Online Heavy Hitter Detector on FPGA

Da Tong, Viktor Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Email: {datong, prasanna}@usc.edu

Abstract—Detecting heavy hitters is essential for many network management and security applications in the Internet and in data centers. Heavy hitter is the entity in a data stream whose amount of activity, such as bandwidth consumption or number of connections is higher than a given threshold. In this work, we propose a pipelined architecture for an online heavy hitter detector on FPGA. It also reports the top $K$ heavy hitters. We design an application specific data forwarding mechanism to handle data hazards without stalling the pipeline. The stream size and the threshold for heavy hitter detection can be configured through run-time parameters. The post place-and-route results on a state-of-the-art FPGA shows that the architecture can achieve a throughput of 84 Gbps supporting 128 K concurrent flows. The proposed architecture can support large number of concurrent flows using external memory while sustaining the same throughput as the on-chip BRAM based implementation.

I. INTRODUCTION

A heavy hitter refers to an entity which accounts for more than a specified proportion of the total activity by all the entities [1]. In the context of computer networking, an entity can refer to a flow, a connection, an IP domain etc. The activity can be measured in terms of number of packets, bytes, connections, number of certain TCP flags, etc [1], [2].

Heavy hitters detection is the basis for many network management and security applications. It has been observed in many network measurement studies that a small proportion of traffic flows accounts for a dominant proportion of the total network traffic [3], [4]. Detecting these "elephant" flows is necessary for traffic policing and traffic load balancing [5]. Applying treatments only to these “elephant” flows can significantly enhance the scalability of differentiated service [6]. During Denial of Service (DoS) attacks, a huge disparity between the numbers of SYN and ACK packets can be observed due to incomplete three way handshakes [2]. Detecting the source IP which initiates a large number of incomplete three way handshakes can be helpful in defending against the DoS attacks.

In recent years, 100 Gbps networking has become a standard. Both the research community and the industry are targeting 400 Gbps and even 1 Tbps networks to support future networking demands [7], [8]. The number of concurrent flows in the networks is also massive. Recent research shows that the number is in the order of millions [9]. Because of the huge amount of streaming data and the real-time constraint on packet processing, the network processing must be performed online, in one pass over the data stream. Therefore heavy hitter detection needs to be an online service providing high throughput while supporting large number of concurrent flows.

State-of-the-art Field Programmable Gate Arrays (FPGAs) offer high operating frequency, unprecedented logic density and very high on-chip memory bandwidth. By exploiting the massive parallelism, FPGA based implementations can achieve extremely high throughput in network processing [10], [11].

In this paper we propose an online heavy hitter detector on FPGA. We summarize our main contributions as follows:

- Application specific architecture for online heavy hitter detection. The proposed architecture can be used as a generic solution for various counter based heavy hitter detection [12], [13], [14].
- A prototype implementation achieving a throughput of 84 Gbps while supporting 128 K concurrent flows.
- An application specific data forwarding mechanism to handle data hazards ensuring the correctness of the architecture without stalling the pipeline regardless of the latency of the memory access.
- Seamless use of external memory to support large number of concurrent flows at the same throughput as the on chip memory based implementation.

The rest of the paper is organized as follows. Section 2 defines our problem. Section 3 reviews the related work in heavy hitter detection. Section 4 describes our online algorithms and the hardware solution. Section 5 evaluates the heavy hitter detector. Section 6 concludes the paper.

II. PROBLEM DEFINITION

A. Heavy Hitter in the context of networking

A heavy hitter is defined as follows [1], [2], [12], [15].

DEFINITION 1. Given an input packet stream $S = \{(k_i, a_i) | i \in [1,N]\}$, $N =$ total number of packets in the stream, $k_i =$ key to identify the $i^{th}$ input packet, $k_i \in \{K_m | m \in [1,D]\}$, $D =$ number of distinct keys, $a_i =$ amount of activity contributed by the $i^{th}$ input packet. For a given threshold $\phi, \phi \in (0,1]$, a heavy hitter is an entity identified by key $K$, whose total activity $\sum_{i | k_i = K} a_i$ is at least $\phi \sum_{i=1}^{N} a_i$.

Without loss of generality, in this work, we use a packet flow as an entity and its bandwidth consumption as the activity measurement. Therefore, $k_i$ is the flow ID to identify the packets and $a_i$ is the packet size of the $i^{th}$ packet in $S$. The network traffic can be viewed as concatenated packet streams of various lengths. The length of each stream, $N$, can
be decided by the users. In each stream, flows are randomly interleaved. The bandwidth consumption of a flow can be represented by the ratio of its carried data and the total amount of data transferred on the link. In a fixed size stream on a link with fixed bandwidth the total amount of bytes transferred is a constant. Therefore the bandwidth threshold can be represented using number of bytes. We use $B$ to denote the threshold in bytes. Given the definition and the assumptions above, we consider the following problems.

- In each packet stream, detect all the heavy hitters for a given threshold.
- In each packet stream, report the top $K$ heavy hitters. $K$ is specified by the user at design time, ($K \geq 1$).

**B. Architecture Specification**

According to Section II-A the inputs to our architecture are streams of $\{\text{flow ID, packet size}\}$ pairs. We assume that the preceding systems extract the packet size from the packet header and assign the flow IDs to the packets according to their 5 tuple information: $\{\text{Source IP, Destination IP, IP protocol, Source Port, Destination Port}\}$.

The heavy hitter detection is performed on-the-fly. The flow ID of the heavy hitter should be reported as soon as its bandwidth consumption exceeds the threshold. The flow IDs of the top $K$ ($K \geq 1$) most bandwidth consuming heavy hitters should be reported at the end of each stream.

Our architecture supports two dynamically configurable parameters.

- $\text{StreamSize}$: defines the size of the streams in clock cycles.
- $\text{Threshold}$: defines the threshold in bytes.

**III. RELATED WORK**

There have been many research works on detecting heavy hitters. The algorithms can be divided into 2 categories, counter based algorithms [12], [13], [14] and sketch based algorithms[15], [16].

Counter based algorithms keep a bounded number of counters. Each counter records the total activity of an entity in the stream. On the occurrence of an activity, the algorithm increments the counter for the involved entity. The heavy hitter is reported when its total activity exceeds the threshold set by the network administrator.

Sketch based algorithms are statistical algorithms. They also keep a fixed number of counters. However they don’t associate each entity with one counter. For a single input, a series of hash functions decide which counters should be incremented. More than one counter can be incremented. The statistical summary for any entity are reported upon receiving query.

Although counter based algorithms and the sketch based algorithms use different counter access, reporting and error control mechanisms, they both need counters to record the activity. It is critical to have a high throughput counter based kernel for heavy hitter detection. We are not aware of any prior application specific hardware solution for online heavy hitter detector which achieves high throughput while supporting large number of flows at the same time.

**IV. ALGORITHM AND ARCHITECTURE**

**A. Overall architecture**

In this section, we introduce the overall architecture of the heavy hitter detector. The key idea is to maintain a counter for each traffic flow and compare the counter value with the threshold. This architecture can be used as a generic kernel to realize various counter based heavy hitter detection algorithms [12], [13], [14].

In the proposed design, we store the partial results for each flow and update them based on the input packet information. The partial results are stored in a Partial Results Table (PRT). Each entry of this table contains partial result of a flow. The flow ID, denoted as FlowID, is used as the index to access the table. During each clock cycle, one entry of the table is updated.

The PRT entry has two fields, $\text{SumCurStream}$ and $\text{TimeStamp}$. $\text{SumCurStream}$ records the total packet size of the flow since the beginning of the stream. $\text{TimeStamp}$ records the time of the most recent update of the PRT entry. We also keep a global time stamp, $\text{StreamTimeStamp}$, to record the start time of each stream. During each update, $\text{TimeStamp}$ is compared with $\text{StreamTimeStamp}$ to determine if the partial result of the current flow needs to be initialized. All the time stamps use relative time in number of clock cycles since the start of the system. To count the number of packets processed in the current stream, we keep a counter which is incremented every clock cycle. $\text{Threshold}$ and $\text{StreamSize}$ are kept as global variables in dynamically reconfigurable registers. The detailed operations at each stage of the pipelined architecture are shown in Algorithm 1.

We map Algorithm 1 into a 4 stage pipeline as illustrated in Figure 1. In Stage 1, the input FlowID is used as the physical address to retrieve the partial results from the memory. In Stage 2, the partial results associated with the input FlowID are updated and the heavy hitter is reported. In Stage 3, the updated
partial results from Stage 2 are written back into the memory. In Stage 4, the reported heavy hitters are processed to keep track of the top $K$ most bandwidth consuming heavy hitters. In our architecture the memory read/write can take multiple cycles and the computation takes one clock cycle. The Data Forwarding Unit (DFU) forwards the correct partial results into the Computation stage when data hazard happens. The DFU and the sorting pipeline is discussed in detail in Section IV-B and IV-C.

Algorithm 1 Operations for heavy hitter detection

Let $PktSize$ = Packet size of the input packet.
Let $CurrentTime$ = Current time.
Let $Threshold$ = Threshold for heavy hitter detection in bytes.
Let $C$ count the clock cycle in the current stream.

Stage 1: Memory Read
1: Retrieve $SumCurStream$ and $TimeStamp$ of FlowID from memory

Stage 2: Computation
1: if $TimeStamp > StreamTimeStamp$ then
2: $SumCurStream = SumCurStream + PktSize$
3: else
4: $SumCurStream = PktSize$
5: end if
6: if $SumCurStream > Threshold$ then
7: Report $FlowID$ /@ - Heavy Hitter - /@.
8: end if
9: $TimeStamp = CurrentTime$
10: if $C + 1 >= StreamSize$ then
11: $C = 0$
12: $StreamTimeStamp = CurrentTime$
13: else
14: $C = C + 1$
15: end if

Stage 3: Memory Write
1: Write $SumCurStream$, $TimeStamp$ back into memory

Stage 4: Sorting
1: Sort the reported heavy hitters

B. Data forwarding unit

1) Architecture of the data forwarding unit: Each incoming packet updates the most recent partial results of a flow with its $FlowID$. Therefore, a data hazard occurs if a packet enters the pipeline before the previous packet with the same $FlowID$ has exited the Memory Write stage.

To handle the potential data hazards, we design a Data Forwarding Unit (DFU) to forward the correct partial results into the Computation stage. Assuming that the memory can support a read and a write operations in each cycle, an architecture that ensures that the throughput is not adversely affected by the data hazards is shown in Figure 2. We assume that the latency for memory read and write is $R$ and $W$ cycles respectively. This design works for memory with any access latency. Therefore, $R$ and $W$ can be any positive integer.

The Memory Read stage includes an $R$-stage pipeline. This pipeline ensures that the input $FlowID$ and its partial results from the PRT are passed to the Computation stage in the same clock cycle. The DFU has shift registers to store the partial results written back to the memory and their associated $FlowIDs$. During each clock cycle the partial result and the $FlowID$ at the Memory Write stage are pushed into the shift registers and the oldest element is removed. It takes $R + W + 1$ cycles to update the partial results of an input $FlowID$. So, in any clock cycle the partial results of the $R + W + 1$ $FlowIDs$ before the current input $FlowID$ have not been updated. If a data hazard occurs, then we need to replace the outdated partial result from the PRT with the updated partial result. Therefore, once those partial results are updated, we store them in the shift registers. To detect if a data hazard occurs, the input $FlowID$ to the Computation stage is compared with all the $FlowIDs$ stored in the shift register. If one or multiple matching $FlowIDs$ are found, then the DFU forwards the partial results of the most recent matching $FlowID$ to the Computation stage. We use a priority encoder to give higher priority to the more recent partial results when generating the select signal to the multiplexer. This ensures that the partial result of the most recent $FlowID$ is always forwarded regardless of the number of matching $FlowIDs$.

Fig. 2: Detailed architecture with the data forwarding unit

Fig. 3: Pipelined data forwarding unit
2) Handling very long memory access latency: The DFU is directly connected to the Computation stage. When $R+W+1$ is very large, we need to include a priority encoder of a large width. Due to its large width, the priority encoder has long latency. Thus, the DFU adds large latency to the Computation stage. This decreases the working frequency of the pipeline. To reduce the width of the priority encoder, we pipeline the DFU to reduce the latency in the Computation stage.

This key idea is illustrated in Figure 3. Each stage processes one pair of FlowID and partial result from the shift register. The more recent pairs are processed at the stages closer to the output of the pipeline. By processing the data in such an order, at any stage we only need to check if a match exists in the current stage, because the data in the current stage is always more recent than the data from an earlier stage. At each stage we also need to process the FlowID and partial result at the Memory Write stage. This is because when a FlowID enters the pipelined DFU, the partial results and the FlowID being processed by the DFU are not available in the shift register. The processing element at each stage outputs the partial results of the most recent matching FlowID by that stage. The operations of the processing elements are shown in Algorithm 2.

Algorithm 2: Operations of the processing element of the pipelined DFU

Let $FlowID_{MW}/FlowID_{Shift} = FlowID$ from the Memory Write stage/shift registers.

Let $PR_{MW}/PR_{Shift}/PR_{Fwd} = $ partial result from the Memory Write stage/shift registers/forwarded from the previous stage.

Let $FlowID_{In} = $ Input FlowID.

1: if $PR_{Fwd}$ is from the shift registers then
2: if $FlowID_{MW} == FlowID_{In}||FlowID_{Shift} == FlowID_{In}$ then
3: if Exactly one of $FlowID_{MW}, FlowID_{Shift}$ matches $FlowID_{In}$ then
4: Output the partial result of the matching $FlowID$
5: else
6: Output $PR_{MW}$ has the highest priority - $\rightarrow$/
7: end if
8: else
9: Output $PR_{Fwd}$
10: end if
11: else
12: if $FlowID_{MW} == FlowID_{In}$ then
13: Output $PR_{MW}$
14: else
15: Output $PR_{Fwd}$
16: end if
17: end if

C. Sorting Pipeline

To keep track of the top $K$ most bandwidth consuming heavy hitters on-the-fly, we design a sorting pipeline as shown in Figure 4. The top heavy hitter is denoted as Top HH in Figure 4. Conceptually the top $K$ heavy hitters are kept in an array of $K$ elements. We map the array into a $K$ stage pipeline. Whenever a heavy hitter is reported at the Memory Write stage, it is input to the sorting pipeline. We refer to the input heavy hitters as candidates. As a candidate traverses through the pipeline, it is processed to decide if it should be inserted at the stage or forwarded to the next stage. If it is not inserted at any stage, it is discarded.

The detailed design of each stage is also shown in Figure 4. When a candidate is inserted, other elements in the array either stay in their current positions or move towards the end of the array. Therefore, the possible values at the $i^{th}$ position of the array are, 1) the current value at this position, 2) the current value at the $(i-1)^{th}$ position, and 3) the candidate. The detailed operations in each pipeline stage are shown in Algorithm 3. For a single input stream, after the last candidate finishes traversing the pipeline, the pipeline contains the top $K$ heavy hitters in descending order.

It takes $K$ cycles to process each candidate. So it takes $K+N$ cycles to process a stream of $N$ packets. Since the packet streams come consecutively, during the first $K$ cycles of a data stream, the sorting pipeline carries out two tasks: 1) completing the sorting of the previous packet stream, 2) sorting the current packet stream. These two tasks need to be independent. Therefore, when processing the last packet of each stream, in each stage, we store the value at that stage into a separate register. This way the results of the previous stream can be preserved even if a candidate from the current stream needs to be inserted.

V. Experimental Results

A. Experimental setup

We implemented our heavy hitter detector on an FPGA. Our target device was Virtex-6 XC6SX475 with -2 speed grade. Our implementation used only on-chip Block RAM. All
Algorithm 3 Operations in the $i^{th}$ pipeline stage for top $K$ heavy hitter detection, $i \in [0, K - 1]$

Let $H = A$ heavy hitter.
Let $H.BW = \text{The bandwidth consumption of } H$
Let $H.FlowID = \text{The FlowID of } H$
Let $H_{\text{new}} = \text{The candidate}$.
Let $H_i = \text{The heavy hitter currently at } i^{th}$ stage
Let $H_i^{\text{recv}} = \text{The heavy hitter received by the stage } i \text{ from the previous stage}$.
Let $H_i^{\text{fwd}} = \text{The heavy hitter forwarded to the next stage at stage } i$.

1: if $i == 0$ then
2: $H_i^{\text{recv}} = H_{\text{new}}$
3: else
4: $H_i^{\text{recv}} = H_{i-1}^{\text{fwd}}$
5: end if
6: if $H_i^{\text{recv}}.BW > H_i.BW$ then
7: if $H_{\text{new}}.FlowID == H_i.FlowID$ then
8: $H_i = H_i^{\text{recv}}$
9: $H_i^{\text{recv}}.BW = 0$/* - Make sure any $H$ is stored at most once - */
10: else
11: SWAP($H_i^{\text{recv}}, H_i$)
12: end if
13: end if
14: $H_i^{\text{fwd}} = H_i^{\text{recv}}$

ENDFUNC

Fig. 5: Area and timing of the pipelined DFU

Fig. 6: Area and timing of the sorting pipelined

Table I: Clock rate for various partial result table sizes

<table>
<thead>
<tr>
<th>PRT size (# of flows)</th>
<th>128 K</th>
<th>256 K</th>
<th>512 K</th>
</tr>
</thead>
<tbody>
<tr>
<td>% of BRAM occupied</td>
<td>24%</td>
<td>48%</td>
<td>96%</td>
</tr>
<tr>
<td>Clock rate (MHz)</td>
<td>172</td>
<td>113</td>
<td>89</td>
</tr>
</tbody>
</table>

TABLE I: Clock rate for various partial result table sizes

B. Data Forwarding Unit and the sorting pipeline

Figure 5 shows the clock rate and the logic resource consumption of the pipelined DFU for various numbers of pipeline stages. As described in Section IV-B2, the number of pipeline stages is equal to the memory access latency. Figure 6 shows the clock rate and the logic resource consumption of the sorting pipeline for various numbers of pipeline stages. As described in Section IV-C, the number of pipeline stages equals the number of supported top heavy hitters. We use the number of occupied slices as the metric for resource consumption. Both the pipelined DFU and the sorting pipeline show similar area and time complexity. As the number of pipeline stages increases, the resource consumption grows linearly and the clock rate sustains high, over 289 MHz.

C. Throughput of the architecture

Table I shows the clock rate for various PRT sizes (# of flows) using one BRAM block for the entire PRT. As the PRT size grows, the clock rate drops dramatically. This is due to the complex routing in large BRAM blocks. A large BRAM block is configured by concatenating small BRAM blocks.

To enhance the clock rate we pipeline the BRAM access. Each BRAM stage stores a portion of the PRT. The input FlowIDs and the partial results traverse the pipeline to complete the memory read and write. Since each stage keeps a small BRAM block, the access latency in each pipeline stage is much lower than keeping the entire PRT in a large BRAM block. The memory access in the pipelined BRAM takes multiple clock cycles. However the clock rate of the Memory Read stage is much higher compared with the non-pipelined BRAM access. This increases the clock rate of the pipeline.

Table II shows the clock rates after we pipelined the BRAM access. The number of BRAM stages varies from 8 to 32. When the number of stages is 8, the clock rate is improved compared with the architecture with the non-pipelined BRAM. However, when the number of BRAM stages increase, the clock rates drops. This is due to the long latency in the DFU.
TABLE II: Clock rate for various partial result table sizes using pipelined BRAM access (MHz)

<table>
<thead>
<tr>
<th>PRT size (# of flows)</th>
<th>128 K</th>
<th>256 K</th>
<th>512 K</th>
</tr>
</thead>
<tbody>
<tr>
<td>% of BRAM occupied</td>
<td>24%</td>
<td>48%</td>
<td>96%</td>
</tr>
<tr>
<td>8 BRAM stages</td>
<td>185</td>
<td>157</td>
<td>97</td>
</tr>
<tr>
<td>16 BRAM stages</td>
<td>163</td>
<td>141</td>
<td>116</td>
</tr>
<tr>
<td>32 BRAM stages</td>
<td>118</td>
<td>101</td>
<td>96</td>
</tr>
</tbody>
</table>

TABLE III: Clock rate for various partial result table sizes using pipelined BRAM and pipelined DFU (MHz)

<table>
<thead>
<tr>
<th>PRT size (# of flows)</th>
<th>128 K</th>
<th>256 K</th>
<th>512 K</th>
</tr>
</thead>
<tbody>
<tr>
<td>% of BRAM occupied</td>
<td>24%</td>
<td>48%</td>
<td>96%</td>
</tr>
<tr>
<td>8 BRAM stages</td>
<td>205</td>
<td>222</td>
<td>102</td>
</tr>
<tr>
<td>16 BRAM stages</td>
<td>252</td>
<td>225</td>
<td>191</td>
</tr>
<tr>
<td>32 BRAM stages</td>
<td>263</td>
<td>251</td>
<td>212</td>
</tr>
</tbody>
</table>

as discussed in Section IV-B2. To further improve the clock rate, we use the pipelined DFU discussed in Section IV-B2. Table III shows that the clock rates are significantly increased after pipelining the DFU.

By using both pipelined BRAM access and pipelined DFU, we can achieve over 200 MHz clock rate even when we use 96% of the BRAMs on the FPGA. Assuming a minimum packet size of 40 bytes, when supporting 512 K concurrent flows using 96% of the BRAM, the architecture can perform both heavy hitter detection and top $K$ heavy hitter detection at a throughput of 68 Gbps. If the PRT size is 128 K, the architecture can achieve a throughput of 84 Gbps.

The architecture can also be implemented using external SRAMs to support a large number of concurrent flows. External memory interfaces can be implemented on FPGA supporting high frequency and sufficient bit width for memory device. Therefore, the memory interface does not limit the bandwidth provided by the external memory. DDR2 SRAM can work at over 400 MHz with 36-bit data width and a burst length of 2 [17]. This memory bandwidth is sufficient to support the architecture to operate at 200 MHz. For example, if we use two 72 Mb DDR2 SRAM chips, with a burst length of 2 and memory address width of 20 bits, we can support up to 1 Million concurrent flows.

VI. CONCLUSION

In this paper, we proposed a pipelined architecture for online heavy hitter detection. The architecture can be used to realize various counter based heavy hitter detection algorithms. The architecture can also keep track of the heavy hitters that have the largest amount of activities. In the experiments, using on-chip BRAM, the proposed architecture can support up to 512 K concurrent flows. The architecture sustains high frequency (200 MHz+) while we scale the percentage of the occupied BRAMs from 24% to 96%. If we use external SRAM, then the architecture can support very large number of concurrent flows at the same throughput as the on-chip BRAM based implementation. As future work, we will apply our architecture to accelerate the statistical heavy hitter detection algorithms, such as sketch based algorithms [15], [16]. We will also extend our architectures to realize tunable trade-off between accuracy and throughput for heavy hitter detection.

VII. ACKNOWLEDGMENT

We would like to thank Sanmukh Rao Kuppennagari for his careful reading of the manuscript and editorial comments.

REFERENCES