# Fast and Scalable Priority Queue Architecture for High-Speed Network Switches

Ranjita Bhagwan, Bill Lin
Center for Wireless Communications
University of California, San Diego

*Abstract* -**In this paper, we present a fast and scalable pipelined priority queue architecture for use in high-performance switches with support for fine-grained quality of service (QoS) guarantees. Priority queues are used to implement highest-priority-first scheduling policies. Our hardware architecture is based on a new data structure called a Pipelined heap, or P-heap for short. This data structure enables the pipelining of the enqueue and dequeue operations, thereby allowing these operations to execute in essentially constant time. In addition to being very fast, the architecture also scales very well to a large number of priority levels and to large queue sizes. We give a detailed description of this new data structure, the associated algorithms and the corresponding hardware implementation. We have implemented this new architecture using a 0.35 micron CMOS technology. Our current implementation can support 10 Gb/s connections with over 4 billion priority levels .**

## I. INTRODUCTION

Future packet-switched integrated-service networks are expected to support a wide variety of real-time applications with diverse quality of service (QoS) requirements. These real-time applications have stringent performance requirements in terms of average throughput, end-to-end delay and cell-loss rate. These requirements must be satisfied at extremely high speeds without compromising on network utilization.

To provide for QoS guarantees in packet-switched networks, a number of service disciplines have been proposed, including [8], [24], [9], [10], [26], [5], [6], [25], [19], [18], [11], [12], [7]. Many of these service disciplines are based on a *prioritization* of the network resources to match different QoS requirements. In these schemes, packets are assigned priority values and are transmitted in a *highest-priority-first* order[1].

To implement this priority-based scheduling policy, priority queues can be used to maintain a real-time sorting of the queue elements in a decreasing order of priorities. Thus, the task of highest-priority-first scheduling can be reduced to a simple removal of the top queue element. However, to maintain this real-time sorting at link speeds, a fast hardware priority queue implementation is essential.

In the literature, several hardware-based sorted priority queue architectures have been proposed: calendar queues [1], binary-tree-of-comparators-based priority queues [21], [22], shift-register-based priority queues [2], [3], [23], and systolic-array-based priority queues [14], [15], [17]. All of these schemes have one or more shortcomings. The calendar queues, for example, can only accommodate a small fixed set of priority values since a large priority set would require extensive hardware support. The other three classes of architectures, based on binary-trees-of-comparators, shift-registers, and systolic arrays, are all generally difficult to scale because the hardware complexity is dependent on the worst-case queue size: each queue element requires a separate comparator datapath and separate register storage.

In this paper, a new *pipelined* priority queue architecture is described. The architecture is based on a novel data structure called a *Pipelined-heap*, or *P-heap* for short. This data structure is similar to a conventional *binary heap* [4]. In the literature, several parallel algorithms have been proposed to make binary heap enqueue and dequeue operations work in constant time ([28], [29]), but these schemes are based on multi-processor software implementations with OS support for data locking and inter-process messaging. We believe implementing these in hardware is both non-trivial and expensive. Our approach has modified the binary heap in a manner to facilitate the pipelining of the enqueue and dequeue operations while keeping the hardware complexity low, thereby allowing these operations to execute in essentially *constant time*.

Our current implementation is designed to support a total link rate of 10 Gb/s, which can correspond to either a single OC-192 connection or a combination of lower speed connections.

In addition to being very fast, our priority queue architecture also scales very well with respect to both the number of priority levels and the queue size. Our current implementation can support $2^{32}$ (over 4 billion) priority levels using a 32-bit priority field. Instead of requiring a separate comparator datapath for each queue element, the number of comparator datapaths needed in our pipelined architecture is logarithmic to the queue size. In addition, the storage required for the queue elements can be efficiently organized into on-chip SRAM modules. Also, our current implementation can support unlimited buffer sizes. Using a 0.35 micron CMOS technology, all the priority queue management functions, including the necessary SRAM components, can be implemented in a single application-specific integrated circuit.

The remainder of this paper is organized as follows. In Section II, we describe briefly the queue manager model for which the P-heap has been devised. In Section III, we describe the P-heap data structure and in Section IV, its enqueue and dequeue algorithms. In Section V, we describe the pipelined operation of the P-heap. In Section VI, hardware implementation and memory organization issues are discussed. Section VII shows the hardware implementation results using a 0.35 micron CMOS technology.

[1]Service disciplines based on th Earliest-Deadline-First scheme can be mapped to the highest-priority-first scheduling order, *i.e.*, an earlier deadline would translate to a higher priority, while a later deadline would map to a lower priority value

## II. The Queue Manager

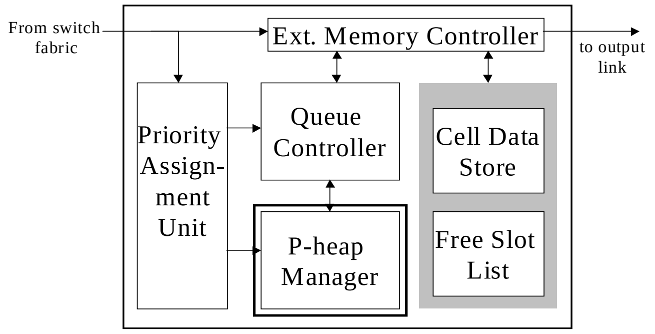

Fig. 1. The output queue manager

To support the operations of the P-heap, a dynamic queue manager is required for each port of the switch. In an output-queued switch, the queues can be maintained dynamically as described in [27]. We use a similar model in our approach, where each output queue is maintained using an Output Queue Manager, as shown in Figure 1. The Output Queue Manager consists primarily of a Priority Assignment Unit, a P-heap Manager and a Queue Controller. The Priority Assignment Unit stamps the incoming packets with a certain priority, which is decided depending upon the scheduling algorithm in use. The P-heap Manager contains the P-heap priority queue. Each of the elements in the P-heap holds a priority value, and it is on these values that the queue is sorted. The Queue Controller maintains a lookup table with entries corresponding to each priority value. Each entry consists of a pointer to a list of packets of the same priority. We refer to this list as a *priority list*. Thus our implementation is one of "per-priority queueing" rather than per-flow queueing, and is more general in the sense that it can handle differing priorities within the same flow. It should be noted that at any point of time, the P-heap only contains the priority values for which the priority list is non-empty, $i.e.$, it contains only the $active$ priority values.

When a new packet needs to be inserted into the queue, the Priority Assignment Unit stamps the packet with a suitable priority value. The Queue Controller determines whether a priority list already exists for the stamped priority value. If it does, it simply adds the new packet to the corresponding priority list. However, if the list does not exist, the Queue Controller creates a new priority list. It also signals the P-heap manager to perform an `enqueue` operation, which inserts the new priority value into the P-heap in a sorted manner. This is done to make sure that the highest priority stays at the top of the P-heap so that whena dequeue of a packet is required, the priority list with the highest priority can be accessed.

When a packet needs to be removed from the queue, the P-heap Manager determines the non-empty priority list of highest priority by looking at the topmost element of the P-heap and sends this priority value to the Queue Controller. The Queue Controller accesses the corresponding priority list and removes a single packet from it. If this causes the priority list to become empty, the P-heap Manager inititates a procedure called `de-queue` which removes the topmost element from the P-heap, while making sure that the P-heap remains sorted.

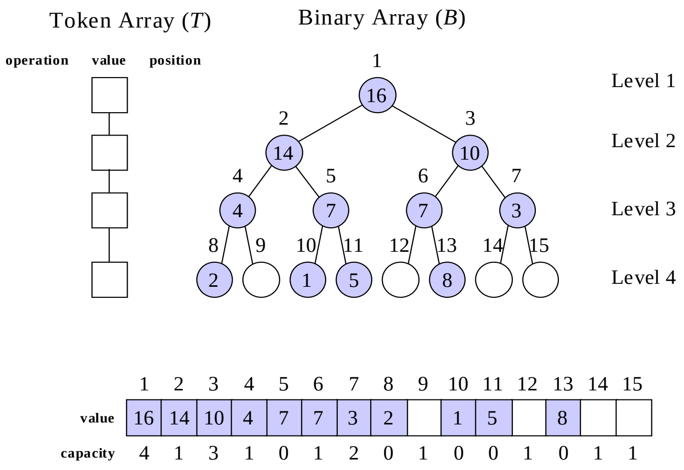

Fig. 2. The P-heap data structure

The former discussion is a brief note on the working of the queue manager. In this paper, however, we concentrate on the P-heap data structure and the operations on it and the following sections give a detailed explanation of the same.

## III. The P-heap Data Structure

A known data structure for maintaining priority queues is the *binary heap* [4]. Its enqueue and dequeue operations use $O(log(n))$ time, where $n$ is the size of the heap, and cannot be easily pipelined. However, it takes a very simple hardware implementation to emulate a binary heap. Keeping this fact in mind, we designed the *Pipelined Heap* or the *P-heap*, which, while preserving the ease of hardware implementation of the conventional binary heap, allows pipelined implementation of priority queue operations, providing, in effect, constant-time operations.

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

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

where $l$ is the *number of levels* in the tree represented by $B$. For example, in Figure 2, $l$ = 4 and hence the number of elements in the binary array $B$, or the number of nodes in the binary tree it represents, is $2^4 - 1$, or 15. 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:

$$parent(i) = \lfloor i/2 \rfloor$$

$$left(i) = 2i$$

$$right(i) = 2i + 1$$

In the rest of the paper, we refer to a node in $B$ with index $i$ as $B[i]$. We formally define the $j^{th}$ level of $B$, $L_j$, as the set of nodes given by

$$L_j = \{B[i] \mid 2^{j-1} \leq i \leq 2^j - 1\}$$

Another difference between $B$ and a binary heap data structure lies in the contents of each node in the tree. While in a binary heap, the nodes contain just the value on which the heap is sorted, 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 Figure 2, 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

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

We define a *trio* for every node in $B$ as follows: The $i^{th}$ $trio$ of $B$, $\Delta_i$, is defined as the set of nodes

$$\{B[i], B[left(i)], B[right(i)]\}$$

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. $B[j].active \Rightarrow B[i].active$

For example, in Figure 2, $B[1].value$ is 16, while $B[2].value$ is 14. This satisfies property 1. Also, no active node has an inactive parent. This conforms with property 2.

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 $l$, which we formalize here as

$$length(T) = l \quad l \in Z^+$$

For example, in Figure 2, the value of $l$ is 4. The length of the token array is also 4. The element $T[i]$ is associated with the level $L_i$ of $B$. For example, $T[1]$ is associated with level 1 of $B$, which is the root. The purpose of the token array is to aid the pipelined operation of the P-heap. Each element $T[i]$ of the token array also comprises of three fields. They are:

- $T[i].operation$: this field holds an $instruction$, depending on the operation to be executed at level $L_i$ of the P-heap. It is useful in the pipelined implementation of P-heaps.
- $T[i].value$: This field may hold a priority value that needs to be inserted into $B$.
- $T[i].position$: this field can hold the index of a node at level $L_i$ of $B$.

The significance of these fields is addressed in the next two Sections.

## IV. Priority Queue Operations on the P-heap

The motivation behind defining a new priority queue data structure is to be able to $pipeline$ the enqueue and dequeue operations on it. The conventional binary heap operations, though very simple to implement, execute in $O(log(n))$ steps,where $n$ is the number of elements in the heap, and they cannot be easily pipelined. On the other hand, architectures like the systolic array can be pipelined, but have extremely high hardware requirements. 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 developed keeping this objective in mind.

### A. *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 $B[i_j] \in L_j$ 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. Figure 3 shows all the valid paths in the given binary array. In this example, there are a total of five valid paths in $B$.

The `enqueue` operation first writes the new value into $T[1].value$, sets $T[1].position$ to 1 and travels through a valid path by making up to $l$ calls to a core procedure called the

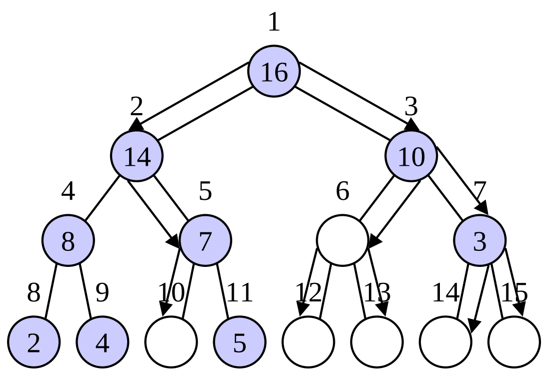

Fig. 3. An example of a binary array $B$ with five valid paths

`local-enqueue`. This procedure takes as input $j$, denoting the level $L_j$ of $B$. The `enqueue` first calls `local-enqueue`(1). The `local-enqueue` algorithm works in the following way:

- The position and value fields of the element in $T[j]$ are read. We refer to them as $i$ and $v$ respectively.
- if $B[i]$ is inactive, $v$ is simply written into $B[i].value$, making $B[i]$ active. This reduces the number of inactive nodes in the sub-tree rooted at $B[i]$ by 1. This number is stored in $B[i].capacity$, which is therefore decremented by 1. The `local-enqueue` operation completes, and since the new value has found a place in $B$, the `enqueue` operation may be stopped too.
- If $B[i]$ is active and the value $v$ is greater than that of $B[i]$, the two are swapped. This step does not violate the P-heap properties as we know that for $\Delta_i$, if all the nodes in it are active,

$$B[i].value \geq B[left(i)].value$$

$$B[i].value \geq B[right(i)].value$$

and therefore

$$v > B[i].value \Rightarrow v > B[left(i)].value$$

$$v > B[i].value \Rightarrow v > B[right(i)].value$$

- The value in $T[j]$ is moved down to $T[j+1].value$.
- The *capacity* of $B[left(i)]$ is read. If it is greater than zero, it means that there are some inactive nodes in the sub-tree rooted at $B[left(i)]$, and $B[left(i)]$ can therefore be part of the valid path to be traversed. The index $left(i)$ is written into $T[j+1].position$, so that in the next iteration of `enqueue`, a call to the procedure `local-enqueue`(j+1) may be made to examine $\Delta_{left(i)}$ for an inactive node.
- If the capacity of the left child is zero, the capacity of $B[right(i)]$ must be non-zero [2], and the index $right(i)$ is written into $T[j+1].position$, so that in the call to procedure `local-enqueue`(j+1), $\Delta_{right(i)}$ may be examined for an inactive node.

[2]If this is not the case, it implies that the queue is full and our algorithm leads to a value being dropped from the priority queue.

```
procedure local-enqueue($j$)
begin
    $i \Leftarrow T[j].position$;
    $v \Leftarrow T[j].value$;
    if $B[i].value$ = false
        $B[i].value \Leftarrow v$;
        $B[i].active \Leftarrow$ true;
        Decrement $B[i].capacity$;
        return done;
    elsif $T[j].value > B[i].value$
        Swap $T[j].value, B[i].value$;
        Move $T[j].value$ to $T[j+1].value$;
    end if;
    if $B[left(i)].capacity > 0$
        $T[j+1].position \Leftarrow left(i)$;
    else
        $T[j+1].position \Rightarrow right(i)$;
    end if;
    return not_done;
end procedure;
```

Fig. 4. The `local-enqueue` algorithm.

The algorithm followed by `local-enqueue` is shown in Figure 4.

The `local-enqueue` procedure is called up to $l$ times before the `enqueue` can complete. This is because the valid path stretches from the root of $B$ to a leaf of $B$, and is therefore $l$ nodes long. We may have to traverse the path starting from the root right till the leaf node to find an inactive node, which would be accomplished by making $l$ calls to the `local-enqueue` procedure. It can also be observed that any instance of `local-enqueue` works on a single trio. This is a significant point which shall be referred to later in the section on the P-heap pipeline.

Going back to the `enqueue` operation, it sets $T[1].value$ to the new value to be inserted, and $T[1].position$ to 1. Following this, `local-enqueue`(1), `local-enqueue`(2),..., `local-enqueue`($l$) may be executed. An example is shown in Figure 5. The new value 9 is stored in $T[1].value$, while $T[1].position$ is set to 1. The operation `local-enqueue`(1) is then executed (Figure 5 (a)). Since 9 is smaller than $B[1].value$,*i.e.* 16, there is no swap. The new value 9 is moved down to $T[2].value$. The capacity of $B[2]$ is examined, and is found to by equal to 1 since $B[10]$ is inactive. $T[2].position$ is therefore set to the index 2. The execution of `local-enqueue`(1) completes.

Now, `local-enqueue`(2) is executed (Figure 5 (b)). $T[2].value$ and $T[2].position$, which have the values 9 and 2 respectively, are read. The value of $B[2]$ is read and is found to be 14. $T[2].value$, 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 $B[4]$ 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 index of the *right child* of $B[2]$, i.e. 5 is written into $T[3].position$, ending the execution of `local-enqueue`(2).

`local-enqueue`(3) is now executed, as shown in Figure 5 (c). The value and position fields of $T[3]$ are read, which are

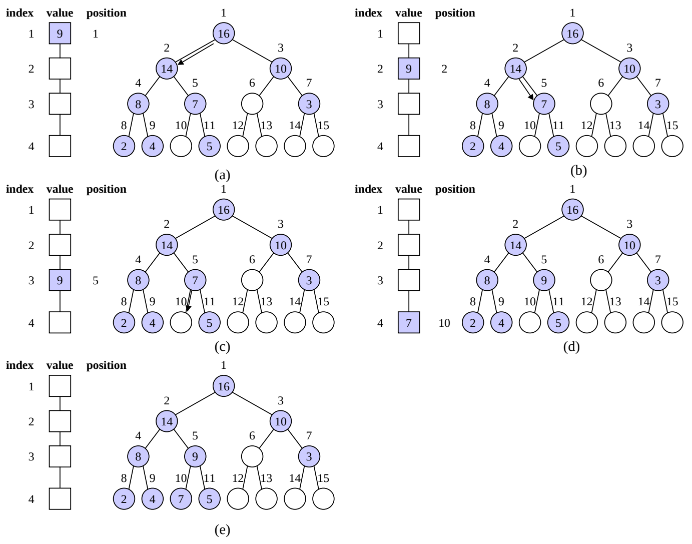
Fig. 5. An example of the `enqueue` procedure

```
procedure enqueue($V$)
begin
    $T[1].value \Leftarrow V$;
    $T[1].position \Leftarrow 1$;
    while $j \leq l$ do
        $temp \Leftarrow$ local-enqueue($j$);
        if $temp =$ done
            return;
        else
            $j \Leftarrow j + 1$;
        end if;
    end while;
end procedure;
```

Fig. 6. The `enqueue` algorithm.

found to be 9 and 5 respectively. The value 9 is compared with $B[5].value$, $i.e.$ 7. Since 9 is larger, the two are swapped. $T[3].value$ now holds the value 7 and it is moved down to $T[4].value$. The capacity of $B[10]$, the left child of $B[5]$, is examined. It is found to be 1 and so the index 10 is written into $T[4].position$.

During the execution of `local-enqueue`(4), shown in Figure 5 (d), it is found that $B[10]$ is inactive, 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 5 (e). The algorithm followed by the `enqueue` operation is given in Figure 6.

## B. 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 within the 1st trio, $\Delta_1$. We need to push down the inactive node to the lower levels till the P-heap property is maintained throughout $B$. To sort the trio, we use a procedure called the `local-dequeue`. This procedure takes as input $j$,

```
Procedure local-dequeue($j$)
begin
    $i \Leftarrow T[j].positon$;
    if both $B[left(i)]$, $B[right(i)]$ are inactive
        return done;
    end if;
    Read the values of the active nodes among $B[left(i)]$
    and $B[right(i)]$;
    Determine the node $B[k]$ with largest value $V$;
    Make $B[i]$ active;
    $B[i].value \Leftarrow V$;
    Make $B[k]$ inactive;
    Increment $B[k].capacity$;
    $T[j+1].position \Leftarrow k$;
    return not_done;
end procedure;
```

Fig. 7. `local-dequeue` algorithm.

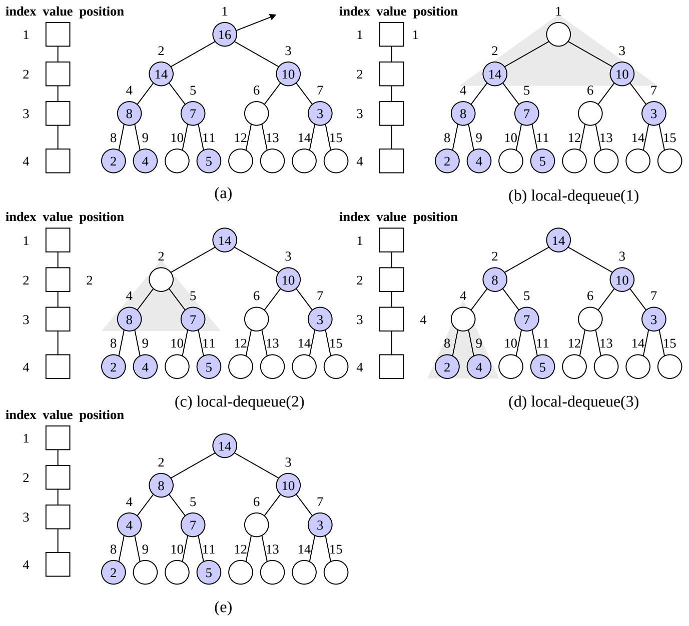
Fig. 8. The P-heap `dequeue` operation

indicating the level $L_j$ of $B$. The initial steps to sort the P-heap are to set $T[1].position$ to 1, increment $B[1].capacity$ and call `local-dequeue`(1). The operation `local-dequeue`($j$) operates as described below:

- $T[j].position$ is read into $i$. The index $i$ corresponds to an inactive node $B[i]$.
- If both the child nodes of $B[i]$ are inactive, property 2 is satisfied, and the `dequeue` operation may halt.
- Otherwise, at least one of the two child nodes is active. The values of the active nodes are read.
- Let us say that $B[k]$ is the node with the larger value $v$ of the two child nodes. $B[i]$ is made active and the value $v$ is written into it, while $B[k]$ is made inactive. In other words, the inactive node is pushed down from position $i$ to $k$. This increases the number of inactive nodes in the sub-tree rooted at $B[k]$ by 1. $B[k].capacity$ is therefore incremented by 1.
- The index $k$ is written into $T[j+1].position$ for future use,

```
procedure dequeue
begin
    $j \Leftarrow 1$;
    $max\_value \Leftarrow B[1].value$;
    $B[1].active \Leftarrow$ false;
    Increment $B[1].capacity$;
    while $j \leq l$ do
        $temp \Leftarrow$ local-dequeue($j$);
        if $temp$ = done
            return;
        else
            $j \Leftarrow j + 1$;
        end if;
    end while;
end procedure;
```

Fig. 9. The dequeue algorithm

```
Procedure local-enqueue-dequeue($j$)
begin
    $i \Leftarrow T[j].position$;
    if both $B[left(i)]$, $B[right(i)]$ are inactive
        return done;
    end if;
    Read the values of the active nodes among all three nodes
        in $\Delta_i$;
    Determine the node $B[k]$ with largest value;
    if $i = k$
        return done;
    end if;
    Swap $B[i].value, B[k].value$;
    $T[j+1].position \Leftarrow k$;
    return not_done;
end procedure;
```

Fig. 10. local-enqueue-dequeue algorithm.

when local-dequeue($j + 1$) is called.

Now, since the node $B[k]$ is inactive, the P-heap property 2 might be violated in the trio $\Delta_k$. This situation can be rectified by calling procedure local-dequeue($j + 1$). It reads $T[j + 1].position$, which is $k$ and moves the inactive node further down if required. Thus, by making up to $l$ calls to local-dequeue, we can ensure that the P-heap properties are satisfied throughout $B$.

An example is shown in Figure 8. The top-most value, 16 is first removed (Figure 8 (a)) and $B[1]$ is made inactive, while its capacity is increased by one. $T[1].position$ is set to 1 and local-dequeue(1) is called, as shown in Figure 8 (b). The operation sorts $\Delta_1$ by moving the inactive node from $B[1]$ down to $B[2]$, which has a larger value than $B[3]$. The value 14 is moved from $B[2]$ to $B[1]$, making $B[1]$ active. The index of the new inactive node, 2, is written into $T[2].position$.

Now, local-dequeue(2) is executed, shown in Figure 8 (c), so that $\Delta_2$ may be sorted to satisfy the P-heap property. The value of $B[4]$, $i.e.$ 8, is moved up to $B[2]$, while the inactive node moves further down to $B[4]$. The index 4 is written into $T[3].position$.

Finally, as shown in Figure 8 (d), local-dequeue(3) is executed which causes the inactive node to move 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 8 (e) shows what $B$ looks like at the end of the dequeue. The code for the dequeue operation is shown in Figure 9.

## C. *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 $V$ to be inserted into the queue. This may violate P-heap property 1 within $\Delta_1$. To sort the heap and satisfy the P-heap properties, the enqueue-dequeue makes up to $l$ calls to the core procedure local-enqueue-dequeue, which takes as input $j$, a level in $B$.

Figure 10 gives the algorithm. $T[1].position$, as in all other operations, is set to 1. The steps given below follow.

- The value of $T[j].position$ is read into $i$.
- If both child nodes of $B[i]$ are inactive, the P-heap properties are satisfied and the enqueue-dequeue operation may halt.
- If not, the active values are read from the three nodes in $\Delta_i$.
- The node $B[k]$ with the largest value $v$ is determined.
- if this node is $B[i]$, the P-heap properties are satisfied and the procedure enqueue-dequeue may halt.
- Otherwise, the values of $B[i]$ and $B[k]$ are swapped.
- The index $k$ is written into $T[j + 1].position$ to aid the call to local-enqueue-dequeue(j+1) in the next iteration.

Since $B[k]$ may now have a value smaller than its original value, $\Delta_k$ might violate the P-heap properties. Thus up to $l$ executions of the local-enqueue-dequeue procedure might be necessary to restore order.

An example is shown in Figure 11. The highest value 16 is dequeued, $i.e$ removed from $B[1]$, and it is replaced with the new value, 9 (Figure 11 (a)). local-enqueue-dequeue(1) is now called, which causes 9 to be swapped with the value of $B[2]$, $i.e.$ 14 (Figure 11 (b)). The index 2 is written into $T[2].position$. This is followed by a call to local-enqueue-dequeue(2), shown in Figure 11 (c), where it is observed that the P-heap properties are already satisfied within $\Delta_2$ and no swaps need to be made. Hence the enqueue-dequeue operation stops. Figure 12 gives the algorithm followed by the enqueue-dequeue operation.

## V. PIPELINING THE P-HEAP OPERATIONS

The operations explained in the previous section have all been designed keeping in mind the need to efficiently pipeline them for constant-time priority queue operation. In this section, we describe the *P-heap pipeline*, and how all three operations can be executed on the P-heap in constant-time.

The local procedures are all constant-time operations and access at most the three nodes in a trio, which belong to two consecutive levels of $B$, and two nodes in $T$, which belong to the same two levels of $T$. For example, local-dequeue(3) ac-

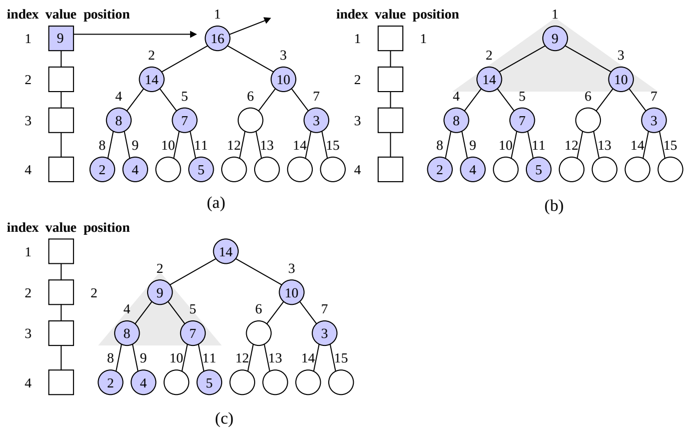

Fig. 11. The P-heap `enqueue-dequeue` operation

```
procedure enqueue-dequeue($V$)
begin
    $j \Leftarrow 1$;
    $max\_value \Leftarrow B[1].value$;
    $B[1].value \Leftarrow V$;
    while $j \leq l$ do
        $temp \Leftarrow$ local-enqueue-dequeue($j$);
        if $temp$ = done
            return;
        else
            $j \Leftarrow j + 1$;
        end if;
    end while;
end procedure;
```

Fig. 12. The `enqueue-dequeue` algorithm

```
function $f(i)$
begin
    if $T[i].operation = $ enq
        local-enqueue($i$);
    elsif $T[i].operation = $ deq
        local-dequeue($i$);
    elsif $T[i].operation = $ edq
        local-enqueue-dequeue($i$);
    end if;
end function;
```

Fig. 13. The function $f$.

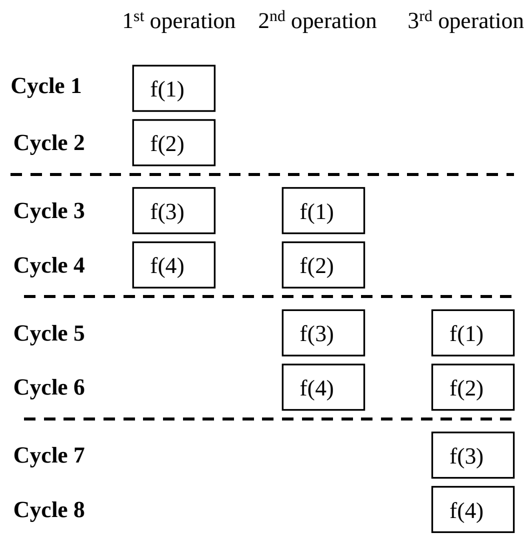

Fig. 14. Working of the P-heap pipeline

cesses only $L_3$ and $L_4$ of $B$, and nodes $T[3]$ and $T[4]$ of $T$. Similarly, `local-enqueue`(2) accesses $L_2$ and $L_3$ of $B$, and $T[2]$ and $T[3]$ of $T$. We show in the remainder of this section that several instances of the three local operations can be executed simultaneously in a pipelined manner.

We define the $i^{th}$ pipeline window $\alpha_i$ as

$$\alpha_i = \{L_i, L_{i+1}, T[i], T[i+1]\} \quad 1 \leq j < l$$

$$\alpha_i = \{L_i, T[i]\} \quad j = l$$

So, we can say that `local-dequeue`($j$) accesses only $\alpha_j$ and that `local-enqueue`($k$) accesses only $\alpha_k$.

We refer back to the definition of the token array $T$, where we introduced the field $T[j].operation$. This field holds one of four instructions,

{`enq, deq, edq, nop`}

one each for `enqueue`, `dequeue`, `enqueue-dequeue` and `no-op`. For example, if $T[3].operation$ = `dequeue`, it means that the `local-dequeue`(3) procedure, used by the `dequeue` operation, needs to be executed on $\alpha_3$ of the P-heap.

We also define function $f(i)$ as shown in Figure 13.

The function $f(i)$ executes one of the three local procedures on pipeline window $\alpha_i$ depending on the operation field of $T[i]$. So it can be said that $f(i)$ accesses only pipeline window $\alpha_i$, since it only executes `local-dequeue`($i$), `local-enqueue`($i$) or `local-enqueue-dequeue`($i$), all of which access only $\alpha_i$.

Based on this definition of $f$, we make the following claims.

*Claim 1:* If we can access every level of $B$ in parallel, the functions $f(1)$, $f(3)$,..., $f(i)$,..., $f(2\{\lfloor \frac{l-1}{2} \rfloor\}+1)$, where $i$ is an odd number, can be executed simultaneously.

*Proof:* If every level of $B$ is independently accessible, all operations for which $\alpha_j \cap \alpha_k = \phi$ can be executed simultaneously. $\alpha_1$, $\alpha_3$,..., $\alpha_{2\{\lfloor \frac{l-1}{2} \rfloor\}+1}$ satisfy this property.

Hence, the functions $f(1)$, $f(3)$., $f(i)$,..., $f(2\{\lfloor \frac{l-1}{2} \rfloor\}+1)$, where $i$ is an odd number, can be executed simultaneously. ■

*Claim 2:* If we can access every level of $B$ in parallel, the operations $f(2)$, $f(4)$..., $f(i)$,..., $f(2\{\lfloor \frac{l}{2} \rfloor\})$, where $i$ is an even number, can be executed simultaneously.

*Proof:* similar to that of claim 1.

■

We define a P-heap *cycle* in the following way:

A P-heap cycle is the maximum time required to execute any instance of the three local operations.

Having made these claims and definitions, we now show how

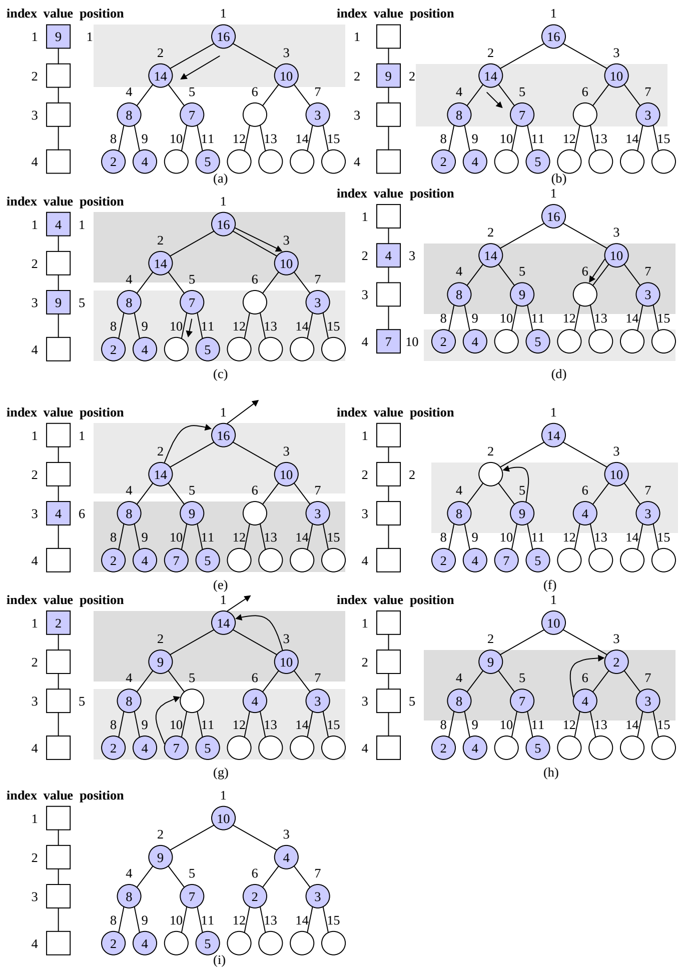

Fig. 15. The pipelined operation of a 4-level P-heap

the P-heap pipeline works on a four-level P-heap in Figure 14. In the Figure, three operations are executed one after the other. In cycles 1 and 2, only the 1$^{st}$ operation is active. In cycle 3, the 2$^{nd}$ operation starts alongside, with the simultaneous operation of $f(3)$ (executed by the 1$^{st}$ operation) and $f(1)$ (executed by the 2$^{nd}$ operation). The functions $f(2)$ and $f(4)$ are executed simultaneously in cycle 4. In cycle 5, the 1$^{st}$ operation ends while the 3$^{rd}$ starts. The two functions $f(1)$ and $f(3)$ are executed together, at the same time. This goes on till the pipeline is finally flushed. Each pipeline stage is therefore *2 cycles wide* and two consecutive instances of $f$ comprise the pipeline stage. A new operation can be started on the P-heap every two cycles.

We now give an example of the P-heap pipeline in action. Figure 15 shows a four-level P-heap with the following operations executed on it.

```
enqueue(9);
enqueue(4);
dequeue;
enqueue-dequeue(2);
```

**Cycle 1:** The new value to be enqueued, 9, is written into $T[1].value$ (Figure 15 (a)). $T[1].position$ is set to 1 and $T[1].operation$ is set to `enq`. This is followed by execution of $f(1)$, which in turn executes `local-enqueue`(1). The value 9, which is smaller than 16, is pushed down to $T[2].value$. The left branch of $B$ is taken, since the left sub-tree has an inactive node. $T[2].operation$ is now set to `enq`.

**Cycle 2:** `local-enqueue`(2) is executed in this cycle (Figure 15 (b)). Since 9 is smaller than $B[2].value$, which is 14, it is moved down to the next level, $i.e.$ $T[3].value$. $B[2].capacity$, 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:** The operation `enqueue`(9) continues at $\alpha_3$ of $B$ (Figure 15 (c)). Since 9 is larger than 7, the value of $B[5]$, the two are swapped and 7 is moved down to $T[4].value$, while $T[4].operation$ is set to `enq`. At the same time, `enqueue`(4) starts operating on $\alpha_1$. Since 4 is smaller than 16, it is just moved down to $T[2].value$. The capacity of $B[2]$ is found to be 0 since it was decremented by the `enqueue`(9) operation. So the right branch is taken for future comparison. $T[2].operation$ is set to `enq`. Thus the two operations that run simultaneously are $f(1)$ and $f(3)$. This conforms with claim 1.

**Cycle 4:** The `enqueue`(9) completes with the execution of $f(4)$ (Figure 15 (d)), having found an inactive node in $B[10]$ to insert the value 7. At the same time, the function $f(2)$ , called by the `enqueue`(4) operation , continues on $\alpha_2$ of $B$. Since 4 is smaller than $B[3].value$ which is 10, there is no swap. The value 4 is moved down to $T[3].value$ and the left branch is taken. $T[3].operation$ is set to `enq`. No new operation starts at this stage.

**Cycle 5:** $f(3)$ is executed by the operation `enqueue`(4) (Figure 15 (e)). It finds an inactive node, $B[6]$, and writes the value 4 into it, thus ending this `enqueue` operation. Since there is no `local-enqueue` necessary at level $L_4$, $T[4].operation$ is set to `nop`. At the same time, the `dequeue` operation starts working on $\alpha_1$ with the procedure `local-dequeue`(1) called by $f(1)$. The highest value, 16, is removed from $B[1]$ and it is made inactive. Since the largest of the values in the trio $\Delta_1$ is 14, it is moved up to $B[1]$, while $B[2]$ is made inactive. $T[2].operation$ is set to `deq`.

**Cycle 6:** The `dequeue` executes $f(2)$ (Figure 15 (f)), in which since $B[2]$ is inactive, $\Delta_2$ is found to violate the P-heap property. The value of $B[5]$, 9, is moved up to $B[2]$ while $B[5]$ is rendered inactive. $T[3].operation$ is set to `deq`.

**Cycle 7:** The `dequeue` continues with the execution of $f(3)$, where the value of $B[10]$, 7, is moved up to $B[5]$ so that the P-heap property is maintained in $\Delta_5$ (Figure 15 (g)). $T[4].operation$ is set to `nop`, since the `dequeue` can be stopped at this stage. The next operation, `enqueue-dequeue`(2), starts alongside by executing $f(1)$. The highest value 14 is removed from $B[1]$ and replaced by 2. Since the P-heap property is violated in $\Delta_1$, the value 2 of $B[1]$ is swapped with the value of $B[3]$, which is 10. $T[2].operation$ is set to

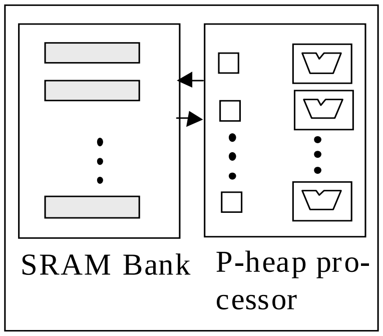

Fig. 16. The P-heap Manager (PHM), with the external DRAM module and the Priority Assignment Unit.

edq.

**Cycle 8:** The dequeue completes in the previous cycle, and so the only operation running now is the enqueue-dequeue (Figure 15 (h)). Since 2 is found to be smaller than the value of $B[6]$, $i.e.$ 4, the two are swapped. This operation completes right here.

Figure 15 (i) 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, dequeue and enqueue-dequeue operations using the P-heap pipelined priority queue.

## VI. Hardware Requirements

In this section, the hardware necessary to implement P-heaps in a high-speed packet switch is described. The P-heap array objects $B$ and $T$ are implemented using the following:

- The binary array $B$ is implemented not as one memory element, but as $l$ SRAM elements, where $l$ is the number of levels in $B$. All the nodes in level $i$ are maintained in SRAM-$i$. Simultaneous memory accesses can be performed to the different levels of $B$, as each one is stored in a different SRAM. This satisfies the condition specified in claims 1 and 2.
- The token array $T$ is represented by $l$ registers, one for each level of $B$.

The priority management functions and the SRAMs required for implementing the P-heap are integrated onto the P-heap Manager(PHM).

The PHM is shown in Figure 16. It 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 token array $T$ is a part of the datapath. The controller holds $l$ comparators, one 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.

The P-heap therefore requires $l$ SRAMs, $l$ registers and $l$ 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, 1024 comparators and additional combinational logic *per packet* in the queue. The hardware requirements increase linearly with the size of the queue. The P-heap of the same size requires only 10 comparators, as opposed to 1024 needed by the systolic array. The additional combinational logic required in a P-heap is *per level*. The hardware required for the P-heap increases logarithmically with the size of the queue. The P-heap requires 10 memory elements. Since the 1st few levels of the P-heap are very small ($L_1$ has only one element, $L_2$ has 2, $L_3$ has 4, etc), they can be stored in registers, while the larger levels can be stored in SRAMs.

Currently available on-chip SRAMs can have sizes up to 256 KB. Using these memory modules along with 32 bit-wide priority values, the P-heap can support about $2^{17}$ different active priority lists at any given time, while allowing for unlimited buffer space.

## VII. Implementation Results

We implemented the P-heap processor engine using the TSMC 0.35 micron CMOS standard-cell technology, along with 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 the size of a P-heap Pipeline cycle defined earlier and multiplying it by 2, since this is the time required between any two consecutive P-heap operations.

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 |

From these figures, we conclude that with 32 bit priorities, the P-heap 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.

An advantage of the P-heap is that the pipeline stage time does not change with increasing size of the queue. The P-heap pipeline stage width is *independent* of the actual length of the queue. The increase in queue length only increases the amount of hardware required, and that too, on a logarithmic scale.

## VIII. Conclusions

In this paper, we presented a fast and scalable pipelined priority queue architecture that can effectively support constant time enqueue and dequeue operations. The presented P-heap data structure and the associated algorithms are well-suited for hardware implementation. In addition to being very fast, the architecture also scales very well to a large number of priority levels and to large queue sizes. Our current implementation can support as many as $2^{32}$ priority values and can vary in size to up to $2^{17}$ entries. The P-heap can therefore be used efficiently in a high-speed packet switch providing fine-grained quality-of-service guarantees.

## References

[1] R. Brown, “Calendar queues: a fast O(1) priority queue implementation for the simulation of event set problem”, *Communications of the ACM*, 31(10):1220-1227, October 1988.

[2] J. Chao, “A novel architecture for queue management in the ATM network”, *IEEE Journal on Selected Areas in Communications*, 9(7):1110-1118, September 1991.

[3] J. Chao and N. Uzun, “A VLSI sequencer chip for ATM traffic shaper and queue management”, *IEEE Journal of Solid-State Circuits*, 27(11):1634-1643, November 1992.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, “Introduction to algorithms”, McGraw-Hill Book Company, ISBN 0-07-013143-0.

[5] R. L. Cruz, “Quality of service guarantees in virtual circuit switched networks”, *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 6, August 1995.

[6] R. L. Cruz, “Service burstiness and dynamic burstiness measures: a framework”, *Journal of High Speed Networks*, vol. 1, no. 2, pp. 105-127, 1992.

[7] A. Demers, S. Keshav, S. Shenker, “Analysis and simulation of a fair queueing algorithm”, *Proceedings of ACM SIGCOMM’89*, pp. 1-12, 1989.

[8] D. Ferrari and D. Verma, “A scheme for real-time channel establishment in wide-area networks”, *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 4, pp. 368-379, April 1990.

[9] N. R. Figueira and J. Pasquale, “Leave-in-time: a new service discipline for control of real-time communications in a packet-switching network”, *Proceedings of ACM SIGCOMM’95*, August 1995.

[10] N. R. Figueira and J. Pasquale, “Rate-function scheduling”, *Proceedings of INFOCOM’97*, pp. 1065-1074, April 1997.

[11] S. J. Golestani, “Congestion-free communication in high-speed packet networks”, *IEEE Transactions on Communications*, vol. 39, no. 12, pp. 1802-1812, December 1991.

[12] C. Kalmanek, H. Kanakia, and S. Keshav, “Rate controlled servers for very high-speed networks”, *Proceedings of IEEE GLOBECOM’90*, vol. 1, pp. 12-20, 1990.

[13] R. Katz, “Contemporary Logic Design”, Addison-Wesley/Benjamin-Cummings Publishing Co., Redwood City, CA, 1993

[14] P. Lavoie and Y. Savaria, “A systolic architecture for fast stack sequential decoders”, *IEEE Transactions on Communications*, 42(2-4):324-334, Feb-Apr 1994.

[15] C. E. Leiserson, “Systolic priority queue”, *Caltech Conference on VLSI*, pp. 200-214, January 1979.

[16] N. McKeown, M. Izzard, A. Mekkittikul, B. Ellersick, M. Horowitz, “The tiny tera: a packet switch core”, *Hot Interconnects Symposium*, Stanford University, August 1996.

[17] S. W. Moon, K. G. Shin, and J. Rexford, “Scalable hardware priority queue architectures for high-speed packet switches”, *Proceedings of Real-Time Applications Symposium*, June 1997.

[18] A. K. Parekh and R. G. Gallager, “A generalized processor sharing approach to flow control in integrated service networks: the multi-node case”, *Proceedings of IEEE INFOCOM’93*, vol. 2, pp. 521-530, March 1993.

[19] A. K. Parekh and R. G. Gallager, “A generalized processor sharing approach to flow control in integrated service networks: the single-node case”, *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pp. 344-357, June 1993.

[20] C. Partridge et al., “A 50-Gb/s IP router”, *IEEE/ACM Transactions on Networking*, vol. 6, no. 3, pp. 237-248, June 1998.

[21] D. Picker and R. Fellman, “A VLSI priority packet queue with inheritance and overwrite”, *IEEE Transactions on VLSI Systems*, 3(2):245-252, June 1995.

[22] J. Rexford, J. Hall, and K. G. Shin, “A router architecture for real-time point-to-point networks”, *Proceedings of International Symposium on Computer Architecture*, pp. 237-246, May 1996.

[23] 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 on Mini and Microcomputers*, 17(1):42-51, 1995.

[24] D. Verma, H. Zhang, and D. Ferrari, “Delay jitter control for real-time communication in a packet switching network”, *Proceedings of IEEE TriCom’91*, pp. 35-43, April 1991.

[25] H. Zhang and D. Ferrari, “Rate-controlled service disciplines”, *Journal of High Speed Networks*, vol. 3, no. 4, pp. 389-412, 1994.

[26] L. Zhang, “VirtualClock: a new traffic control algorithm for packet switching networks”, *ACM Transactions on Computer Systems*, vol. 9, no. 2, pp. 101-124, May 1991.

[27] Y. Chen and J. S. Turner, “Dynamic Queue Assignment in a VC queue Manager for Giggabit ATM Networks”, *Proceedings of the IEEE ATM Workshop*, 1998.

[28] V. N. Rao and V. Kumar, “Concurrent Access to Priority Queues”, *In IEEE transactions of Computers*, vol.37, Dec 1988.

[29] S. K. Prasad, S. I. Sawant, “Parallel Heap: A Practical Priority Queue for Fine to Medium-Grained Applications on Small Multiprocessors”, *The Seventh Symposium on Parallel and Distributed Processing*, San Antonio, TX 1995.