A FPGA Based Efficient Pipelined Binary Heap

Monjur Alam and Sushil K. Prasad
Department of Computer Science
Georgia State University
Email: malam5@students.gsu.edu, sprasad@gsu.com

***Abstract*—Heap can be used as a priority queue implementation for wide variety of algorithms like routing, anomaly prioritization, shortest path, scheduling, etc. It is imperative to design an efficient parallel operation of priority queue to ensure Quality-of-Service (QoS) guarantees. There are some intrinsic limitation for the realization of truly parallel operation for such algorithm. Some authors provide parallel solutions [18], [19], [20]; but they do not focus on a serious issue like *hole*. All previous approaches incur a lot of hardware cost in producing *hole* in the heap. Moreover, these could result an unbalanced incomplete binary heap.**

In this paper we propose a hardware realization of parallel binary heap. The heap is implemented in a pipelined fashion for FPGA platform. Compared to existing FPGA implementations, we have three advantages. We ensure the tree structure to be complete binary heap by providing *hole* minimization technique which results from min-delete operations. The proposed design takes $O(1)$ time for min-delete and insert operations by ensuring minimum waiting times between two consecutive operations. Moreover, hardware sharing between two consecutive pipelined level incurs reduced hardware cost. We analyze various design issues and comparative hardware complexity of the proposed design. Our design reduce $\frac{1}{2x}$ hardware cost comparing to the existing designs.

***Keywords*-Parallel Binary Heap, FPGA, Verilog;**

I. Introduction

Network based anomaly detection [25] involves score calculation followed by ranking of all packets based on that score. To handle high network congestion, it is important to provide an efficient interface for prioritization of packets based on the score assigned. As software based implementation presented by
citeweb provides slower interfaces, a hardware based prioritization interface is necessary. Based on the packet priority, the interface will take drop or pass decisions. For high speed traffic, it is required to process these tasks in parallel.

To resolve the first class of difficulty several authors [21], [23] come up with hardware based solution. The intention is to provide very fast interface to process network data. To achieve this goal, Das *et. al.* [21], [22] comes up with hardware based solution for anomaly detection. The work comprises of a new Feature Extraction Module (FEM) which summarizes the network behavior. It also incorporates an anomaly detection mechanism using Principal Component Analysis (PCA) as the outlier detection method. The authors of [23] propose an FPGA-based reconfigurable architecture for feature extraction of large high-speed networks.

All of the approaches mentioned suffer from two basic problems:

1) There is no efficient implementation to deal with huge network congestion.
2) prioritization of network traffics are not maintained.

Implementation of parallel priority queue will solve this requirement. A priority queue (PQ) 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. PQs are the most basic component for scheduling, mostly used in routers, event driven simulators [18], etc. There are several hardware based PQs implementations that are usually implemented by either ASIC chips [8], [9], [15] or FPGA [18], [19], [20]. But, all of them suffer from some limitations and not applied to all applications.

In the literature, several hardware-based priority queue architectures have been proposed [14], [15]. All of these schemes have one or more shortcomings. 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. The major drawback of this design is that it takes at least 3 clock cycles to complete a single stage. Moreover, it never addresses the *hole* generated by parallel *min-delete* operation followed by an *insertion*. The calendar queues implemented by [8] can only accommodate a small fixed set of priority values since a large priority set would require extensive hardware support.

A. Our Contribution

We have implemented a software based anomaly detection mechanism where a score is assigned to each packet. We apply Markov based model for score calculation. A FPGA based parallel binary heap is implemented for score prioritization. We present the various design issues and hardware complexity. The pipeline architecture ensures no waiting time for any operation except the *deletion* one, which takes three clock cycles. To optimize this operation cycles, we combine insert and delete as $replacement$ operation which is not much differed from *Insert-Only* (I) operation. Each

of *insert* and *delete* operation takes $O(1)$ time. We also evaluate the design trade-offs of the proposed priority queue implementations. One elegance of our design is to optimize "hole" created by *delete* operation. We have seen that, for worst case, there could be $\sim$2x performance degradation and almost $\sim$2x hardware overhead if we do not tackle $hole$ creation during parallel operation. We minimize the *hole* at the time of *insertion*. Due to limited scope, we only discuss the FPGA implementation of parallel binary heap and we omit the anomaly part.

A Summary of the contents of the sections to follow: Section II contains an overview and the art of literature related to the 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 illustrates the implementation result along with the performance comparison with existing designs. Section V concludes the paper and identifies some directions for future research.

II. Preliminary and Related Work

Software based score prioritization of network packets are presented by Kruegel *et. al* [25]; where the packets with maximum score gets high priority to be processed next. Each time, the score is calculated "on the fly" and it is compared with other set of precalculated scores. Effectively, there is a processing delay to come up with a decision. Moreover, processing parallel packet is not possible here, as the "on the fly" calculation is highly serialized process.

A. Priority Queue

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

- **Insert:** A number n_i is inserted into the set of candidate number N in Q, provided that the new list 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.

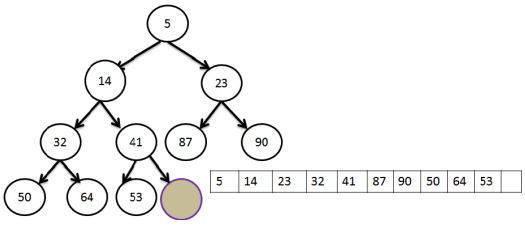

Figure 1. Binary min heap with its array representation.

1) Priority Queue Implementation: Priority queue can be implemented by using binary heap data structure.

Definition 2.1: A min-heap is a binary tree H such that (i) the data contained in each node is less than (or equal to) the data in that nodes children and (ii) 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 insertion algorithm on the binary min heap H is as follow:

- Place the new element in the next available position (say 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 (i) the new elements parent is smaller than or equal to the new element, or (ii) the new element reaches the root ($H[1]$).

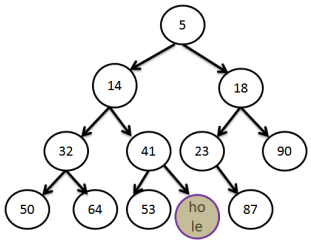

Figure 2. New heap structure after insertion 18.

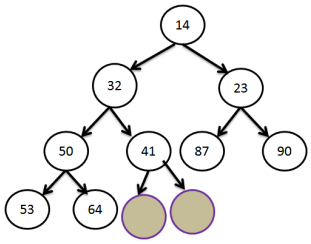

Figure 3. New heap structure after single deletion operation from the original heap shown at figure 1.

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

The deletion algorithm is as follow:

- Return the root $H[1]$ element.
- Replace the root $H[1]$ by the last element at the last level (say $H[i]$).

- Compare 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 $H[2i+1]$, un till the parent become less than its children or it 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 that the 5 is no anymore after the *deletion*. Moreover, heap is re-structured according to the *deletion* 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 this data structure to show $O(p)$ operations are required 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 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 based architectures

1) Hardware Based Priority Queue: There have been several hardware based priority queue implementations described in the literature [8], [9], [10], [11], [12], [13], [14], [15]. Pipelined based ASIC implementations can attain $O(1)$ execution time [11], [12]. Due to several limitations like cost and size, most of the ASIC implementations does not support a large number of nodes to be processed. These implementation are also limited in scalability. In [13], the author claims the pipelined heap presented be the most efficient one. However, this implementation incurs high hardware cost. The design is not flexible, more specifically, it is designed with a fixed heap size. 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 to compare nodes in different level 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 can connect 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 require much hardware, more specifically, it require $O(n)$ comparators for n nodes. To overcome the hardware complexity, a recursive processor is implemented by [16]; where hardware is drastically reduced by compromising execution timing cost. 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 some pragmatic solution of so 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. FPGA based pipelined heap is presented by Ioannou *et. al* [18]. This architecture is very much scalable and can run for 64K nodes without compromising performance. The major drawback of this design is that it takes at least 3 clock cycles to complete a single stage. More over, it never address the *hole* generated by parallel *delete* operation followed by an *insertion*.

III. FPGA Based Parallel Heap Implementation

Like an array representation, heap can be represented by hardware register or FPGA latch. Each level of the heap can be virtually represented by each latch. The size of the latch 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 the different latches do represent the different levels. Here, root node can be stored by L_1, the next level with two elements can be stored in L_2 and the last level with 3 elements can be stored in L_4, although the last level can have max 8 elements.

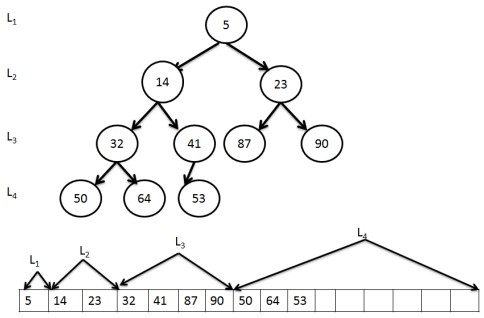

Figure 4. Storage in FPGA of deferent nodes in binary heap

A. Insert Operation

We have already discussed the insert operation which is intimated from the last available node of the heap. If this bottom up insertion be done in parallel with other operations such as delete, may make the Heap inconsistence. To clarify this lets consider min heap example presented in figure 4.

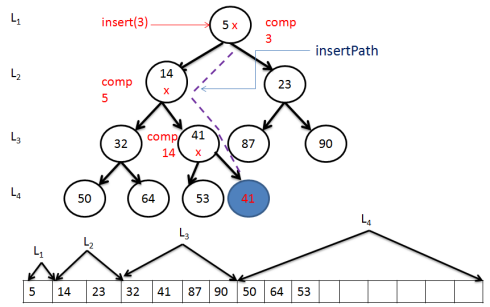

Figure 5. Insertion path

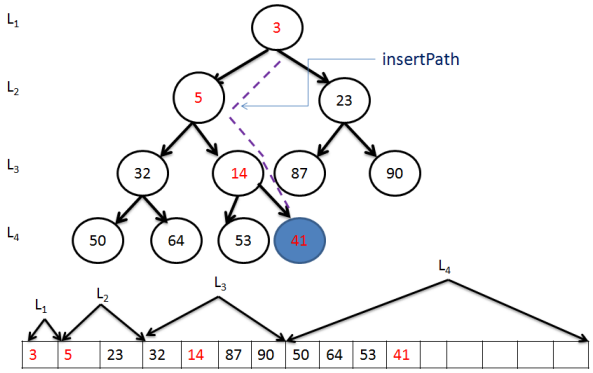

Figure 6. Contain of latch (L) after insertion completed

If we insert element 3 in the heap followed by delete one element from heap then what will happen? Let us assume nodes at each level get updated by a single clock cycle. That means, in worst case, total 4 clock cycles are required to complete the insert operation in this situation. So, delete operation either has to wait for 4 clock cycles or it will delete wrong element (5 in this case) at the root. So, it is incumbent to insert from root and go down. But, we need to know the path for the new inserted element, otherwise the tree will not hold complete binary tree conditions anymore. We have adopted a nice algorithm presented by Vipin *et. al* [7] in our design. The algorithm is as follow:

- Let k be the last available node that 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 insertion path for 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. So, starting from root, the path should be $root \rightarrow left \rightarrow right \rightarrow right$ and this can be demonstrated by the Figure 5. Figure 6 represents binary tree as well as latch after this insert operation.

B. Delete Operation

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

1) For sequential operation, it works perfectly fine. For, parallel execution of insert/del, $hole$ can be created here. The situation happens after any $insert$ followed by $delete$ operation.

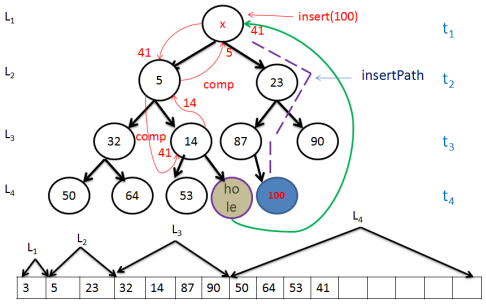

Figure 7. Hole is the resultant for parallel operation of insert-delete

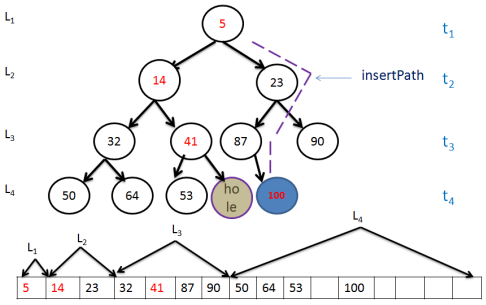

Figure 8. Contain of latch (L) after parallel operation of insert-delete

From the Figure 7 we can illustrate this scenario clearly. Let at t_1, the operation insert with element 100 is encountered and it is denoted by $insert(100)$. Obviously, the element will be inserted at the last node of last level which is 12. Let, after one clock cycle of $insert$, $delete$ is encountered (say at t_2). At, that time, $insert$ was modifying at L_2. So, due to $delete$, hole will be created at node 10th as shown in Figure 7. Eventually, when $insert(100)$ will finish, the element 100 will occupy at the position of $H[12]$, but, $H[11]$ will become empty. This situation is illustrated by Figure 8.

Let us assume that $insert$ instruction comes at time t_i and $delete$ instruction comes at t_j, where $i, j = 1, 2, 3, \cdots$ and $j > i$. Let, operation of either $insert$ or $delete$ takes one clock cycle at any level to complete

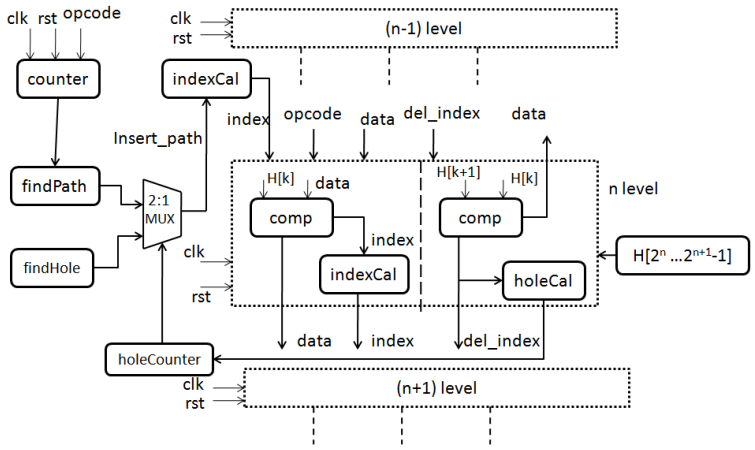

Figure 9. Top Level Architecture of insert-delete

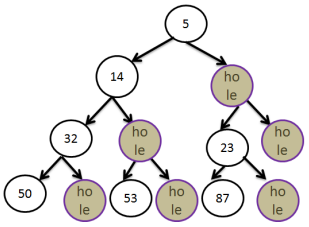

Figure 10. Worst case scenario for *hole* creation.

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 heap. In worst case scenario, both time and hardware cost becomes double for such *hole* output (Figure 10). We should carefully avoid this situation.

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.

So, we should intentionally avoid the root replacement by the last element. Let us delete root first and keep it as it is. Fill the root with its least child and follow the algorithm. In this case, we can save one cycle and hardware cost, more specifically, can minimize the path delay. Now, our aim is to minimize *hole* by adding logic.

C. Insert-Deletion Logic Implementation

Figure 9 illustrates the top level architecture of *insertion-delete* operation. 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 *deletion*

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:     else
11:       data = data
12:     end if
13:   end for
14: else
15:   Remove H[1]
16:   while (leftChild[del_index] ≠ NULL&rightChild[del_index] ≠ NULL)
      do
17:     if (leftChild[del_index] < rightChild[del_index]) then
18:       H[del_index] = leftChild[del_index]
          del_index = del_index * 2
19:     else
20:       H[del_index] = rightChild[del_index]
          del_index = del_index * 2 + 1
21:     end if
22:   end while
23:   hole_counter = hole_counter + 1
      hole_reg[hole_counter] = del_index
24: end if
```

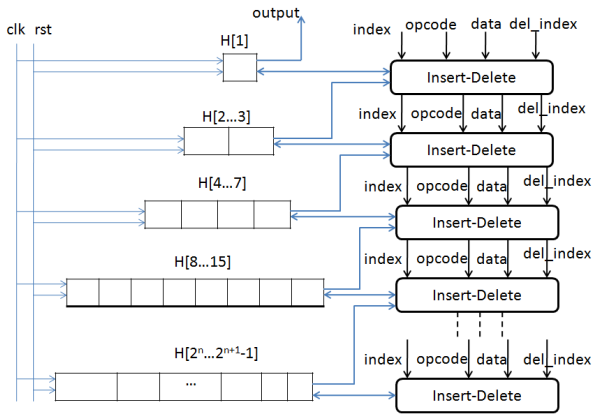

Figure 11. Pipeline Design Overview

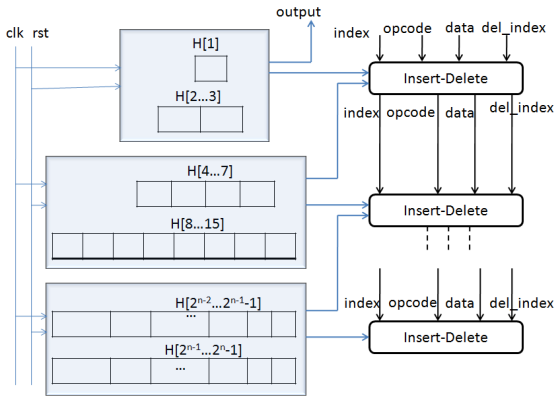

Figure 12. Sharing $Insert - Delete$ hardware resulting combinational logic reduction by $\sim \frac{x}{2}$.

operation. The *indexCal* block is used to find the insertion path. We have modified the existing path finding algorithm proposed by [7]. We first consider the *holeReg* to obtain insertion path. The *holeReg* contains the *holes* created at *deletion* 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. Based on the comparison, either the node is updated by present data and the node is passed to the next level as present data, or the node remains unchanged and the present data is passed to the next level.

Deletion : We maintain *del_index* to find the last deleted node. For example, initially, *del_index* becomes 1 which means root is deleted. The comparator finds the min element between $H[del\_index * 2]$ and $H[del\_index * 2 + 1]$ and that min gets replace to $H[del\_index]$. 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 min one. Each time *holeCal* finds if there is a valid child for *del_index*. 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 at Algorithm 1. We use 2:1 multiplexer to select the path based on the value of *holeCount*. The logic for *findPath* is illustrated at Algorithm 2. The *indexCal* block is implemented based on the value of *findPath* and the logic is illustrated at 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. There is some difficulty to realize this expression in hardware. We express this logic by Algorithm 3.

We have used global clock(clk) and global reset(rst) signal for the each logic block except the combinational logic parts. The clk and rst signals are not mentioned at each figure due

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**

to place limitation. The function of *findHole* is basically an implementation of stack register and its return value is presented at Algorithm 4.

1) Pipeline Design: To achieve high throughput we need to start one operation before completing the previous operation. So, many operations can be in progress in the tree. To achieve so, we consider our design to take a single clock cycle to perform each stage. For any stage, only one operation (*insert, delete*) can be execute at any time t. That is why all the 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 *insertion or deletion* based on the signal *opcode*. Each level takes only one clock cycle to perform each operation. Each level sends *data* and *opcode*

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 $findHole(hole\_counter)$

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

Algorithm 5 Algorithm for $indexCal(insert\_path)$

```
for (i = 0 to insert_path bits) do
    if (bit == 0) then
        index_i = 2*index(i − 1)
    else
        index_i = 2*index(i − 1) + 1
    end if
end for
```

to the next level to perform. There is a global clock and global reset attached to each stage. All the level contains the same logic hardware except the first level.

D. Optimization Technique

We need to know the operations at each level at each clock cycle to provide more optimization. We make each individual operations like *Read, Write* and *compare (comp)* to complete in separate single clock cycle. Each level has to perform these three basic operations resulting three clock cycles in total. We pre-compute data for a level such a way that there are maximum overlap between consecutive two levels in case of *insertion*. For any level β, 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, $\beta + 1^{st}$ level perform *Read* at $t+1$ time. Now, β level performs write operation at $t+2$, while $\beta + 1^{st}$ level finish *comp* and generates the index to be read by the $\beta + 2^{nd}$ level. At time $t+3$, the $\beta + 1^{st}$ level will perform *Write* operation while the $\beta + 2^{nd}$ level will complete the *comp* operation and will make *index* available for the $\beta+3^{rd}$ level. By this ways, we find there are two operations overlaps between two consecutive levels in 3 cycles. Effectively, it results of writing at each clock cycle after initial latency of two clock cycle at the first level. The Figure 13 illustrates this situation. We can see that, while level L_2 perform *comp* at clock 2, then level L_3 performs the *Read*. The level L_2 completes *Write* at clock 3, while level L_3 completes the *comp* followed by *Write* at clock 4. We make *comp* operation by β and *Read* operation by $\beta+1$ 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.

1) Hardware Sharing: Unlike, the *insert*, the *delete* operation of any level waits for data from its next level. As the min element of a certain levels go up to the upper level, the data will be available to write after performing the *comp* operation of that level. In general, if *Read* operation executes at t time by level β, then it executes *comp* operation at $t+1$ (except root level). As the *comp* generates the next *index* to be read by the next level. So, $\beta+1^{st}$ level perform *Read* at $t+1$ time. But, the level β can not perform *Write* operation at $t+2$, because the data from $\beta+1^{st}$ level will be written at the level β; and the resultant of *comp* by $\beta+1^{st}$ level will be available after $t+2$; that means the level β can perform *Write* only at time $t+3$. At $t+2$ the level β becomes idle. For each level, we can see that there is such idle state. For example, while level L_2 perform *comp* at clock 2, then level L_3 performs the *Read* (Figure 14). Level L_2 becomes idle at clock 3 while L_3 performs *comp* at that time. Eventually, the level L_2 performs *Write* at clock 4 after the data available by the level L_3 performs. From the Figure 14, we can see that at clock 3 the data from L_2 is written at level L_1. That means, the level L_2 suffers at a temporary *hole* at clock 3. This *hole* at level L_2 is compensated while the level L_3 write at L_2 at clock 4. But, the the level L_3 suffers from temporary *hole*. While a level has temporary *hole*, the level is in inactive state; that means there could not be any operation to be performed at that level at that time. In general, for any time t, the β level can not be completed if $\beta+1$ level can not finish the task of *comp* at $t+1$. That means we can share hardware between the levels β and $\beta+1$.

Figure 12 illustrates the hardware sharing where a common *Insert-Delete* block is used for two consecutive levels.

2) Replacement Technique: There are four possible sequence of operations like: *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 each stage. We introduce some tricks for $IDID\cdots DIDI\cdots$ sequences. In such cases, it just becomes a $replacement$ technique without thinking *hole* optimization.

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

So, time complexity for these sequence is exactly same as *Insert-Only*. The worst case situation happens in case of *Delete-Only* ($D\ D\ \cdots D$) sequences. Although, the $replacement$ ensures the $hole$ free architecture, the random sequences ($IIDDIDIIDDDI\cdots$) results obvious $hole$ outcome.

IV. Implementation Result

The proposed design has been simulated by ISim for implementation on Xilinx Sparttan6 XC6SLX4 hardware platform, where 32MB on-chip memory is used. Conceptually, we can simulate 2^{25} nodes; we predicts up to 2^{32} nodes based on simulation results. Throughput (τ) is calculated as:

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

where ω is the bit length, f is the clock frequency and χ is the number of clock cycle required to compute *insert-delete*.

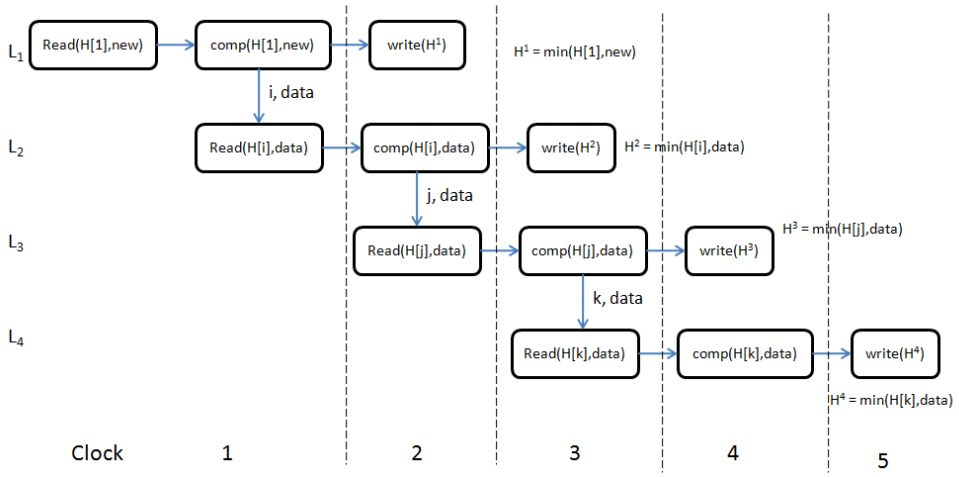

Figure 13. Parallel insert operation: illustrates operations at each level at each clock

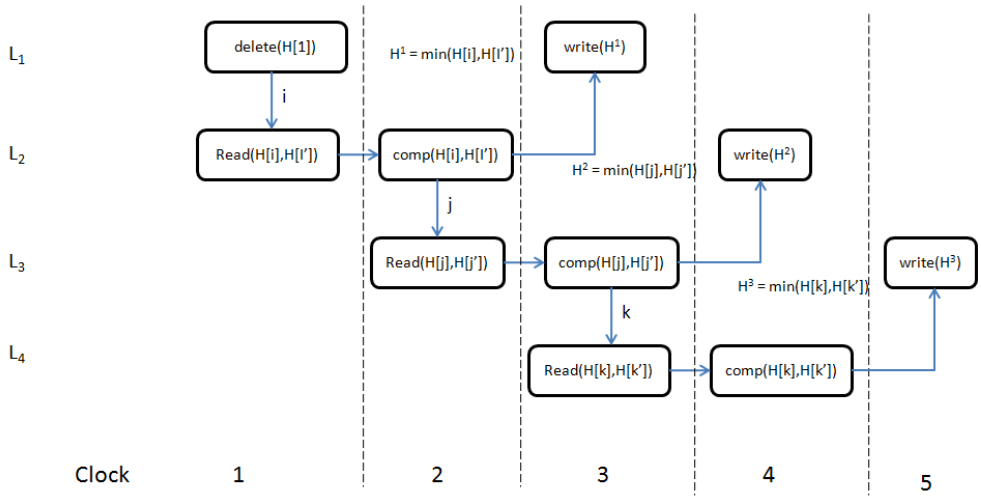

Figure 14. Parallel delete operation: illustrates operations at each level at each clock.

Table I
VARIATION OF FREQUENCY, EXECUTION TIME AND THROUGHPUT WITH NUMBER OF LEVEL

Number of Level (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
28	157.45	19.05	4.39
32	143.69	20.87	4.57

Table I demonstrates the performance result obtained from simulation for worst case situation. The execution time per level is calculated as:

$$t = \frac{3}{f} \quad (2)$$

where β is the number of level and f is the frequency. We use the number of level (β) and bit length (ω) interchangeably. Number of elements in the heap will be $2^{\omega} - 1 = 2^{\beta} - 1$. Form the table, 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 143.69 MHz $\beta = 32$. The parameter, execution time is directly proportion to frequency and inversely proportion to β. For example, it takes $\frac{3}{143.69} = 20.87$ ns when $\beta = 32$. Because, it takes 3 cycles (worst case) each stage to complete the task. The relation of throughput is a little bit complex. We can see that, it is directly proportion to frequency which is inversely proportion to β. But, it is directly proportion to β it self. As

we have designed a fully pipelined architecture, the output can be obtained in each clock cycles as shown at Figure 14. We obtain throughput, for example, $143.69 \times 32 = 4.59$ GB/Sec when $\beta = 32$.

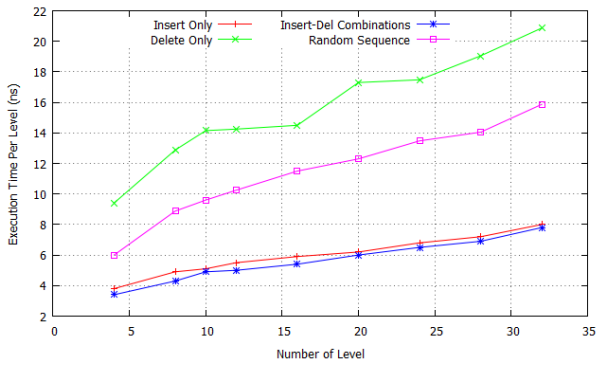

Figure 15. Execution time for different sequence of operations.

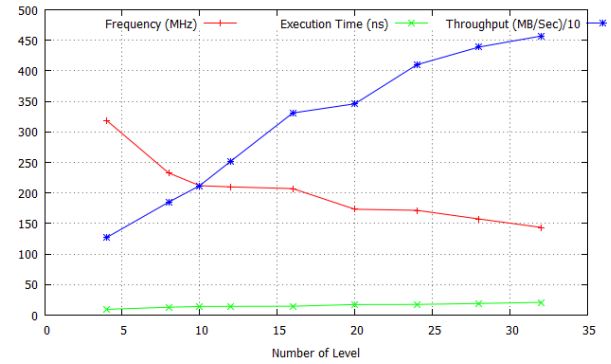

Figure 16. Different performance matrices

Figure 15 depicts the execution time at each level for different sequence of operations. We could see that both the $II \cdots II$ and $IDID \cdots DIDI$ takes almost same time. Conceptually, the later sequence takes one more extra clock cycle for the first level. The rest of the level it takes exactly same clock cycle that the *Insert-Only* operation takes. Only difference is that, the $IDID \cdots DIDI$ sequence does not care about *hole* optimization logic. So, it operates on minutely higher frequency (180 MHZ) than $IIII \cdots$ sequence (171.6 MHZ), in case of 24 levels. We can see that the random sequence operates on some middle stage between the worst and the best case. Figure 16 illustrates the graphical presentation of different efficiency parameter with variation of β for worst case situation. From the figure, it is clear that throughput increases even though frequency decreases with the increase of β.

A. *Hardware Cost*

We can visualize hardware cost with some parameters like [18]:

$$C = \beta \times (\kappa + F) + 2^{\beta} \times M$$

where C is the cost for β levels. κ is the numbers of comparators used, F is the number of *flip-flop* for each level and M represents the memory bits. For accessing memory bit, we use static RAM. Xilinx provides 2x512 K static RAM. So, effectively, we can simulate 2^{34} nodes. As we have addressed two levels of optimization like :*hole* minimization and *hardware sharing*; our design results very much cost effective comparing to the traditional designs [14], [18], [9]. We used, for example, 1970 number of Look-up tables (LUTs), 2870 number of slices with 800 *flip-flop* register to simulate 2^{32} number of nodes.

Table II demonstrates comparative analysis of our proposed design with existing ones. As different designs address different issues and implemented in different platform, it will be not fair to have direct comparison. We could see that, our design performs worst comparing to [17] in terms of total number of LUT used. But, as the design of [17] is implemented by using hardware-software co-design, it is very slow in execution ($O(\log n)$. Our design is very much comparable to [18], [9]. The design of [9] ensures high throughput with low clock frequency by using cell sizes of 424 bits. Unlike [18], [9], our design stands at moderate value of throughput and frequency by ensuring balanced complete binary tree.

V. Conclusion and Future Work

We implement a web based anomaly detection device. The anomaly is detected based on score calculation. The incoming network packets are captured and parsed the packets. The entire anomaly detection engine is based on software. Only the hardware part is the prioritization of anomalous packets.

We propose a hardware realization of parallel binary heap as an application of web based anomaly prioritization. The heap is implemented in pipelined fashion in FPGA platform. The propose design takes $O(1)$ time for all operations by ensuring minimum waiting time between two consecutive operations. We propose two levels of hardware optimization: by sharing hardware between two consecutive levels and by minimizing *hole*. In fact, *hole* minimization can ensure balanced heap structure as well. We present the various design issues and hardware complexity.

The work presented in this paper leaves several directions for future research. We presented some of these idea here.

- We present 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

Table II
HARDWARE COST AND PERFORMANCE COMPARISON.

Design	Comparator (κ)	Flip-flop (F)	SRAM (F)	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
Our	$\frac{\beta}{2}$	β	β	**1970**	**143.69**	**4.57**	$O(1)$	**Yes**

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 complex in terms of FPGA implementation.

- On-chip memory has a limited size (32 MB in our case, which means 2^{25} nodes can be simulated). To make the design more scalable, we can configure this into FPGA-ARM-core where FPGA is integrated into ARM processor. External RAM can be used, keeping in mind that this would lead slow design for extra memory access time.

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, vol. 49, no. 1, pp. 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. Nageshwara 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.

[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