High Throughput Sketch Based Online Heavy Change Detection on FPGA

Da Tong, Viktor Prasanna

Ming Hsieh Department of Electrical Engineering

University of Southern California

Los Angeles, CA 90089, USA.

Email: datong@usc.edu, prasanna@usc.edu

Computer Engineering Technical Report Number CENG-2015-04

Ming Hsieh Department of Electrical Engineering – Systems
University of Southern California
Los Angeles, California 90089-2562

Aug 2015

High Throughput Sketch Based Online Heavy Change Detection on FPGA*

Da Tong and Viktor Prasanna

University of Southern California, Los Angeles CA 90089, USA
{datong, prasanna}@usc.edu

Abstract. Significant changes in traffic patterns often indicate network anomalies. Detecting these changes rapidly and accurately is a critical task for network security. Due to the large number of network participants and the high throughput requirement of today's networks, traditional per-item-state techniques are either too expensive when implemented using fast storage devices (such as SRAM) or too slow when implemented using storage devices with massive capacity (such as DRAM). Sketch, as a highly accurate data stream summary technique, significantly reduces the memory consumption when supporting a large number of items. Therefore, the Sketch based techniques can be supported by the fast on-chip storage of state-of-the-art computing platforms to achieve high throughput. In this work, we propose a fully pipelined Sketch based architecture on FPGA for online heavy change detection. Our architecture forecasts the activity of the network entities based on their history, then reports the entities whose difference between their observed activities and the forecast activities exceed a threshold. The post place-and-route results on a state-of-the-art FPGA show that our architecture sustains high throughput of 96 - 103 Gbps using various configurations for online heavy change detection.

1 Introduction

Abrupt heavy changes in the network traffic patterns are often caused by network anomalies such as attacks or network failures. The "heavy change" refers to the significant inconsistency between the observed behavior and the normal behavior (usually derived based on history) in a short period of time. Detecting the heavy changes is a very important technique to detect network anomalies such as DoS attacks [1], network changes [2], etc.

In recent years, 100 Gbps networking has become a standard. 400 Gbps is well accepted as the next mile stone [3]. For such a high throughput requirement, there is a large amount of streaming data to process with tight real-time data processing constraints. Thus, it is desirable that the data stream processing, including heavy change detection, be performed online and at high throughput.

To perform heavy change detection, both the current state and the history of the network need to be maintained. Thus, given the current scale and complexity

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

of the network, traditional per-item-state techniques need a large amount of memory. For example, the number of concurrent flows on a typical backbone link is around 1 M [4]. If we use one 32-bit counter for the current state and three 32-bit counters for the history, then we need 128 Mb memory in total. Such a large memory footprint can only be handled by external storage devices such as DRAM. Due to the relatively low memory bandwidth provided by DRAM, it is very difficult for the per-item-state solution to achieve a very high throughput.

To achieve a high throughput, we need a solution that is able to support a large number of items on the on-chip storage of state-of-the-art computing platforms. Sketch is a data stream summarization technique with sub-linear memory complexity [5]. When supporting a large number of items, its memory consumption is much lower than the per-item-state counter based techniques with very little accuracy trade-off [2], [5]. In [2], the authors proposed a heavy change detection technique based on K-ary Sketch. The technique shows excellent accuracy with very low memory consumption. In this paper, we modify the original algorithm proposed in [2] to support online feature and propose an architecture to accelerate our modified algorithm.

To achieve high possible throughput, we choose FPGA as our target platform. State-of-the-art Field Programmable Gate Arrays (FPGAs) offer high operating frequency, unprecedented logic density and massive on-chip memory bandwidth. It is a promising platform to support the parallel arithmetic operations and memory operations needed for many networking applications [6], [7].

Our main contributions are as follows:

- An online solution for Sketch based heavy change detection based on the algorithm proposed in [2]
- A high throughput architecture for our proposed online heavy change detection solution on FPGA. It sustains 96 - 103 Gbps throughut for various configurations.
- A fully pipelined architecture for online univariate time series forecasting based on a sliding window of data stream history.
- 5× to 250× throughput improvement compared with other existing architectures for heavy change detection on various computing platforms.

The rest of the paper is organized as follows. Section 2 defines our problem. Section 3 reviews the related work in heavy change detection. Section 4 gives a detailed explanation of our proposed algorithm. Section 5 presents our architecture. Section 6 evaluates our architecture. Section 7 concludes the paper.

2 Problem Definition

In this section we define the online heavy change detection problem considered in this paper.

We use the cash register streaming model [2] to describe the network traffic. Let $S = p_0, p_1 ...$ be a stream of items. Each item $p = < k, a >$ consists of a key k, and a positive activity a. Associated with each unique key value k is a time

varying signal $A[k]$. The arrival of each new data item p causes the underlying signal $A[k]$ to be updated: $A[k]+ = a$. Let I denote a time interval containing a consecutive sequence of items from S. Then the stream S can be viewed as a series of concatenated time intervals I_0, I_1 ... . We want to find the keys whose total activities change dramatically over time intervals. These keys are referred to as heavy change keys.

Formally, the heavy change keys can be defined as follows:

Definition 1 *Given an input data stream S, for any time interval I_j in S, for any key k in I_j, let $O_k^{I_j}$ denote the observed total activity of key k in time interval I_j, let $F_k^{I_j}$ denote the forecast total activity of key k in time interval I_j. If $|O_k^{I_j} - F_k^{I_j}|$ is greater than a threshold Φ, then we define key k as a heavy change key.*

Our goal is to find the heavy change keys in each interval through online processing at a high throughput. In this paper, online processing means that every clock cycle one input item is processed and the processing for one interval starts right after the processing for the last interval completes.

The threshold Φ is application specific. There are various ways to generate the threshold in the literature. [8], [9] use a fraction of the total differences as the threshold. [2] uses a scaled second moment of the total difference as the threshold. [1] determines the threshold based on the features of its target application. It is also possible for some applications to determine the threshold in advance (for example, using machine learning techniques). Therefore in this paper, we don't specify the method to generate the threshold. Instead, we assume that for any time interval, a threshold is available by the end of that interval. This assumption is general enough whether the threshold is generated at the end of the time interval, during the interval or pre-determined.

According to Section 1, to achieve a high throughput, we need to use Sketch based solutions. As a technique based on random compression, Sketch based solutions cannot guarantee to identify all the heavy change keys. However the technique proposed in [2] has been proved to be able to generate a sufficiently good approximation of the true heavy change key sets. Therefore, we develop our solution based on the offline algorithm proposed in [2].

As discussed above, neither the total activity of each key nor the threshold are available before the end of the time interval, thus we can determine the heavy change keys only after we have processed the input stream in the interval. However, Sketch does not preserve the keys in the input stream, so we need to go through the input stream again to test each input key [2]. We use the "first/second pass" to refer to the first/second time we go through the input stream. Note that the activity has been recorded in the first pass, we only need to go through the keys in the second pass. So for each unique key, a copy needs to be made in the first pass and tested in the second pass. Recording and retrieving the input keys are not closely related to the main topic of this paper, so we assume that an existing system performs these operations. We discuss its memory space and bandwidth requirement in Section 5.4 to show that for a

typical performance requirement, state-of-the-art computing platform is able to support such a system.

As a summary, our goal is to design a high throughput architecture on FPGA for online heavy change detection based on the solution proposed in [2]. The architecture runs two passes over the input data stream. In the first pass, the architecture summarizes the data stream to record the total activity. In the second pass, the heavy change keys are identified according to the threshold available by the end of the first pass.

3 Related Work

[2] first applied sketch to heavy change detection in data streams. The authors proposed a data struture called K-ary Sketch and use it to summarize the time intervals in the data stream. Time series forecast models are used to forecast the "normal behavior" of the items in the current interval based on their observed behavior in the past time intervals. To identify the heavy changes, the absolute difference between the observed behavior and the forecast behavior is compared with the threshold based on the second moment of the forecast errors. Through thorough experiments, the authors show that by carefully choosing the parameters, their Sketch based solution detects the heavy change keys with very low false positive and false negative rates.

The K-ary Sketch used in [2] is irreversible, thus a second pass over the input keys is needed to identify the heavy change keys. To address this problem, [8] proposed a reversible sketch data structure so that the detected heavy change keys can be recovered from the sketch without the second pass. The main ideas are intelligently partitioning the input keys and designing the hash functions, so that the keys can be recovered through set intersections.

Heavy change detection has been accelerated on various computing platforms in the research community, such as CPU [10], GPU [10], NetFPGA [11] and FPGA [8]. None of these works achieves a throughput close to 100 Gbps. In Section 6, we make a detailed comparison of our architecture and the acceleration techniques proposed in these works.

4 Algorithm

Our online algorithm is based on the offline algorithm proposed in [2]. We chose this algorithm because it has been empirically proved to be very accurate for heavy change detection. Both algorithms are based on K-ary Sketches. Thus in the following subsections, we first introduce the k-ary Sketch, then the change detection based on the K-ary Sketch.

4.1 K-ary Sketch

K-ary Sketch uses a two dimensional array of counters to record the total activity of each item (identified by k) in a data stream. Let D and W denote the

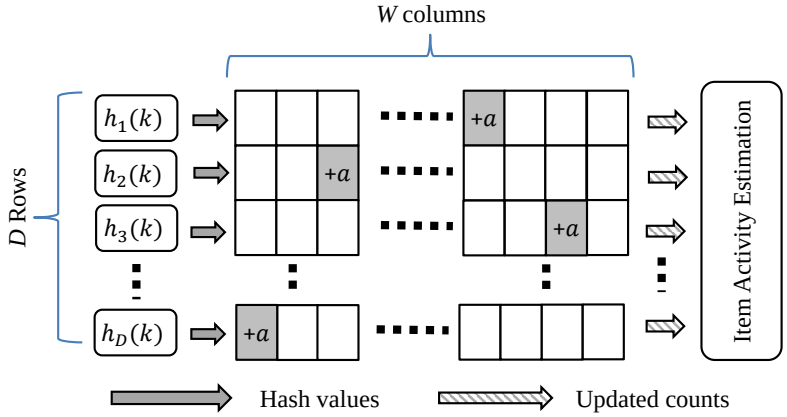

Fig. 1. K-ary Sketch

number of rows and columns in the two dimensional array. Let C denote the 2-dimensional array, then $C(d, w)$ is the counter value at the d-th row and w-th column of the 2-dimensional array. As shown in Figure 1, for any item $< k, a >$ in S,

$$\forall d \in [D], C(d, h_d(k)) = C(d, h_d(k)) + a \tag{1}$$

Let $Sum(S)$ denote the total activity recorded by the sketch. The estimation of the total activity associated with key k is:

$$median_{d \in [1,D]} \{ \frac{C(d, h_d(k)) - sum(S)/W}{1 - 1/W} \} \tag{2}$$

The estimation can be performed upon query to any item at any time during the data stream.

4.2 Time Series Forecast

As defined in Section 2, change detection is based on the absolute difference between the observed activity and the "normal" activity. In [2], the "normal" activity is generated using time series forecast models based on the observed activities in the past intervals. The authors explored the following commonly used models: Moving Average Model, S-Shaped Moving Average Model, Exponentially Weighted Moving Average Model, Non-seasonal Holt-Winters Model, and Auto Regressive Integrated Moving Average Model (ARIMA). (Due to the space limitation, we do not include the details of these models here.) These models can all be be abstracted as the linear combination of the previous activities. Thus, using the same notations as in Definition 1, the forecast "normal" activity of k in interval I_j can be represented as:

$$F_k^{I_j} = \sum_{t=j-N}^{j-1} c_t O_k^{I_t} \tag{3}$$

where c_t is the weight for the corresponding interval. For different forecasting models, c_ts are different. These coefficients can be determined by off-line training and then programmed into the architecture. Therefore a general architecture can be designed to support all these models. The general architecture is introduced in Section 5. In practice we only need the severl most recent past intervals to do the forecast. For example the ARIMA model normally needs only the last 2 intervals [2]. In Equation 3, N denotes the number of necessary past intervals.

4.3 Online Sketch Based Change Detection

Given Equation 3, the absolute difference between the observed and the forecast activity can be represented as:

$$|O_k^{I_j} - F_k^{I_j}| = |O_k^{I_j} - \sum_{t=j-N}^{j-1} c_t O_k^{I_t}| = |\sum_{t=j-N}^{j} c_t O_k^{I_t}| \tag{4}$$

In the Sketch based algorithm proposed in [2], all the $O_k^{I_t}$s are recorded using K-ary Sketches and the linear combination is performed directly at the sketch level by combining every entry. Therefore using the notation of K-ary Sketch, Equation 4 can be represented as:

$$C_e^{I_j}(d, w) = \sum_{t=j-N}^{j} c_t C^{I_t}(d, w) \tag{5}$$

(C^{I_t} denotes the two dimensional array in the sketch for interval I_t). The result of the combination is stored in another K-ary Sketch referred to as "error sketch". This sketch estimates the absolute forecast error according to Equation 2.

The offline algorithm proposed in [2] is shown in Algorithm 1. As mentioned in Section 2, we need 2 passes over S to complete the change detection. More specifically as shown in Algorithm 1, in the first pass, the activity in the current interval is recorded using Equation 1 and the error sketch is constructed using Equation 4. In the second pass, the forecast errors for all the keys are estimated based on the error sketch using Equation 2 and then compared with the threshold.

Figure 2 illustrates our modified version of Algorithm 1 to realize online change detection. S^{I_j} denotes the sub-stream of S in interval I_j. As shown in the figure, we are maintaining a sliding window of $N + 1$ intervals, 1 current interval for observation and N most recent past intervals for forecasting. Every time we start processing a new interval, the window slides 1 interval forward and the least recent past interval is discarded. The major modifications are:

– Performing the first pass for the current interval and the second pass for the last interval at the same time. This way, after completing the first pass for any interval, we can immediately start the first pass of the next interval rather than holding the input stream for the second pass of the current interval.

Algorithm 1 Offline Change Detection [2]

Fisrt Pass:
for Each item $< k, a >$ in I_j **do**
  for all $d \in [1, D]$ **do**
    $C^{I_j}(d, h_d(k)) = C^{I_j}(d, h_d(k)) + a$
  end for
end for
for all $d \in [1, D]$ **do**
  for all $w \in [1, W]$ **do**
    $C_e^{I_j}(d, w) = \sum_{t=j-N}^{j} c_t C^{I_t}(d, w)$
  end for
end for
End Fisrt Pass

Second Pass:
for All the keys k in S **do**
  if $|median_{d\in[1,D]}\{\frac{C_e^{I_j}(d,h_d(k))-sum(S)/W}{1-1/W}\}| \geq \Phi$ **then**
    Report k as heavy change key
  end if
end for
End Second Pass

– Updating the error sketch incrementally as the activity is recorded. As shown in Figure 2, every time the activity of a key is recorded, its entry in the error sketch is updated. Note that by doing this we do not update all the entries in the error sketch as in Algorithm 1, but only the entries of the keys that appear in the interval. Doing this is both safe and necessary for online processing. It is safe because, as defined in Definition 1, we are only interested to the keys that appear in the interval. It is necessary because if we construct the error sketch after completing the activity recording, we need to hold the input stream until the error sketch is constructed. This is against our goal for online processing. If we don't hold the input stream after recording the activity, the window slides forward and the least recent past interval is discarded. However, this interval is necessary to construct the error sketch. We can assign additional storage to keep this interval, but higher memory consumption is always not desired.

With these modifications, the algorithm shown in Figure 2 is able to perform the online change detection as defined in Section 2. In Section 5 we introduce our architecture to accelerated it.

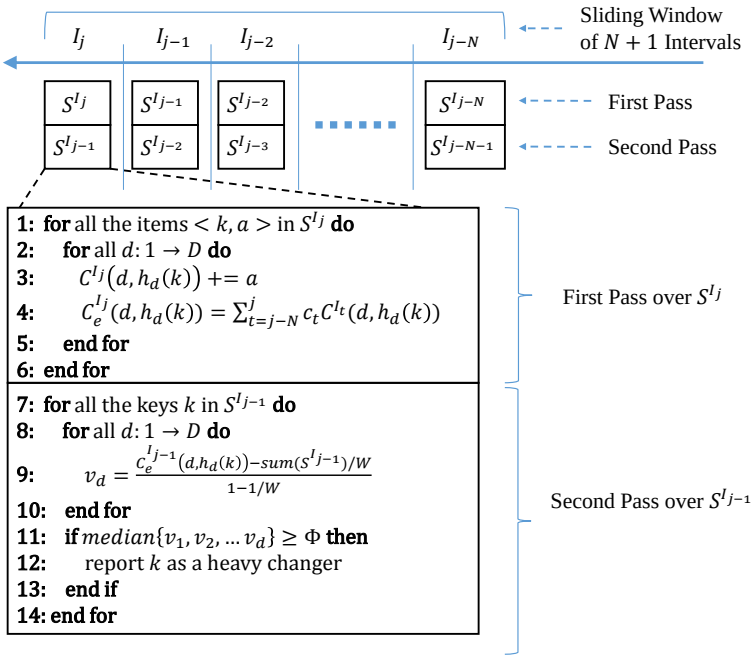

Fig. 2. Online Change Detection

5 Architecture

5.1 Overall Architecture

The overall architecture based on Figure 2 is shown in Figure 3. In our online algorithm both first and second pass iterate through the rows of various sketches. Since each iteration step only involves the same rows of the involved sketches, we map each iteration step to one super pipeline stage.

At any super stage, the the input item in the first pass goes through hashing and then the module for sketch update and linear combination. These steps perform Line 3 and generate the RHS of Line 4 in Figure 2. The input key in the second pass is hashed, then goes to the error sketch to help estimating the forecast error. Since the two passes are processed at the same time, we need to maintain two error sketch modules. Let $ES0$ and $ES1$ denote the two error sketch modules. In any interval I_t, if $ES0$ constructs the error sketch for the first pass over S^{I_t}, then $ES1$ is used to estimate the forecast error for the second pass over $S^{I_{t-1}}$. Later in I_{t+1}, the constructed error sketch in $ES0$ is used to estimate the forecast error for S^{I_t}, and $ES1$ constructs the error sketch for $S^{I_{t+1}}$. A switch is implemented to route the two passes to its destination error sketch. The switch and the two error sketches together perform the assignment of Line 4 and Line 9 in Figure 2. Line 11 is distributed to the super stages of the pipeline. The basic idea is that: at any super stage, a sorted array of the vs from the

earlier stages is taken as the input, the v generated at this stage is inserted into the input array at the appropriate position to keep the resulting array sorted. This resulting array is forwarded to the next super stage. The output array from the last super stage is a sorted array of the vs from all the rows. The pivot of this array is the median value to compare with the threshold. The mux in Figure 3 selects the output for the second pass to insert to the sorted array. In the following subsections, we introduce the detail of the modules in Figure 3.

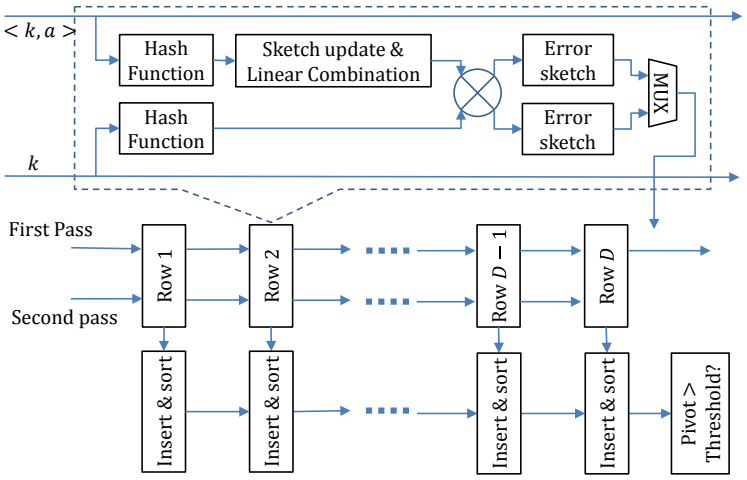

Fig. 3. Overall Architecture

5.2 Hashing

We choose the hash functions from a hash function family called H_3 [12]. The hash function delivers highly even distribution of hash keys to hash values. It is defined as follows [12]:

Definition 2 *Let A = the key space, the set of all possible bit-vectors of M bits, B = the hash value space, the set of all possible bit-vectors of H bits, Q = set of all possible $M \times H$ boolean matrices. For a given $q \in Q$ and $x \in A$, let $q(m)$ = the m-th row of q, and let $x(m)$ = m-th bit of x. The hash function $h_q(x) : A \rightarrow B$ is defined as:*

$$h_q(x) = x(1) \cdot q(1) \oplus x(2) \cdot q(2) \oplus ... \oplus x(M) \cdot q(M)$$

$\cdot$ denotes the bitwise AND operation. $\oplus$ denotes the bitwise XOR operation. The set $\{h_q | q \in Q\}$ is called H_3 class.

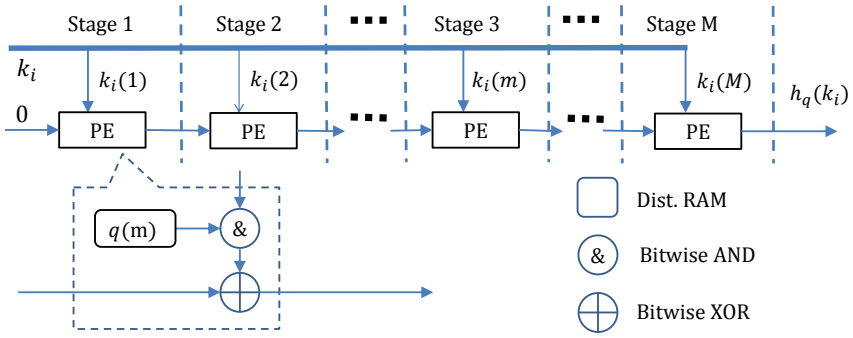

Fig. 4. Pipelined hash computation

To implement D hash functions for D rows, D matrices are chosen from Q. Different q's lead to different performance with respect to evenly distributing the hash keys among hash values.(Selection of q's is not the focus of this paper.)

Given any $q \in Q$, our proposed architecture for h_q is illustrated in Figure 4. As defined in Definition 2, the H_3 hash computation needs M sets of bitwise AND and bitwise XOR operations. We map each set of operations to one pipeline stage. At any stage $m \in [1, M]$, $q(m)$ is stored in distributed RAM. A bitwise AND operation is first performed on the m-th bit of the input k and $q(m)$. Then, the result of the bitwise AND operation is used to XOR with the partial hash value forwarded from the previous stage.

5.3 Sketch Update & Linear Combination

The architecture for sketch update and linear combination is shown in Figure 5. As discussed in Section 4.3, we need to maintain a sliding window of $N + 1$ intervals. We map each interval to a super stage with an interval index. This index is the number of intervals from the head of the sliding window (i.e. the current interval) to the interval that corresponds to this super stage. When the processing for one interval completes, the window slides forward to start the processing for the next interval and the least recent past interval is discarded. We use the super stage for the discarded interval to record the activity of the next interval and preserve it until it is discarded. Figure 6 illustrates an example for a sliding window of 4 intervals. Every time the window slides, the interval indices of all super stages increment by 1. The stage with index 0 is the head of the window and the stage with interval index 3 is to be discarded. In the following subsections, we first discuss the linear combination of sketches assuming that the super stages always preserve the correct data as illustrated in Figure 6. Then we show how to handle the sketch update and the sliding of the window to makes the assumption valid.

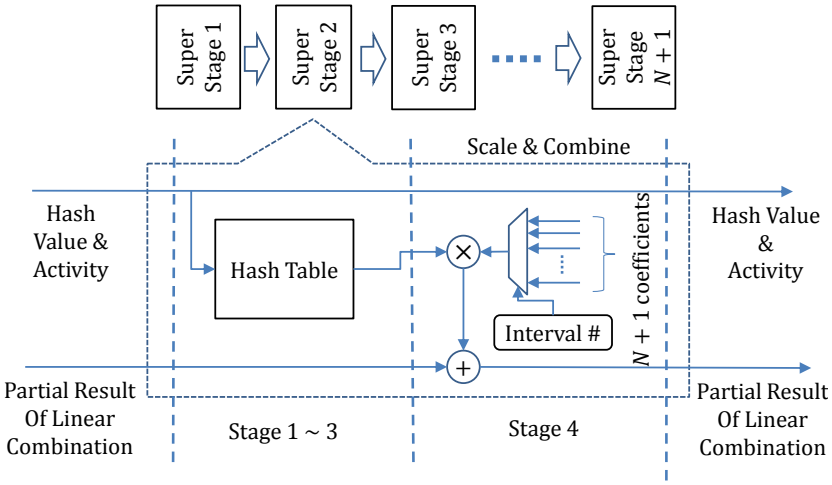

Fig. 5. Sketch Update and Linear Combination

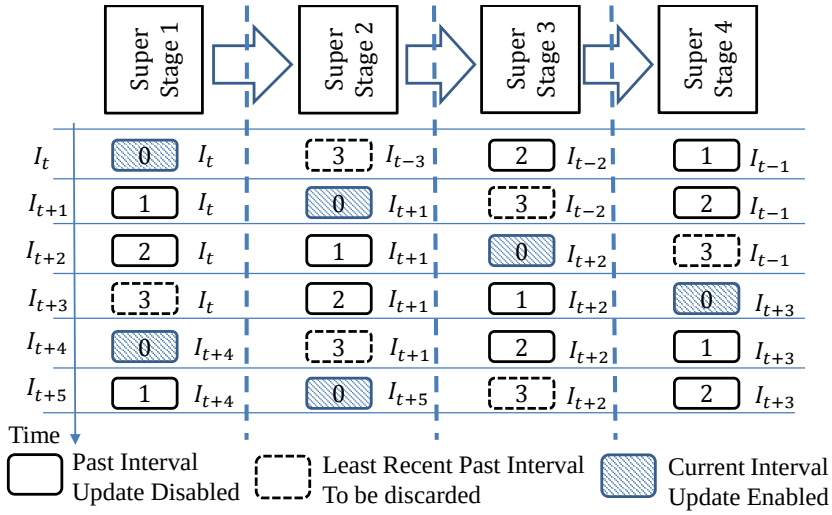

Fig. 6. Mapping Time Intervals to Pipeline Super Stages for Sliding Window

Linear Combination of Sketches As shown in Figure 5 each super stage has 4 actual pipeline stages. The first 3 stages maintain the hash table containing the recorded activity of the corresponding interval. The 4th stage stores the coefficients (c_is in Figure 2) for all the intervals and select the one corresponding to the interval index to scale the activity value from the hash table. The scaled value is added to the partial combination result from the last super stage. Then the updated partial combination result is forwarded to the next super stage. This way, the output of super stage $N + 1$ is RHS of Line 4 in Figure 2.

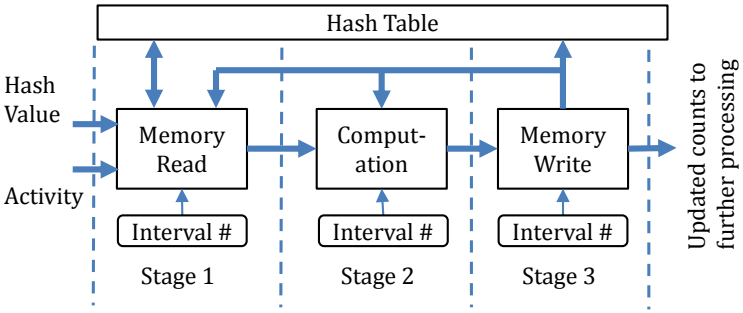

Fig. 7. Hash Table

Hash Table Maintenance The architecture of the hash table is shown in Figure 7. We implement the hash tables using the on-chip dual-port BRAM, one port for memory read, one port for memory write. When the super stage corresponds to the current interval, for each input item, we need to retrieve the corresponding hash table entry, update it and write it back to the hash table to execute Line 3 of Figure 2. We map these operations to a 3 stage pipeline. In Stage 1, the activity counts are retrieved from the BRAM using the hash value. In Stage 2, the counts are updated. In Stage 3, the updated counts from Stage 2 are written back into the memory. The pipelined architecture takes 3 clock cycles to perform 1 hash table update. Since each clock cycle we are taking in one hash value, it is possible that the up-to-date count for the input hash value is still being processed in the pipeline and the activity count retrieved from BRAM is out-dated. To solve this problem, we forward the updated counts at the memory write stage to the memory read stage and the computation stage. If the hash value in memory read stage or computation stage matches the hash value in the memory write stage, then the up-to-date activity count is being written back to the memory. When such situation occurs, the forwarded counts from the memory write stage is used for further processing rather than the outdated activity count from the BRAM. When a super stage corresponds to a past interval, the memory write and the data forwarding are both disabled, because there are no need to update the activity record for a past interval.

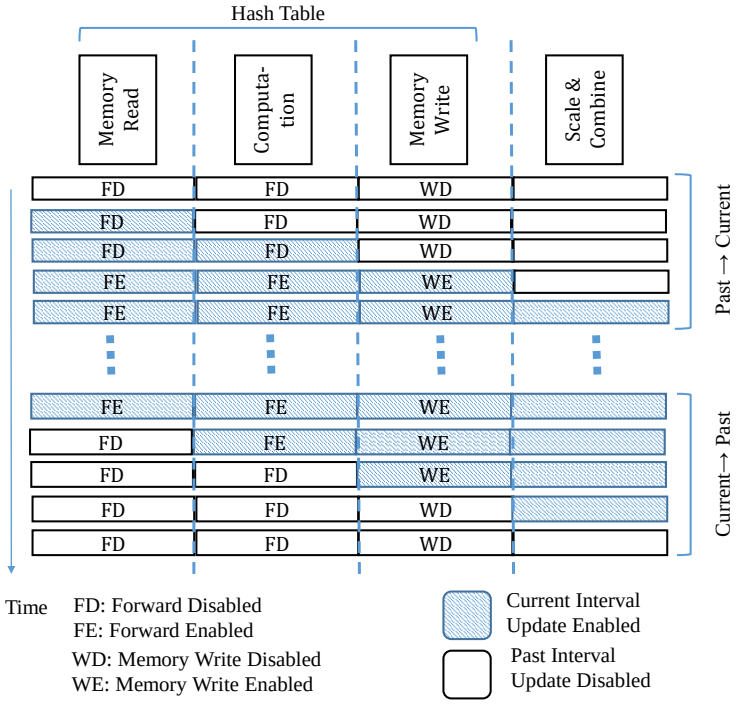

Fig. 8. Controls During the Sliding of Windows

Handling the Sliding Window Since each super stage has 4 actual pipeline stages, the sliding of the window is not instant: it takes 4 clock cycles for the new interval from entering the super stage to completely occupying the super stage. During this 4 clock cycles, the new interval and the last interval co-exist in the super stage, we need to control the data forwarding and the memory operations to make sure the two neighboring intervals do not interfere with each other's processing.

Figure 8 shows the control policy during the transition between the past and the current intervals. When the super stage is fully occupied by the past interval, data forwarding and memory write are both disabled. For the first item in the current interval, the memory write is enabled, but the data forwarding is kept disabled because the forwarded data belongs to the last interval. For the second item in the current interval, the data forwarding at memory read stage is kept disabled to avoid mistakenly receiving the forwarded data from the past interval. The data forwarding at computation stage is enabled in case that the second item updates the same memory location as the first item. For the third and later items in the current interval, both memory write and the data forwarding are enabled. In the last stage of the super stage, the appropriate coefficient is selected according to the interval index at that stage. The interval indices is set by the first item in the interval. During the transition from the

current to the past interval, once the last item of the current interval leaves the corresponding stage, the data forwarding and the memory write is disabled, and the interval indices are incremented. During the transition between two past intervals, all the controls stay the same.

One last issue is that during the current interval when any hash table entry is accessed for the first time, it contains the activity record from the last interval. We need to use zero as the activity count to update it instead of its original count. To decide if this entry has ever been updated in the current interval, we associate each hash table entry with the ID of the interval in which it is last updated. This ID is not the interval index, it is a unique ID among all the intervals. When we update the hash table entry, if the recorded ID is different from the current interval ID, then this entry has never been updated in the current interval.

5.4 Heavy Change Determination

As discussed in Section 5.1, the result of the linear combination is recorded by the error sketch in the first pass and then used to estimate the forecast error in the second pass. The median of the outputs from the error sketches is found by sorting and picking the pivot. In this section we discuss the details about the error sketch, the sorting module, and the solution to make the second pass of the input stream available.

Error Sketches The update to the error sketch is independent of the preserved value in the error sketch. When any entry needs to be updated the new value simply overwrites the old value. Therefore its architecture only has one memory stage to perform both read and write, and a stage to perform Line 9 of Figure 2.The memory write is enabled during the first pass and disabled during the second pass.

Insert and Sort Figure 9 shows the architecture of the insert and sort module. It uses the module for Row 4 as an example. The input is a sorted array of 3 items and one new item to insert. The output is a sorted array of 4 items. Because the new value is inserted into a sorted array, at any position of the output array, there are only 3 possible value: 1. the value at the same position in the input array, 2. the new value, 3. the value at the neighboring position towards the start of the array. The rules to choose the values for the positions in the output array are shown in Algorithm 2 (suppose we sort the values in ascending order):

Second Pass over the Input Data Stream As discussed in Section 2, we need to store the keys during the first pass and retrieve them during the second pass. Since the first pass for the current interval and second pass for the last interval need to run at the same time, we need to have two storage modules running in

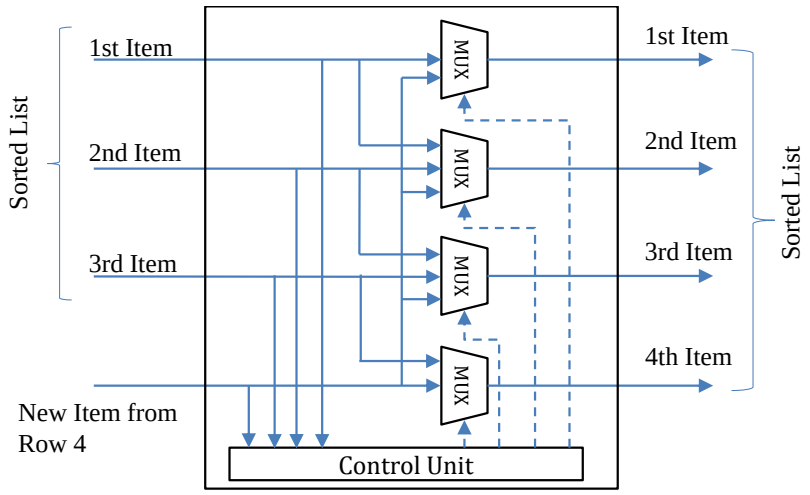

Fig. 9. Insert & Sort Module

Algorithm 2 Insert and Sort Algorithm for Row R

Input: Sorted Array of $R-1$ items $I[R-1]$
Input: New item to insert v
Output: Sorted Array of R items $O[R]$
Given a position p
if $p = 0$ **then** ▷ The start of the output array
  if $v \geq I[p]$ **then**
    $O[p] = I[p]$ ▷ New item in $[1, R-1]$
  else
    $O[p] = v$ ▷ New item at 0
  end if
else if $p = R-1$ **then** ▷ The end of the output array
  if $v \geq I[p]$ **then**
    $O[p] = v$ ▷ New item at $R-1$
  else
    $O[p] = I[p-1]$ ▷ New item in $[0, R-2]$
  end if
else ▷ Other positions of the output array
  if $v < I[p-1]$ **then**
    $O[p] = I[p-1]$ ▷ New item in $[0, p-1]$
  else if $v < I[p]$ **then** ▷ $v \geq I[p-1]$
    $O[p] = v$ ▷ New item at p
  else
    $O[p] = I[p]$ ▷ New item in $[p+1, R-1]$
  end if
end if

parallel taking turns to perform the storing and retrieving. As shown in Section 6, the highest clock rate our architecture is 327 MHz. At such a frequency, suppose the key is 128 bits, the required memory bandwidth is 80 Gbps in total for both storing and retrieving. Such a bandwidth can be supported by state-of-the-art FPGA platform. In [13], the FPGA platform provides 24 GB memory with 115 Gbps (384 bits $\times$ 300 MHz) memory bandwidth. 24 GB memory is able to support 2 modules each storing 750 M unique 128-bit keys, which is more than enough for a typical problem size of several million concurrent flows [4].

6 Evaluation

6.1 Experimental Setup

We implemented our heavy change detector on a Xilinx Virtex-7 xc7vx690tffg1930-3 FPGA. All reported results are post place-and-route results using Xilinx Vivado 2014.3. The implementation is RTL implementation, no high level synthesis tool is used. Throughput is our target performance metric. We use Gigabit per second (Gbps) as the unit for throughput. When we calculate the throughput based on the clock rate, we assume that every clock cycle the architecture can take in one packet and all packets have minimum packet size of 40 Bytes. Thus, the reported throughput is the minimum throughput. We also report the FPGA resource usage. The logic resources on Xilinx FPGA devices are organized using slices containing LUTs and registers. All the hash tables are maintained in BRAM. Thus the usage of slice LUT, slice register, and BRAM reflect the resource consumption of the implemented design. We report the usage in percentage. The device we use has 433 K slice LUTs, 866 K slice registers and 1470 36-Kb BRAM blocks.

Our architecture can be configured using the following 5 parameters:

- **The width of key k** determines the number of unique keys the detector can support. Using K-bit keys, the architecture can support 2^K unique keys.
- **The width W and depth D** of the two dimensional counter array determine the accuracy of the algorithm, i.e. false positive rate and false negative rate.
- **The length of the interval C** determines the width of the counters in the sketch entries. The worst case is that all the items in the stream goes to the same entry. Therefore, each counter needs to be able to support the entire stream in the interval.
- **The width of the window $N + 1$** determines the number of 2-dimensional arrays the architecture needs to maintain.

In the following subsections we first report the performance using a typical configuration. Then we vary these parameters to test the scalability of our architecture.

For the typical configuration, we choose 16, 32, 64, 128 bits for the key width. These numbers cover most of the key types in network application (IPv4, IPv6,

Port numbers, 5-tuple information[1], etc). For W and D, it is empirically proved in [2] that using $D = 5$ and $W = 32K$, K-ary Sketch excellently guarantees low false negative rate and false positive rate. Total stream size is proportional to the interval length. As mentioned in [2], 5 minutes is a typical time interval length for change detection because it is a reasonable trade-off point between responsiveness and computational overhead. However since our proposed solution is an pipelined online hardware solution, shorter interval length does not lead to additional computational overhead. Therefore we can use shorter intervals for a faster response rate. So we use 1 minute, another interval length used in [2]'s experiments as a typical interval length. The throughput of our design is around 100 Gbps. For such an interval length on a 100 Gbps link, we need to support a stream of size 6 Gb (0.75 GB). The corresponding counter width is 40 bits (supporting up to 1 TB). The window width depends on the forecast model. ARIMA model is the most accurate model in [2], it needs no more than 2 past intervals. Therefore we use $N + 1 = 3$ for the typical configuration.

Table 1. Throughput comparison of various sketch acceleration techniques

Platform	Device	Key Width (bit)	Window Size (# of intervals)	Interval Length (sec.)	D	W	Throughput (Gbps)
NetFPGA [11]	—	32	4	60	3	2^{15}	4
GPU [10]	1 Radeon HD 5870	16	3	60	5	2^{16}	2.875
CPU [10]	1 AMD Athlon 64 X2 at 2 GHz	16	3	60	5	2^{16}	0.4
FPGA [8]	3 Xilinx Virtex 2000E	—	—	—	5	2^{12}	22
Our Architecture	1 Xilinx Virtex 7 XC7VX690	32	4	60	5	2^{15}	106

6.2 Performance Using Typical Configuration

The throughput and resource usage using the typical configuration is shown in Figure 10. For all the key width, our architecture achieves 100 Gbps through-

[1] the 5-tuple information refers to source/destination IP address, source/destination port number and transport layer protocol

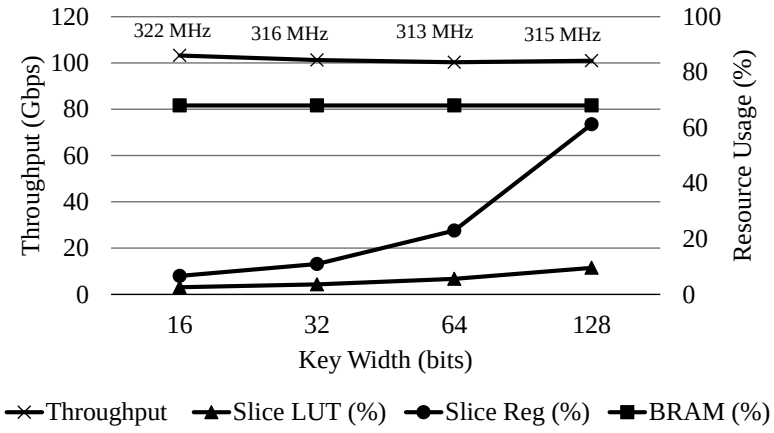

Fig. 10. Performance Using Typical Configuration

put. This demonstrates that our architecture can support almost all types of commonly used keys at high throughput.

The usage of the slice registers grows fast with the increment of key width. This is because the key width is the same as the number of pipeline stages for the hash computation. Carrying the input key and the partial hash value through the pipeline consumes a large amount of registers. Through comparing the resource usage in Figure 10, 11 and 12 we can notice that the key width has the biggest affect on the slice register usage. The BRAM usage stays the same for all 4 key width because the key width doesn't affect the size of the 2-dimensional array or the number of arrays. The usage of slice LUT also varies little when the key width increases because the logic and arithmetic operations in the hash computing is simple and consume little slice resource.

6.3 Scalability

Window Size Figure 11 shows the throughput and the resource usage for various window sizes. For the window size of 2 and 3 intervals, the throughput achieves 100 Gbps. As the window size increases, the throughput drops slightly. For each interval in the window, we need to implement one 2-dimensional array, thus the BRAM usage grows linearly as the window size increases. The slice LUT and slice register usage stays almost the same for all the window sizes. As discussed in Section 4.2, our architecture is a general architecture for various time series forecast models. Thus Figure 11 demonstrates that our architecture achieves high throughput for various time series forecast models based on various window size.

Interval Length Figure 12 shows the throughput and the resource usage for various interval length. For all the interval length, our architecture can achieve

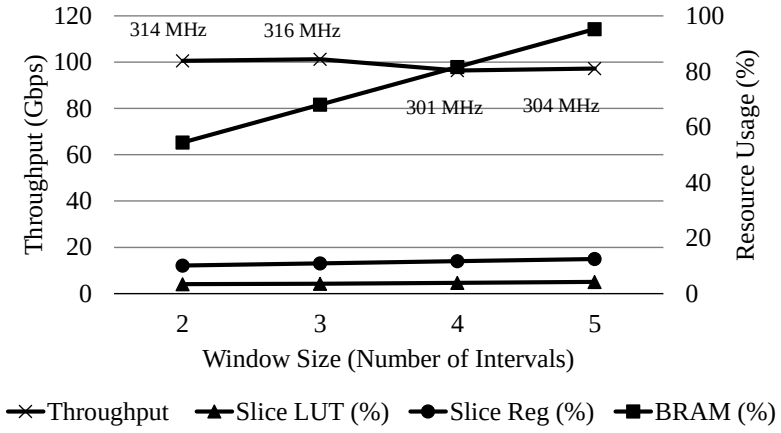

Fig. 11. Performance for Various Window Size

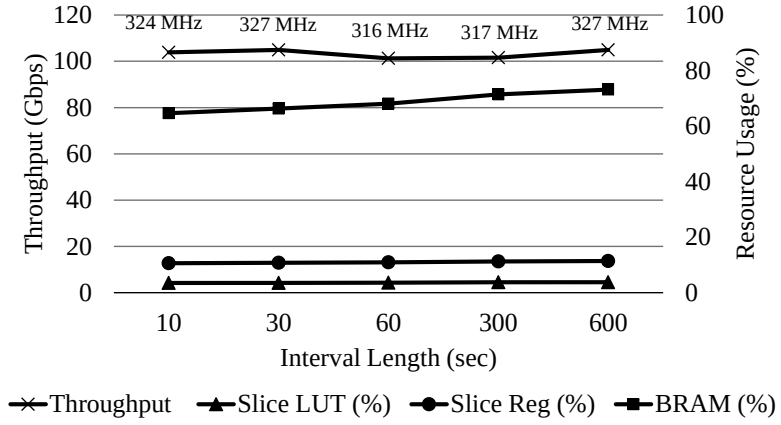

Fig. 12. Performance for Various Interval Length

100+ Gbps. Shorter interval length leads to faster response rate for change detection. Figure 12 shows that our architecture can be used for various application scenarios where various response rates are needed.

6.4 Comparison with Other Sketch Acceleration Techniques

In Table 1, we compare the throughput of our architecture with other architectures for heavy change detection on various platforms [11], [10], [8]. For the publications that used different assumption when computing their throughput, we re-compute the throughput based on the reported results in the original publication assuming 40-bytes minimum packet size. A blank cell means the information is not available in the paper. As shown in Table 1, our architecture achieves a much higher throughput than all other techniques. The throughput improvement is 5× - 250×.

7 Conclusion

In this work, we proposed a solution for online heavy change detection and accelerated it using a deeply pipelined architecture. The architecture can consistently achieve around 100 Gbps throughput using various configurations. Our results demonstrate that through hardware acceleration, online heavy change detection can be performed at a very high throughput for various application scenarios using various key widths, various time series forecast models, and with various response rates. As future research, we will study the application of our solution with the help of High Level Synthesis Tools on FPGA and on other high performance computing platforms such as multicore platforms or heterogeneous platforms.

References

1. Y. Gao, Z. Li, and Y. Chen, "A dos resilient flow-level intrusion detection approach for high-speed networks," in *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, 2006, pp. 39–39.
2. B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: Methods, evaluation, and applications," in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '03, 2003, pp. 234–247.
3. M. Attig and G. Brebner, "400 gb/s programmable packet parsing on a single FPGA," in *Seventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2011.
4. C. Estan and G. Varghese, "New directions in traffic measurement and accounting," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 323–336, Aug. 2002.
5. 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.

6. 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. ACM, 2013, pp. 255–264.
7. A. Bitar, J. Cassidy, N. Enright Jerger, and V. Betz, "Efficient and programmable ethernet switching with a noc-enhanced fpga," in *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '14. ACM, 2014, pp. 89–100.
8. R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P. A. Dinda, M.-Y. Kao, and G. Memik, "Reversible sketches: Enabling monitoring and analysis over high-speed data streams," *IEEE/ACM Trans. Netw.*, vol. 15, no. 5, pp. 1059–1072, Oct. 2007.
9. G. Cormode and S. Muthukrishnan, "What's new: finding significant differences in network data streams," *Networking, IEEE/ACM Transactions on*, vol. 13, no. 6, pp. 1219–1232, Dec 2005.
10. T. Wellem, Y.-K. Lai, C.-C. Lee, and K.-S. Yang, "Accelerating sketch-based computations with gpu: A case study for network traffic change detection," in *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, Oct 2011, pp. 81–82.
11. Y.-K. Lai, N.-C. Wang, T.-Y. Chou, C.-C. Lee, T. Wellem, H. T. Nugroho *et al.*, "Implementing on-line sketch-based change detection on a netfpga platform," in *1st Asia NetFPGA Developers Workshop*, 2010.
12. M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *Computers, IEEE Transactions on*, vol. 46, no. 12, pp. 1378–1381, Dec 1997.
13. Z. Istvan, G. Alonso, M. Blott, and K. Vissers, "A flexible hash table design for 10gbps key-value stores on fpgas," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013, pp. 1–8.