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 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. Holes are created from parallel min-delete operations. Moreover, holes could result an unbalanced incomplete binary heap, which leads to a longer response time. In this paper, we propose a hardware realization of parallel binary heap. The proposed technique makes three key contributions. (1) We ensure the tree structure be complete binary heap by providing *hole* minimization technique which reduces the hardware cost by 37.76% in terms of number of lookup tables (LUTs). The hole minimization technique also allows the proposed design to take $O(1)$ time for min-delete and insert operations by ensuring minimum waiting times between two consecutive operations. (2) Hardware sharing between two consecutive pipelined levels reduces hardware cost even further by 3.70%. (3) Introducing a *replacement* operation, we reduce the response time by 30.36%. As a result, the proposed technique incurs 78.50% less overhead while achieving the same performance level when compared to existing techniques.**

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

I. Introduction

A priority queue is one of popular data structures that is being used for various applications including routing, anomaly prioritization, shortest path search, and scheduling [25], [21], [22], [23]. A priority queue is a data structure in which each element has a priority and a dequeue operation removes and returns the highest priority element in the queue. It 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] to handle 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 is presented by Ioannou *et. al* [18]. This architecture is scalable and can run for 64K nodes without compromising performance, but it takes at least 3 clock cycles to complete a single stage. The calendar queues [8] incur significant hardware cost when they need to support a large priority set.

Moreover, all of exisiting works do not address a *hole* generated by parallel *min-delete* operations followed by *insertion* operations. Since holes occupy storage elements but do not have valid data, retaining holes incurs 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. The hole minimization techinque reduces hardware cost by 37.76% in terms of number of lookup tables (LUTs) and an average response time by 37.76%.
- **Hardware sharing:** The hardware cost is reduced by sharing hardware between two consecutive pipelined levels. The hardware sharing technique contributes 3.70% reduction in the hardware cost.
- **Replacement operation:** When a *min-delete* operation immediately followed by an *insertion* operation is detected, a *replacement* operation substitutes these two operations. In this way, we reduce an average response time by 30.36%.

The rest of this paper is organized as follows: Section II contains an overview and the art of literature related to this work. Our proposed design including different design trade-off is presented in section III. We also propose several optimization technique in terms of performance efficiency and hardware minimization. Section IV demonstrates the implementation results along with the performance comparison with existing designs. Section V concludes the paper and identifies some directions for future research.

II. Background and Related Work

In this section, basic algorithms of a priority queue implementation is presented and related works are discussed.

A. Background

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

- **Insert:** A number is inserted into Q, provided that the new list should maintain the priority queue.
- **Min-delete:** Find out the minimum number in Q and delete that number from Q. Again, after deletion the property of priority queue should be kept unchanged.

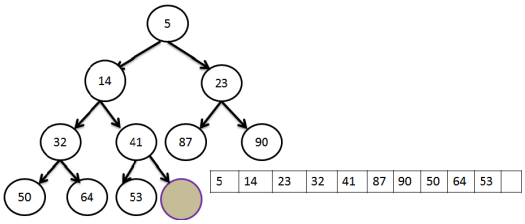

Figure 1. Binary min heap with its array representation.

A priority queue can be implemented by using a binary heap data structure. A min-heap is a binary tree H such that (1) the data contained in each node is less than (or equal to) the data in that nodes children and (2) the binary tree is complete.

Figure 1 shows the binary min heap (H). The root of H is $H[1]$, and 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 follow:

- Place the new element in the next available position (e.g. i) in the H.
- Compare the new element $H[i]$ with its parent $H\lfloor i/2 \rfloor$. If $H[i] < H\lfloor i/2 \rfloor$, then swap it with its parent.
- Continue this process until either (1) the new elements parent is smaller than or equal to the new element, or (2) the new element reaches the root ($H[1]$).

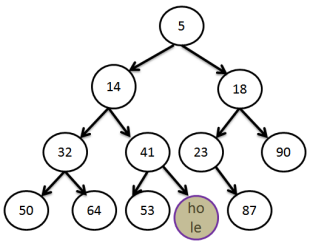

Figure 2. New heap structure after insertion 18.

Figure 2 shows the new heap structure after insertion of 18 at the heap presented in Figure 1.

The min-delete algorithm is as follow:

- Return the root $H[1]$ element.

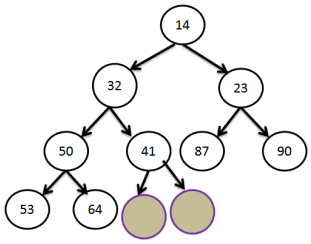

Figure 3. New heap structure after single deletion operation from the original heap shown at Figure 1.

- Replace the root $H[1]$ by the last element at the last level (e.g. $H[i]$).
- Compare the root with its children and replace the root by its min child.
- 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 reaches to the leaf node.

Figure 3 depicts the heap structure of single deletion operation from the original heap shown at Figure 1. We can see that 5 was the root element at Figure 1. The updated Figure 3 depicts when the root is removed by *min-delete*. Moreover, heap is re-structured according to the *min-delete* algorithm presented above.

B. Related Work

Several authors have theoretically proved that parallel heap as an efficient data structure to implement 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 processor used. The idea is designed for EREW PRAM shared memory model of computation. The many-core architecture by [3] in GPGPU platform provides multi-fold speed up. 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 in case of multi-core architectures

There have been several hardware based priority queue implementations described in the literature [8], [9], [10], [11], [12], [13], [14], [15]. Pipelined hardware implementations can attain $O(1)$ execution time [11], [12]. Due to several limitations like 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 the literature. The common drawback of these two implementations is using a large number of comparator ($O(n)$). The responsibility of comparators used here is comparison of 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.

It requires a global communicator hardware which connects with all levels. The implementation based on *Systolic Arrays* [14] needs a bigger storage buffer to hold pre-processed data. These approaches are not scalable and requires $O(n)$ comparators for n nodes. To overcome the hardware complexity, a recursive processor is implemented by [16] where hardware is drastically reduced by compromising execution time. Bhagwan and Lin [9] designed a physical heap such that commands can be pipeline between different levels of heap. The authors in the paper [8] give a pragmatic solution of called *fanout* problem mentioned in [10]. The design presented in [17] is very efficient in terms of hardware complexity. But, as the design is implemented by using hardware-software co-design, it is very slow in execution ($O(\log n)$).

For the FPGA-based priority queue implementation, Kuacharoen *et. al* [20] implemented the logic presented in [10] by incorporating some extra features to ensure the design to be acted as a real-time operating system task scheduler. The major limitation of this paper 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. A FPGA-based pipelined heap is presented by Ioannou *et. al* [18]. This architecture is scalable and can run for 64K nodes without compromising performance, but it takes at least 3 clock cycles to complete a single stage. Moreover, it never address 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. Each level of a heap can be virtually represented by an array of latches. 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; the next level with two elements is stored in L_2; the next level is in L_3; and the last level with 3 elements is stored in L_4, although the last level can have max 8 elements.

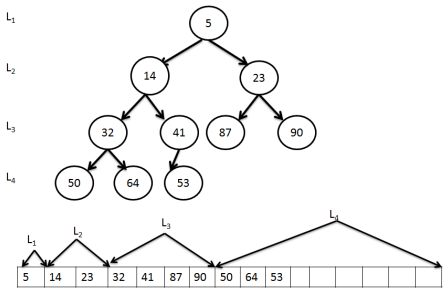

Figure 4. Storage in FPGA of deferent nodes in binary heap

A. Insert Operation

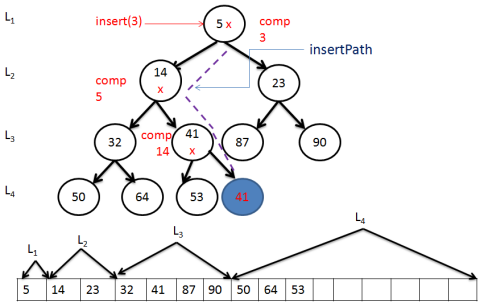

Figure 5. Insert path

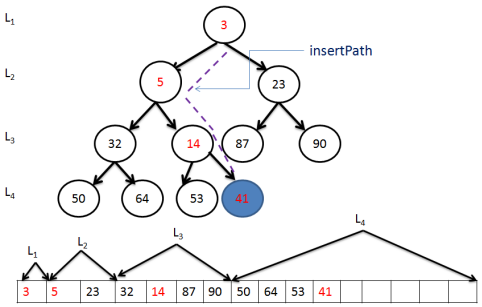

Figure 6. Contain of latch (L) after insertion completed

We have already discussed the *insert* operation which is intimated from the last available node of a heap. If this bottom-up insertion is done in parallel with other operations such as min-delete operations, it may cause inconsistency in the heap. To clarify this, let us consider the example presented in Figure 4. Suppose we insert element 3 in the heap followed by deleting one element from heap. Let us assume nodes at each level get updated in a single clock cycle. That means, in worst case, total 4 clock cycles are required to complete an insert operation in this situation. Before it completes, if a min-delete operation comes, it has to wait for up to 4 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 have adopted an algorithm presented by Vipin *et. al* [7] in our design. The algorithm is as follows:

- Let k be the last available node that a new element can be placed. Let j be the first node of the last level. Then binary representation of $k - j$ will give you the path.
- Let $k - j = B$, which binary representation is $b_{\beta-1}b_{\beta-2}\cdots b_2b_1$. 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 to be inserted. In this case, node at 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 $root \rightarrow left \rightarrow right \rightarrow right$, which is illustrated by the Figure 5. Figure 6 represents a binary tree as well as latches after this insert operation.

B. Min-delete Operation

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

1) For sequential operation, it works perfectly fine. For, parallel execution of insert/delete, $hole$ can be created here. The situation happens after any $insert$ followed by $delete$ operation.
2) While you replace root by last element of heap, it requires extra one clock cycle to write that element at root. Moreover, we need to compare three elements, root and its two children or any node and its children. For hardware perspective, it is cost efficient to compare two elements rather than to compare three elements. More over, it incurs the path delay longer.

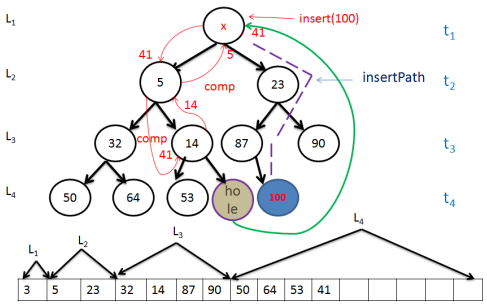

Figure 7. Hole is the resultant for parallel operation of insert-delete

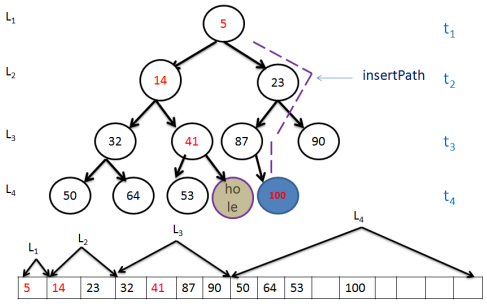

Figure 8. Contain of latch (L) after parallel operation of insert-delete

To solve the first issue, we proposed a hole minimization technique. Figure 7 illustrates the scenario of hole creation clearly. At t_1, the insert operation with element 100 is encountered and it is denoted by $insert(100)$. Obviously, the element will be inserted at the last node of the last level which is 12. After one clock cycle of *insert*, *min-delete* is encountered at t_2. At that time, $insert(100)$ was being processed at L_2. So, due to the *min-delete* operation, a hole will be created at 10th 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. This situation is illustrated by Figure 8.

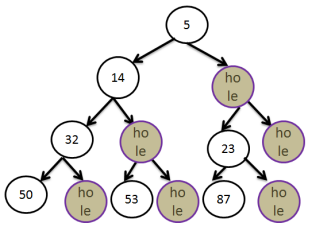

Figure 10. Worst case scenario for $hole$ creation.

To be general, 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. It is obvious that only single node gets modified (if any) for all levels. In general, for any $insert - delete$ combination, $hole$ will be created if $(t_j - t_i) < \beta$, where β is the depth of the heap. In the worst-case scenario, both time and hardware cost become double because of $hole$ (see Figure 10). We should carefully avoid this situation.

The *holes* are removed while an *insert* operation is processed. We check the *hole* existence by checking a register. If a *hole* exists, we fill it by new element. We apply *insert* algorithm with a modification that data will be inserted at the position of a *hole* instead of the last available node. We fill the last *hole* by inserted data in case of multiple hole presence, for the ease of implementation. Details of the algorithm to minimize holes are described in the following 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 and fill the root with its least child and follow the algorithm described in the previous section. In this way, we can save one cycle and minimize the path delay.

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.

Figure 9 illustrates the top level architecture of the priority queue employing the *insert-delete* logic. The *counter* is used to maintain the total number of elements present in the heap. It is incremented by one for *insert* operation and decremented by one for *min-delete* operation. The *indexCal* block

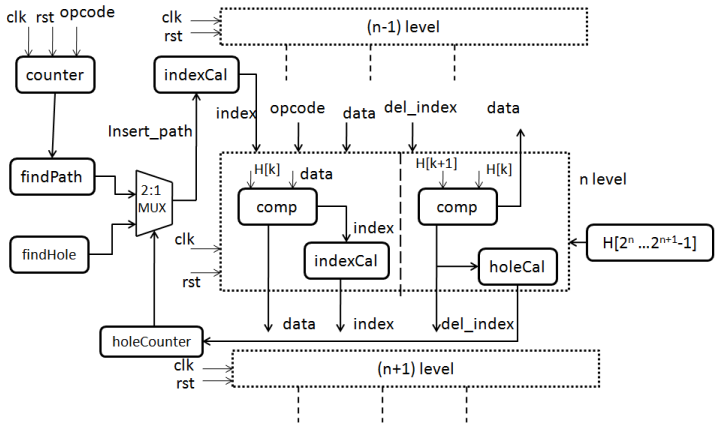

Figure 9. Top Level Architecture of insert-delete

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

1: **if** (opcode == 1) **then**
2: counter = counter +1;
3: **if** $(holeCounter > 0)$ **then**
4: insert_path = $findPath(counter,\ holeCounter)$
5: **end if**
6: **for** (0 **to** number of level) **do**
7: $index$ = $indexCal(insert\_path)$
8: **if** $(data < H[index]$ **then**
9: $H[index] = data$
$data = H[index]$
10: **end if**
11: **end for**
12: **else**
13: Remove $H[1]$
14: **while** $(leftChild[del\_index] \neq$ NULL & $rightChild[del\_index] \neq$ NULL) **do**
15: **if** $(leftChild[del\_index] < rightChild[del\_index])$ **then**
16: $H[del\_index] = leftChild[del\_index]$
$del\_index = del\_index * 2$
17: **else**
18: $H[del\_index] = rightChild[del\_index]$
$del\_index = del\_index * 2 + 1$
19: **end if**
20: **end while**
21: $holeCounter = holeCounter + 1$
$holeReg[holeCounter] = del\_index$
22: **end if**

is used to find the insert path. We have modified the existing path finding algorithm proposed by [7]. We first consider the *holeReg* to obtain insert path. The *holeReg* contains the *holes* created at *min-delete* operation. We maintain a *holeCounter* to identify a valid *hole*. Based on the *index*, the heap node is accessed and the node is compared with the present data.

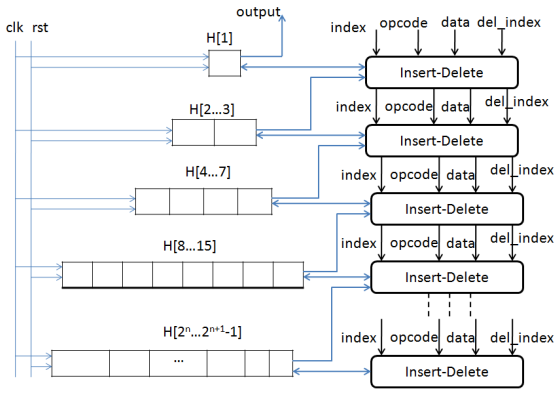

Figure 11. Pipeline Design Overview

Based on the comparison, either the node is updated by present data and the node data is passed to the next level, or the node remains unchanged and the present data is passed to the next level.

We maintain *del_index* to find the last deleted node. For example, initially, *del_index* is 1 which means 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 is replaced with $H[del\_index]$. Now, *del_index* gets modified with the index of min element. Again, the comparator finds the min of the ancestors of the new index and replaces the node of new index with that of the min element. Each time *holeCal* finds whether there is a valid child for *del_index* or not. If there is no valid child, then *holeCounter* is incremented by 1 and *holeReg* is updated with *del_index*. By this way, we maintain *hole*.

The *insert-delete* parallel algorithm is presented in Algorithm 1. We use a 2:1 multiplexer to select the path based on the value of *holeCount*. The logic for *findPath* is illustrated in Algorithm 2. The *indexCal* block is implemented based on the value of *findPath* and the logic is illustrated in

Algorithm 5. The *findNode* logic calculates the first node of the last level. To calculate the first node of the last level is noting but the mathematical expression of $log(n)$ where n is the number of elements of the binary heap. The hardware implementation of this logic is presented in Algorithm 3. The function of *findHole* is an implementation of stack register and 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 $indexCal(insert\_path)$

```
1: for (i = 0 to insert_path bits) do
2:    if (bit == 0) then
3:       index_i = 2*index_(i−1)
4:    else
5:       index_i = 2*index_(i−1) + 1
6:    end if
7: end for
```

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 be started from the top (root) of the tree 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*. Each level takes only one clock cycle to perform each operation. Each level sends *data* and *opcode* to the next level to perform. All the level contains the same logic hardware except the first level.

D. Optimization Technique

In this section, we present two optimization techniques. One is reducing hardware cost even further by sharing hardware in consecutive levels. The other is reducing response time by introducing a *replacement* operation.

1) Hardware Sharing: We make each basic operation e.g. *read, write* and *compare (comp)* complete in single clock cycle. Each level has to perform these three basic operations resulting in three clock cycles in total. We precompute data for a level such a way that there is maximum overlap between consecutive two 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}. In this way, we find there are two operations overlap between two consecutive levels in 3 cycles. Effectively, it results in writing at each clock cycle after 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.

A *insert* or *min-delete* operation of any level waits for data from its next level. As the min element of a certain levels goes up to the upper level, the data will be available to be written after performing a *comp* operation of that level. In general, if *read* operation is executed at t time by level L_i, then it executes *comp* operation at $t+1$ (except the root level). As *comp* generates the next *index* to be read by the next level, level L_{i+1} performs *read* at $t+1$. However, level L_i can not perform a *write* operation at $t+2$ because the data from level L_{i+1} will be written at level L_i; and the resultant of *comp* by level L_{i+1} will be available after $t+2$. That means level L_i can perform *write* only at time $t+3$. At $t+2$, level L_i becomes idle. For each level, we can see that there is such idle state. In the example of Figure 13, while level L_2 performs *comp* at clock 2, level L_3 performs *read*. Level L_2 becomes idle at clock 3 while L_3 performs *comp* at that time. Eventually, level L_2 performs *write* at clock 4 after the data becomes available by level L_3. From Figure 13, we can see that at clock 3 the data from L_2 is written at level L_1. That means, level L_2 suffers at a temporary *hole* at clock 3. This *hole* at level L_2 is compensated while level

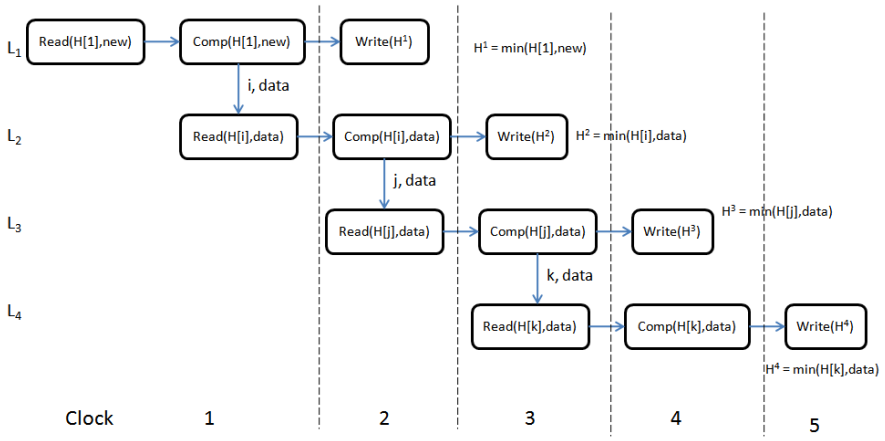

Figure 12. Parallel insert operation: illustrates operations at each level at each clock

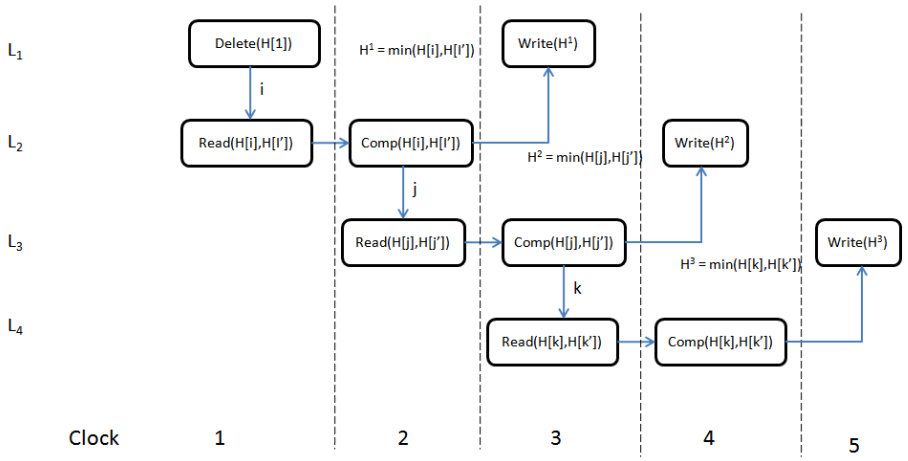

Figure 13. Parallel delete operation: illustrates operations at each level at each clock.

L_3 writes to L_2 at clock 4. However, level L_3 also suffers from a temporary *hole*. While a level has temporary *hole*, the level is in an inactive state, which means there could not be any operation to 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} can not finish the task of *comp* at $t+1$. In this situation, we can share hardware between the levels L_i and L_{i+1}. Except for this situation, a common *insert-delete* block can be shared by two consecutive levels, as illustrated in Figure 14.

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 ($I\ I\ \cdots I$) takes 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 algorithm of the *replacement* operation is as follows:

- Let X be the root element and Y be the element to be inserted in a request sequence $ID\cdots DI$.

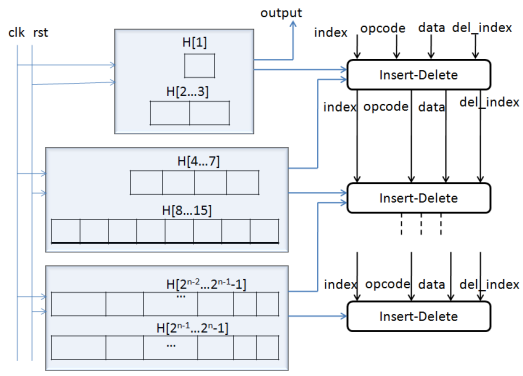

Figure 14. Sharing *Insert-Delete* hardware between two consecutive levels.

- Delete $min(X,\ Y)$
- $H[1] \leftarrow min(X,\ Y)$
- Continue this replacement for each level by comparing $H[i]$ with $H[2i]$ and $H[2i+1]$, until the parent become

Table I
VARIATION OF FREQUENCY, EXECUTION TIME AND THROUGHPUT WITH THE 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

less than its children or it reaches to the leaf node. Time complexity for these sequence is exactly same as *Insert-Only* where no holes are generated.

IV. EVALUATION

The proposed design has been simulated by ISim for implementation on Xilinx Sparttan6 XC6SLX4 hardware platform, where 32MB on-chip memory is used. The data path is simulated by using Verilog and Python script. The test bench is generated through Python and it send to 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 opcode. 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 is calculated as:

$$t = \frac{3}{f} \quad (1)$$

where f is the frequency. Throughput (τ) is calculated as:

$$\tau = \frac{\omega \times f}{\chi} \quad (2)$$

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 proportion to the bit length (β). We obtain maximum frequency = 318.8 MHz for $\beta = 4$, and minimum frequency 171.6 MHz $\beta = 24$. Execution time is directly proportion to frequency and inversely proportion to β because it takes 3 cycles (worst case) per stage to complete the task. Increasing the number of levels, both clock frequency and execution time increase, but the throughput also increases. 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 reducing the hardware cost, and the *replacement* operation shorten the response time further.

Table II shows results of the hardware cost measurement. Before applying the hole minimization technique, the number of comparators, filp-flops, LUTs and slices is 32, 1400, 3165, and 4870, respectively. By applying the technique, it is 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 the responsem time as shown in Table III. The hole minimization technique reduces the response time from 2010 to 1920, which corresponds to 4.48% reduction. By introducing the *replacement* operation, the response time is further reduced 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 second to the 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.

C. Comparison with Existing Techniques

The three techniques proposed in this paper (hole minimization, hardware sharing, and replacement operation) offers more efficient implementation of a priority queue when compared with previous works. Table IV compares the hardware cost and response time with existing techniques.

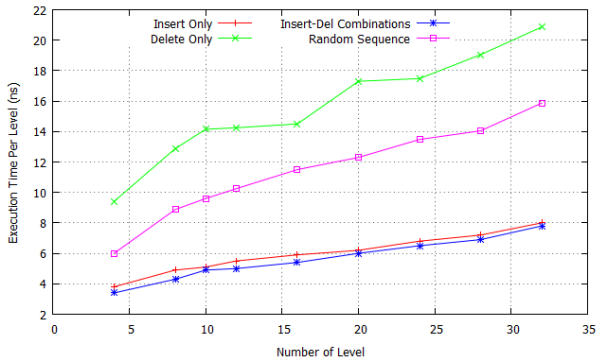

Figure 15. Execution time for different sequence of operations.

Since different designs address different issues and implemented in different platform, the comparison is made 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), which offers 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 pipelined fashion in hardware. Thed propose design takes $O(1)$ time for all operations by ensuring minimum waiting 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 also ensure a balanced heap structure, which contributes to reducing the response time. The *replacement* operation reduces the 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, the proposed 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 used the binary heap where each node has maximum two children. 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 of *insert* or *delete*; we need to assure the heap construction along with the sorted list of each node. There could be a lot of scope to have parallel operation, but it would be little 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.

[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.

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**

[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