An Efficient Hardware Implementation of Parallel Binary Heap

Monjur Alam*, Junghee Lee†, Zhe Cheng Lee‡ and Sushil K. Prasad§
*Georgia Institute of Technology, Atlanta, Georgia, Email: [malam31@gatech.edu]
†University of Texas at San Antonio, San Antonio, Texas, Email: junghee.lee@utsa.edu
‡Soteria Systems LLC, Atlanta, Georgia, Email: zlee@soteriasystemsllc.com
§Georgia State University, Atlanta, Georgia, Email: sprasad@gsu.edu

***Abstract*—A heap can be used as a priority queue implementation for a wide variety of algorithms like routing, anomaly prioritization, shortest path search, and scheduling. A parallel implementation of a heap is expected to offer higher throughput and Quality-of-Service (QoS) than a serial implementation. Parallel solutions have been proposed [18], [19], [20], but they incur significant hardware cost by producing *holes* in a heap via parallel *min-delete* operations. Holes result in an unbalanced incomplete binary heap, which leads to longer response time. In this paper, we propose a hardware realization of parallel binary heap. Our proposed technique makes three key contributions. (1) We provide a *hole* minimization technique that 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). It also allows the proposed design to take $O(1)$ time for min-delete and insert operations by ensuring minimum wait time between two consecutive operations. (2) Sharing hardware between two consecutive pipelined levels reduces hardware cost even further by 3.70%. (3) We introduce a *replacement* operation, which reduces the response time by 30.36%. As a result, the proposed technique incurs 78.50% less overhead while achieving the same performance level compared to existing techniques.**

***Keywords*-Parallel Binary Heap, Hardware Implementation, Priority Queue**

I. Introduction

A priority queue is a popular data structure used for various applications such as routing, anomaly prioritization, shortest path search, and scheduling [25], [21], [22], [23]. 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 concept is a basic component for scheduling used in most routers and event-driven simulators [8], [18].

There are several hardware-based implementations of a priority queue [8], [9], [14], [15], [18], [19], [20] for handling a large volume of elements. The *Systolic Arrays* and *Shift Registers* based approaches [14], [15], for example, are not scalable and require $O(n)$ comparators for n nodes. FPGA-based pipelined heap, presented by Ioannou *et. al* [18], is scalable and can run for 64K nodes without compromising performance, but it takes at least three clock cycles to complete a single stage. Calendar queues [8] incur significant hardware cost when supporting a large priority set.

Moreover, all of the existing works do not address *holes* generated by parallel *min-delete* operations followed by *insert* operations. Since holes occupy storage elements but do not have valid data, retaining holes introduces additional overhead. Holes also lead to an unbalanced tree, which may result in a long response time.

Toward this end, this paper proposes an efficient hardware implementation of a parallel priority queue. The contributions of this paper are summarized as follows:

- **Hole minimization:** The proposed approach minimizes holes in a parallel priority. Our hole minimization technique reduces hardware cost by 37.76% in terms of number of lookup tables (LUTs) and reduces average response time by 37.76%.
- **Hardware sharing:** Sharing hardware between two consecutive pipelined levels reduces the hardware cost. The hardware sharing technique contributes a 3.70% 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. Section III presents our proposed design, including different design trade-offs. 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

In this section, we present basic algorithms of a priority queue implementation and discuss related works.

A. Background

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.

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

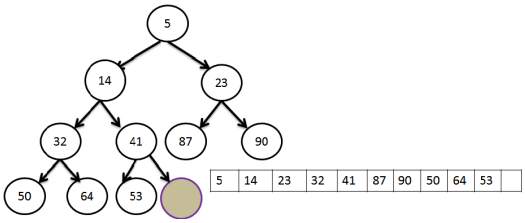

Figure 1. Binary min heap with its array representation.

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 e.g. i in H.
2) Compare the new element, $H[i]$, with its parent, $H[i/2]$. If $H[i] < H[i/2]$, then 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 ($H[1]$).

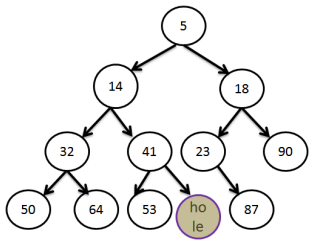

Figure 2. New heap structure after inserting 18.

Figure 2 shows the new heap structure after inserting 18 into 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 e.g. $H[i]$.

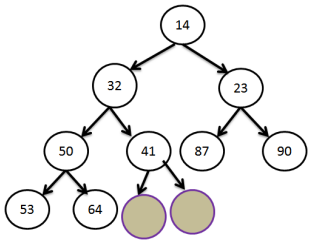

Figure 3. New heap structure after a delete operation from the original heap shown at Figure 1.

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 the 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 to the leaf node.

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

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.* [1], [4] 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 [3] in GPGPU platform provides multi-fold speedup. Another theoretical approach [5] deploys a pipeline or tree of processors ($O(\log n)$, where n is the number of nodes). The implementation of this algorithm [6] is expensive for multi-core architectures.

There have been several hardware-based priority queue implementations described in literature [8], [9], [10], [11], [12], [13], [14], [15]. Pipelined hardware implementations can attain $O(1)$ execution time [11], [12]. 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* [14], [15] 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 in different levels with $O(1)$ step complexity. For the *Shift Register* [15] based implementations, when new data comes for processing, it is broadcasted to all levels. This requires a global communicator hardware, which connects to all levels. The implementation based on *Systolic Arrays*

[14] 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, a recursive processor is implemented by [16] where hardware use is drastically reduced by compromising execution time. Bhagwan and Lin [9] designed a physical heap such that commands can be pipelined between different levels of the heap. The authors in the paper [8] give a pragmatic solution of the *fanout* problem mentioned in [10]. The design presented in [17] is very efficient in terms of hardware complexity but is 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* [20] implemented the logic presented in [10] 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 [19], and it ensures high scalability and high throughput. The FPGA-based pipelined heap presented by Ioannou *et. al* [18] is scalable and can run for 64K nodes without compromising performance, but it takes at least three clock cycles to complete a single stage. Moreover, it never addresses the *hole* generated by parallel *min-delete* operations followed by *insert* operations.

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 can be represented as $2^{\beta-1}$, where β is the level, assuming that root is the level 1. Figure 4 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.

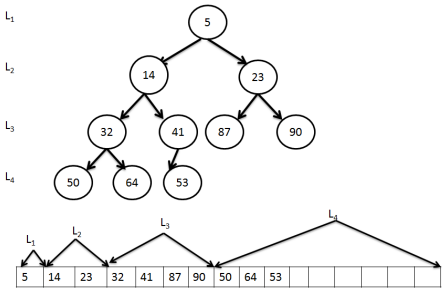

Figure 4. Storage in FPGA of deferent nodes in binary heap

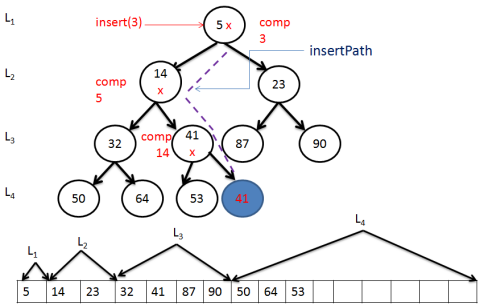

Figure 5. Insert path

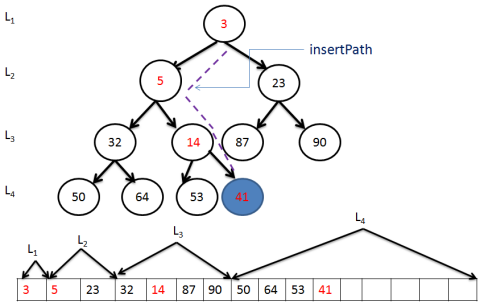

Figure 6. Container of latches L after insertion completes.

A. Insert Operation

We have already discussed the *insert* operation, which is intimated from the last available node of a heap. If a bottom-up insertion is done in parallel with other operations such as min-delete operations, it may cause an inconsistency in the heap. To clarify, let us consider the example presented in Figure 4. 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, in the worst case, a total of four clock cycles is required to complete only an insert operation in this situation. Before it completes, if a *min-delete* operation comes, it has to wait up to four clock cycles to avoid deleting a wrong element (5 in this case) from the root. 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 complete binary tree conditions. We adopt an algorithm presented by Vipin *et. al* [7] 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 5 shows the insert path for a new element 3 to be inserted. 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 5. Figure 6 represents the binary tree and latches after this insert operation.

B. Min-delete Operation

There is one conventional approach to delete an element from a heap. Since a min element resides at the root, deletion always happens from the root, and the last element of the heap replaces the root. There are two difficulties here:

1) For sequential operation, it works fine. Parallel execution of insert/delete, however, can create a *hole*. This happens after any $insert$ followed by $delete$ operation.
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, three-element comparisons cause a longer path delay.

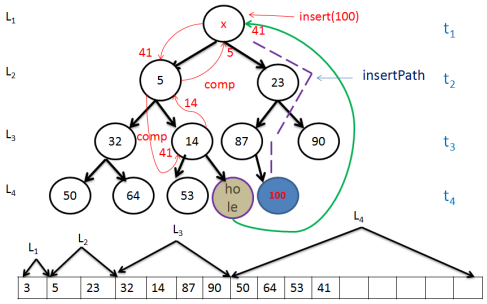

Figure 7. A hole results from parallel operation of insert-delete

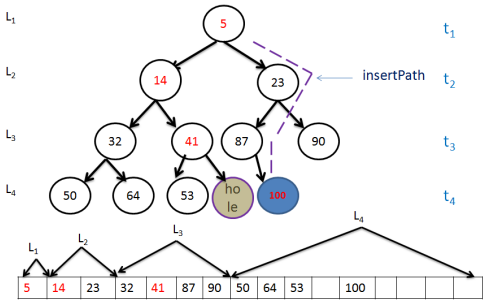

Figure 8. Container of latches (L) after parallel operation of insert-delete

Figure 7 illustrates the first issue clearly. Index 1 corresponds to the root. At t_1, the insert operation with element 100, denoted by $insert(100)$, is encountered. The element will be inserted at the last node of the last level, which is at index 12. After one clock cycle of *insert*, *min-delete* is encountered at t_2. But, at that time, $insert(100)$ is being processed at L_2, so the last element is still the $11th$ node. The *min-delete* operation will create a hole at the $11th$ node, as shown in Figure 7. Eventually, when $insert(100)$ finishes, element 100 will occupy at the position of $H[12]$, but $H[11]$ will become empty. Figure 8 illustrates this situation.

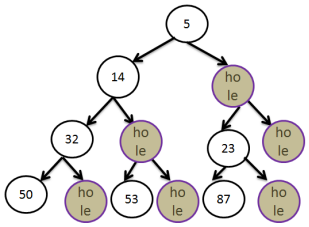

Figure 10. Worst case scenario for *hole* creation.

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 $j > i$. Either *insert* or *min-delete* takes one clock cycle at any level to complete tasks at that level. Only a single node gets modified (if any) for all levels. In general, any *insert-delete* combination will create a $hole$ if $(t_j - t_i) < \beta$, where β is the depth of the heap. In the worst-case scenario, shown in Figure 10, both time and hardware cost double due to *holes*. We should be careful to avoid this situation.

The naive way to avoid a hole from an *insert-delete* combination is to stall *min-delete* from entering the pipeline until *insert* completes. When the latter operation begins, the correct last element would replace the root. However, this method turns every *insert* that is followed by a *min-delete* into a sequential operation. In the worst-case scenario, where a binary heap of many levels encounters a sequence with nothing but *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 a register, and if one exists, we fill it with the new element. We apply the *insert* algorithm, except we modify it so that the data will be inserted at the position of a *hole* instead of the last available node. In the case of multiple holes, we fill the last *hole* with inserted data 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.

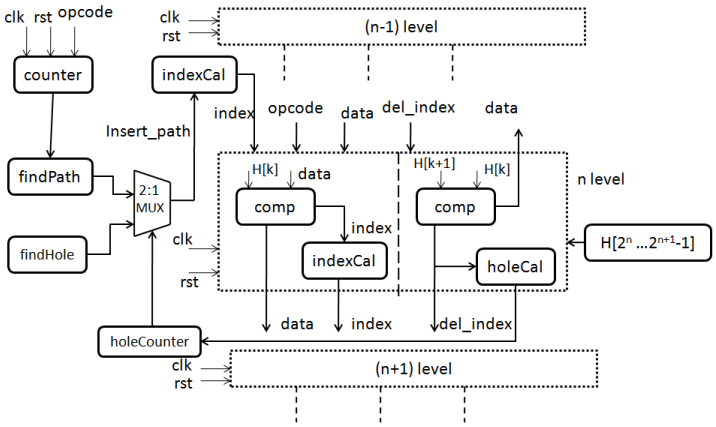

Figure 9. Top Level Architecture of insert-delete

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

```
1:  if (opcode == INSERT_OP) then
2:      counter = counter + 1;
3:      insert_path = findPath(counter, holeCounter)
4:      for (i = 0 to number of levels) do
5:          index = indexCal(insert_path, i, index)
6:          if (data < H[index_i] then
7:              H[index_i] = data
                data = H[index_i]
8:          end if
9:      end for
10: else
11:     Remove H[1]
12:     while (leftChild[del_index] ≠ NULL &
        rightChild[del_index] ≠ NULL) do
13:         if (leftChild[del_index] <
            rightChild[del_index]) then
14:             H[del_index] = leftChild[del_index]
                del_index = del_index * 2
15:         else
16:             H[del_index] = rightChild[del_index]
                del_index = del_index * 2 + 1
17:         end if
18:     end while
19:     holeCounter = holeCounter + 1
        holeReg[holeCounter] = del_index
20: end if
```

Figure 9 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

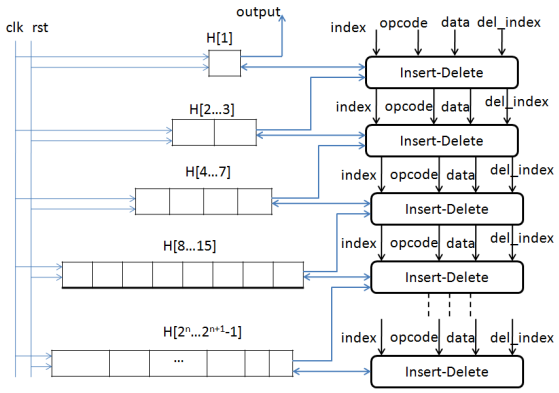

Figure 11. Pipeline Design Overview

incremented by one for an *insert* operation and decremented by one for a *min-delete* operation. Modifying the existing path-finding algorithm proposed by [7], we consider *holeReg* during insertion to obtain 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 with 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 deleted. 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 *del_index*. If

not, then *holeCounter* is incremented by 1, and *holeReg* is updated with *del_index*. 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 a mathematical expression of $log(n)$ 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, *findHole* is an implementation of a stack register. Its return value is presented at Algorithm 4.

Algorithm 2 Algorithm for $findPath$(counter, holeCounter)

```
1: if (holeCounter > 0) then
2:
3:    return findHole(holeCounter)
4: else
5:    leaf_node = findNode(counter)
6:    return (counter − leaf_node)
7: end if
```

Algorithm 3 Algorithm for $findNode(counter)$

```
1: for (i = 0; 2^i < counter; i = i + 1) do
2:    leaf_node = i + 1
3: end for
4: return leaf_node
```

Algorithm 4 Algorithm for $findHole(holeCounter)$

```
1: return holeReg[holeCounter]
```

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

```
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
```

D. Pipeline Design

To achieve high throughput, we need to start one operation before completing the previous operation so that multiple operations can be in progress at the same time. Our implementation takes a single clock cycle to perform an operation at each stage. For any stage, only one operation (*insert-delete*) can be executed at any given time t. That is why all operations must start from the top of the tree (root) and proceed towards the bottom (leaf).

Figure 11 illustrates the basic pipeline architecture of our binary heap. Each level performs *insert* or *min-delete* based on the signal *opcode*. The level takes only one clock cycle to perform each operation. It sends *data* and *opcode* to the next level to perform. All levels, except the first, contain the same logic hardware.

E. Optimization Technique

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.

1) Hardware Sharing: We make each basic operation–*read, write* and *compare*–complete in one clock cycle. Each level has to perform these three basic operations, resulting in three clock cycles in total. We pre-compute data for a level such that there is a maximum overlap between two consecutive levels in case of *insert* operations. For any level L_i, if *read* operation executes at t time, then it executes *comp* operation at $t+1$. The *comp* generates the next *index* to be read by the next level. So, level L_{i+1} performs *read* at $t+1$ time. Now, level L_i performs a write operation at $t+2$, while level L_{i+1} finishes *comp* and generates the index to be read by level L_{i+2}. At time $t+3$, level L_{i+1} will perform a *write* operation while level L_{i+2} will complete a *comp* operation and will make *index* available for level L_{i+3}. This way, we find there are two operations overlapping between two consecutive levels in three cycles. Effectively, it results in a write at each clock cycle after an initial latency of two clock cycles at the first level. Figure 12 illustrates this situation. In this example, while level L_2 performs *comp* at clock 2, level L_3 performs *read*. Level L_2 completes *write* at clock 3, while level L_3 completes *comp* followed by *write* at clock 4. A *comp* operation is processed by level L_i and a *read* operation is done by level L_{i+1} at same clock cycle t by using the concept of different edge of clock. Level L_2, for example, performs *comp* at positive edge of clock 2 and level L_3 performs *read* at negative edge of clock 2.

An *insert* or *min-delete* operation of any level waits for data from its previous level. As the min element of certain levels goes up to the upper level, the data will be available to write after performing a *comp* operation of that level. At all levels except root, if a *read* operation is executed at t time by level L_i, then that level executes a *comp* operation at $t+1$. As *comp* generates the next *index* to be read by the next level L_{i+1}, L_{i+1} performs *read* at $t+1$. However, L_i cannot perform a *write* operation at $t+2$ because the data from L_{i+1} will be written at level L_i; the resultant of *comp* by L_{i+1} will be available after $t+2$. That means L_i can perform *write* only at time $t+3$. At $t+2$, L_i becomes idle.

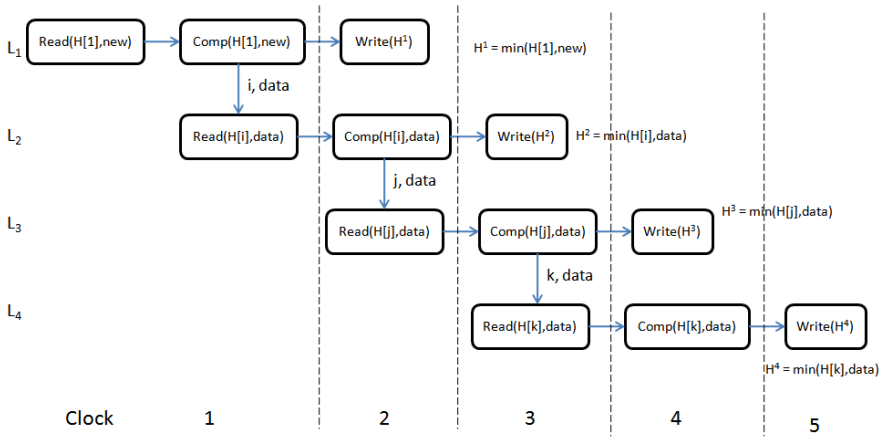

Figure 12. Parallel insert operation: illustrates operations at each level at each clock

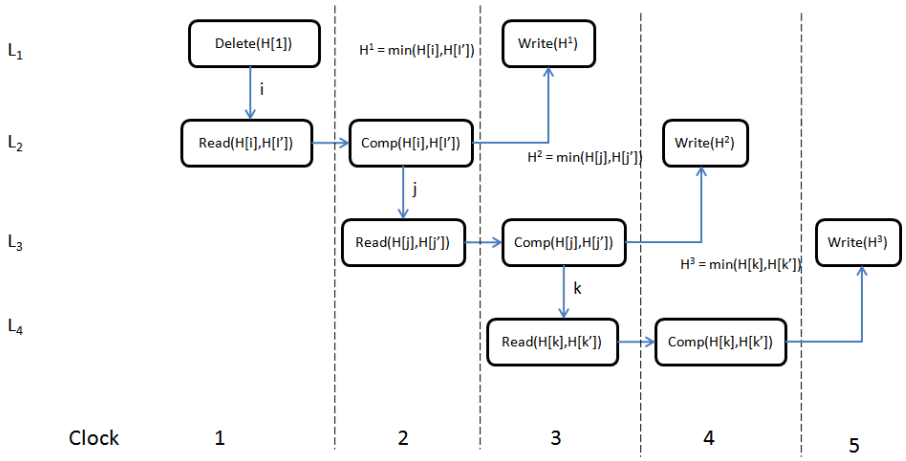

Figure 13. Parallel delete operation: illustrates operations at each level at each clock.

For each level, we can see that there is such an idle state. In the example of Figure 13, while L_2 performs *comp* at clock 2, L_3 performs *read*. L_2 becomes idle at clock 3 while L_3 performs *comp* at that time. Eventually, L_2 performs *write* at clock 4 after the data becomes available by level L_3. From Figure 13, we can see that the data from L_2 is written at level L_1 at clock 3 . That means L_2 suffers a temporary *hole* at clock 3. This *hole* at L_2 is compensated while L_3 writes to L_2 at clock 4. However, L_3 also suffers a temporary *hole*. While a level has a temporary *hole*, the level is in an inactive state, which means no operation could be performed at that level at that time. In general, for any given time t, the level L_i can not 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}. Except for this situation, two consecutive levels can share a common *insert-delete* block, as illustrated in Figure 14.

2) Replacement Technique: There are four possible sequences of operations: *Insert-Only* (I), *Delete-Only* (D),

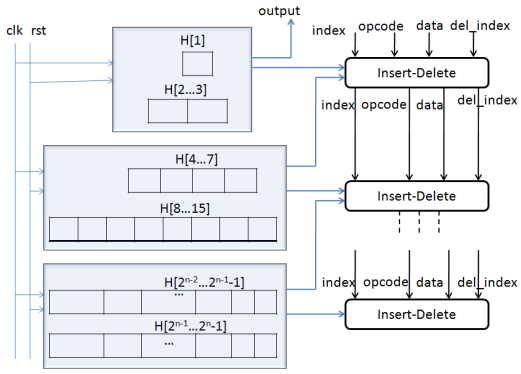

Figure 14. Sharing *Insert-Delete* hardware between two consecutive levels.

Insert-Delete (ID) and *Delete-Insert* (DI). The *Insert-Only* sequences ($I\ I\ \cdots I$) take single clock cycle to operate at each stage. For $IDID\cdots DIDI\cdots$ sequences, we introduce a *replacement* operation that does not cause *holes*. The

Table I
VARIATION IN FREQUENCY, EXECUTION TIME, AND THROUGHPUT WITH NUMBER OF LEVELS.

Number of Levels (β)	Frequency (f) (MHz)	Execution Time (ns)	Throughput (τ) (GB/Sec)
4	318.8	9.41	1.27
8	232.8	12.88	1.85
10	212	14.15	2.12
12	210	14.25	2.52
16	207.2	14.5	3.31
20	173.4	17.3	3.46
24	171.6	17.48	4.10

algorithm of the *replacement* 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.

IV. EVALUATION

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 simulate the data path using Verilog and Python script. We generate the test bench through Python, which is sent to the ISim simulator. The instruction for *insert/delete* is executed based on the 1-bit opcode value. Internal logic of FPGA determines the *replacement* operation based on two consecutive opcodes. Unless otherwise stated, a random sequence of *insert/delete* operations is used.

A. Sensitivity Analysis

We explore how the results (frequency, execution time, and throughput) change with the number of levels. The execution time per level, t, is calculated as:

$$t = \frac{3}{f} \tag{1}$$

where f is the frequency. Throughput, τ, is calculated as:

$$\tau = \frac{\omega \times f}{\chi} \tag{2}$$

where ω is the bit length, f is the clock frequency and χ is the number of clock cycles required to compute one *insert-delete* 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; it is inversely proportional to bit length (β). We obtain maximum frequency = 318.8 MHz for $\beta = 4$, and minimum frequency 171.6 MHz for $\beta = 24$. Execution time is directly proportional to frequency and inversely proportional to β, because it takes at worst three cycles per stage to complete the task. An increase of the number of levels leads to slower clock frequency and execution time but higher throughput. As we have designed a fully pipelined architecture, the output can be obtained in each clock cycle as shown in Figure 13. For the rest of experiments, we use $\beta = 24$.

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	780	1840	2730

Table III
RESPONSE TIME RESULTS.

Design	Response Time (ms)
Without hole minimization	2010
With hole minimization	1920
With replacement	1337

B. Hardware Cost and Response Time

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. By applying the technique, these numbers are reduced by 0.00%, 42.14%, 37.76%, and 41.07% respectively. The hardware sharing technique reduces it 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 1920, which corresponds to 4.48% reduction, as shown in Table III. Introducing the *replacement* operation further reduces the response time by 30.36%.

The impact of the *replacement* operation is dependent on the sequence of operations. Figure 15 compares the response time according to the sequence of operations. When only *insert* operations come, there will be no hole created, which gives us the second-best response time. The best case is when insert/delete operations come in an alternative fashion. In this case, all pair of insert/delete operations can be replaced with the replacement operation. The worst case is when only delete operations come right after only insert operations are processed. In this case, none of operations can benefit from the replacement operations while many holes

are created. The response time of a random sequence is in-between the best and worst cases.

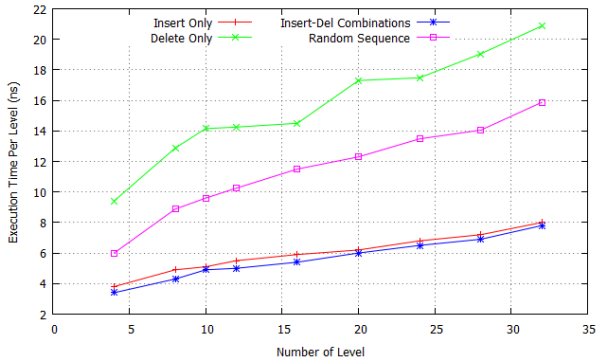

Figure 15. Execution time for different sequence of operations.

C. Comparison with Existing Techniques

The three techniques proposed 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 [14], 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 [14] requires 2^{β} comparators and $2^{\beta+1}$ flip-flops, the proposed design incurs much less overhead ($\frac{\beta}{2}$ and β), offering better scalability.

Reference [17] offers seemingly less overhead, but it is, in fact, *underestimated*. While reference [17] 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 [18], [9] 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 the proposed 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.

V. Conclusions

In this paper, we propose a hardware 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 sharing hardware 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 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 [4]. In that case, each node of the heap will have n sorted data (except the last node). Each time an *insert* or *delete* occurs, we need to maintain the heap construction along with the sorted list of each node. Parallel operatons could have many applications, and they could be more complex in terms of FPGA implementation.

References

[1] S. Prasad, “Efficient parallel algorithms and data structures for discreteevent simulation”, PhD Dissertation, 1990.

[2] Sushil Prasad , I. Sagar Sawant, “Parallel Heap: A Practical Priority Queue for Fine-to-Medium-Grained Applications on Small Multiprocessors”. Proceedings of 7th IEEE Symposium on Parallel and Distributed Processing (SPDP)1995.

[3] Xi He, Dinesh Agarwal, Sushil K. Prasad: “Design and implementation of a parallel priority queue on many-core architectures”. HiPC 2012: 1-10

[4] N. Deo and S. Prasad, “Parallel heap: An optimal parallel priority queue”, The Journal of Supercomputing, vol. 6, no. 1, pp. 87-98, 1992.

[5] G. S. Brodal, J. L. Tradff, and C. D. Zaroliagis, “A parallel priority queue with constant time operations”, Journal of Parallel and Distributed Computing, 49(1): 4-21, 1998.

[6] A. V. Gerbessiotis and C. J. Siniolakis, “Architecture independent parallel selection with applications to parallel priority queues”, Theoretical Computer Science, vol. 301, no. 1 Vol 3, pp. 119-142, 2003.

[7] V. N. Rao, Vipin Kumar: “Concurrent Access of Priority Queues”, IEEE Trans. Computers 37(12): 1657-1665, 1988

[8] S.-W. Moon, J. Rexford, and K. G. Shin, “Scalable hardware priority queue architectures for high-speed packet switches”, IEEETC: IEEE Transactions on Computers, vol. 49, 2000.

[9] R. Bhagwan and B. Lin, “Fast and scalable priority queue architecture for high-speed network switches”, in INFOCOM, 2000, pp. 538-547.

[10] H. J. Chao and N. Uzun, “A VLSI sequencer chip for ATM traffic shaper and queue manager”, IEEE Journal of Solid-State Circuits, vol. 27, no. 11, pp. 1634-1642, November 1992.

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 (τ) (GB/Sec)	Execution Time	Complete Tree ?
[14]	2^{β}	$2^{\beta+1}$	0	8560	-	-	$O(1)$	Yes
[17]	$2 \times \beta$	$2^{\beta+1}$	0	1411	-	-	$O(\log n)$	Yes
[18]	$2 \times \beta$	$2 \times \beta$	$2 \times \beta$	-	180	6.4	$O(1)$	No
[9]	$2 \times \beta$	$2 \times \beta$	$2 \times \beta$	-	35.56	10	$O(1)$	No
Proposed	$\frac{\beta}{2}$	β	β	**1840**	**171.6**	**4.10**	$O(1)$	**Yes**

[11] 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 Transactions on Very Large Scale Integration Systems, vol. 16, pp. 781-791, 2008.

[12] H. Wang and B. Lin, "Succinct priority indexing structures for the management of large priority queues", in Quality of Service, 2009. IWQoS. 17th International Workshop on, july 2009, pp. 1-5.

[13] 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. Proceedings, april 2006, pp. 1-12.

[14] S.-W. Moon, K. Shin, and J. Rexford, "Scalable hardware priority queue architectures for high-speed packet switches", in Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE, jun 1997, pp. 203-212.

[15] R. Chandra and O. Sinnen, "Improving application performance with hardware data structures", in Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, april 2010, pp. 1-4.

[16] Yehuda Afek, Anat Bremler-Barr, Liron Schiff: "Recursive design of hardware priority queues". Computer Networks 66: 52-67 (2014)

[17] Chetan Kumar, Sudhanshu Vyas, Ron K. Cytron, Christopher D. Gill, Joseph Zambreno, Phillip H. Jones: "Hardware-software architecture for priority queue management in real-time and embedded systems". IJES 6(4): pp. 319-334 (2014)

[18] A. Ioannou and M. G. Katevenis, "Pipelined heap (priority queue) management for advanced scheduling in high-speed networks", IEEE/ACM Transactions on Networking (ToN), vol. 15, no. 2, pp. 450-461, 2007.

[19] Muhuan Huang, Kevin Lim, Jason Cong: "A scalable, high-performance customized priority queue", FPL 2014: 1-4

[20] P. Kuacharoen, M. Shalan, and V. J. Mooney, "A configurable hardware scheduler for real-time sys- tems", in Engineering of Reconfigurable Systems and Algorithms, T. P. Plaks, Ed. CSREA Press, 2003, pp. 95-101.

[21] Abhishek Das, David Nguyen, Joseph Zambreno, Gokhan Memik, Alok N. Choudhary: "An FPGA-Based Network Intrusion Detection Architecture". IEEE Transactions on Information Forensics and Security 3(1): 118-132 (2008)

[22] Abhishek Das, Sanchit Misra, Sumeet Joshi, Joseph Zambreno, Gokhan Memik, Alok N. Choudhary: "An Efficient FPGA Implementation of Principle Component Analysis based Network Intrusion Detection System." DATE 2008: 1160-1165

[23] Sailesh Pati, Ramanathan Narayanan, Gokhan Memik, Alok N. Choudhary, Joseph Zambreno: "Design and Implementation of an FPGA Architecture for High-Speed Network Feature Extraction". FPT 2007: pp. 4-56

[24] M. Abadeh, J. Habibi, Z. Barzegar, and M. Sergi, "A parallel genetic local search algorithm for intrusion detection in computer networks", Eng. Appl. of AI, Vol. 20, No. 8, pp. 1058-1069, 2007.

[25] Christopher Krgel, Giovanni Vigna: "Anomaly detection of web-based attacks". ACM Conference on Computer and Communications Security 2003: 251-261