "A systolic architecture for fast stack sequential decoders", In IEEE Trans. on Communications, 42(2-4):324-334, Feb-Apr 1994.
[12] C. E. Leiserson, "Systolic priority queue", In Caltech Conference on VLSI, pp. 200-214, January 1979.
[13] N. McKeown, M. Izzard, A. Mekkittikul, B. Ellersick, M. Horowitz, "The tiny tera: a packet switch core", In Hot Interconnects Symposium, Stanford University, August 1996.
[14] S. W. Moon, K. G. Shin, and J. Rexford, "Scalable hardware priority queue architectures for high-speed packet switches", In Proc. of Real-Time Applications Symposium, June 1997.
[15] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated service networks: the single-node case", In IEEE/ACM Trans. on Networking, vol. 1, no. 3, pp. 344-357, June 1993.
[16] Y. Tamir, H.C. Chi, "Symmetric crossbar arbiters for VLSI communication switches", In IEEE Trans. on Parallel and Distributed Systems, vol. 4, no. 1, pp.13-27, 1993.
[17] D. Verma, H. Zhang, and D. Ferrari, "Delay jitter control for real-time communication in a packet switching network", In Proc. of IEEE TriCom'91, pp. 35-43, April 1991.
[18] L. Zhang, "VirtualClock: a new traffic control algorithm for packet switching networks", In ACM Trans. on Computer Systems, vol. 9, no. 2, pp. 101-124, May 1991.
[19] D. Picker and R. Fellman, "A VLSI priority packet queue with inheritance and overwrite", In IEEE Transactions on VLSI Systems, 3(2):245-252, June 1995.
[20] J. Rexford, J. Hall, and K. G. Shin, "A router architecture for real-time point-to-point networks", In Proceedings of International Symposium on Computer Architecture, pp. 237-2 46, May 1996