An Efficient FPGA-Based Pipelined Binary Heap

***Abstract*—A heap can be used as a priority queue implementation for a wide variety of algorithms such as routing, anomaly prioritization, shortest path search, and scheduling. A parallel implementation of a heap is expected to offer higher throughput and quality of service than a serial implementation. Several parallel hardware solutions have been proposed, but they incur significant hardware cost by producing *holes* in a heap via parallel *insert-delete* operations. Holes result in an unbalanced, incomplete binary heap, which leads to longer response time, and they also waste hardware resources.**

In this paper, we demonstrate the significant consequences of holes in terms of hardware and response time. We propose a FPGA realization of a pipelined binary heap that addresses holes. Our primary proposed technique, a hole minimization technique, forces the tree structure to be a complete binary heap. This technique reduces the hardware cost by 37.76% in terms of number of lookup tables (LUTs) and average response time by 14.48%. It allows the proposed design to take $O(1)$ time for *min-delete* and *insert* operations by ensuring minimum wait time between two consecutive operations. Our proposed heap makes two other contributions. (1) Sharing hardware between two consecutive pipelined levels reduces hardware cost even further by 7.70%. (2) We introduce a *replacement* operation, which reduces the response time by 30.36%. As a result, the proposed heap incurs 78.50% less overhead while achieving similar performance compared to existing techniques.

***Index Terms*—Parallel Binary Heap, FPGA, Priority Queue, Hole Minimization**

I. Introduction

A priority queue is a popular data structure used for various applications such as routing, anomaly prioritization, shortest path search, and scheduling [6, 7, 19]. It is a data structure in which (1) each element has a priority, and (2) a dequeue operation removes and returns the highest priority element from the queue.

The concurrent priority queue proposed by Vipin *et. al* [20] leads a new direction of parallel heap processing. It shows how insert and delete operations can be processed from the root node; in particular, the authors formulate the insertion path from the root to the last node. Although the paper does not show fine-grained cycle operations during the concurrent process, it theoretically shows the complexity of concurrent heap processing by addressing the locking meachnism at system-level implementation.

Moving forward from the concurrent software priority queue, there are several parallel hardware implementations of a priority queue [2, 4, 11–13, 17, 18] for handling a large volume of elements, though they have individual drawbacks. The *Systolic Arrays* and *Shift Registers* based approaches [4, 18], for example, are not scalable and require $O(n)$ comparators for n nodes. The FPGA-based pipelined heap presented by Ioannou *et. al* [12] is scalable and can run for 64K nodes, but it stalls parallel insert-delete operations. Calendar queues [17] incur significant hardware cost when supporting a large priority set. A hybrid priority queue architecture proposed by [14] operates on $O(log\ n)$ time.

Moreover, all of the existing works on hardware-based parallel priority queues share a common problem: they do not address *holes* created as a result of parallel insert-delete operations. Holes occupy storage elements but do not have valid data. Retaining them not only introduces additional overhead, but it also leads to an unbalanced tree, which may result in longer response time. Although holes can be eliminated by rebalancing the tree, doing so requires a non-negligible number of cycles, during which no further insert or delete operations can be processed.

Toward this end, we demonstrate the signficant, negative consequences of holes with respect to hardware cost and response time and shows how holes can be minimized. We propose an efficient, parallel FPGA-based binary heap with the following summarized contributions:

- **Hole minimization:** The proposed approach minimizes holes created by parallel operations on a priority queue. Our hole minimization technique reduces hardware cost by 37.76% in terms of number of lookup tables (LUTs) and average response time by 14.48%.
- **Hardware sharing:** Sharing hardware between two consecutive pipelined levels reduces the hardware cost. The hardware sharing technique contributes a 7.70% cost reduction.
- **Replacement operation:** Upon detection of a min-delete operation immediately followed by an insert operation, a *replacement* operation substitutes these two operations. This method reduces average response time by 30.36%.

The rest of this paper is organized as follows: Section §II contains an overview of literature related to this work and discusses holes in parallel implementations in more details. Section §III presents our proposed design with hole minimization. We also propose several optimization techniques for performance efficiency and hardware minimization. Section §IV describes the implementation results along with performance comparisons with existing designs. Section §V concludes the paper and identifies some directions for future research.

II. Background and Related Work

A priority queue is a data structure that maintains a collection of elements using the following set of operations by a minimum priority queue Q:

- **Insert:** Insert a number into Q, provided that the new list should maintain the priority queue.

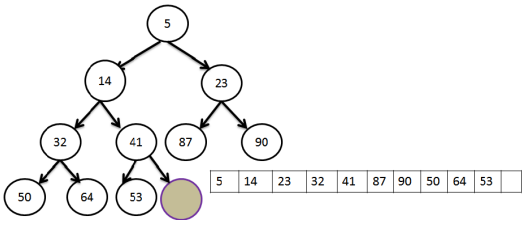

Fig. 1: Binary min heap with its array representation.

- **Min-Delete:** Find the minimum number in Q and delete it from Q. After deletion, the property of priority queue should be kept unchanged.

A priority queue can be implemented by using a binary heap data structure. A min-heap is a complete binary tree H such that the data contained in each node is less than or equal to the data in that node's children. Figure 1 shows the binary min heap H. The root of H is $H[1]$. Given the index i of any node in H, the indices of its parent and children can be determined in the following way:

$$\begin{aligned} parent[i] &= \lfloor i/2 \rfloor \\ leftChild[i] &= 2i \\ rightChild[i] &= 2i+1 \end{aligned}$$

The insert algorithm on the binary min heap H is as follows:

1) Place the new element in the next available position i.e. i in H.
2) Compare the new element, $H[i]$, with its parent, $H[i/2]$. If $H[i] < H[i/2]$, swap their values.
3) Continue this process until either the new element's parent is smaller than or equal to the new element, or the new element reaches the root, $H[1]$.

Figure 2 shows the new heap structure after inserting 18 into the original heap from Figure 1.

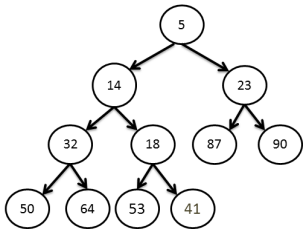

Fig. 2: New heap structure after inserting 18.

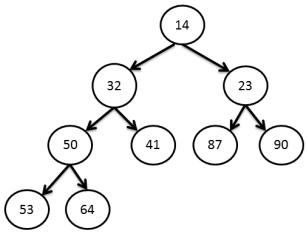

Fig. 3: New heap structure after a *min-delete* in the original heap from Figure 1.

The min-delete algorithm is as follows:

1) Return root element, $H[1]$.
2) Replace the root with the last element at the last level i.e. $H[i]$.
3) Compare the new root with its children. Unless the root is smaller than its children, swap the root and its min child.
4) Continue swapping for each level by comparing $H[i]$ with $H[2i]$ and $H[2i+1]$ until the parent becomes less than its children or reaches the leaf node.

Figure 3 depicts the heap structure after a min-delete operation on the original heap from Figure 1. We can see that 5 was the previous root element. The heap is restructured according to the min-delete algorithm.

A. *Pipelined Binary Heap*

As a binary heap accumulates more levels of nodes due to insertions, processing incoming operations becomes slower. This slowdown can be avoided by pipelining operations in the heap. Multiple operations can then be handled in parallel, resulting in a speedup.

To clarify using an example, suppose a pipelined binary heap receives a sequence of two *inserts*, and the last level can hold two more elements. These two indices share the same parent index. Assume that the placement of a new element and the comparison with swapping each take a clock cycle. Upon encountering the first operation, the new element is inserted at the last level. While the first inserted element is compared to its parent and swapped if necessary, the element from the second operation is inserted. The second element is compared and potentially swapped with the (updated) parent, while the first element is handled at the next level. From this example, a pipelined binary heap can ideally process one operation at each level.

Difficulties arise when processing insert and delete operations in parallel. *Inserts* and *min-deletes* normally start at the last and first level of the heap respectively and progress their tasks in opposite directions. Due to the opposite flows of the two types of operations, doing them in parallel may cause an inconsistency in the heap. Although the insert algorithm can be modified to operate in the same direction as *min-delete*, as we would discuss in Section §III, parallel *insert-deletes* (a parallel *insert-delete* is an *insert* immediately followed by a *min-delete*) nevertheless can create *holes* in the heap.

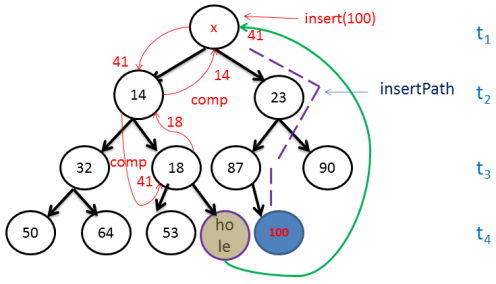

Fig. 4: A hole is created from parallel insert-delete operation

A hole occupies a node in the heap, but it does not contain valid data. Figure 4 demonstrates how a hole is created during

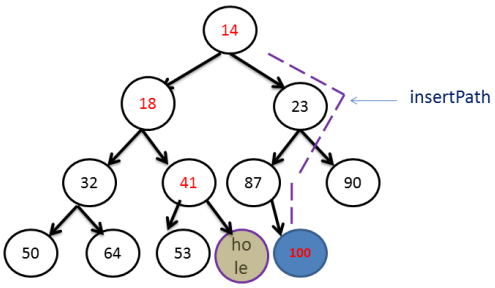

Fig. 5: The result of the parallel insert-delete operation

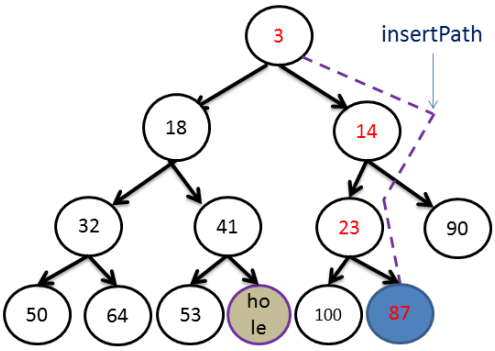

Fig. 6: Inserting 3 after the parrallel *insert-delete*

a parallel insert-delete operation on the heap from Figure 2. At time t_1, the insert operation with element 100, denoted by $insert(100)$, is encountered. Again, we would discuss the algorithm for top-down insertion later, but we can see that the new element 100 will be at the last node of the last level, which is at index 12, in four cycles. After one clock cycle of *insert*, *min-delete* is encountered at t_2. But, at that time, $insert(100)$ is being processed at the second level, so the last element is still the 11^{th} node. The min-delete operation will create a hole at $H[11]$. Eventually, when $insert(100)$ finishes, element 100 will occupy the position of $H[12]$, but $H[11]$ will become empty. Figure 5 illustrates the aftermath of the insert-delete operation.

When another *insert* is encountered, instead of the hole at $H[11]$, $H[13]$ will be occupied instead, as shown in Figure 6. This is because the last occupied element is $H[12]$, and the pipelined heap typically inserts a new element in the next available node following the last occupied one.

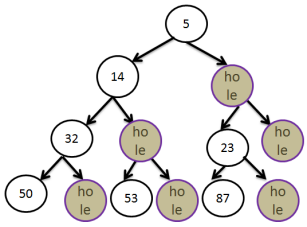

Fig. 7: Bad case scenario for hole creation.

Holes unnecessarily increase the depth of the heap, which delays response time for further operations. The most significant negative impact, however, is that they waste storage resources. In a bad case scenario, shown in Figure 7, hardware cost double.

Of course, in serial priority queues, the possibility of hole creation is nonexistent. The serial queue must process an *insert* or *min-delete* before beginning the next, so the last node is updated before any *min-deletes*.

We discuss more about parallel insert-delete operations and holes in Section §III.

B. Related Work

Several authors have theoretically proven that the parallel heap is an efficient data structure for implementing a priority queue. Prasad *et. al.* [8] theoretically illustrate that this data structure requires $O(p)$ operations with $O(\log n)$ time for $p \leq n$, where n is the number of nodes and p is the number of processors used. This idea is designed for the EREW PRAM shared memory model of computation. The many-core architecture by [10] in GPGPU platform provides multi-fold speedup. Another theoretical approach [3] deploys a pipeline or tree of processors ($O(\log n)$, where n is the number of nodes). The implementation of this algorithm [9] is expensive for multi-core architectures.

There have been several hardware-based priority queue implementations described in literature [2, 4, 5, 16–18, 21, 22]. Pipelined hardware implementations can attain $O(1)$ execution time [16, 21]. However, due to several limitations such as cost and size, most hardware implementations do not support a large number of nodes to be processed. Thus, these implementations are limited in scalability. The *Systolic Arrays* and the *Shift Registers* [4, 18] based hardware implementations are well known in literature. The common drawback of these two implementations is the usage of a large number of comparators ($O(n)$). These comparators are responsible for comparing nodes from different levels with $O(1)$ step complexity. For the *Shift Register* [4] based implementations, when new data comes for processing, it is broadcasted to all levels. This requires global communicator hardware, which connects to all levels. The implementation based on *Systolic Arrays* [18] needs a larger storage buffer to hold preprocessed data. These approaches are not scalable, and they require $O(n)$ comparators for n nodes. To overcome the hardware complexity, [1] implements a recursive processor where hardware use is drastically reduced by compromising execution time. Bhagwan and Lin [2] design a physical heap such that commands can be pipelined between different levels of the heap. The authors in [17] give a pragmatic solution of the *fanout* problem mentioned in [5]. The design presented in [15] is very efficient in terms of hardware complexity but very slow in execution ($O(\log n)$), as it is implemented using hardware-software co-design.

For the FPGA-based priority queue implementation, Kuacharoen *et. al* [13] implement the logic presented in [5] by incorporating extra features to ensure that the design would act as a real-time operating system task scheduler. The major limitation of this work is that it deals with very few priority levels and a small number of items in the queue at any given time. A hybrid priority queue is implemented by [11], and it

ensures high scalability and high throughput. The ASIC-based pipelined heap presented by Ioannou *et. al* [12] is scalable and can run for 64K nodes without compromising performance, but it takes at least three clock cycles in the average case to complete a single stage. Kumar *et. al* [14] propose a new hybrid priority queue architecture that can be managed in hardware and/or software; its main drawback is that it operates on $O(\log n)$ time.

However, all aforementioned designs never address the *holes* created from parallel insert-delete operations.

While our proposed design is similar to [20], the latter a software approach as opposed to hardware. It, again, does not address holes created by parallel insert-delete operations. Instead, it circumvents the issue by using locking.

In the next section, we present our proposed design with hole minimization, including several optimization techniques for performance efficiency and hardware minimization.

III. Efficient Parallel Heap Implementation

Like an array representation, a heap can be represented by hardware registers or FPGA latches. An array of latches can virtually represent each level of a heap. The number of latches at each level is $2^{\beta_k - 1}$, where β_k is the k^{th} level, assuming that root is in level 1. Figure 8 shows how latches represent levels. In this example, the root node is stored in L_1; two elements are stored in the next level, L_2; the level after that is L_3, storing four elements; and the last level, L_4, has three elements, although it can have a maximum of eight.

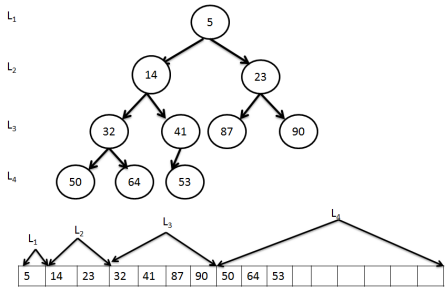

Fig. 8: Storage in FPGA of different nodes in binary heap

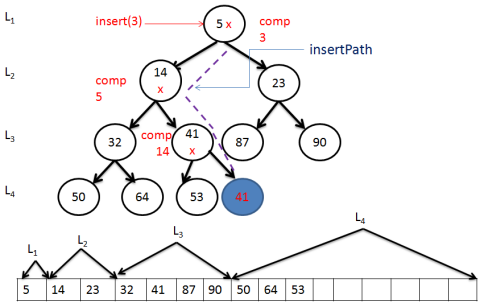

Fig. 9: Insert path

A. Insert Operation

The *insert* operation is intimated from the last available node of a heap. However, if a bottom-up insertion is done

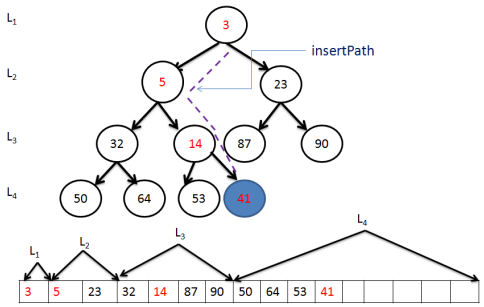

Fig. 10: Container of latches L after insertion completes.

in parallel with other operations such as *min-delete*, it may cause an inconsistency in the heap. To clarify, let us consider the heap in Figure 8. Suppose we insert element 3 into the heap and then immediately delete one element. Let us assume nodes at each level get updated in a single clock cycle. That means a total of four clock cycles is required to complete only an insert operation in this situation. Before it completes, if a *min-delete* comes, that operation has to wait up to four clock cycles to avoid deleting a wrong element from the root (5 in this case). Thus, it is incumbent to insert from the root and go down. However, we need to know the exact path for the newly inserted element, otherwise the tree will no longer hold the conditions of a complete binary tree. We adopt an algorithm presented by Vipin *et. al* [20] in our design. The algorithm is as follows:

1) Let k be the last available node where a new element can be placed. Let j be the first node of the last level. Then, the binary representation of $k - j$ will give the path for the insertion.
2) Let $k - j = B$, and represent B in binary form $b_{\beta-1}b_{\beta-2}\cdots b_2 b_1$. Starting from root, scan each bit of B starting from $b_{\beta-1}$;
   - **if** $b_i == 0$ ($i \in \{\beta-1, \beta-2, \cdots, 2, 1\}$), **then** go to left
   - **else** go right

Figure 9 shows the insert path for a new element 3. In this case, the node at index 11 should be filled up. The first node of the last level is at index 8. So, $11 - 8 = 3$, which can be represented as 011 in binary. Starting from root, the path should be $root \rightarrow left \rightarrow right \rightarrow right$, as illustrated by the Figure 9. Figure 10 represents the binary tree and latches after this insert operation.

B. Min-Delete Operation

One conventional approach for deleting an element from a heap is to delete from the root and replace it with the last element. There are two difficulties here:

1) For sequential operation, this method works fine. Parallel execution of *insert-delete* (a *min-delete* that follows an *insert*), however, can create a hole.
2) Replacing the root with the last element of the heap requires an additional clock cycle to write that element into the root node. Additionally, we need to compare three

elements: root and its two children, or any node and its children. From a hardware perspective, it is cost-efficient to compare two elements rather than three. Moreover, 3-element comparisons cause a longer path delay.

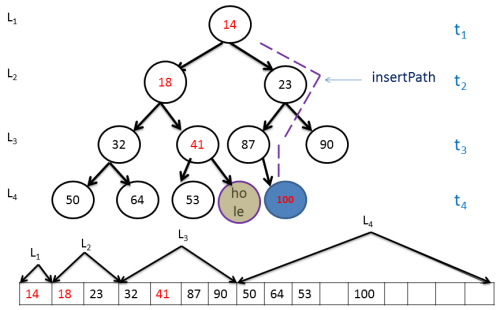

Fig. 11: Container of latches L after parallel *insert-delete*.

We already show the first issue in the previous section, when we delete on the next cycle after inserting 100. Figure 11 illustrates the result from the earlier example again, with the addition of latches.

For the general case, let us assume that an insert operation comes at time t_i and a min-delete operation comes at t_j, where $i, j = 1, 2, 3, \cdots$ and $j > i$. All operations take one clock cycle at any level to complete their respective tasks at that level. Only a single node, if any, gets modified for all levels. Any insert-delete combination will create a hole if $(t_j - t_i) < \beta\tau$, where β is the current depth of the heap and τ is the cycle time per level.

The naive approach to avoid a hole from an insert-delete combination is to stall *min-delete* from entering the pipeline until *insert* completes. After insertion completes, the last element will update to the newly filled node. Thus, when the delete operation begins, the heap will correctly identify the new node as the last element to replace the root with instead of the node before. However, this method turns every *insert* that is followed by a *min-delete* into a sequential operation. The number of cycles the delete operation must wait for the insert operation becomes the heap's current depth. In the worst-case scenario, where a binary heap of many levels encounters a sequence containing only insert-delete combinations, performance degrades significantly.

To solve the first issue, we propose a hole minimization technique. With this technique, the holes are removed while an insert operation is processed. We check for the existence of a hole by checking special-purpose registers, which each stores the location of a hole. Each register is of bit size $\beta_{k_{max}}$. If a hole exists, we fill it with the new element. We apply the insert algorithm, except we modify it so that data will be inserted at the position of a hole instead of the last available node. In the case of multiple holes, we choose the insert path to the last hole for implementation ease. Details of this algorithm are described in the next subsection.

To address the second issue, we intentionally avoid the root replacement by the last element. We delete root first and keep it as it is. Then, we fill the root with its least child and follow the min-delete algorithm described in the previous section. This way, we can save one cycle and minimize the path delay. Because this method does not guarantee that the last node to be empty, it may create a hole for any *min-delete* regardless whether the operation follows an *insert*. However, we can still remove the holes using the same hole minimization technique.

C. Insert-Delete Logic Implementation

Insert-delete logic is implemented to realize the proposed priority queue. The logic handles the *insert* and *min-delete* operations for each level.

Algorithm 1 $Insert - Delete(data, opcode)$

```
if (opcode == INSERT) then
    counter = counter + 1
    (insert_path, holeCounter) = findPath(counter, holeCounter)
    for (i = 1 to β) do
        index = indexCal(insert_path, i, index)
        if (data < H[index] then
            swap H[index] and data
        end if
    end for
else
    H[1] = NULL
    while (leftChild[del_index] ≠ NULL && rightChild[del_index] ≠ NULL) do
        if (leftChild[del_index] < rightChild[del_index]) then
            H[del_index] = leftChild[del_index]
            del_index = del_index * 2
        else
            H[del_index] = rightChild[del_index]
            del_index = del_index * 2 + 1
        end if
    end while
    counter = counter − 1
    holeCounter = holeCounter + 1
    holeReg[holeCounter] = del_index
end if
```

Algorithm 2 $findPath(counter, holeCounter)$

```
if (holeCounter > 0) then
    holeCounter = holeCounter − 1
    return (holeCal(holeCounter), holeCounter)
else
    leaf_node = findNode(counter)
    return (counter − leaf_node, holeCounter)
end if
```

Figure 12 illustrates the top-level architecture of the priority queue employing the insert-delete logic. The *counter* maintains the total number of elements present in the heap. It is incremented by one for an insert operation and decremented

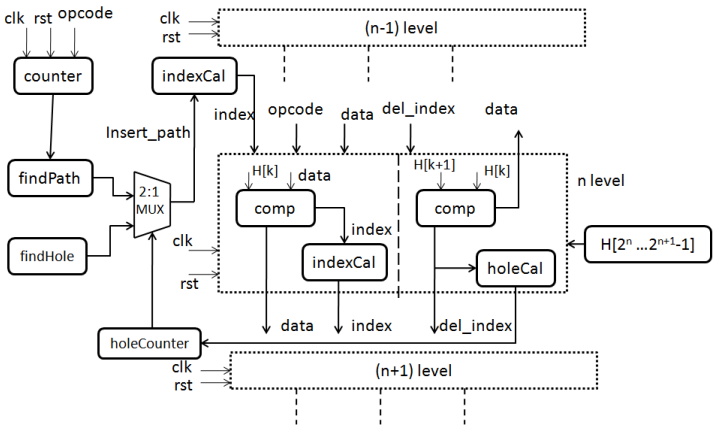

Fig. 12: Top Level Architecture of *insert-delete*

Algorithm 3 $findNode(counter)$

1: **for** $(i = 1; 2^i \leq counter; i = i + 1)$ **do**
2: $leaf\_node = i + 1$
3: **end for**
4: **return** $leaf\_node$

Algorithm 4 $holeCal(holeCounter)$

1: **return** $holeReg[holeCounter]$

by one for a min-delete operation. Modifying the existing path-finding algorithm proposed by [20], we consider *holeReg* during insertion to obtain the insert path. The *holeReg* contains holes created from min-delete operations. We maintain *holeCounter* to identify a valid hole with *holeReg*. The *indexCal* block finds the insert path. The heap node at *index* is accessed and compared to the present data. Based on the comparison, either the present data updates the node and the previous node data is passed to the next level, or the node remains unchanged and the present data is passed to the next level.

We also maintain *del_index* to find the last deleted node. For example, initially, *del_index* is 1, meaning the root is empty. The comparator finds the min element between $H[del\_index * 2]$ and $H[del\_index * 2 + 1]$, and that min element replaces $H[del\_index]$. Now, *del_index* updates to the index of min element. Again, the comparator finds the min child from the new index and replaces the node at the new index with that of the min element. Each time, *holeCal* determines whether a valid child exists for $H[del\_index]$. If not, then *holeCounter* is incremented by 1, and *holeReg* is updated with *del_index*.

Algorithm 5 $indexCal(insert\_path, level, index)$

1: **if** (bit_{level} of $insert\_path == 0$) **then**
2: **return** $index = 2*index$
3: **else**
4: **return** $index = 2*index + 1$
5: **end if**

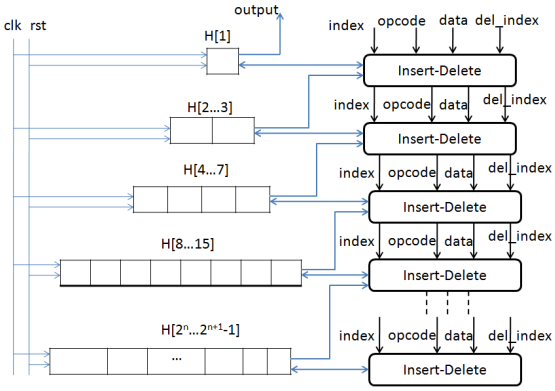

Fig. 13: Pipeline design overview

In this manner, we maintain *hole*.

Algorithm 1 presents the insert-delete parallel algorithm. We use a 2:1 multiplexer to select the path based on the value of *holeCounter*. The logic for *findPath* is illustrated in Algorithm 2. Within *findPath*, the *findNode* logic calculates the first node of the last level. That node can be determined using the mathematical expression $log(n)$ rounded down, where n is the number of elements in the binary heap. Algorithm 3 presents the hardware implementation of this logic. The *indexCal* block uses the output of *findPath* as well as the current level and index to generate the next index for the insertion path. Algorithm 5 illustrates this logic. Functionally, *holeCal* is an implementation of a stack register. Its return value is presented at Algorithm 4.

D. Pipeline Design

To achieve high throughput, we need to start one operation before completing the previous so that multiple operations can be in progress at the same time. Figure 13 illustrates the basic pipeline architecture of our binary heap. Each level executes *insert* or *min-delete* based on signal *opcode*, and only one operation can be executed there at any given time t. It sends *data* and *opcode* to the next level to perform. All levels, except the first, contain the same logic hardware.

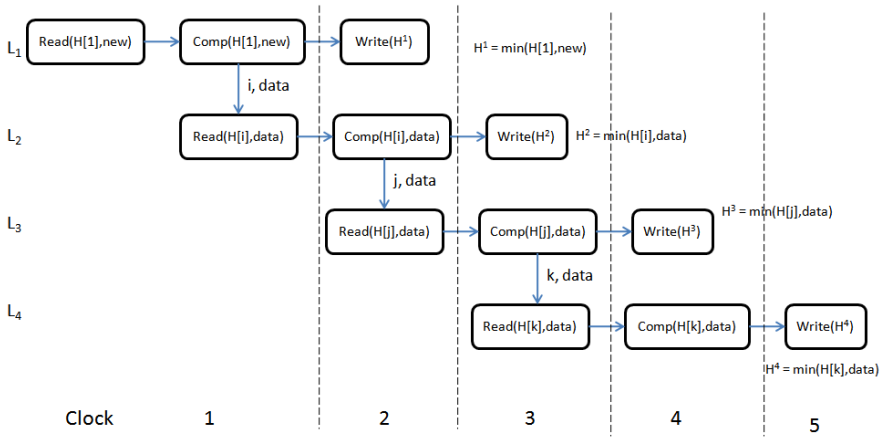

Fig. 14: Parallel insert operation: illustrates operations at each level at each clock

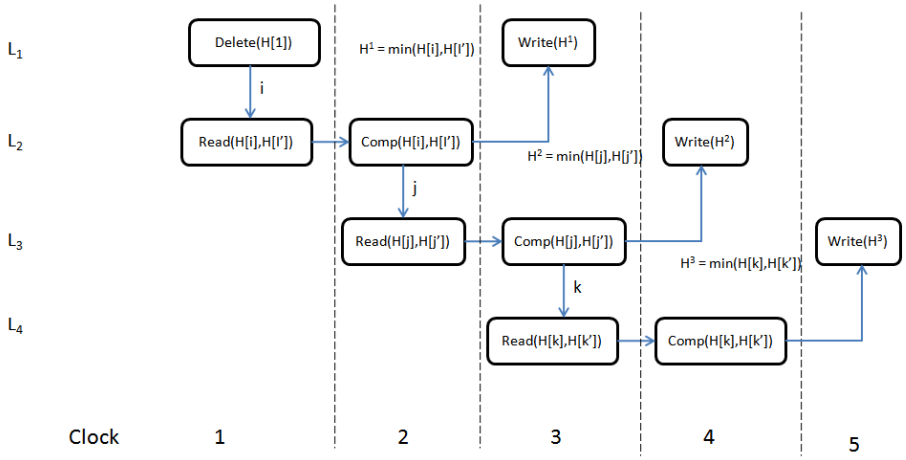

Fig. 15: Parallel delete operation: illustrates operations at each level at each clock.

Within an operation, each level has to perform three tasks in this order: *read*, *compare*, and *write*. Each task individually takes one clock cycle to execute, meaning each insert or min-delete operation performs three cycles of meaningful work at a level. During *comp*, the level sends the data and opcode to the next level so that the latter can begin its tasks. *Comp* for one level and *read* for the next level is started at positive edge and negative edge of the same clock cycle respectively. This way, the next level can read the data and opcode generated from the previous level at the next clock cycle while the previous performs *write*.

Figure 14 illustrates the pipeline flow for a parallel insert operation, broken down to the three tasks.

Cycle t**:** At any level, L_i, suppose *read* performs at this clock cycle.

Cycle $t+1$**:** *Comp* completes in L_i, generating the next *index* to be read by the next level, L_{i+1}. L_{i+1} performs *read* within the same clock cycle.

Cycle $t+2$**:** L_i performs *write*. L_{i+1} finishes *comp*, which generates the index to be read by L_{i+2}.

Cycle $t+3$**:** L_i can accept a new operation. L_{i+1} performs *write*. L_{i+2} completes *comp*, which generates the index for L_{i+3}.

The pattern repeats for subsequent levels until the insert operation finishes. Effectively, this results in a *write* at each clock cycle after an initial latency of two clock cycles at the first level.

Figure 15 illustrates a parallel min-delete operation. In all levels except the first, *read* and *write* behave similarly as with the insert operation. However, level L_i does not perform *write* at $t+2$ but at the following cycle, $t+3$, instead. This is because in our min-delete algorithm, the last element of the last level does not replace root after the root data is removed, so the min child has to move up to the empty node on each level. As the min child is determined by *comp* in the next level L_{i+1}, L_i has to wait one cycle so that the comparison result from L_{i+1} can be written to L_i. That means L_i suffers a temporary hole at $t+2$. This hole is filled as L_{i+1} writes to L_i at $t+3$. However, L_{i+1} also suffers a temporary *hole*, but it is compensated by the next level and so on.

E. Optimization Techniques

In this section, we present two optimization techniques: (1) sharing hardware in consecutive levels to reduce hardware cost

even further, and (2) introducing a *replacement* operation to reduce response time.

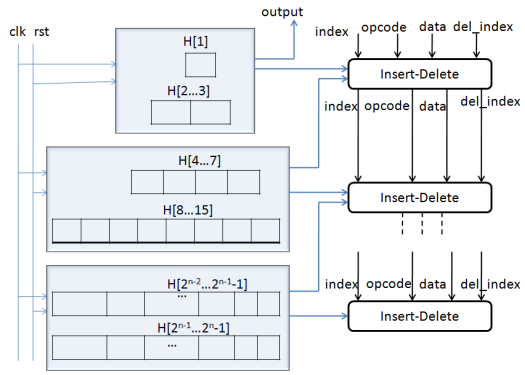

Fig. 16: Sharing *Insert-Delete* hardware between two consecutive levels.

1) Hardware Sharing: We have seen in Figure 15 that for each level, there is an *idle* state. For example, while level L_2 becomes idle at clock 3 after performing *comp* at clock 2. This idle state occurs because L_2 is waiting on the result from level L_3 as the latter is performing *comp*. Eventually, L_2 performs *write* at clock 4 after the data becomes available by L_3.

While a level is in an idle state, no operation could be performed at that level at that time. In general, for any given time t, the level L_i cannot be completed if level L_{i+1} cannot finish *comp* at $t+1$. In this situation, we can share hardware between the levels L_i and L_{i+1}, as shown in Figure 16.

2) Replacement Technique: There are four possible sequences of operations: *Insert-Only* (I), *Delete-Only* (D), *Insert-Delete* (ID) and *Delete-Insert* (DI). The *Insert-Only* sequences ($I\ I\ \cdots I$) take single clock cycle to operate at each stage. While our insert-delete logic minimizes holes created from $IDID\cdots DIDI\cdots$ sequences, we can optimize for scenarios where the insert or delete request immediately follows after the cycle where the other is encountered. Here, we introduce a *replacement* operation that does not cause holes. The algorithm of this operation is as follows:

1) Let X be the root element and Y be the element to insert from a request sequence $ID\cdots DI$.
2) Delete $min(X,\ Y)$
3) $H[1] \leftarrow min(X,\ Y)$
4) Continue this replacement for each level by comparing $H[i]$ with $H[2i]$ and $H[2i+1]$, until the parent becomes less than its children, or it reaches to the leaf node.

The time complexity for these sequences is exactly the same as for *Insert-Only* where no holes are generated. Our replacement operation is ignored if the second request occurs more than one cycle after the first. Thus, this optimization can work in parallel with our hole minimization technique.

F. Case Studies

1) Root Replacement by Last Element: Some authors replace the root with the last element at *min-delete* [2, 5] for their proposed heaps. As we discuss before, though it may seem

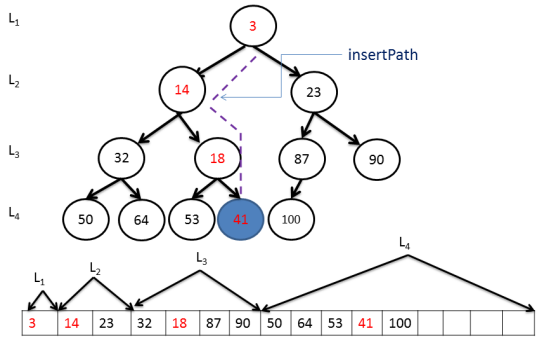

Fig. 17: Using our hole minimization technique, the hole that exists between 53 and 100 in Figure 5 gets filled with 41 after inserting 3 into the heap.

that a hole will not be created with this approach, this is not the case. Figure 4 demonstrates the hole being created during a parallel insert-delete operation on the heap from Figure 2 using those authors' operations. When inserting 100, the insert path goes through $H[1] \rightarrow H[3] \rightarrow H[6] \rightarrow H[12]$. At the time the *insert* is being procesed, the *min-delete* comes and replaces the root with the element at node $H[11]$, creating a hole at node $H[11]$, as shown at Figure 5. We consider the node at $H[11]$ as a hole because it does not contain any data when its next neighbor, $H[12]$, does. This situation could be avoided if the root was replaced by $H[12]$. However, that requires having *min-delete* wait until *insert* completes. So, any consecutive *insert-delete* will lead to hole creation with the root replacement mechanism for other heaps.

2) How Holes are Minimized: When a hole is created in our proposed heap, its location is saved into a hole register. The inserted node goes to that location instead of the last available location. For example, with the heap from Figure 11, the hole index 11 is stored at a hole register. Now, let us insert 3 into the heap. The last available insert node location would be index 13 in a naive pipelined heap implementation. But instead, we use the hole index 11 for constructing the insert path. According to our algorithm, the insert path is $H[1] \rightarrow H[2] \rightarrow H[5] \rightarrow H[11]$. This hole is filled by the element 41 during the insert operation, as shown in Figure 17. Meanwhile, for the naive implementations, as the index 13 would be considered for the insert path, their path would be $H[1] \rightarrow H[3] \rightarrow H[6] \rightarrow H[13]$, which does not gaurantee the hole minimization.

3) Worst Case Scenario: To fill any possible number of holes, $2^{\beta_{k_{max}}-1}-1$ registers are required, where $\beta_{k_{max}}$ is the maximum depth of the heap. That is because the last level can have up to $2^{\beta_{k_{max}}-1}$ nodes, and of course, no holes exist at the current level if all nodes at that level are empty. However, dedicating this many registers to address the worst case scenario wastes hardware storage, which is the very problem we are addressing. Moreover, this number is only theoretical.

For a 32-max-level heap, we believe that 512 registers is enough based on our evaluations using standard benchmarks. We find that the worst empirical case scenario creates 512 holes

while the average case creates 10 to 12 holes. In a theoretical event where 512 registers cannot prevent all holes from being removed, our heap still saves hardware resources compared to other parallel hardware implementations of this maximum depth.

As with other implementations, the presence of holes in our heap doesn't invalidate the correctness of the heap itself.

IV. Evaluation

TABLE I: Variation in frequency and throughput with number of levels.

Number of Levels (β)	Frequency (f) (MHz)	Throughput (Π) (GB/Sec)
4	318.8	1.27
8	232.8	1.85
10	212	2.12
12	210	2.52
16	207.2	3.31
20	173.4	3.46
24	171.6	4.10

TABLE II: Hardware cost results. LUT stands for look-up table.

Design	Comparator	Flip-flop	LUT	Slice
Without hole minimization	32	1400	3165	4870
With hole minimization	32	810	1970	2870
With hardware sharing	16	750	1840	2730

TABLE III: Response time results.

Design	Response Time (ms)
Without hole minimization	2010
With hole minimization	1720
With replacement	1337

We simulate the proposed design with ISim and implement the proposed design on the Xilinx Sparttan6 XC6SLX4 hardware platform, using 32MB on-chip memory. We use Verilog and Python script to simulate the data path. Through Python, we generate the test bench, which is sent to the ISim simulator. The instruction for *insert/delete* is executed based on the 1-bit opcode value. Internal logic of the FPGA determines the *replacement* operation based on two consecutive opcodes. Unless otherwise stated, a random sequence of *insert* and *delete* operations is used.

A. Sensitivity Analysis

We explore how the results (frequency and throughput) change with the number of levels. Throughput, Π, is calculated as:

$$\Pi = \frac{\omega \times f}{\chi} \quad (1)$$

where ω is the bit length, f is the clock frequency, and χ is the number of clock cycles required to compute one operation. We use the number of levels (β) and bit length (ω) interchangeably. The number of elements in the heap is $2^{\omega} - 1 = 2^{\beta} - 1$.

From Table I, we found that the obtained clock frequency is not constant but instead inversely proportional to bit length. This is expected since more time is required to store larger data. We obtain a maximum frequency of 318.8 MHz from $\beta = 4$, and minimum frequency of 171.6 MHz from $\beta = 24$. An increase in the number of levels leads to slower clock frequency but higher throughput. As we have designed a fully pipelined architecture, the output can be obtained in each clock cycle as shown in Figure 15. For the remaining experiments, we use $\beta = 24$.

B. Hardware Cost and Response Time

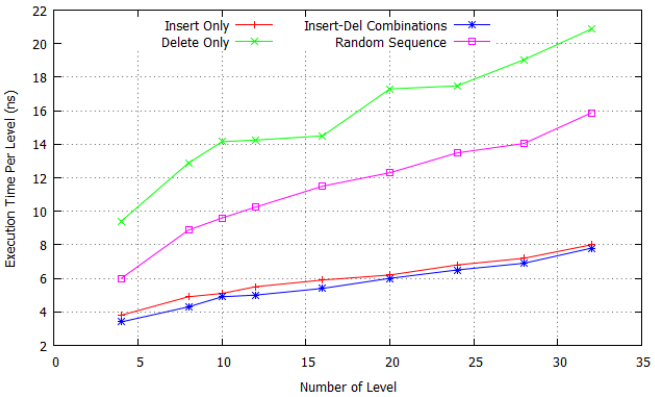

Fig. 18: Execution time for different sequence of operations.

The hole minimization technique reduces both hardware cost and response time. The two optimization techniques further improve them. The hardware sharing technique contributes to the hardware cost reduction, and the *replacement* operation shortens the response time further.

Table II shows the results of the hardware cost measurements. Before applying the hole minimization technique, the number of comparators, flip-flops, LUTs, and slices is 32, 1400, 3165, and 4870 respectively. Applying the technique reduces these numbers by 0.00%, 42.14%, 37.76%, and 41.07% respectively. The hardware sharing technique reduces them even further by 50.00%, 3.70%, 6.60%, and 4.88% respectively.

The hole minimization technique also improves response time, reducing it from 2010 to 1720, which corresponds to 14.48% reduction, as shown in Table III. Introducing the *replacement* operation further shortens the response time by 30.36%.

The impact of the *replacement* operation depends on the sequence of operations. Figure 18 compares the response time according to the sequence of operations. When only insert operations come, there will be no holes created, giving us the second-best response time. The best case is when insert-delete operations come in an alternative fashion. In this case, the replacement operations substitute for all pairs of insert-delete operations. The worst case is when only delete operations come right after only insert operations are processed. In this case, none of the operations can benefit from the replacement operations. The response time of a random sequence is between the best and worst cases.

TABLE IV: Hardware cost and performance comparison with previous works. n denotes the number of nodes.

Design	Comparator (κ)	Flip-flop (F)	SRAM (M)	LUT	Max Frequency (f) (MHz)	Throughput (II) (GB/Sec)	Execution Time	Complete Tree ?
[18]	2^{β}	$2^{\beta+1}$	0	8560	-	-	$O(1)$	Yes
[11]	$2 \times \beta$	$2^{\beta+1}$	0	1411	-	-	$O(\log n)$	Yes
[14]	-	$2 \times \beta$	$2 \times \beta$	3161	-	-	$O(\log n)$	No
[12]	$2 \times \beta$	$2 \times \beta$	$2 \times \beta$	-	180	6.4	$O(1)$	No
[2]	$2 \times \beta$	$2 \times \beta$	$2 \times \beta$	-	35.56	4.8	$O(1)$	No
Proposed	$\frac{\beta}{2}$	β	β	**1840**	**171.6**	**4.10**	$O(1)$	**Yes**

C. Comparison with Existing Techniques

The three techniques we propose in this paper (hole minimization, hardware sharing, and replacement operation) offer a more efficient implementation of a priority queue compared to previous works. Table IV compares the hardware cost and response time with existing techniques. Since different designs address different issues and are implemented on different platforms, we make the comparison based on complexity analysis.

When compared to reference [18], our design offers significantly lower overhead while achieving a similar performance level. The number of LUTs used to implement a priority queue is reduced by 78.50%. The complexity of the execution time of both designs is $O(1)$. However, while reference [18] requires 2^{β} comparators and $2^{\beta+1}$ flip-flops, our design incurs much less overhead ($\frac{\beta}{2}$ and β respectively), offering better scalability.

Kumar *et. al* [14] do not report neither throughput nor frequency, our design is better than their design in terms of LUT count. Moreover, it operates on $O(\log n)$ time.

Reference [11] offers seemingly less overhead, but the result is, in fact, *underestimated*. While reference [11] is a hybrid approach combining hardware and software, the overhead in this table accounts only for hardware. In addition, the response time of the hybrid approach is not scalable with the number of nodes.

References [2, 12] do not report the number of LUTs, but we can compare their hardware cost by complexity analysis. They require $2 \times \beta$ comparators, flip-flops, and SRAM, whereas our design needs only $\frac{\beta}{2}$ comparators and β flip-flops and SRAM. The hole minimization and hardware sharing techniques have achieved aggressive optimization in hardware cost. Both the designs perform better throughput; as [2] is implemented by using TSMC 0.35 micron CMOS standard-cell technology and [12] is implemented at ASIC platform.

The significantly higher frequency our proposed heap offers compared to Bhagwan and Lin's [2] is expected. The latter's insert algorithm checks whether at least one empty node exists somewhere in the left subtree of each node the insert operation propagates through. If there is no empty node, the insert path moves to the right subtree. This means that the insert operations will always fill empty nodes starting from the left, moving down. Although this algorithm does guarantee that holes are *eventually* minimized, it does not do so in the next immediate *inserts*. It also inherently creates more holes than it minimizes and creates an imbalanced tree. Both of these effects not only waste hardware resources unless the heap is full, but they also result in longer response time.

In comparison, insert operations in our heap always fill all empty nodes in each level before moving down to the next new level, and they fill any holes that are created immediately. Thus, our heap more efficiently uses hardware storage and has lower response time than [2].

V. Conclusions

In this paper, we propose a FPGA-based realization of a priority queue that is based on a binary heap. The heap is implemented in a pipelined fashion in hardware. Our design takes $O(1)$ time for all operations by ensuring minimum wait time between two consecutive operations. We propose two techniques for hardware optimization: hole minimization and hardware sharing between two consecutive levels. In addition, hole minimization can ensure a balanced heap structure, which contributes to a reduction in response time. The replacement operation, a different optmization technique we introduce, reduces response time further by substituting a pair of insert and delete operations with one replacement operation that does not create a hole. As a result, our design achieves a similar performance while offering markedly lower overhead.

The work presented in this paper leaves several directions for future research. For example, we use the binary heap where each node has two children at maximum. In many cases, each node may have n number of items [8]. In that case, each node of the heap will have n sorted data (except the last node). Each time an insert or delete operation occurs, we need to maintain the heap construction along with the sorted list of each node.

References

[1] Y. Afek, A. Bremler-Barr, and L. Schiff. Recursive design of hardware priority queues. *Computer Networks*, 66:52–67, 2014.

[2] R. Bhagwan and B. Lin. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings IEEE INFOCOM 2000, The Conference on Computer Communications, Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Reaching the Promised Land of Communications, Tel Aviv, Israel, March 26-30, 2000*, pages 538–547, 2000.

[3] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operations. *J. Parallel Distrib. Comput.*, 49(1):4–21, 1998.

[4] R. Chandra and O. Sinnen. Improving application performance with hardware data structures. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Workshop Proceedings*, pages 1–4, 2010.

[5] H. J. Chao and N. Uzun. A vlsi sequencer chip for atm traffic shaper and queue manager. *IEEE Journal of Solid-State Circuits*, 27(11):1634–1642, 1992.

[6] A. Das, S. Misra, S. Joshi, J. Zambreno, G. Memik, and A. N. Choudhary. An efficient FPGA implementation of principle component analysis based network intrusion detection system. In *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14*, pages 1160–1165, 2008.

[7] A. Das, D. Nguyen, J. Zambreno, G. Memik, and A. N. Choudhary. An fpga-based network intrusion detection architecture. *IEEE Trans. Information Forensics and Security*, 3(1):118–132, 2008.

[8] N. Deo and S. K. Prasad. Parallel heap: An optimal parallel priority queue. *The Journal of Supercomputing*, 6(1):87–98, 1992.

[9] A. V. Gerbessiotis and C. J. Siniolakis. Architecture independent parallel selection with applications to parallel priority queues. *Theor. Comput. Sci.*, 1-3(301):119–142, 2003.

[10] X. He, D. Agarwal, and S. K. Prasad. Design and implementation of a parallel priority queue on many-core architectures. In *19th International Conference on High Performance Computing, HiPC 2012, Pune, India, December 18-22*, pages 1–10, 2012.

[11] M. Huang, K. Lim, and J. Cong. A scalable, high-performance customized priority queue. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, pages 1–4, 2014.

[12] A. Ioannou and M. Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Trans. Netw.*, 15(2):450–461, 2007.

[13] P. Kuacharoen, M. Shalan, and V. J. Mooney. A configurable hardware scheduler for real-time systems. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, June 23 - 26, 2003, Las Vegas, Nevada, USA*, pages 95–101, 2003.

[14] N. G. C. Kumar, S. Vyas, R. K. Cytron, C. D. Gill, J. Zambreno, and P. H. Jones. Hardware-software architecture for priority queue management in real-time and embedded systems. *IJES*, 6(4):319–334, 2014.

[15] N. G. C. Kumar, S. Vyas, R. K. Cytron, C. D. Gill, J. Zambreno, and P. H. Jones. Hardware-software architecture for priority queue management in real-time and embedded systems. *IJES*, 6(4):319–334, 2014.

[16] K. McLaughlin, S. Sezer, H. Blume, X. Yang, F. Kupzog, and T. G. Noll. A scalable packet sorting circuit for high-speed WFQ packet scheduling. *IEEE Trans. VLSI Syst.*, 16(7):781–791, 2008.

[17] S. Moon, J. Rexford, and K. G. Shin. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Trans. Computers*, 49(11):1215–1227, 2000.

[18] S. Moon, J. Rexford, and K. G. Shin. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Trans. Computers*, 49(11):1215–1227, 2000.

[19] S. Pati, R. Narayanan, G. Memik, A. N. Choudhary, and J. Zambreno. Design and implementation of an FPGA architecture for high-speed network feature extraction. In *2007 International Conference on Field-Programmable Technology, ICFPT 2007, Kitakyushu, Japan, December 12-14, 2007*, pages 49–56, 2007.

[20] V. N. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Trans. Computers*, 37(12):1657–1665, 1988.

[21] H. Wang and B. Lin. Succinct priority indexing structures for the management of large priority queues. In *17th International Workshop on Quality of Service, IWQoS 2009, Charleston, South Carolina, USA, 13-15 July 2009.*, pages 1–5, 2009.

[22] X. Zhuang and S. Pande. A scalable priority queue architecture for high speed network processing. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 23-29 April 2006, Barcelona, Catalunya, Spain*, 2006.