Dynamically Configurable Online Statistical Flow Feature Extractor on FPGA

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

***Abstract*—Statistical information of network traffic flows is essential for many network management and security applications in the Internet and data centers. In this work, we propose an architecture for a dynamically configurable online statistical flow feature extractor on FPGA. The proposed architecture computes a set of widely used statistical features of the network traffic flows on-the-fly. We design an application specific data forwarding mechanism to handle data hazards without stalling the pipeline. We prove that our architecture can correctly process any sequence of packets. Users can dynamically configure the feature extractor through run-time parameter. The post place-and-route results on a state-of-the-art FPGA device show that the feature extractor can achieve a throughput of 96 Gbps for supporting 64 K concurrent flows.**

I. Introduction

Measuring statistical flow features is the basis of many network management and security applications. State-of-the-art solutions for traffic engineering in data center networks achieve the best performance when per-flow statistics are available [1]. Statistics of network flows are also essential inputs to machine learning based traffic classification algorithms. It has been shown that using statistical flow features greatly improves the accuracy of machine learning traffic classifiers especially when classifying P2P applications [2]. Therefore, statistical flow feature extraction is an essential service of the data plane of the network.

In recent years, 100 Gbps networking is becoming a standard. Both the research community and the industry are targeting 400 Gbps networks [3], [4]. The number of concurrent flows in the networks is massive. Recent research shows that the number is on the order of millions [5]. Because of the huge amount of streaming data and the real-time constraint on packet processing, it is desirable that the network processing be performed online, in one pass over the data stream. Therefore statistical feature extraction needs to be an online service providing high throughput while supporting a large number of concurrent flows.

Due to its extremely high parallelism and on-chip memory bandwidth, Field Programmable Gate Array is well suited for high performance network processing [6], [2], [7]. In this paper we propose a dynamically configurable online statistical flow feature extractor on FPGA. The feature extractor computes a set of widely used flow features on-the-fly. To meet the requirements of various applications, the window size for feature extraction is dynamically configurable. We evaluate the throughput of our architecture on a state-of-the-art FPGA device.

This work was partially supported by the US NSF under grant CCF-1116781.

We summarize our main contributions as follows:

- Hardware solution for online statistical flow feature extraction. A prototype implementation using BRAM achieves a throughput of 96 Gbps while supporting 64 K concurrent flows.
- An application specific data forwarding mechanism to handle data hazards in the pipelined architecture. The mechanism ensures the correctness of the architecture without stalling the pipeline.
- Seamless use of external SRAM to support large number concurrent flows at the same throughput as the BRAM based implementation.

The rest of the paper is organized as follows. Section 2 defines our problem. Section 3 reviews the related work in statistical flow feature extraction. Section 4 describes our online algorithms and the hardware architecture. Section 5 evaluates the feature extractor. Section 6 concludes the paper.

II. Problem Definition

Table I shows the definition of a representative set of flow features. The proposed design computes these features on-the-fly for each flow in the network. In Table I, P_i is the packet information of the i^{th} input packet of a flow. P_i can be packet size, time stamp, etc. These flow features have been widely used in recent researches [8], [2], [9]. The feature extraction is performed within a predefined window for each flow. The flow features are reported once the extraction is completed.

In online feature extraction, each packet needs to go through three steps: packet capturing, flow labeling and feature computation [10]. In packet capturing the necessary header information is retrieved from the packet. In flow labeling the retrieved packet information is attached with a flow ID based on the packet's 5-tuple information. During the feature computation, each feature value of the corresponding flow is updated using the header information of the input packet. We focus on feature computation in this work. We assume that packet capturing and packet labeling are handled by preceding systems.

The proposed design supports a dynamically configurable parameter, $WindowSize$, to define how many packets should

TABLE I: List of Statistical Features

Total number of packets	N
Sum of the values	$\sum_{i=0}^{N-1} P_i$
Average value	$Avg_P = \frac{\sum_{i=0}^{N-1} P_i}{N}$
Variance of the values	$Var_p = \frac{\sum_{i=0}^{N-1}(P_i - Avg_P)^2}{N}$
Maximum value	$P_{max} = \underset{i\in[0,N-1]}{max}\{P_i\}$
Minimum value	$P_{min} = \underset{i\in[0,N-1]}{min}\{P_i\}$

be processed before reporting the feature values of the flow identified by the current input $FlowID$.

III. Related Work

There have been few efforts in measuring network flow statistics in the research community [10] as well as in the industry [8].

In [10], the authors propose an open source parallel software tool for extracting feature values of network flows in real-time. They implement their algorithm on serial, parallel, pipelined and hybrid architectures. Through analysis and experiments, they compare the performance of the four architectures and provides insight into the parallelized feature extraction in software. However, they only achieve a throughput of 750 Mbps. This is not sufficient to meet the data rates in current networks.

Cisco Netflow [8] is a standard network protocol for collecting IP traffic information. It is the most widely used flow measurement solution at the present time. Netflow has two drawbacks. First it only reports the flow record when the flow ends. This prevents it from serving many network security applications which need to make a decision at an early phase of the flow. Second, it cannot process every packet on high bandwidth links. Sampling techniques to handle high speed links cannot guarantee high accuracy of the flow features.

We are not aware of any prior application specific architecture for flow feature extraction which achieves high throughput, supports large number of flows, and accurately computes the flow features at the same time.

IV. Algorithm and Architecture

A. Algorithms and pipelined architecture

In the proposed design, we store the partial results 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 results of a flow. The $FlowID$ is used as the index to access the table.

During feature extraction, the architecture updates the partial results according to the input packet information. Some of the operations in Table I need to be performed every time the PRT entry is updated. Other operations only need to be performed once when the feature values need to be

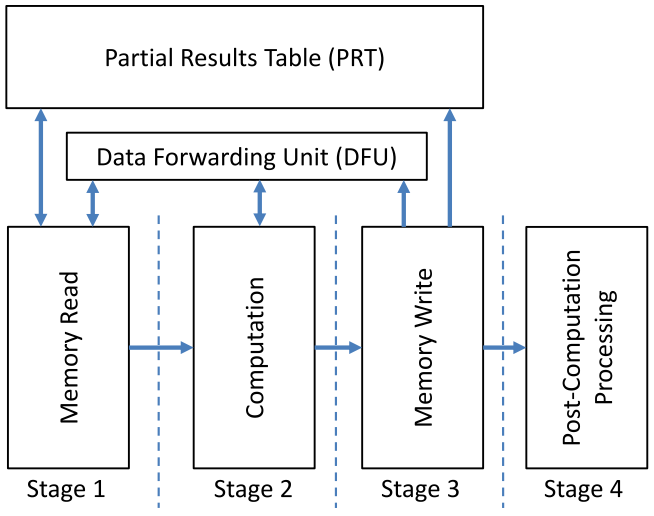

Partial Results Table (PRT)
Data Forwarding Unit (DFU)
Memory Read
Computation
Memory Write
Post-Computation Processing
Stage 1
Stage 2
Stage 3
Stage 4

Fig. 1: Pipelined architecture of the feature extractor

reported. For average value, $Avg_P = \frac{\sum_{i=0}^{N-1} P_i}{N}$, only the sum operation needs to be performed for every update. The division only needs to be performed when the feature value needs to be reported. For variance, we can modify the definition as $Var_P = \frac{\sum_{i=0}^{N-1}(P_i - Avg_P)^2}{N} = Avg_{P^2} - Avg_P^2 = \frac{\sum_{i=0}^{N-1} P_i^2}{N} - Avg_P^2$. In this new equation, only the sum of the squares of the input value needs to be performed for every update. The square of Avg_P, the division and the subtraction can be performed only once when the feature value needs to be reported. The operations of the pipeline stages 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 result from the memory. In Stage 2, the partial result associated with the input $FlowID$ is updated. In Stage 3, the updated partial result from Stage 2 is written back into the memory. In Stage 4, one-time operations are performed to complete the feature extraction. In the proposed architecture the memory read/write and the computation take one clock cycle each. The Data Forwarding Unit (DFU) forwards the correct partial results into the Computation stage. It is discussed in detail in Section IV-B.

B. Forwarding mechanism

In the online algorithm, each incoming packet updates the most recent partial results of a flow with the same $FlowID$. 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 these data hazards, we design a Data Forwarding Unit (DFU) to forward the correct partial results into the Computation stage.

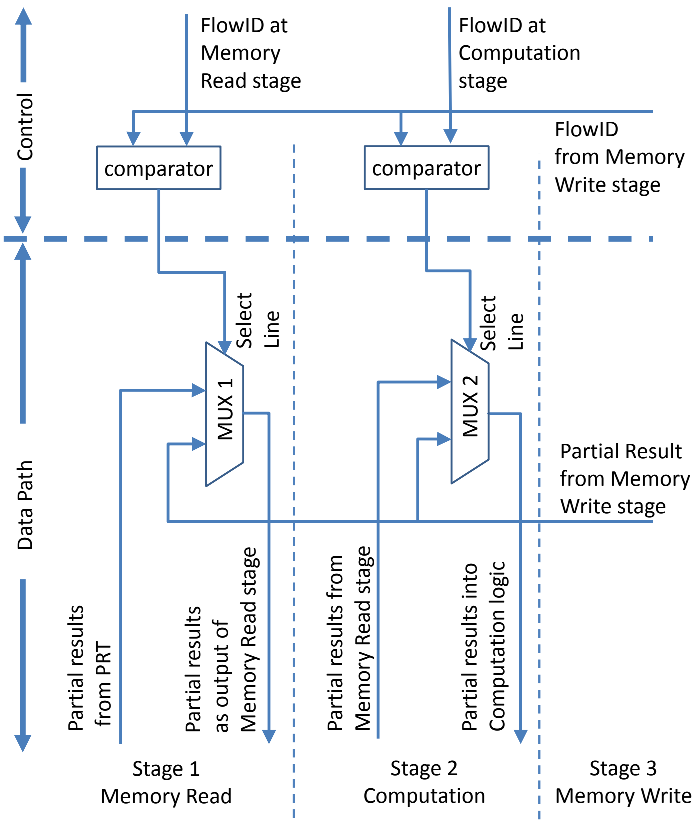
1) Single cycle memory access: The architecture of the DFU for memory with single cycle access latency is shown in Figure 2. Two comparators determine whether the $FlowID$s at the Memory Read and the Computation stage match the $FlowID$ at the Memory Write stage. When no match occurs,

Algorithm 1 Operation for online feature extraction

```
For a flow identified by a certain $FlowID$
Let $WindowSize$ = the window size for feature extraction
Let $NumPkts$ = the number of processed packets
Let $Max/Min$ = the maximum/minimum $PacketInfo$ in the current window
Let $Sum$ = the sum of $PacketInfo$ since the start of the flow
Let $SumSqr$ = the sum of the squares of $PacketInfo$ since the start of the flow
Stage 1: Memory Read
 1: Retrieve partial results of the $FlowID$ from memory
 2: $PacketInfoSqr = PacketInfo * PacketInfo$
Stage 2: Computation
 1: if $PacktInfo > Max$ then
 2:     $Max = PacketInfo$
 3: end if
 4: if $PacktInfo < Min$ then
 5:     $Min = PacketInfo$
 6: end if
 7: $NumPkts = NumPkts + 1$
 8: $Sum = Sum + PacketInfo$
 9: $SumSqr = SumSqr + PacketInfoSqr$
Stage 3: Memory Write
 1: if $NumPkts == WindowSize$ then
 2:     $Max = 0$
 3:     $Min$ = Maximum possible value of $PacketInfo$
 4:     $NumPkts = 0$
 5:     $Sum = 0$
 6:     $SumSqr = 0$
 7: end if
 8: Write $Max, Min, NumPkts, Sum, SumSqr$ back into memory
Stage 4: Post-Computation Processing
 1: if $NumPkts == WindowSize$ then
 2:     $Avg = Sum/NumPkts$
 3:     $Var = SumSqr/NumPkts - Avg^2$
 4:     report $Max, Min, Sum, Avg, Var$
 5:     Set the $FeatureValuesValid$ flag
 6: end if
```

Multiplexer 1 (MUX 1) forwards the partial results from the PRT as the output of the Memory Read stage and Multiplexer 2 (MUX 2) forwards the output of the Memory Read stage to the Computation stage. If a match occurs between the $FlowID$s of the Memory Read stage and the Memory Write stage, Multiplexer 1 forwards the partial results from the Memory Write stage as the output of the Memory Read stage instead of the partial results from the memory. If a match occurs between the $FlowID$s of the Computation stage and the Memory Write stage, Multiplexer 2 forwards the partial results from the Memory Write stage into the Computation stage instead of the partial results from the Memory Read stage.

2) *Multi-cycle memory access:* The DFU for memory with multi-cycle access latency is shown in Figure 3. We assume



Control
FlowID at Memory Read stage
FlowID at Computation stage
FlowID from Memory Write stage
comparator
comparator
Select Line
Select Line
MUX 1
MUX 2
Data Path
Partial Result from Memory Write stage
Partial results from PRT
Partial results as output of Memory Read stage
Partial results from Memory Read stage
Partial results into Computation logic
Stage 1 Memory Read
Stage 2 Computation
Stage 3 Memory Write

Fig. 2: DFU for single cycle memory access

that the memory can support a read and a write operations in each cycle. The DFU contains shift registers to store the previously updated partial results and their associated $FlowID$s. In each clock cycle, the current partial result and the $FlowID$ at the Memory Write stage are pushed into the data forwarding unit, all the entries in the shift registers shift to the end and the oldest partial result and $FlowID$ are removed. In each clock cycle, the $FlowID$ in the Computation stage is compared with all the $FlowID$s stored in the DFU. If a matching $FlowID$ is found, then its associated partial result is forwarded into the Computation stage instead of the partial result returned by the memory. In the case that multiple matches occur, the partial result associated with the most recent matching $FlowID$ is forwarded. A priority encoder is used to give higher priority to a more recent $FlowID$ and partial result for generating the select signal to the multiplexer. Therefore, the multiplexer always picks the partial result of the most recent $FlowID$ among all the matching $FlowID$s.

C. *Pipelined PRT access*

If we use a single BRAM block to store the PRT, we need to implement a large BRAM block. On FPGA, a large BRAM block is configured by concatenating small BRAM blocks. The long wires significantly increase the routing delay, which in turn degrades the clock rate.

To increase the clock rate we map the PRT into a pipeline. An example of the mapping is shown in Figure 4. The PRT

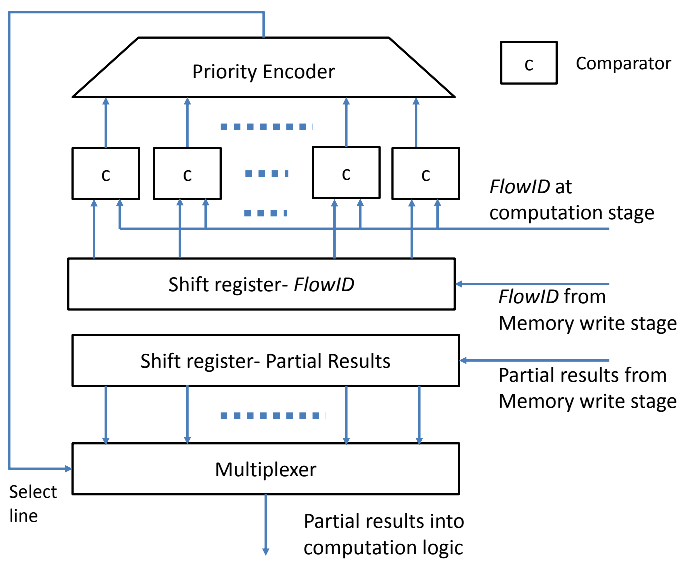

Priority Encoder
c
Comparator
c
c
c
c
FlowID at computation stage
Shift register- FlowID
FlowID from Memory write stage
Shift register- Partial Results
Partial results from Memory write stage
Multiplexer
Select line
Partial results into computation logic

Fig. 3: DFU for multi-cycle memory access

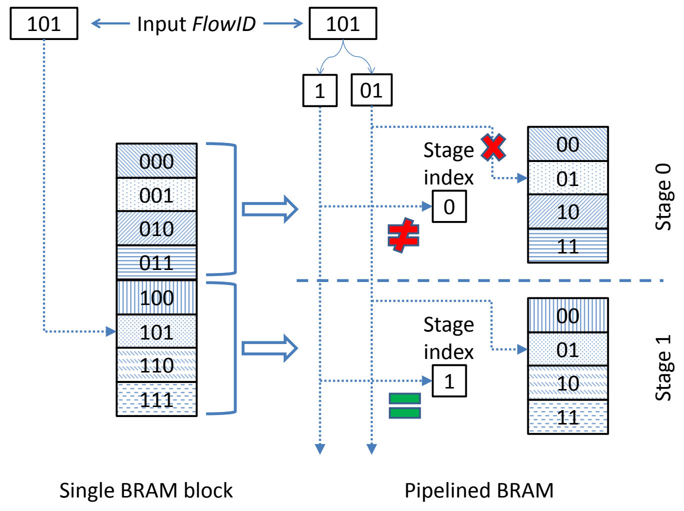


Fig. 4: An example of pipelined PRT access

in the example has 8 entries and is mapped into a two stage pipeline. Generally, if the pipeline has 2^M stages, then the first M bits of the PRT entry index (i.e. the $FlowID$ of the flow store in the entry) are used as the stage index. The PRT entries with the same index are stored in the same stage. The rest of the bits of the entry index are used as the physical address to store the entry in that stage. In any stage, when the first M bits of the input $FlowID$ match the stage index, the memory read and write are performed. Since each stage keeps $1/2^M$ of the PRT entries, 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.

D. *Proof sketch*

In this section, we prove that the proposed DFU can ensure the correctness of the computation without stalling the pipeline for any input sequence of $FlowID$s.

1) Single cycle memory access: As described in Section IV-A, memory read/write and the computation take one clock cycle each. Therefore during any clock cycle there are 4 packets in the pipeline. So we only need to consider a window of 4 packets in the input sequence. Let N denote the size of the number of packets in the input stream. Let P_n denote the n^{th} packet in the input sequence, $n \in [1, N]$. For a window of 4 packets, P_n through P_{n-3}, there are 3 basic cases in which data hazard can occur:

- Case 1: P_n and P_{n-1} have the same $FlowID$.
- Case 2: P_n and P_{n-2} have the same $FlowID$.
- Case 3: P_n and P_{n-3} have the same $FlowID$.

Other cases can be viewed as a combination of these cases. For example, if P_n, P_{n-1} and P_{n-3} have the same $FlowID$, then this case can be viewed as Case 1 followed by Case 2. In Case 3, when P_n enters the Memory Read stage, P_{n-3} has already completed the memory write. The updated partial results are available for P_n. Therefore, Case 3 does not cause a data hazard.

Figure 5 illustrates how the DFU handles the data hazard in Case 1. When P_n enters the Memory Read stage, the correct partial results are being updated by P_{n-1} in the Computation stage. So the partial results returned from PRT are not the correct data to be used. During this clock cycle no match of $FlowID$s is detected by the DFU. Therefore, as described in Section IV-B, the incorrect partial results from the PRT are forwarded as the output of the Memory Read stage. During the next clock cycle P_n enters the Computation stage and P_{n-1} enters the Memory Write stage. The DFU detects the matching $FlowID$s between the Computation stage and the Memory Write stage. It then forwards the correct partial results from the Memory Write stage into the Computation stage. Thus, P_n is processed based on the correct partial results.

The correctness of the computations for Case 2 can be proved similarly. The DFU directly forwards the partial results from the Memory Write stage to either Memory Read stage or Computation stage in one clock cycle. Therefore no stalling is required.

2) Multi-cycle memory access: In this section we prove that the DFU introduced in Section IV-B2 can ensure the correctness of the architecture without stalling the pipeline regardless of the memory access latency. We assume that the latency for memory read and write is R and W cycles respectively.

As described in Section IV-B2, the DFU can make sure to forward the most recent partial result stored in the shift registers which has the same $FlowID$ as the input $FlowID$ to the Computation stage. Therefore, for a given input, as long as we store a sufficient number of previously updated partial results to cover all possible data hazards, the correctness of the computation can be guaranteed.

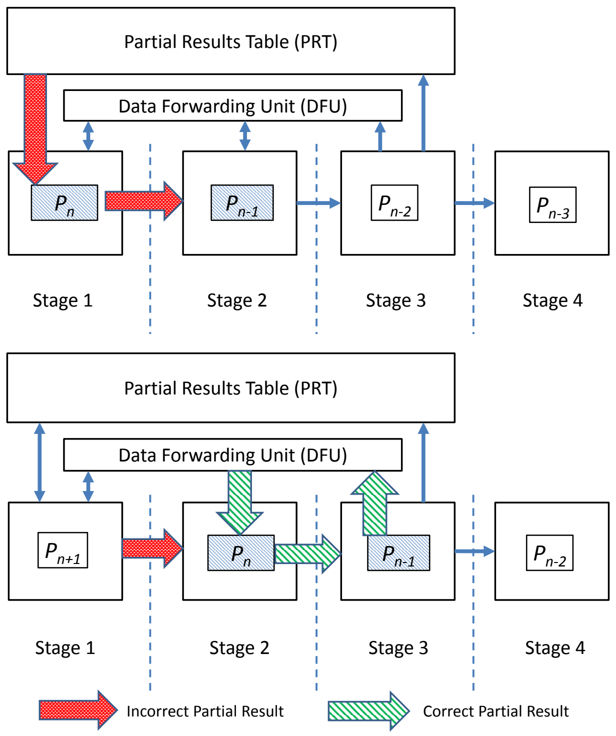


Fig. 5: Proof sketch: Case 1

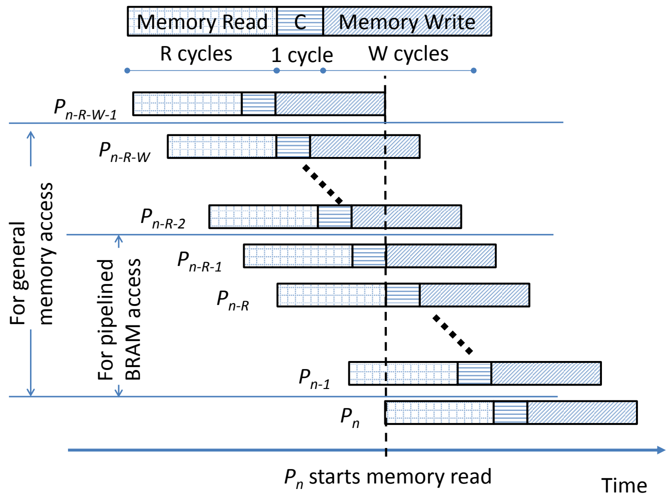


Fig. 6: Processing of $R + W + 2$ packets using memory of multi-cycle access latency

TABLE II: Clock rate for various partial table sizes

PRT size (# of flows)	32 K	64 K	128 K	256 K
Clock rate (MHz)	250	153	135	89

Figure 6 illustrates the processing of $R + W + 2$ packets. When P_n enters the pipeline, $P_{n-R-W-1}$ just completes the memory write. So the partial results updated using the packets before P_{n-R-W} are available to P_n at the Memory Read stage. Therefore data dependencies between P_n and the packets before P_{n-R-W} do not cause data hazards. We don't need to store these partial results in the DFU.

Partial results updated using P_{n-R-1} through P_{n-1} are not available to P_n at the Memory Read stage since they are either in the Memory Read stage or in the Computation stage. We need to store these partial results in the DFU.

The availability of the partial results updated using P_{n-R-W} through P_{n-R-2} depends on the memory. If we use a general memory with W-cycle write latency, the data to write into the memory is not available for retrieving until the operation is completed in W cycles. Using such a device, the partial results updated using P_{n-R-W} through P_{n-R-2} cannot be available to P_n in the Memory Read stage. We need to store these partial results in the DFU. If we use the pipelined BRAM access described in Section IV-C ($R = W$), the memory write at each stage takes one cycle. When P_n reads from any BRAM stage all the partial results written into that stage during the previous clock cycles are available for retrieving. Therefore, the partial results updated using P_{n-R-W} through P_{n-R-2} are available to P_n. We don't need to store these partial results in the DFU.

Given the analysis in this section, we can configure the DFU as follows to ensure the correctness of the computation:

- For memory with R-cycle read latency and W-cycle write latency, store the most recent $R + W + 1$ partial results and their $FlowID$s in the DFU.
- For pipelined BRAM access with R-cycle read latency and W-cycle write latency ($R = W$), store the most recent $R + 1$ partial results and their $FlowID$s in the DFU.

V. Experimental Results

A. Experimental setup

We implemented the proposed design on FPGA. Our target device is Virtex-6 XC6SX475 with -2 speed grade. The implementation uses only on-chip Block RAM for storage. BRAM can be configured to be single or dual ported with various word sizes. In the proposed design, each BRAM block has two Read/Write ports. We use one port as the read port, and the other as write port to realize concurrent read and write. All reported results are post place and route results using Xilinx ISE 14.2.

B. Throughput

Table II shows the clock rate for various PRT size (# of flows). As the PRT size grows, the clock rate drops dramatically. This is because of the complex routing in large BRAM blocks as mentioned in Section IV-C.

Table III shows the clock rates after pipelining the PRT access using 16 stages. The clock rate for all PRT size is significantly improved compared with the non-pipeline PRT access. As shown in Table III, when PRT is small, using 8 pipeline stages achieves higher clock rate than using 16 stages. When PRT is large, using 16 stages achieves higher clock rate. This is because as we increase the number of pipeline stages

TABLE III: Impact of pipelined PRT access on clock rate

PRT size (# of flows)	32 K	64 K	128 K	256 K
8 stages	355	303	200	132
16 stages	291	256	237	220

TABLE IV: Impact of number of bits to represent the input packet information on clock rate

PRT size (# of flows)	32 K	64 K	128 K	256 K
16 bits	291	256	237	220
12 bits	277	274	236	246
8 bits	259	280	229	247

to decrease the latency in each PRT stage, we need to increase the width of the priority encoder in the DFU. The wider the priority encoder, the longer the latency it adds to the DFU. This lowers the working frequency of the entire pipeline. So when we increase the number of pipeline stages from 8 to 16:

- For small PRT, the latency in each PRT stage becomes smaller than the latency in DFU. Due to the increased latency in DFU, the clock rate drops.
- When the PRT is large, the latency in each PRT stage stays larger than the latency of each DFU stage. Due to the decreased latency of each PRT stage, the clock rate increases.

Therefore, to optimize the throughput, we need a larger number of stages for a large PRT than for small PRT. Assuming a minimum packet size of 40 bytes, for supporting 64 k concurrent flows, the architecture achieves a throughput of 96 Gbps.

We also implemented the architecture using different number of bits to represent the input packet information. The results are shown in Table IV. The architecture sustains high frequency for all the bitwidth in Table IV.

The architecture can also be implemented using external SRAMs to support a large number of concurrent flows. DDR2 SRAM can work at over 400 MHz with 36-bit data width and a burst length of 2 [11]. Using a dual ported SRAM interface, the SRAM controller allows two read and two write operations per cycle at over 200 MHz. Considering two sets of SRAM chips and interfaces, we can read and write 144 bits per clock cycle. This memory bandwidth is sufficient to support the architecture to operate at 200 MHz. 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 statistical feature extraction. The proposed architecture can support both functions at a throughput of 96 Gbps while supporting 64 K concurrent flows using BRAM. If we use external SRAM, then the number of supported flows can be increased, bounded only by the capacity of the memory.

The architecture can be extended to perform online heavy hitter detection [12], [13]. A heavy hitter is a traffic flow whose activity accounts for more than a pre-defined proportion of the total activity by all the flows. The PRT can be used to store the total activity of each traffic flow. The Computation stage can be extended as the counter for the activity of each traffic flow. The Post-computation stage can be extended to check the total activity of each traffic flow against the threshold for heavy hitter detection.

As future work, we will generalize our assumptions on the latency of Computation stage and the DFU to multiple clock cycles. The generalized assumption will enable us to design a deep pipelined Computation stage, which may achieve higher throughput than the proposed architecture in this work. We will also extend our architecture to support sketch based network measurement applications [14].

References

[1] M. Moshref, M. Yu, A. Sharma, and R. Govindan, "Scalable rule management for data centers," in *10th USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[2] D. Tong, L. Sun, K. Matam, and V. Prasanna, "High throughput and programmable online trafficclassifier on FPGA," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '13, 2013.

[3] "FP3: Breakthrough 400G network processor," http://www.alcatel-lucent.com/fp3/.

[4] M. Attig and G. Brebner, "400 gb/s programmable packet parsing on a single FPGA," in *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, 2011.

[5] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '02, 2002.

[6] T. Wolf, R. Tessier, and G. Prabhu, "Securing the data path of next-generation router systems," *Computer Communications*, vol. 31, no. 4, apr 2011.

[7] H. Le and V. Prasanna, "Scalable high throughput and power efficient Ip-lookup on FPGA," in *17th IEEE Symposium on Field Programmable Custom Computing Machines, 2009. FCCM '09.*, 2009.

[8] "Introduction to CISCO IOS NetFlow - a technical overview," http://www.cisco.com/en/US/products/ps6601/products_ios _protocol_group_home.html.

[9] D. Angevine and A. N. Zincir-Heywood, "A preliminary investigation of Skype traffic classification using a minimalist feature set," in *International Conference on Availability, Reliability and Security*, 2008.

[10] S. Li and Y. Luo, "High performance flow feature extraction with multi-core processors," in *Networking, Architecture and Storage (NAS), 2010 IEEE Fifth International Conference on*, 2010.

[11] Renesas Products, http://www.renesas.eu/products/memory/ fast_sram/qdr_sram/index.jsp.

[12] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, ser. IMC '04, 2004.

[13] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Finding hierarchical heavy hitters in data streams," in *Proceedings of the 29th international conference on Very large data bases - Volume 29*, 2003.

[14] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005.