Flow Monitoring in High-Speed Networks using Two Dimensional Hash Tables

David Nguyen and Gokhan Memik

Department of Electrical and Computer Engineering Department
Northwestern University
Evanston, Illinois
{dnguyen, memik}@ece.northwestern.edu

Abstract. Flow monitoring is a required task for a variety of networking applications including fair scheduling and intrusion/anomaly detection. However, due to the complexities of implementing efficient flow monitoring hardware, most routers do not implement hardware-based flow monitoring. Existing flow monitoring techniques are implemented in software, which cannot be utilized for real-time monitoring in high-speed networks. In this paper, we present the design of a flow monitoring scheme, which is based on a two-dimensional hash table technique. The hash tables on the second dimension are accessed in parallel to increase the accuracy. Since they are accessed in parallel, the overall access latency is also reduced. We present four novel techniques based on this two-dimensional hash table scheme. We also present the FPGA implementation of these designs. We show that our design achieves high accuracy for high-speed networks. Particularly, using a simulation environment that processes packet traces, we show that on average our implementation can find the flow information within 8% of the actual value while achieving link speeds exceeding 60 Gbps for a workload with constant packet sizes of 40 bytes.

Keywords: Network Monitoring, FPGAs, Reconfiguration, Hash Tables, Flow Monitoring.

1. Introduction

There is a tremendous growth in the complexity of networking applications. New applications do not only emerge from the industry, but also from individuals in the public domain. With this growth, today’s routers (particularly edge routers) are expected to perform much more complicated tasks compared to a decade ago. While core routers are still designed with one purpose in mind-to move traffic as quickly as possible-the requirements at the edge are diverse. In addition, the edge routers must support continually evolving requirements due to the introduction of new applications. Some of these applications (such as QoS, fair packet scheduling, intrusion/anomaly detection) require flow information\(^1\) [15]. Therefore, there is an increasing need for efficient flow monitoring techniques that can be used in high-speed networks. Flow monitoring enables several applications such as accounting and billing, network planning, peering assignments, traffic engineering, network monitoring, user monitoring and profiling,

\(^1\) In this paper, the terms flow and session are used interchangeably; both correspond to a TCP session. Hence, flow information is statistics (such as total traffic generated) about a TCP connection collected at a router.
and security analysis. The flow information is particularly useful for intrusion-detection systems (IDS) and firewalls. However, despite the advantages of utilizing the flow information, most routers do not implement applications based on accurate flow information, because there still does not exist flow monitoring implementations that can satisfy both the accuracy and the latency requirements; existing solutions are either too slow or too inaccurate.

In this paper, we investigate a hash-based flow monitoring technique that can be utilized in high-speed networks. Specifically, we will develop four novel techniques based on a two dimensional hash table that is used to gather flow information. By using several packet traces, we show that our design can achieve the desired accuracy. We implement our design using a Xilinx VirtexII XC2V8000 chip and also show that speeds up to 73 Gbps can be achieved without compromising the accuracy. We show that different designs perform better for certain workloads/requirements. Therefore, the unit can be modified to adapt to different packet flows and requirements. By utilizing the reconfiguration capabilities of FPGA’s, such modifications can be implemented effectively. Particularly, in this paper we make the following contributions:

- Propose a novel application of two-dimensional hashes,
- Present four novel designs based on hashing and implement it on a representative FPGA device,
- Investigate the accuracy of each design by using various packet traces, and
- Show different modifications on the design to adapt to different workloads and tasks.

The requirements on the flow monitoring application vary according to the task executed on the router. For example, for a fair scheduling algorithm, the flows with large amounts of data are more important than the flows with small amounts of data. On the other hand, a firewall might ignore the large data flows and concentrate on the flows with little data to capture malicious packets. Therefore, any flow monitoring hardware design should be flexible to adapt to different applications. In addition, the behavior of the design also changes according to the packets that are processed. Therefore, adaptability is also necessary for obtaining high accuracy in different workloads. Because of these adaptability requirements, FPGAs are attractive alternatives to implement such designs.

There is a plethora of flow monitoring algorithms [7] such as NetFlow [5]. However, almost all of the existing algorithms are implemented in software. Designing hardware for these programs is very hard if not impossible. Because of this limitation, they cannot be used for real-time monitoring in high-speed networks. With high-speed networks at gigabit rates and the increase of network attacks, detection systems must act quickly when an attack occurs. Current software trace dump tools are not fit for this type of hostile environment that requires quick decision making and anomaly detection. One solution is to use FPGAs to implement a high-performance, reconfigurable solution for flow monitoring. With so many flows to keep track of, a quick, efficient design and estimation model can satisfy the requirements of a swift response detection system.

Currently, high performance FPGAs offer competitive solutions to other networking hardware implementations. Programmable logic provides high flexibility, a desirable feature for networking applications, and high-speed execution of tasks through specialized functionality in hardware. Certain network processor vendors have offered FPGA based implementations such as the SpeedRouter chip from IP Semiconductors [24],
and the Comet Stream Processor from Fujitsu [9]. FPGAs are also used to implement certain key networking tasks, taking the role of an accelerator [4, 14, 18]. Other types of reconfigurable logic have been used by various vendors for different network processing elements [3, 20, 23].

This paper is organized as follows. In the next section, we explain the flow monitoring techniques. Section 3 presents the FPGA implementation. In Section 4, we discuss the experimental results. Section 5 gives an overview of the related work. Section 6 concludes the paper with a summary.

2. Flow Monitoring Unit (FMU)

In this section, we present an overview of the Flow Monitoring Unit (FMU). In the next section, we will present the hardware implementation of different versions of our design. Our algorithm is based on two-dimensional hashing. Our main goal is to use multiple hash units in parallel and thereby reduce the probability that two different flows are mapped to the same hash bucket. Since the hash functions can be accessed in parallel, we also reduce the access latency compared to a similar design with a single hash table. We first discuss the queries that can be serviced by the FMU. We then discuss the overall architecture.

2.1. Queries

The host machine uses the FMU unit by sending queries or requests to it. Then, the FMU returns the result, which is used by the host to implement a specific application. This interface allows the unit to be utilized in various processing engines. Currently, two queries are implemented in the FMU. These are:

\[ \text{UPDATE}(k, v) \] : Increase the value of the key \( k \) by \( v \).

\[ \text{GET}(k) \] : Return the value for the key \( k \).

The key \( k \) in these queries can be any combination of 5 packet header fields: source IP, destination IP, source port, destination port, and protocol. As shown in Figure 1, our design combines all of these fields. The value is the hash value stored for each key. The meaning of this value depends on the application that uses the FMU. For example, the host can use the FMU to store the number of packets for each flow. In that case, all the \( \text{UPDATE} \) queries are of the form \( \text{UPDATE}(k, 1) \). In another case, these values may correspond to the size of each packet.

These two types of queries are essential for many networking tasks. In addition, other queries can be easily implemented by executing a combination of these queries. For example, to reset the hash value for a certain key, the host can execute “\( \text{UPDATE}(k, - \text{GET}(k)) \)” query in two steps. In the first step, the value is read into the host and in the second step the \( \text{UPDATE} \) query is executed. Note that since the output of the hash is not exact, the \( \text{GET} \) query actually returns an estimate. Therefore, the success of any unit is measured in the error of these estimates. We will discuss this further in Section 4.

2.2. Flow Monitoring Techniques

As mentioned in the previous sections, FMU is based on hashing. The advantage of hashing is its constant access time, i.e. regardless of the size of the set being accessed, the access takes the same amount of time. This is not true for “exact solutions” without size limitations. Therefore, there are no bandwidth guarantees for such designs. Hav-
ing constant latency also aids the scheduling of the tasks and hardware pipelining when the FMU is used by a host processor.

For a one-dimensional hash table $T$ with size $S$, the $\text{GET}(k)$ query returns the value $T[h(k)]$, where $h(k)$ is the hash value for the key $k$. Similarly, $\text{UPDATE}(k, v)$ query changes the value of $T[h(k)]$ to $T[h(k)] + v$. For a two-dimensional hash function with dimensions $N \times S$, there are $N$ hash tables each with $S$ elements. Each of these tables is addressed by a different hash function. In such a configuration, queries perform similar operations. However, they are applied to all hash tables in parallel. Particularly, $\text{UPDATE}(k, v)$ query changes the values of all $T_i[h_i(k)]$ to $T_i[h_i(k)] + v$, where $i = 0 \to N-1$. The $\text{GET}(k)$ function, on the other hand, intelligently combines the results from each hash table. For this purpose, we have designed four novel techniques:

**Min FMU:** The simplest of the techniques is called the **min FMU (MIFMU)**. Min FMU reads the corresponding values from all the tables and returns the smallest of them. Particularly, $\text{GET}(k)$ function returns the value:

$$\min \{ T_i[h_i(k)] \}, \ \forall i \in \{0,\ldots,N-1\}$$

**Median FMU:** The second technique is the **median FMU (MEFMU)**. This technique tries to find the correct value by considering all values entered into the tables. Particularly, it returns the value

$$\text{median} \{ T_i[h_i(k)] - \frac{\text{sum}}{S} \}, \ \forall i \in \{0,\ldots,N-1\}$$

Sum in the above equation is the sum of all values entered into any one of the hash tables. Note that this value is equivalent for all hash tables. The intuition behind the MEFMU is to single out the large flows. Once we subtract the average output value from the specific value stored, the information for the flows with large output is estimated accurately.

**Collision Estimate FMU:** Both MIFMU and the MEFMU have drawbacks particularly for flows with small values. Therefore, we developed the **collision estimate FMU (CEFMU)**. The CEFMU tries to estimate the number of collisions (i.e. number of cases where two different flows are mapped to the same hash bucket) for each hash bucket and returns the output values according to this collision estimate. The collision estimate is calculated using a counter and the last key value used for each hash bucket. Whenever there is an access to a hash bucket, the current key is compared against the last key value. If these values are not equal, we increment the collision counter by $v$. Then, the CEFMU returns the value:

$$\min \{ T_i[h_i(k)] - C_i[h_i(k)] \}, \ \forall i \in \{0,\ldots,N-1\}$$

To implement the CEFMU, we use an additional table $(C_i[h_i(k)])$ that stores the collision counters. The goal in this technique is to estimate the collisions and subtract this value from the value in the hash bucket. This significantly increases the accuracy of the FMU for flows with small values.

**Hybrid FMU:** The final technique is the **hybrid FMU (HYFMU)**, which runs the CEFMU and the MIFMU techniques in parallel and returns one of them. The decision is based on the output values estimated by each technique. In our experiments, we have

---

2 In our implementation, to achieve equal timing, we utilized the same hash function with different hash seeds to generate the effect of different hash functions.
seen that for flows with small values, the CEFMU is more accurate than the MIFMU. For the flows with large outputs, on the other hand, the MIFMU is the most successful technique. Therefore, by intelligently selecting the correct output, the HYFMU achieves high accuracy for all flow types. The selection of the hybrid is based on the values of each technique. Particularly, HYFMU selects the CEFMU if its output is small. Otherwise, MIFMU is selected. To implement this scheme, HYFMU subtracts the output of the CEFMU from the output of the MIFMU. If the difference is smaller than 100, then MIFMU output is used, otherwise CEFMU output is selected.

3. FPGA Implementation

We implemented the FMU using the Synplify Pro synthesis tool [22] and Xilinx Design Manager implementation tools. We chose Xilinx VirtexII XC2V8000 FPGA as our target chip. The hardware of the FMU consists of four major parts: 1) the hash function, 2) the hash table (i.e. memory), 3) the selection mechanism used in the GET queries, and 4) the control logic. The first two major blocks are depicted in Figure 1, which presents a single hash function. On the left-hand side of the figure, the inputs to the unit are depicted: the 5-tuples from the packet as explained in the previous section, the clock signal, the query type, and the input value used for UPDATE queries. The 5-tuple inputs are used by the hash function to find the address to access the memory. The query type is used to generate the memory write enable signal. For the memory, we use 32-bit values. In other words, the input and output values to the memory are always 32 bits. The overall FMU architecture is presented in Figure 2. The FMU replicates N hash functions (each hash function is identical to Figure 1). Then, the outputs (glob_output of Figure 1) of each are fed into a selection mechanism as shown in Figure 2. According to the FMU type, the selection mechanism returns the corresponding output.

For the selection of the particular hash function used in our system, we have tested several functions. We found the function by Jenkins [11] most effective, because it consistently resulted in fair distribution of flows to hash buckets. In addition, the hardware implementation of the function was simple enough to be implemented on an FPGA. After implementing the one dimensional hash unit, we observed that the total memory size required by the unit is the limiting factor on its size. In other words, the total memory size of the hash tables dictated the total number of hash functions we can implement in parallel. This limit was 80,000 entries for the particular chip we used. Hence, there was an abundance of hardware resources to be used for hash functions. Therefore, rather than sharing the hash function resources, we decided to create an instance of a hash unit for each parallel hash function as shown in Figure 2. Each hash function uses 1019 four-input LUTs. The hash table is implemented using the Block RAM feature in Xilinx Virtex FPGAs. The selection mechanism uses 192, 352, 212, and 556 four-input LUTs for MIFMU, MEFMU, CEFMU, and HYFMU, respectively.

The two-dimensional hash tables are more suitable to be implemented in hardware compared to one-dimensional tables. Consider a total memory size of 10K entries. One can implement a one-dimensional hash table with 10K entries, or implement 10 hash tables each with 1K elements. As we will show in the next section, the two-dimensional hash outperforms the one-dimensional hashing in terms of accuracy. In addition, it also has smaller latency because 1K tables will be accessed in parallel compared to a single access to a 10K table. This difference can be seen in Table 1, where the first line represents a one-dimensional hash table.
Table 1 presents the critical path for different FMU designs. The total delay for the hash function was initially 50 ns, which was the slowest component in our design. We modified the design by applying extensive pipelining. In this design, processing of different packets can be initiated every cycle. Therefore, pipelining increases the overall throughput significantly. The rightmost column of Table 1 presents the corresponding maximum bandwidth that can be supported by various FMU designs for GET queries. This value is calculated for constant packet sizes of 40 bytes. The UPDATE queries are independent of the FMU type (because they do not use the selection mechanism) and can be executed on 73.9 Gbps link speeds.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Critical Path Delay</th>
<th>Max. Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>N = 1, S = 8K</td>
<td>6.63 ns.</td>
<td>48.3 Gbps</td>
</tr>
<tr>
<td>N = 4, S = 2K, MIFMU</td>
<td>4.33 ns.</td>
<td>73.9 Gbps</td>
</tr>
<tr>
<td>N = 4, S = 2K, MEFMU</td>
<td>4.99 ns.</td>
<td>64.1 Gbps</td>
</tr>
<tr>
<td>N = 4, S = 2K, CEFMU</td>
<td>4.33 ns.</td>
<td>73.9 Gbps</td>
</tr>
<tr>
<td>N = 4, S = 2K, HYFMU (combining min and CE)</td>
<td>4.99 ns.</td>
<td>64.1 Gbps</td>
</tr>
</tbody>
</table>

4. Experimental Results

In this section, we present the simulation results for the different FMU techniques (MIFMU, MEFMU, CEFMU, and HYFMU as explained in Section 2.2) we have de-
developed. To test the performance of each technique, we have implemented a simulator that reads a packet trace and models the events of the techniques. During the simulations, the hash values are incremented by one for each packet, i.e. for each packet \texttt{UPDATE}(k, 1) query is executed, which counts the number of packets in each flow. The simulator also finds the exact number of packets in each flow off-line. We report the results in error rate, which is the average error over all the flows. To find this number, we sum the absolute values of the differences between the estimated number of packets in a flow and the exact number of packets in the same flow. Then, this total error sum is divided by the number of packets in the flow to find the error rate for one flow. The error rate for a configuration is the average error over all flows. For each simulation, we report the average error observed in 5 traces from the NLANR [19] repository: ANL-1075669897, APN-1076296463, BUF-1003116118, BWY-1077399092, MEM-1076772087.

4.1. Comparison of FMU Techniques

First, we compare different FMU techniques. Figure 3 presents the average error rates for the four FMU techniques while varying the table size from 500 entries to 16000 entries. The number of parallel hash functions (N) is set to 4. We see that for most configurations the CEFMU technique has the best performance. The error drops below 10% for a table size of 16,000 entries. Note that, for N=4, this is the largest table size we can generate. For this table size, the hybrid technique (HYFMU) gives the best performance with an error rate of 7.3%. The reason for this lies in the selection methodology for the HYFMU. HYFMU relies on the values returned by each technique. If there are errors in either MIFMU or CEFMU, the flow classification can be made incorrectly, resulting in an increase in the overall error rate. However, if they both have small error rates, HYFMU makes correct classifications.

![Figure 3: Average error rates of various FMU techniques.](image)

![Figure 4: Error rates for small and large flows for MIFMU and CEFMU.](image)

Figure 4 presents the error rates for large and small flows within each trace for the same configuration. Any flow with a number of packets exceeding 0.1% of the total
number of packets is categorized as a large flow. All other flows are categorized as small. For the five traces we have simulated, on average 10% of the packets belong to a large flow. Figure 4 presents the average error of CEFMU and MIFMU techniques for the large and small flows. The results clearly indicate that the MIFMU technique is more successful for large flows, whereas CEFMU is better for small flows. In the MIFMU technique, we simply return the minimum hash value found. Within this value, large flows usually overwhelm the small flows. Particularly, if the corresponding bucket is used by another small flow, its effect on the value returned remains small for large flows. Therefore, the errors for the large flows remain low. However, for the small flows, even if another small flow is mapped to the same hash bucket, the error rate suddenly increases. This is particularly true for small N values, where collisions are more frequent. The CEFMU, on the other hand, can eliminate such effects for small flows. However, for large flows, the CEFMU usually overestimates the effect of collisions, and therefore has higher error rates.

![Figure 5. Error rates for a total table size of 4K entries.](image)

4.2. Sensitivity Analysis

In the second set of experiments, we fix the total size of the hash tables and vary the number of parallel hash functions to investigate the optimal parallelism. The results for a total table size of 4K entries are presented in Figure 5. In this example, when N is 2, there are 2 parallel hash functions accessing tables containing 2K entries. Similarly, when N is 4, each function accesses a table with 1K entries, etc. The results for a total table size of 32 K entries are presented in Figure 6. Regardless of the total table size, for all techniques except MEFMU, increasing the number of parallel hash functions initially results in a reduction in the error rate. Particularly, for the table size of 32K entries, the error rate reduces from 69% to 17%, from 59% to 11%, and from 63% to 19% for MIFMU, CEFMU, and HYFMU, respectively. However, further division of
the tables result in an increase of error rates for the MIFMU technique. The CEFMU technique, on the other hand, improves its performance slightly until N reaches 4. The reason for this is the fact that CEFMU is more immune to collisions. When we reduce the size of each table, the number of collisions in each function increases. The CEFMU is designed to perform well under these circumstances. Therefore the effects of more collisions are significantly less for CEFMU.

5. Related Work

Many networking applications have found their way to hardware implementations [10, 16]. With link speeds increasing and the multitude of network applications, future solutions place a premium on both performance and flexibility. FPGA's qualify for both these requirements. Current generation of FPGAs can operate at speeds ranging from 50 MHz to 250 MHz and have capacity on par with large ASIC designs. For example, FPGA's have been used in developing platforms for experimentation of active networks [8] for such services as detection of Denial-of-Service (DoS) attacks, real-time load balancing for e-commerce servers, real-time network based speed recognition servers for v-commerce, etc. Also, high speed front-end filters and security management applications for ATM firewalls have found their way onto FPGA's to reduce performance penalties at the IP level [17].

As for flow monitors, TCP/IP Splitter [21] has been implemented as part of the FPX (Field-Programmable Port Extender) project to perform flow classification, checksums, and packet routing. However, this implementation is limited to 3 Gbps monitoring. Jupiter T-Series routers also implement a propriety flow monitoring mechanism. Although the router runs at 10 Gbps link speeds, the monitoring is limited to 250K packets per second for each physical interface card. It is also limited in the maximum number of flows (400K) and flow creation rate (12K new sessions per second). Before these, flow monitoring tools have been in software, such as HTTPDUMP.

The theory of single dimensional hashing, is well established and documented in the literature [2, 12]. Multi-dimensional hash functions are also known. Recently, software techniques are proposed to estimate changes in flow information [1, 6, 13]. Some of these techniques are based on hashing. However, these techniques cannot be used for capturing information of flows with small amount of packets. In addition, they are implemented in software and cannot be used in high-speed networks.

6. Conclusions

Flow monitoring is one of the important tasks in computer networks. There is a plethora of algorithms to implement flow monitoring. However, almost all of these techniques are implemented in software and designing hardware for them is very hard if not impossible. With the increase in the link speeds and the wider usage of flow information, hardware flow monitoring is becoming essential for most state-of-the-art routers. In this paper, we presented a flow monitoring design implemented using FPGAs. The unit is based on two-dimensional hashing. Because it uses hashing, it has two major advantages over alternative techniques. First, high speeds can be achieved. Second, the access latency to any data is constant. Clearly, the disadvantage of hashing is its inaccuracy. We have shown four novel techniques of varying complexity to address this problem. Particularly, we have presented different selection mechanisms among hash functions that are executed in parallel. The best technique (HYFMU), which combines collision estimation and min. estimation, can process 200M packets
per second (corresponding to a link speed of 64 Gbps for 40-byte packets), while hav-
ing an average error rate of 7.3%.

References