An FPGA-Based In-NIC Cache Approach for Lazy Learning Outlier Filtering

Ami Hayashi
Dept. of ICS, Keio University,
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Japan
Email: hayashi@arc.ics.keio.ac.jp

Hiroki Matsutani
Dept. of ICS, Keio University,
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Japan
Email: matutani@arc.ics.keio.ac.jp

***Abstract*—As data sets grow rapidly in size and the number, an outlier detection that filters unnecessary normal information becomes important. In this paper, we propose to move the outlier detection from an application layer to a NIC (Network Interface Card). Only anomalous items or events are delivered for a network protocol stack and the other packets are discarded at the NIC. The demands for storage and computation costs at a host are thus drastically reduced. We employ lazy learning algorithms for the outlier detection, because they can be applied to complex reference data including different clusters. However, it is challenging to offload the lazy learning to NIC hardware because of high computational cost and huge reference data. In this paper, we propose to cache only a frequently-accessed portion of reference data in the NIC. This idea can be applied to lazy learning algorithms in general. LOF (Local Outlier Factor) and KNN (K-Nearest Neighbor) are thus implemented on an FPGA (Field Programmable Gate Arrays) based NIC. Simulation results of the proposed system using LOF with 100,000 reference data show that 45% to 90% of queries are hit to the proposed cache and filtered at the NIC. The results are corresponding to 1.82x to 10x throughput improvements on the outlier filtering compared to that of a software-based execution.**

I. Introduction

Data sets grow rapidly in size with the advances in information and communication technologies and mobile and sensing devices. Network bandwidth has continued to grow and it becomes possible to collect enormous data in a high rate (e.g., ten Gbps). For this reason, it is a major issue to achieve a high performance on-the-fly processing of continuously generated data, called stream processing.

Online outlier detection is one of the most popular stream processing. Only anomaly data are extracted online from the data received continuously with an outlier detection algorithm. This becomes more important as sensor data are increasing in size due to the advance of IoT technology. Warning and anomaly detection systems that handle enormous sensor data often rely on online outlier detection. If we drop the anomaly data, we may overlook emergency situations such as service failure, though anomaly data may appear at a very low frequency; thus a high precision is required for the outlier detection. Outlier detection using lazy learning is a non-parametric algorithm which can be applied to various reference data. However, it is a critical issue that its high computational cost would become a bottleneck of stream processing. In this paper, we aim for high efficiency of the online outlier detection using lazy learning.

In this paper, we offload the outlier detection to the NIC (Network Interface Card) hardware. Figure 1 illustrates an overview of our system. Sample data (e.g., temperature data such as 20.0) generated from the clients, such as sensor nodes, to the server through a network are examined by an outlier detection algorithm at the server side NIC. Only anomaly data are delivered to a network protocol stack and the other packets (most data) are discarded at the NIC. This method reduces not only host workload by outlier detection but also host workload by processing of network protocol stack. However, it is not trivial to offload the outlier detection using lazy learning to the NIC because of complex computation and huge reference data required for the lazy learning. In particular, there is no existing work that offloads LOF (Local Outlier Factor) to the NIC hardware to the best of our knowledge. In this paper, we propose to cache only a frequently-accessed portion of reference data in the NIC. This idea is essential to offload LOF to FPGA (Field Programmable Gate Array) based NIC (FPGA NIC) for online outlier filtering and also applicable to the other lazy learning algorithms.

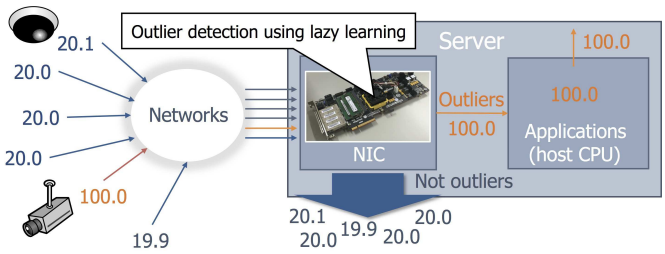

Fig. 1. Overview of outlier filtering NIC

The rest of this paper is organized as follows. Section II introduces outlier detection based on lazy learning algorithms and the FPGA-based accelerators. Section III illustrates our outlier filtering NIC using lazy learning algorithms and Section IV evaluates it in terms of resource utilization and performance. Section V summarizes this paper.

II. Background and Related Work

A. Outlier Detection Based on Lazy Learning

We employ lazy learning algorithms for an outlier detection. These algorithms process queries (new sample data) mostly without preprocessing to reference data, contrary to that based on eager leaning which processes queries after generalization of reference data. Lazy learning has the advantage that can be applied to complex reference data since it is possible to make a lot of local approximations. We adopt two lazy learning algorithms, LOF and KNN (K-Nearest Neighbor), in this paper.

B. Outlier Detection Using Local Outlier Factor

LOF, proposed by Breunig et.al., is a non-parametric outlier detection algorithm based on density [1]. Because LOF employs the density as an index, it can detect outliers even if there are

datasets with different distribution models. The algorithm is described below briefly.

Here we want to detect whether a query (new sample data) p is outlier by comparing it with reference data D. First, k-distance of p (denoted as $k\_distance(p)$) is calculated from distance between p and data $o \in D$ (denoted as $d(p,o)$). $o \in D$ is selected so that the following two conditions are satisfied, assuming k is a positive integer parameter given by the user.

(Condition 1) k or more data ($o' \in D$) satisfy the following relationship.

$$d(p,o') \leq d(p,o) \tag{1}$$

(Condition 2) $k-1$ or less data ($o' \in D$) satisfy the following relationship.

$$d(p,o') < d(p,o) \tag{2}$$

Then, k-distance neighborhood of p (denoted as $N_{k\_distance(p)}(p)$) is defined by the following equation.

$$N_{k\_distance(p)}(p) = \{q \in D | d(p,q) \leq k\_distance(p)\} \tag{3}$$

In other words, $N_{k\_distance(p)}(p)$ is a set of neighbor data around p, which includes at least k data.

Reachability distance of p from $o \in D$ (denoted as $reach\_dist_k(p,o)$) is defined by the following equation.

$$reach\_dist_k(p,o) = \max\{k\_distance(o), d(p,o)\} \tag{4}$$

Reachability distance of p from $o \in D$ is the distance between p and $o \in D$, but the distance is at least $k\_distance(o)$ (calculated from o and neighborhoods of o).

Local reachability density of p (denoted as $lrd_k(p)$) is defined by the following equation.

$$lrd_k(p) = \frac{|N_{k\_distance(p)}(p)|}{\Sigma_{o \in N_{k\_distance(p)}(p)} reach\_dist_k(p,o)} \tag{5}$$

LOF of p is calculated from local reachability densities $lrd_k(p)$ and $lrd_k(o)$, both of them are calculated by Equation (5). LOF of p (denoted as $LOF\_k(p)$) is defined by the following equation.

$$LOF_k(p) = \frac{\Sigma_{o \in N_{k\_distance(p)}(p)} \frac{lrd_k(o)}{lrd_k(p)}}{|N_{k\_distance(p)}(p)|} \tag{6}$$

The query p is detected as an outlier when $LOF\_k(p)$ is far from 1 because we can estimate that the density of p is too sparse compared to that of its neighborhoods. In other words, the proposed system detects outliers by calculating LOF of input queries p (sample data sent from clients to the server) to the reference data D accumulated in the server.

C. Outlier Detection Based on k Nearest Neighbor

KNN is a classic and practical lazy learning algorithm [2]. An outlier is detected based on whether the distance between the query and its k-th closest reference data is large or not. In other words, it detects outliers based on k-distance. It computes distances between the query and all the reference data and sorts them in the closest order. As the k-th closest distance becomes large, the possibility of outlier becomes high. Compared to LOF, although the computation is simple, it would not be suitable to deal with reference data including various clusters whose variance is different.

D. FPGA-Based Accelerators

FPGA-based acceleration for machine learning algorithms is an emerging research topic. There are prior works that offload various algorithms on FPGAs. For example, Random forest is one of the most popular machine learning algorithms [2]. Essen et.al. offloaded Random forest to FPGAs in order to improve the performance [3]. They employed CRF (Compact Random Forest) that consists of trees whose depth is limited in order to efficiently implement it on FPGAs.

In [4], the authors offloaded an outlier detection using Mahalanobis distance on an FPGA NIC. The outlier detection algorithm used in [4] is too naive. More practical outlier detection algorithms that require huge reference data, such as LOF, cannot be applied to [4]. Various applications can take advantages of FPGA-based outlier detection. For example, Das et.al. implemented a FEM (Feature Extraction Module) and an outlier detection with PCA (Principal Component Analysis) on FPGAs to accelerate NIDS (Network Intrusion Detection System) application [5].

When we offload LOF and KNN to FPGA NIC, a huge memory resource to store all reference data is required for the FPGA NIC. In addition, a huge sorter whose length is equivalent to the total number of reference data is required to find the neighborhoods of the query. Due to these difficulties, there is no prior work that offloads LOF algorithm to FPGAs to the best of our knowledge, although there are accelerators with other devices such as GPUs [6]. On the other hand, there is prior work that accelerates KNN algorithm by offloading it onto FPGAs with limited sizes of reference data (e.g., 10,000 $\sim$ 20,000). They are introduced as follows.

Pu et.al. proposed an implementation of FPGA based heterogeneous computing architecture suitable for KNN with OpenCL [7]. An efficient implementation by parallelizing the distance computation and bubble sort achieved a high performance. EER (Energy Efficiency Ratio) of their implementation on an FPGA is 804x higher compared to that with a CPU and 3x higher compared to that with a GPU. Manolakos et.al. proposed flexible IP cores for FPGA implementation of KNN classifier [8]. They also proposed an efficient implementation of the distance computation and sort. In [8], users can select a suitable design from two proposed designs in according to the magnitude of parameters and numbers of reference data and features. Their proposed system processes ten thousands of queries per seconds. In addition to the above works, various FPGA implementations of KNN have been proposed such as [9] and [10].

As described above, lazy learning algorithms have been accelerated by GPUs. For example, a GPU implementation of LOF proposed by [6] achieved a more than 100x speed up over software and that of KNN proposed by [7] also achieved a 410x speed up. GPUs are typically used as slave devices of a host machine and thus input data are fed by the host CPU to the accelerator devices via PCI-Express. In contrast, this paper focuses on outlier filtering of packets coming from high bandwidth networks. To this end, FPGA NIC is a practical choice rather than GPUs since we can reduce the CPU workload of host application by discarding unnecessary data (not outliers) at the NIC. We propose a dataset cache (described in Section III) in order to implement LOF and KNN on the FPGA NIC with limited on-board memory. Thanks to the dataset cache, the required memory resource and sorter circuit can be reduced in proportional to the dataset cache size since our proposed outlier

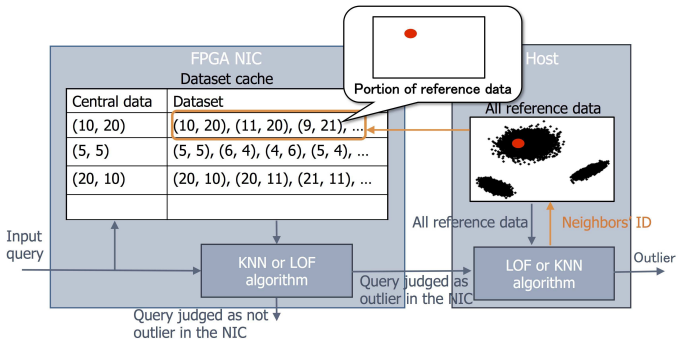

Fig. 2. Outlier filtering NIC using lazy learning

detection uses the dataset cache as reference data. Please note that our proposed dataset cache is orthogonal to the above mentioned optimizing techniques. We are expecting further improvements in cost and performance by combining various optimization on distance computation and sorting with our dataset cache approach.

E. Scalable-Effort Classifier

Venkataramani et.al. proposed scalable-effort classifiers for energy-efficient machine learning [11]. Their approach scales computational load depending on types of input queries. That is, it dynamically adjusts the computational effort depending on the difficulty of the input data, and the large majority can be classified correctly with very low effort. In this paper, we propose to detect outliers and discard them at the NIC by introducing a dataset cache that caches frequently-accessed portion of reference data in the NIC. Outlier queries are transferred to the host and processed again with all the reference data by the host application. In other words, queries which are clearly detected as normal are discarded using dedicated hardware at the NIC, while other queries are processed using the full calculation by the host application. In this respect, our proposed approach faces the same direction as [11] but we focus on completely different and practical network-based systems where easy queries are processed by the NIC.

III. Design and Implementation

In this section, the outlier filtering NIC with dataset cache is proposed.

A. Outlier Filtering NIC Using Lazy Learning

Queries (sample data) are processed by the outlier detection module using lazy learning in the FPGA NIC as illustrated in Figure 1. A query is discarded at the NIC when it is not detected as an outlier, while it is delivered to a network protocol stack in the host when it is detected as an outlier. Figure 2 illustrates an overview of the proposed outlier filtering NIC using lazy learning (i.e., details of "Server" part in Figure 1). Reference data are accumulated in the host application. In our proposal, the outlier detection is performed based on a part of frequently-accessed reference data cached in the NIC. This in-NIC cache is called "dataset cache." The dataset cache has two parameters: the number of lines (the maximum number of datasets cached) and length of a line (the maximum number of data in a dataset). These parameters can be customized to fit to the available memory resource.

B. Processing Flow

An input query is processed with the following five steps.

1) A dateset supposed to be the nearest to the input query is selected.
2) Outlier detection is performed using LOF or KNN with the selected dataset as reference data.
3) The query is discarded when it is not an outlier, while it is passed to the host application when it is an outlier.
4) The host application performs an outlier detection again using complete reference data.
5) Neighbor data of the query are cached in the dataset cache of the NIC when the query is not detected as an outlier by the recalculation.

Each of these steps is described below.

Step 1: Distances between the query and central data of each line (described in Step 5) are calculated and the line with the shortest distance is selected. Dataset stored in the selected line is used as reference data.

Step 2: The query is processed by the outlier detection module with reference data selected in the previous step. The computation of this step is different between LOF and KNN, which will be illustrated in Section III-C.

Step 3: Only the query detected as an outlier is passed to a network protocol stack of the host. Filtering non-outlier data at the NIC reduces the workload of the host drastically.

Step 4: The query transferred from the NIC is then processed by an outlier detection again with all the reference data in the host application. Such a query has been detected as an outlier by the outlier detection with only frequently-used reference data cached in the NIC. If its neighborhoods have not been cached in the NIC, the outlier detection at the NIC may not work correctly. In this case, the distances between the query and the neighborhoods (or the density) are not calculated correctly, so the query may be detected as outlier erroneously. Thus, it is essential to perform the outlier detection again with full reference data to judge whether it is really outlier or not.

Step 5: This step is executed when the query passed from the NIC is not an outlier. In other words, when the query is detected as an outlier at the NIC but it is not detected as an outlier at the host, Step 5 is executed. As mentioned in Step 4, an erroneous detection at the NIC occurs when an appropriate dataset for the query has not been cached at the NIC. Hence, in this step, n neighbor data of the query are inserted to the dataset cache in the NIC as a new dataset. Here, n is the line length of the dataset cache, which means the number of data included in a dataset. At this time, central data of the cache line where the dataset is stored are regarded as the values of the nearest data. Central data are used when the dataset is selected for the outlier detection in the NIC (described in Step 1). Updating the cache takes a certain number of cycles, because n data are sent from the host application to the NIC; thus the dataset cache cannot be updated by the query received from the NIC until the previous update has been completed. We adopt LRU (Least Recently Used) as the replacement algorithm and the replacement is performed line by line.

In the proposed system, an input query is processed using LOF or KNN with a frequently-accessed portion of the reference data cached in the NIC. The query is discarded at the NIC when it is not detected as an outlier. The query detected as an outlier at the NIC is transferred to the host. It is processed again using LOF or KNN with all the reference data by the host application.

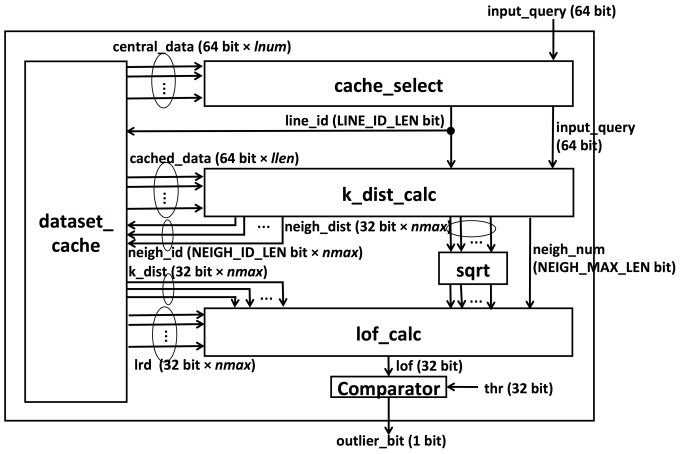

Fig. 3. Overview of outlier detection module

The proposed system reduces not only host workload for LOF or KNN algorithm but also that for network protocol processing. Although LOF requires a large reference dataset which may not be fit to the FPGA NIC, by introducing our dataset cache we can offload it to the NIC. Since the proposed system can detect a part of queries as non-outliers and discard them at the NIC, we can significantly reduce the data size at the NIC and host workload.

C. Design of Outlier Detection Module

In this section, a design of our outlier detection module including the dataset cache (i.e., details of "FPGA NIC" part in Figure 2) is illustrated. We first show the outlier detection using LOF and then we summarize necessary changes for the outlier detection of KNN.

Figure 3 shows an overview of the outlier detection module based on LOF. In this figure, $lnum$ is the number of the lines in the dataset cache, $llen$ is length of the line, and $nmax$ is the maximum number of neighborhoods. LINE_ID_LEN should be wide enough to represent a line ID (depending on $lnum$). NEIGH_ID_LEN should also be width enough to represent neighbors' ID (depending on $llen$). In this illustration, we assume that the number of features is two and length of a feature is 32-bit for simplicity.

This module consists of three modules and the dataset cache. First, an input query is processed in cache_select module and a line_id is selected. This process corresponds to Step 1 in Section III-B. Then the following procedure is corresponding to Step 2. A dataset referred by the outlier detection is selected by line_id and data included in this dataset are transferred to k_dist_calc module. This module finds neighbor data and then it outputs 1) neighbors' ID (neigh_id), 2) squared distances between the query and neighborhoods (neigh_dist), and 3) the number of neighborhoods (neigh_num). Neighbors' k-distance (k_dist) and local reachability density (lrd) are read from the dataset cache by neigh_id and line_id. As these neighbors' information, square roots of distances between the query and the neighborhoods (i.e., neigh_dist) and neigh_num are fed to lof_calc module. Reachability distances between the query and its neighborhoods, its local reachability density, and its LOF (lof) are calculated and lof is outputted from lof_calc module. Then, the query is judged as to whether it is an outlier or not by comparing lof and an user-specified threshold (thr). The result is outputted as a 1-bit signal.

When we employ KNN for the outlier detection, the square root circuits and lof_calc module are not necessary since the distance between the query and k-th nearest data is directly compared to thr. In addition, memory and wires for neighbors' k_distance and lrd are not necessary.

We will illustrate the overview of sub-modules (cache_select, k_dist_calc, and lof_calc modules) and the proposed dataset cache below.

1) cache_select Module: cache_select module is used for both LOF and KNN algorithms. Central data (central_data) and the query (input_query) are inputted to this module and ID of dataset referred by the outlier detection are outputted as line_id from this module. First, squared distances between central_data and input_query are computed. To compute them, $4 \times feat$ DSP Slices are consumed, because a distance calculator consists of $feat$ (the number of features) multipliers each of which consumes 4 DSP Slices. Hence, a cache_select module requires $4 \times feat \times lnum$ DSP Slices. Then, to find the minimum distance (central data closest to the query), these distances are compared in a tournament manner. It takes $\log_2 lnum$ cycles to find the minimum value.

2) k_dist_calc Module: Neighborhoods of the query (input_query) are selected from the reference data (cached_data) in k_dist_calc module. This module outputs IDs of neighborhoods (neigh_id), squared distances between the query and neighborhoods and the number of neighborhoods (neigh_num). First, squared distances are computed as well as cache_select module. k_dist_calc module requires $4 \times feat \times llen$ DSP Slices. Second, a partial sort is performed by partial_sort module. This module outputs top $nmax$ (maximum number of neighborhoods) smallest values and their IDs among all the input values. In other words, the output values are squared distances and their IDs of $nmax$ neighborhoods. The number of neighborhoods is also calculated with a circuit including a comparator.

In the case of using KNN, only the squared distance between the query and k-th nearest data is required. Therefore, circuits to compute the other neighbors' information are not required.

We illustrate partial_sort module below. The partial sort here requires a sorter that outputs only the top $nmax$ smallest values. In this paper, a partial sorter based on merge sort is implemented in partial_sort module because of the simplicity and relatively low execution time. Figure 4 shows a detail of partial_sort module. Please note that width of "data" in this figure is (NEIGH_ID_LEN+32)-bit although not specified in this figure. This module is divided into multiple stages (separated by broken lines in this figure) and these stages are pipelined. First, the input values (each of which has an ID based on its input port) are compared to their next values and stored in registers in ascending order. Second, the pairs of these registers are sorted. As shown in Figure 4, a value pointed by an address of a register is compared to one after another and a lower value is copied to another register. The first address points to a minimum value in the register. The address is incremented when a value pointed by the address is copied. The final (sorted) sequence of values is obtained by repeating these reordering steps. Please note that this is a partial sorter, so there is no need to sort values except the top $nmax$ smallest values. In other words, in each stage, the reordering steps are stopped as soon as the top $nmax$ smallest values have been identified in the stage. Assuming, for example, $nmax = 8$ in Figure 4, only the top-eight smallest values are selected from two input sequences each of which consists of eight values (i.e., sixteen

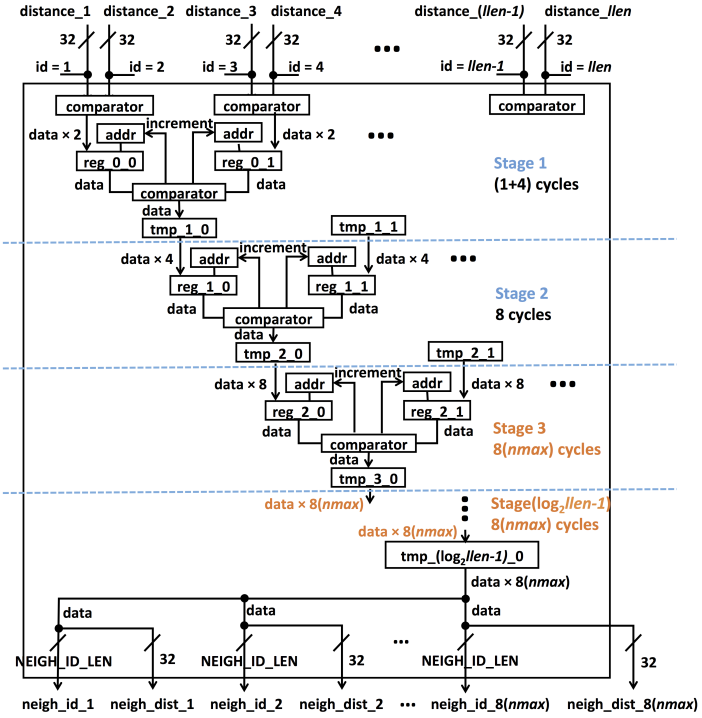

Fig. 4. partial_sort module

values in total). Then only the eight values (not sixteen values) are passed to the next stage. In this way, the number of cycles and the area are reduced compared to the conventional sorter circuit.

The number of cycles for the partial sort equals to the maximum number of neighborhoods ($nmax$). In our implementation, the number of cycles for the partial sort affects the performance of outlier detection at the proposed NIC.

3) lof_calc Module: LOF of the input query is computed based on distances between the query and neighborhoods (neigh_dist), neighbors' k-distances (k_dist), neighbors' local reachability density (lrd), and the number of neighborhoods (neigh_num) in lof_calc module. Processing in this module corresponds to Equations (4), (5), and (6) in Section II-B. This module requires two multipliers and one divider. A multiplier consists of 4 DSP Slices and a divider consists of 14 DSP Slices, so this module requires 22 DPS Slices in total.

4) Dataset Cache: The dataset cache has three purposes: 1) to retain and provide central_data, 2) to retain and provide cached_data based on a given line_id, and 3) to retain and provide k_dist and lrd based on a given line_id and neigh_id. Regarding the first purpose, it continuously outputs the same values as long as there is no update on the cache. This part is implemented with distributed RAMs. On the other hand, the main body of the dataset cache is related to the second and third purposes. These parts are implemented with BRAMs because of the amount of data to be stored.

The dataset cache has separated read and write ports. The read port is used to fetch the cached data, while the write port is used to update the cache. Here we mainly explain the cache read procedure since it determines the outlier detection performance.

Regarding the second purpose, it outputs all the feature values (excluding k_dist and lrd) included in the line selected by a given line_id. Although only a single read port is needed for this purpose, multiple BRAM instances are consumed because

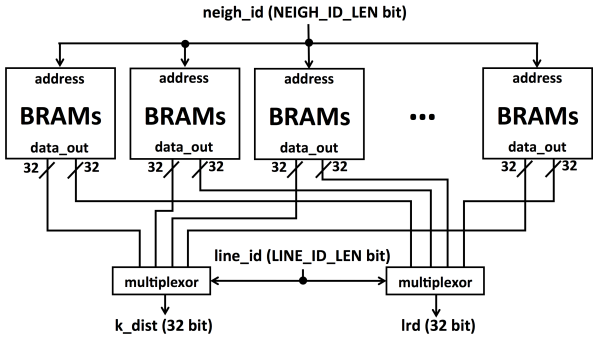

Fig. 5. Dataset cache for k_dist and lrd

of its wide output data width. For example, it requires a 1kB width if a single line contains 128 64-bit data.

The third part is illustrated in Figure 5. This part outputs neighbors' information of a specified line. In our design, a separated BRAM instance is used to store a single line. The output of the dataset cache is thus selected by a given line_id among outputs from all the BRAM instances. Please note that there are $nmax$ neighbors' IDs in each line, which means that $nmax$ cycles are needed to read out the neighbors' information from the cache line. We can reduce the number of BRAM access cycles by distributing a single line data to multiple BRAM instances, although the performance bottleneck currently lies in the partial sorter, not the dataset cache. In the case of KNN, this cache is not required.

IV. Evaluations

The proposed outlier filtering NIC based on LOF and KNN is evaluated in terms of hardware amount, hit ratio, and the maximum throughput.

A. FPGA Resource Utilization

We evaluate the proposed outlier filtering NIC based on LOF and KNN in terms of the LUT, BRAM, and DSP utilizations by varying the dataset cache parameters (i.e., the number of lines and the line length).

1) Environment: We assume NetFPGA-SUME [12] as a target FPGA NIC board, so target FPGA device of the design synthesis is Virtex-7 XC7VX690T. Xilinx ISE Design Suite 13.4 is used for the design synthesis.

We employ Reference NIC design provided by the NetFPGA team [12] as a baseline NIC function and add our outlier detection modules to the NIC. Since Reference NIC is not our contribution, experimental results in this section focus only on our outlier detection modules using lazy learning. Square root circuits, multipliers, and dividers in our modules are generated by Core Generator provided by Xilinx ISE. Memory resources for the dataset cache are implemented as BRAMs.

2) Results: Tables I, II, and III show the LUT, BRAM, and DSP utilizations of the proposed outlier detection module using LOF and KNN, respectively. In addition, Tables IV and V show the LUT and DSP utilizations of cache_select module, k_dist_calc module, lof_calc module, and a square root circuits, where "128_128" and "64_64" mean that the line length and the number of lines are both 128 and 64, respectively. Their ratios over the whole outlier detection module are also shown in these tables.

Again, the challenges to offload the lazy learning to the NIC are twofold: 1) necessity to retain huge reference data in the

TABLE I
NUMBER OF LUTS USED (LUT UTILIZATION)

			Line length	
			64	128
Number of lines	LOF	64	64,816 (15.0%)	106,451 (24.6%)
		128	78,926 (18.2%)	121,304 (28.0%)
	KNN	64	51,966 (12.0%)	90,336 (20.9%)
		128	64,636 (14.9%)	103,008 (23.8%)

TABLE II
NUMBER OF BRAMS USED (BRAM UTILIZATION)

			Line length	
			64	128
Number of lines	LOF	64	129 (8.8%)	193 (13.1%)
		128	193 (13.1%)	257 (17.5%)
	KNN	64	64 (4.4%)	128 (8.7%)
		128	64 (4.4%)	128 (8.7%)

TABLE III
NUMBER OF DSPS USED (DSP UTILIZATION)

			Line length	
			64	128
Number of lines	LOF	64	1,046 (29.1%)	1,588 (44.1%)
		128	1,588 (44.1%)	2,070 (57.5%)
	KNN	64	1,024 (28.4%)	1,536 (42.7%)
		128	1,536 (42.7%)	2,048 (56.7%)

TABLE IV
NUMBER OF LUTS USED IN EACH MODULE (RATIO TO WHOLE OUTLIER DETECTION MODULE USING LOF)

	cache_select	k_dist_calc	lof_calc	sqrt
128_128	17,276 (14.2%)	84,621 (69.8%)	2,782 (2.3%)	7,120 (5.9%)
64_64	8,277 (12.8%)	41,762 (64.4%)	2,782 (4.3%)	7,120 (11.0%)

TABLE V
NUMBER OF DSPS USED IN EACH MODULE (RATIO TO WHOLE OUTLIER DETECTION MODULE USING LOF)

	cache_select	k_dist_calc	lof_calc	sqrt
128_128	1,024 (49.5%)	1,024 (49.5%)	22 (1.1%)	0 (0%)
64_64	512 (48.9%)	512 (48.9%)	22 (2.1%)	0 (0%)

NIC and 2) necessity to implement a large sorter that sorts distances between the query and all the reference data in the NIC.

Regarding the first challenge, as shown in Table II, the BRAM utilization of outlier detection module using LOF is 17.5% when the dataset size is 128×128. Regarding the second challenge, as shown in Table I, the LUT utilization of outlier detection module using LOF is 28.0% when the dataset cache size is 128×128. This result shows that the proposed design is practical to implement. One notable advantage of the proposed dataset cache is that we can reduce the number of reference data to be sorted, resulting in a significant resource saving.

B. Hit Ratio

Here we evaluate the proposed system in terms of hit ratio and discuss the relationship between input queries and the hit ratio. In this paper, the hit ratio is defined as "the number of queries judged as non-outlier at the NIC / (the number of queries judged as non-outlier at the NIC + the number of queries judged as outlier at the NIC but not at the host)," as shown in Figure 6. In other words, the hit ratio is true negative ratio.

1) Environment: To evaluate the hit ratio, we performed simulations using R library. The reference data in the host and input queries are prepared as follows.

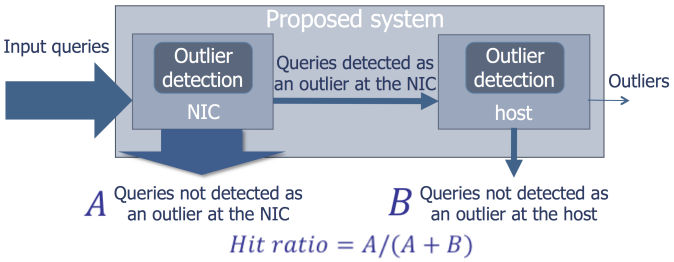

Fig. 6. Relationship between detection results of input queries and hit ratio

- Number of features is two (32-bit floating point number for each).
- Dataset model consists of ten clusters (Gaussian distribution or uniform distribution).
- Each cluster contains 10,000 data (100,000 data in total).
- Input queries are generated with the same parameters of reference data.

Figures 7 and 8 show reference data used in the experiments. X-axis and Y-axis in the figures show the two feature values. We used four types of input queries generated in following conditions.

1) Queries are belonging to all the clusters evenly.
2) 90% of queries are belonging to a specific cluster.
3) 90% of queries are belonging to one of two specific clusters.
4) 90% of queries are belonging to one of three specific clusters.

2) Results: We evaluate the hit ratio when the size of dataset cache is 128×128 and $k = 10$, while thr is varied. Figures 9 and 10 show simulation results using LOF, while Figures 11 and 12 show those using KNN.

The outlier detection is performed in the host application only if the query is detected as an outlier in the NIC. Thus, as the hit ratio increases, more computational cost in the host can be reduced. Assuming all the queries are not outlier, for example when hit ratio is 90%, computational cost (i.e., outlier detection and network protocol stack processing) is reduced by 90%. As shown in these graphs, when the queries are concentrated on a smaller cluster, the hit ratio becomes higher. In particular, the proposed system can reduce more than 90% of workload in the host when the queries are concentrated on a single cluster. Furthermore, the hit ratio decreases slowly when the locality of input queries becomes lower. The dataset cache used in this evaluation retains only one-sixth of entire reference data in the host (100,000 data) since $128 \times 128 = 16,384$ data can be cached. LOF (or $k\_distance$) is not calculated exactly compared to using the original algorithm when the nearest neighborhoods have not been cached. However, LOF would not become high compared to that of outliers when reference data relatively close to the input query have been cached; thus such a query is not detected as an outlier. Thanks to such relatively close neighborhoods cached, the hit ratio is kept high enough. These results demonstrate that our proposed dataset cache is suitable for outlier detection using lazy learning at the NIC.

As shown in these graphs, as thr becomes higher, the hit ratio is increased. When thr is high, an input query would not be judged as an outlier unless distance between the query and the dataset cached in the NIC is quite large. However, as thr changes, the queries detected as outliers are changed. For

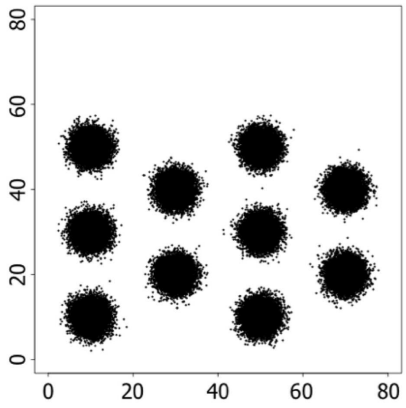

Fig. 7. Distribution of reference data based on Gaussian distribution

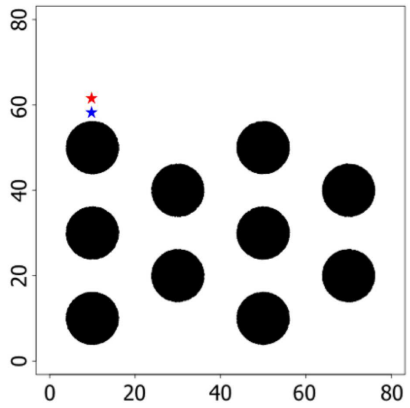

Fig. 8. Distribution of reference data based on uniform distribution

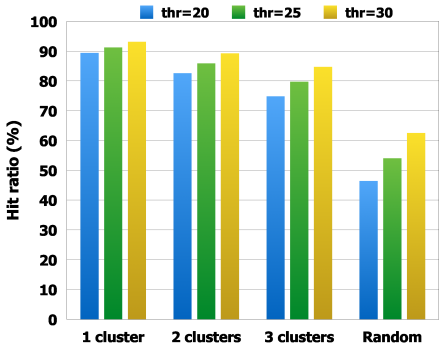

Fig. 9. Characteristic of input queries vs. hit ratio with outlier detection using LOF (Gaussian distribution)

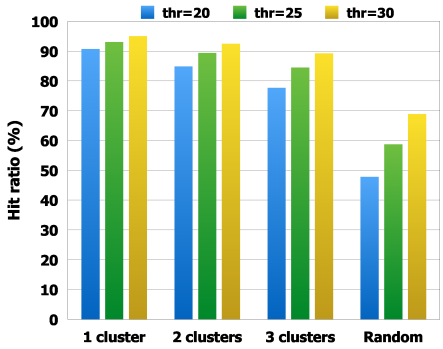

Fig. 10. Characteristic of input queries vs. hit ratio with LOF outlier detection using (uniform distribution)

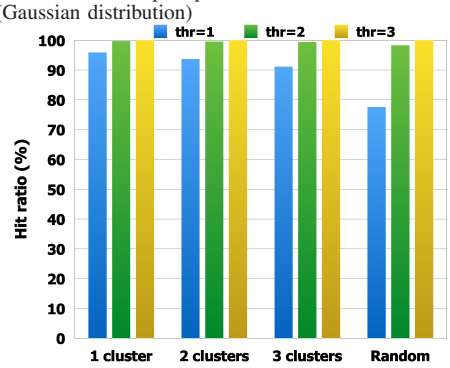

Fig. 11. Characteristic of input queries vs. hit ratio with outlier detection using KNN (Gaussian distribution)

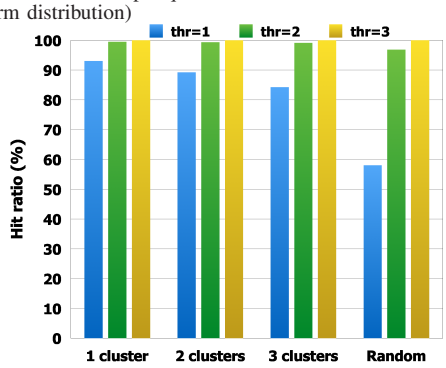

Fig. 12. Characteristic of input queries vs. hit ratio with outlier detection using KNN (uniform distribution)

example, in Figure 8 (KNN is used), both the red and blue stars are detected as outliers with $thr = 1$, while the red star is detected as an outlier but blue star is not with $thr = 3$.

C. Performance Estimation

The query detected as an outlier at the NIC is recalculated by a host application. For example, assuming the hit ratio is 80%, 20% of queries are processed at the host application again while 100% of queries are processed at the NIC. Therefore, throughput of outlier detection with the proposed system $T_{Proposal}$ [Queries/sec] is represented as the following equation.

$$T_{Proposal} = \min\{T_{NIC}, T_{Host}/(1 - P_{Hit})\}, \quad (7)$$

where T_{Host} is the throughput of outlier detection at the host, T_{NIC} is that of outlier detection at the NIC, and P_{Hit} (0∼1) is the hit ratio. The throughput improvement of the proposed system compared to an outlier detection with only a host application is represented as the following equation.

$$T_{Proposal}/T_{Host} = \min\{T_{NIC}/T_{Host}, 1/(1 - P_{Hit})\} \quad (8)$$

T_{NIC} is calculated from the number of cycles and the maximum operation frequency of the outlier detection module. It is mainly affected by the number of cycles for the partial sorting. The partial sorting takes $nmax$ cycles per query, as mentioned in Section III-C2. It is not affected by the size of dataset cache nor other parameters. In this section, we assume $k = 10$ and $nmax = 16$. Maximum operation frequencies of

the outlier detection module with LOF and KNN are shown in Table VI. Evaluation environment is the same as that in Section IV-A. These results demonstrate that the maximum frequency is approximately 130MHz; thus T_{NIC} is approximately 8,000,000 [Queries/sec].

TABLE VI
MAXIMUM OPERATION FREQUENCY

			Line length	
			64	128
Number of lines	LOF	64	134.3MHz	132.5MHz
		128	123.1MHz	131.4MHz
	KNN	64	160.5MHz	159.6MHz
		128	160.1MHZ	152.0MHZ

T_{Host} in the case of using LOF is evaluated in the following environment.

- DMwR::lofactor of R library (extended to the proposed system)
- Intel Core i7 (2.5GHz)
- 16GB RAM (DDR3, 1600MHz)
- OS X 10.9.5

Table VII shows the throughput of outlier detection with all the reference data in the host. As the number of reference data increases, the software performance degrades greatly due to the large amount of distance computations and sortings. From these results, in the case of LOF, T_{NIC} is 8,000,000, T_{Host} is 49$\sim$3,585, and P_{Hit} ranges 0.9$\sim$0.45 as shown in Figure 9. Using Equation (8), the proposed system achieves 10$\sim$1.82 times higher performance compared to the software.

TABLE VII
THROUGHPUT OF OUTLIER DETECTION USING LOF BY HOST APPLICATION

Number of reference data	Throughput [Queries/sec]
1,000	3,585
10,000	528
100,000	49

Here, we evaluated T_{Host} by a single thread execution without further optimizations, so a performance improvement might be obtained when we employ more optimized software implementations. However, T_{NIC} is much higher than T_{Host}, and thus the advantage of the proposed system will not be changed.

D. Discussions on Precision

Queries detected as outliers are recalculated by host application, so false positive pattern (i.e., non-outlier detected as an outlier at the NIC) does not affect the precision, while false negative pattern (i.e., true outlier not detected as an outlier at the NIC) affects the precision. The false positive pattern is likely when neighborhoods of an input query are not cached, while the false negative pattern is expected to be quite rare as illustrated below. Assuming a query p is an outlier, p cannot be detected as an outlier only when densities of neighborhoods cached in the NIC are close to the density of p. However, because, in our design, only frequently-used reference data are cached in the NIC, the neighborhoods cached in the NIC should be far from outliers. Because of this contradiction, the false negative pattern should be quite rare and actually we have not observed any false negative patterns throughout all the experiments in this paper.

V. SUMMARY

The purpose of this paper is to perform the outlier detection using lazy learning on FPGA NICs in order to filter outliers. Since most queries are filtered at the NIC efficiently, CPU workloads for network protocol processing and outlier detection of the received queries is greatly reduced. However, it is challenging to offload lazy learning algorithms to NICs because of the high computational cost and huge reference data needed for the outlier detection. In this paper, we solved this problem by caching frequently-accessed reference data in the FPGA NIC.

We designed and implemented the outlier detection based on LOF and KNN, and evaluated them in terms of resource utilization and maximum operation frequency. We also demonstrated the feasibility to implement them on Xilinx Virtex-7 FPGA. The query detected as an outlier at the NIC is passed to a host application and an outlier detection using the full reference data accumulated in the host is performed. Therefore, as the hit ratio (true negative ratio) increases, the CPU workloads of the host is reduced. Simulation results using 100,000 reference data showed that 45%$\sim$90% queries are hit to the proposed dataset cache in the NIC. This corresponds to 1.82x$\sim$10x performance improvement compared to that without the outlier detection in the NIC. As future work, we will increase the number of feature dimensions. We are now improving the dataset cache so that it can handle high dimensional data.

Acknowledgements This work was supported by JST PRESTO and NEDO (Grant-in-Aid for Development of Cross-Sectoral Technologies to Promote the IoT).

REFERENCES

[1] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "LOF: Identifying Density-Based Local Outliers," in *Proceedings of the International Conference on Management of Data (SIGMOD'00)*, May 2000, pp. 93–104.
[2] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, "Top 10 Algorithms in Data Mining," *Knowledge and Information Systems*, vol. 14, no. 1, pp. 1–37, Dec. 2007.
[3] B. V. Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?" in *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM'12)*, Apr. 2012, pp. 232–239.
[4] A. Hayashi, Y. Tokusashi, and H. Matsutani, "A Line Rate Outlier Filtering FPGA NIC using 10GbE Interface," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 4, pp. 22–27, Sep. 2015.
[5] A. Das, D. Nguyen, J. Zambreno, G. Memik, and A. Choudhary, "An FPGA-Based Network Intrusion Detection Architecture," *IEEE Transactions on Information Forensics and Security*, vol. 3, no. 1, pp. 118–132, Mar. 2008.
[6] M. Alshawbkeh, B. Jang, and D. Kaeli, "Accelerating the Local Outlier Factor Algorithm on a GPU for Intrusion Detection Systems," in *Proceedings of the Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU'10)*, Mar. 2010, pp. 104–110.
[7] Y. Pu, J. Peng, L. Huang, and J. Chen, "An Efficient KNN Algorithm Implemented on FPGA Based Heterogeneous Computing System Using OpenCL," in *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM'15)*, May 2015, pp. 167–170.
[8] E. S. Manolakos and I. Stamoulias, "Flexible IP Cores for the k-NN Classification Problem and Their FPGA Implementation," in *Proceedings of the International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW'10)*, Apr. 2010, pp. 1–4.
[9] H. Hussian, K. Benkrid, C. Hong, and H. Seker, "An Adaptive FPGA Implementation of Multi-Core K-Nearest Neighbour Ensemble Classifier Using Dynamic Partial Reconfiguration," in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL'12)*, Aug. 2012, pp. 627–630.
[10] Y.-J. Yeh, H.-Y. Li, W.-J. Hwang, and C.-Y. Fang, "FPGA Implementation of kNN Classifier Based on Wavelet Transform and Partial Distance Search," in *Proceedings of the Scandinavian Conference on Image Analysis (SCIA'07)*, Jun. 2007, pp. 512–521.
[11] S. Venkataramani, A. Raghunathan, J. Liu, and M. Shoaib, "Scalable-Effort Classifiers for Energy-Efficient Machine Learning," in *Proceedings of the Design Automation Conference (DAC'15)*, Jun. 2015.
[12] "The NetFPGA Project," http://netfpga.org/.