Real-Time Inferring Network Traffic Patterns

Guang CHENG

Abstract— **It is vitally important for applications in detecting DoS attacks, traffic management, and network security to real-time automatically identify traffic patterns in backbone networks with high speed links carrying large numbers of flows. Our objective is to determine traffic patterns that use up a disproportionate fraction of network resources. This paper first analyzes the major time and space cost in computing high volume clusters under different hierarchical structures, and then proposes a variable hierarchical structure to identify network traffic patterns in a top-down fashion. We evaluate our model using real trace files from the CERNET backbone link and demonstrate the improved efficiency of our approach in comparison to previous work on clustering traffic patterns.**

Key Words —**Network Measurement, Traffic Pattern, Data Mining**

I. Introduction

There is an essential need for a network manager to possess the structure of traffic mix that is taking up our limited bandwidth. A comprehensive knowledge of such traffic characteristics is naturally the premise of optimizing, deploying and managing network resources. Analysis over raw traffic data is require to extract better insight which is supposed to yield more meaningful report to manager.

Eventually, there is an inherent contradiction between the level of detail provided by traffic data and the capacity of humans to absorb knowledge beneath that detail. Packet-level information does reflect all what is happening on network but it at the same time almost tells little useful message that we probably want more. A good traffic report is expected to present a global summarization in a time-saving manner that will potentially stop the manager from "hunting for needles". Mining traffic aggregation could be used as one feasible solution. It is vitally important for applications in detecting DoS attacks, traffic management, and network security to real-time automatically identify traffic patterns in backbone networks with high speed links carrying large numbers of flows. Our objective is to determine traffic patterns that use up a disproportionate fraction of network resources.

We take those significant traffic patterns whose traffic volume exceeds a pre-defined fraction of the total traffic as a summarization of traffic mix because they are dominant resource consumers. This paper first analyzes the major time and space cost in computing high volume clusters under different hierarchical structures, and then proposes a variable hierarchical structure to identify network traffic patterns in a top-down fashion. We evaluate our model using real trace files from the CERNET backbone link.

The rest of the paper is organized as follows. We describe related work in Section II. Section III defines the clusters and their properties which are chosen to do traffic patterns. We describe the variable hierarchical algorithm we use in Section IV. Experimental evaluation based on the proposed method is presented in Section V. The conclusion is given in Section VI.

II. Related work

C.Estan in [1] proposed an offline algorithm that aggregates traffic over hierarchical domain which is built in terms of set inclusion on every individual dimension from the "five-tuple definition" of network flow. Instead of using individual flows or other predefined aggregates, they dynamically define traffic clusters, so that any meaningful aggregate of individual flows is a traffic cluster. The difference with ours is that their algorithm is considered as an offline system instead of a real-time one. Wang in [2] concluded that the major computational overhead of Estan's algorithm is spent on potential sorting operation while building the hierarchical tree from leaves to root because there is an inevitable need to put all leaves in order before actual building procedure. To eliminate the need for potential sorting, Wang improved Estan's approach using a top-down way to build the whole hierarchy tree and each node maintains a set of flows that the node covers. However, their strategy limits the search range during the creation of all child nodes that belong to a same parent.

Zhang and P.Truong in [3] and [4] proposed similar real-time online approximate algorithms. A "split threshold" is introduced to suppress the growth of hierarchical tree. Each arriving flow will at most cause one counter update and one new node creation so the overall algorithm cost is decreased. In 2005, Ken Keys[5] computes multiple summaries to aggregate the flows according to the source IP, the destination IP, the source port and the destination port. In 2008, Cheng [6] gave an algorithm to detect superspreaders adaptively on different sampling probabilities, which is used to maintain those recorded IP addresses in a limited memory. These algorithms only provide traffic summaries, but do not keep any original flow information as Cisco NetFlow [7] does.

This work was supported by the National Grand Fundamental Research 973 program of China under Grant No. 2009CB320505, the National Nature Science Foundation of China under Grant No. 60973123, the National Science & Technology Pillar Program during the eleventh five-year plan period under Grant No. 2008BAH37B04, and the Nature Science Foundation of Jiangsu Province under Grant No. BK2008288.

Guang CHENG is a professor of the School of Computer Science and Engineering, Southeast University, Nanjing, 210096, PRC. His office phone: 86-25-83794000; fax: 86-2583614842; e-mail: gcheng@njnet.edu.cn.

III. Problem specification

Let $S=\{\alpha_1,\alpha_2,\alpha_3,\cdots\}$ be an input stream of flows that arrive sequentially. Each item $\alpha_i=(k_i,u_i)$ consists of a key k_i and a positive volume update $u_i\in I$. Associated with each k is a volume counter A[k]. Every new arrival of flow (k_i,u_i) will cause volume counter A[k] to be updated: $A[k_i]=A[k_i]+u_i$.

Definition 1 (Hi gh-Volume Clus ter): Given a set of incoming flows $S=\{\alpha_1,\alpha_2,\alpha_3,\cdots\}$ (multi-set) with a total traffic volume $sum=\sum_i u_i$ and a threshold $\varphi(0<\varphi\le 1)$. Let $V_k=\sum_{(i:k_i=k)}u_i$ denote the traffic volume associated with key k in flow set S, a high-volume cluster is defined as $\{k\mid V_k\ge\varphi\cdot sum\}$.

Definition 2 (Hierarchi cal High-Volume Cluster): Given a set of incoming flows $S=\{\alpha_1,\alpha_2,\alpha_3,\cdots\}$ whose keys k_i are drawn from a hierarchical domain D of height h. For any prefix p in hierarchy D, let $elem(D,p)$ be the set of elements that descendents of p. Let $V(D,p)=\sum_k V_k : k\in elem(D,p)$ denote the total traffic volume associated with prefix p. The set of Hierarchical High-Volume Clusters is defined as a set of prefixes $\{p\mid V(D,p)\ge\varphi\cdot sum\}$.

In the conventional five-tuple flow definition, source and destination IP dimension can naturally form a hierarchy domain Dom_{ip} in terms of IP prefix whose leaves are individual IP addresses. For port and protocol dimensions, simple hierarchies are used to reflect inherent semantics. The goal of unidimensional traffic aggregation is to breakdown total traffic along hierarchy domains and to identify all high-volume clusters.

Tree can be used to express the logical structure of hierarchical domain. Every node in the hierarchy tree is associated with a set of flows that share same semantic importance. The root of hierarchy tree represents all possible values on this dimension. All leave nodes are associated with individual value on this dimension. Internal nodes between root and leaves are intermediate results while doing the aggregation. The sets denoted by two nodes are disjoint unless one of the nodes is a parent of the other.

We can find all high-volume clusters of a dimension by maintaining the entire hierarchical domain in memory and then traverse each node to see if it is above the threshold. This approach works for simple dimensions such as port and protocol dimension because the hierarchies are small. However IP dimension covers a 32-bit space, it is not possible to fit it all into precious measurement resource. A practical solution is to maintain a hierarchy tree containing only individual IP and their ancestors along the paths to root. The scope of this partial hierarchy domain is acceptable in real scenario.

Estan, based on this idea, presented in [1] his unidimensional aggregation algorithm on IP address dimension. Estan's algorithm is divided into two phases. First we traverse all individual flows in raw traffic and create leave nodes in the hierarchy. Merge is required if a second flow of a same IP address is found to ensure the uniqueness. The complete hierarchy tree is built according to prefix relationship among individual flows by the end of this pass. All leave counters are set to their actual traffic and the counters of internal nodes are set to 0. In the second phase of algorithm, a post-order traversal is applied to hierarchy tree to set the traffic counters of all internal nodes to the sum of their descendents. High-volume clusters are identified as the traverse process leaves a node.

Prefix length increases by 1 bit as hierarchy depth drills down every layer. So a binary tree could be used to maintain such hierarchical relation. In Estan's algorithm, prefix length changes from 8 bit to 32 bit. With a root node that directly connects to /8 prefix, the depth of hierarchy tree is 26. The cost of keeping every node of hierarchy tree in memory is expensive. The major time cost consists of the creation of new nodes and the counter update operation. This binary tree storage strategy keeps cluster and its corresponding traffic at any prefix length. The total number of nodes in binary tree could be very large, but the advantage of this approach is when we attempt to access any internal node we do not need to compute its actual traffic. We can extend this binary tree structure to k-tree, so the change of prefix length increases accordingly.

Let *child* be the number of descendent of a given node, *layer* be the depth of hierarchy tree. Because the number of all leave node is equal to IP address space, so $child=2^{(32/layer)}$. All possible combinations of *child* and *layer* are listed in Table 1.

TABLE 1: POSSIBLE LAYER AND CHILD COMBINATION

layer	child
32	2
16	4
8	16
4	256

Analyze a direct approach to traffic aggregation problem: If all child and layer parameters are given, then all nodes (if not existed) are created and updated their corresponding traffic counter (if already existed or being created) along the path from root to leaf. An example which includes 2 flows: 202.112.25.69(100) and 202.112.23.167(80) that arrived sequentially is illustrated in Figure 1 below. This example shows how the creation and update operation work. Colored node in the Figure suggests that node is newly created.

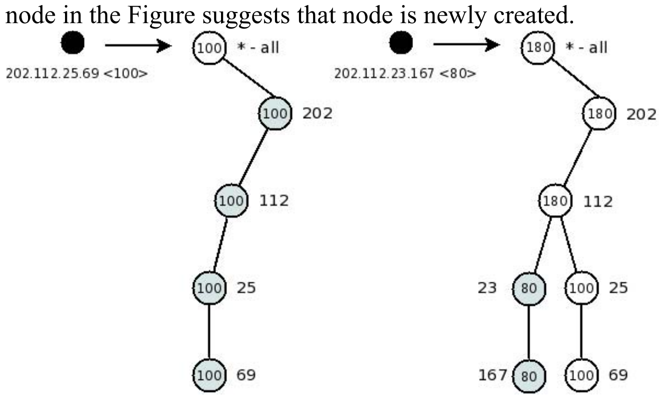

Figure 1 A Example of the Algorithm

We obtain the following algorithm time and space performance results through experiment using continuous

Netflow data split into five-minute grain staring from 09:00 to 09:20 Jul 15th, 2009 in Figure 2. Processing time of our direct approach generally rises as the aggregation layer increases. There is a prominent exception when layer changes from 4 (256 descendents per node) to 8 (16 descendent per node). It is because the creating a new node takes much more time than updating the traffic counter. Though the number of traffic counter that needs update when layer grows from 4 to 8 doubles, the actual time of creation reduces by a large scale because some of the nodes along the path are probably already existed due to previous arriving flows. The individual addresses appeared in our raw traffic are usually concentrate to several active subnets so the collision happens a lot, in fact. As a result, the process time is brought down to a decreased tendency. By carefully examining the graph, we can also discover that processing time only increased by less than 50% when layer jumps from 4 to 32 exponentially. This also proves creation is far slower than updating from another angle. From this fact reflected from our experiment we can learn that during the aggregation, creating new node takes up most of the running time, updating traffic counter has little effect to overall algorithm processing time.

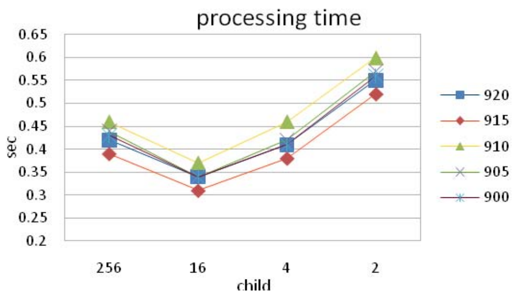

Figure 2 Processing Time among different Aggregation Layers Algorithm

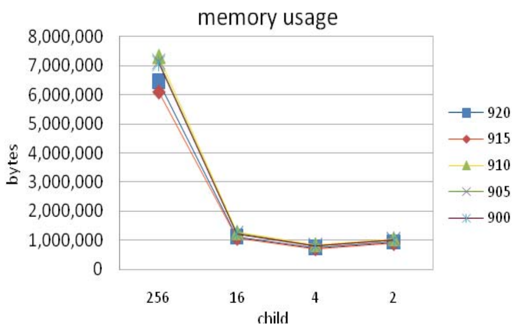

Figure 3 Memory Usage among different Aggregation Layers Algorithm

The memory usage of our direct approach shows general lowering trend as the child count decreases from 256 to 2 in Figure 3. Fewer aggregation layer reduces internal nodes between leaf nodes and root, but the side-effect also leads to more empty field in link list which used to maintain adjacency relationship of parent and child. For instance, only 0.8% of algorithm's committed memory is used to store available data while the rest is totally wasted on empty link when a node has 256 descendents; for the binary case, average memory utilization ratio rises drastically to 55% though the number of internal node is much larger the former condition. The result is listed in Table 2 below concerning memory usage against layer and child combination according to experiments using 5 minutes Netflow data from 09:00 to 09:05 on Jul 15th, 2009. This sample contains 31622 flow records after merge.

TABLE 2: MEMORY USAGE RATIO AMONG DIFFERNET AGGREGATION LAYERS

Number of Aggregation layers	Sub-node Number of each layer	intra-nodes	effective pointers	usage ratio
4	256	27422	59043	0.008411
8	16	69732	101353	0.090842
16	4	154775	186396	0.301076
32	2	325092	356713	0.548634

The prefix length grows by one-bit in Estan's algorithm. In other words, each node has at most two descendents. This layout can improve the memory efficiency. While doing the traffic aggregation, a post-order traversal method is introduced to avoid unnecessary counter update other than updating all counters along its path. As we demonstrated above, access a node and update its traffic counter press little influence on overall processing time. So adopting a post-order traversal strategy to reduce time of update actually does not bring as much benefit as we expected it will. Next, all leaf node has to be created before building the entire hierarchy tree, this restriction will cause two other problem. First, lots of processing time is spent on creating internal nodes, though we introduce post-order traversal to prevent redundant access, the major time and space cost is not well coped with at all. Second, among all nodes in hierarchy tree, only a very small amount of them are high-volume clusters. It is very unnecessary to keep all internal nodes in the memory and take care of them during the algorithm.

According to the two reasons above, this article proposed a top-bottom iterating algorithm for unidimensional traffic aggregation. The general time and space performance can be largely improved by avoiding the creation of all non-high-volume clusters. In section 4, this algorithm will be discussed in detail.

IV. VARIABLE HIERARCHY ALGORITHM

Our algorithm uses a similar hierarchy tree structure to the one in Estan's paper. The root node in the hierarchy has 256 descendents which connects directly to 256 /8 prefixes sub-tree. Each sub-tree is a binary tree. The depth of the hierarchy tree is 26. Unlike Estan's approach, our algorithm splits up total traffic from top to bottom. Only high-volume clusters are left in the hierarchy once the algorithm finishes executing.

Hash table is used to collect all incoming flows. During this stage, replicated flows are merged and the number of individual IP is counted. When we finish the initial work, all flows records in hash table are dumped into a list and sorted using its IP address field as the sorting key. Flows that belong to any given prefix will be stored one after another because the sorting will put successive IP addresses together in order. Note the extra sorting procedure will not increase the total running time of our algorithm. This is because no matter which direction, top-bottom or bottom-top, you decide to build the hierarchy tree, there is always a potential cost for sorting all flows in order hidden in the algorithm. Take Estan's approach as an example, when we create all leaf nodes, sorting is still necessary so that we are able to build the hierarchy upon these scattered individual leaf nodes. In our earlier direct approach though we do not need to sort individual flows in advance, but after the hierarchy is built all leaf nodes are in an ordered status. The cost for sorting are diluted to determine the "walking path from root to leaf" for every arriving flow. The same truth applies to any other offline aggregation method. We can conclude that sorting cost is inevitable regardless which method you choose. Besides, we found sorting merely consumes a small portion of all processing time through experiment.

In addition, we can employ some optimizations to sorting. While we are counting the number of individual IP address, in the meantime we can record the distribution of all flows on each /8 prefix (0.0.0.0/8-255.0.0.0/8). When dumping individual flows from hash table, we can directly put a flow to the segment in Figure 4 (a relative position) according to /8 prefix it belongs. Let $A_i = Count\{p \mid p \in prefix(i)\}$ be the number of individual flow in prefix i.0.0.0/8, $D_i = \sum_{k=0}^{i-1} A_k$ be the accumulated distribution of each /8 prefix. Every incoming flow is inserted sequentially at the position starting from D_i. After the dumping completes, all flow are placed in ordered relatively on /8 prefix level.

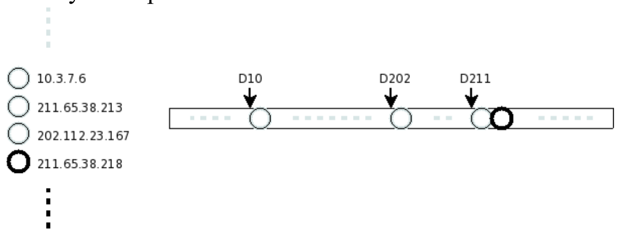

Figure 4 Record a Flow to the flow buffer

Then a quick sort is applied to every /8 prefix segment in the list. The index of flows that belong to prefix i.0.0.0/8 in flow list range from D_i to $D_{i+1}-1$. The number of flows in each /8 prefix is A_i. Let A be the total amount of individual IP, then $A = \sum_{i=0}^{255} A_i$. Because the time complexity of quick sort is $O(N \log N)$, so the overall cost for quick sort on all /8 prefix segments is $\sum_i [A_i \cdot \log A_i] < \sum_i A_i \times \log A = A \log A$.

It is faster than sorting the whole flow list. Furthermore, by analyzing the /8 prefix distribution in raw traffic data captured on the border router of JSERNET, we found less than 2% of all /8 prefixes whose total occurrence is over 50%. For five-minute Netflow data (sampling rate 256), the total occurrence of over 70% /8 prefix is below 1000 on average. Because the efficiency of various sorting algorithms largely depends on the quantity of input data, we can use different sorting methods according to changing input data amount.

Let $T_k^i (1 \le k \le A_i)$ denote the volume of kth flow. The volume $V(p)$ of a given prefix p can be easily computed as $V(p) = T_e^i - T_s^i$ via two random memory accesses and a subtraction. Figure 5 give an example to compute the volume of the 202.112.25.0/24 prefix.

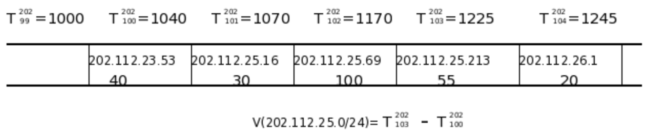

Figure 5 Compute the volume of the 202.112.25.0/24 prefix

Algorithm allocates memory needed for maintaining high-volume clusters all at once before we start. Because the traffic of each node at a certain level in hierarchy tree does not overlap with others, there are at most $\lceil 1/\varphi \rceil$ high-volume clusters for all levels. As the prefix length changes from 8 to 32, the maximum number of high-volume cluster for a given threshold φ is at most $25 \times \lceil 1/\varphi \rceil$. That is far less than the number of nodes contained in an entire hierarchy tree. A 2-d matrix of size $[25, \lceil 1/\varphi \rceil]$ is allocated to maintain all high-volume clusters and an array of 25 elements is also allocated to indicate the last position on each level of high-volume cluster cache. For every element in high-volume cluster cache, we keep its prefix, prefix length, start and end flow index in sorted flow list, its traffic volume and the pointers to its descendent in the next level of cache.

Our algorithm first checks every /8 prefix, writes it into the first level of high-volume cluster cache if its traffic exceeds the threshold. The algorithm then proceeds with high-volume clusters found in the first layer. For each high-volume cluster, our algorithm increases prefix length by 1 bit and attempts to check the traffic volume of its descendents (if any). It iterates through level to level until all high-volume clusters are found. Due to this top-bottom iteration manner, if the traffic volume of a given prefix does not exceed threshold, all of its descendents will not be examined later. The time and memory overhead are greatly reduced compared with an entire hierarchy tree, because the number of high-volume clusters is much smaller. Figure 6 gives the fast aggregation algorithm.

```
1.   layer = 1;
2.   for i = 0 to 255
3.      if(volume(D_i, D_{i+1}-1) >=φ.sum)
4.         add_hv(HH[layer, last[layer]++]);
5.      endif
6.   endfor
7.   for layer = 1 to 25
8.      child = layer + 1;
9.      for current = 0 to last[layer]
10.        pivot = find_pivot(HH[layer, current].s, HH[layer, current].e, child);
11.        if( volume(HH[layer, current].s, pivot)>=φ.sum)
12.           add_hv(HH[child, last[child]++]);
13.        endif
14.        if( volume(pivot+1, HH[layer, current].e) >=φ.sum)
15.           add_hv(HH[child, last[child]++]);
16.        endif
17.     endfor
18.  Endfor
```

Figure 6 Mining Traffic Patterns Algorithm

A fragment of algorithm example is shown below. Suppose the high-volume threshold is 100. 10.8.0.0/28 is a high-volume cluster in the cache, the algorithm is iterating through it and is about to check its descendents. Prefix 10.8.0.0/28 contains 8 individual IP in sorted flow list, whose traffic volume are labeled in Figure 7. When computing the two descendents of 10.8.0.0/28, these 8 individual flows are divided into two /29 prefix and their traffic volumes are checked.

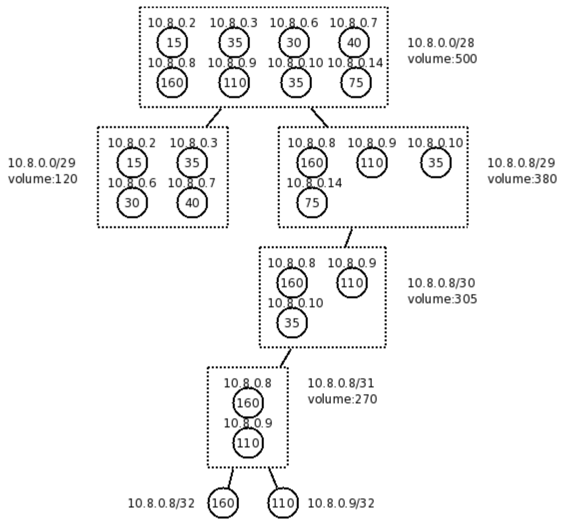

Figure 7 A Prefix 10.8.0.0/28 Example

V. Experiment

In this section we make a comparison concerning the processing time in Figure 8 and memory usage in Figure 9 between our algorithm and Estan's method through experiment using real traffic data captured on JSERNET border. The advantage of our fast unidimensional aggregation algorithm which is attained by avoid creating non-high-volume cluster is demonstrated in the comparison.

The test suit contains 12 five-minute netflow data in V5 format which is export by the router on 8am Jul 15th, 2009 spanning one hour. The total processing time suggests their quantitative relation. The processing time of our fast unidimensional aggregation algorithm consists of sorting and aggregation cost. A large portion part of total processing time is spent on sorting; finding high-volume clusters can be accomplished in a very short time.

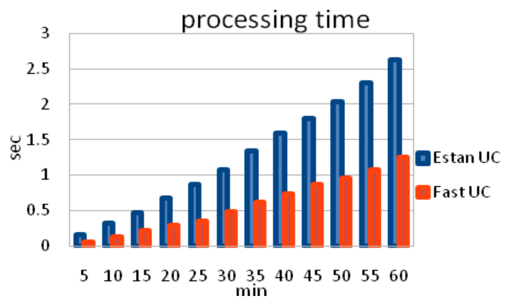

Figure 8 Processing Time Comparison among different Algorithms

The memory usage for high-volume cache in our algorithm is a function of high-volume threshold, so it is of fixed size for any given threshold; the memory that used to store sorted flow is equal to all leaf nodes in Estan's algorithm. All memory used for unnecessary non-high-volume clusters which cover almost all internal nodes are saved.

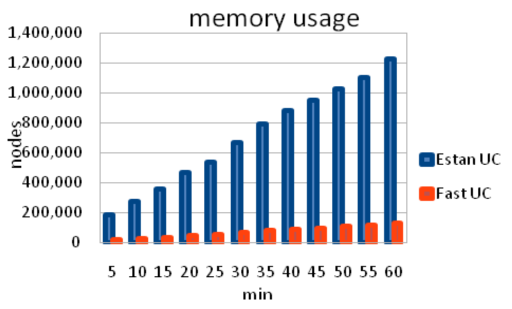

Figure 9 Memory Usage Comparison among different Algorithms

VI. Conclusion

It is a most complicated condition to cluster traffic patterns on IP dimension, so it takes most time to run in traffic clustering problem. By analyzing the time and space performance of a simple direct paradigm of aggregation algorithm we concluded that the major resource consumption is spent on creating all internal nodes, in which only very small portion is actual high-volume cluster. Sorting operation is introduced in our algorithm. The general time and space performance is greatly improved by avoiding the creation of all non-high-volume clusters in a top-down hierarchical algorithm.

References

[1] Cristian Estan, S. Savage, and G. Varghese. Automatically Inferring Patterns of Resource Consumption in Network Traffic. in Proceedings of ACM SIGCOMM, 2003.

[2] Jisheng Wang, David J. Miller, and George Kesidis, Efficient Mining of the Multidimensional Traffic Cluster Hierarchy for Digesting: Visualization, and Anomaly Identification, IEEE J. Selected Areas of Comm., Vol 24, No. 10, pp. 1929-1941, Oct. 2006.

[3] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick Duffield, Carsten Lund, Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Applications, In Internet Measurement Conference, Taormina, Italy, October 2004.

[4] Patrick Truong, Fabrice Guillemin, A Heuristic Method of Finding Heavy Hitter Prefix Pairs in IP Traffic, IEEE Communications Letters, Vol. 13, No. 10, pp. 803-805, Oct. 2009.

[5] Keys K, David M, Estan C, et al. A Robust System for Accurate Real-time Summaries of Internet Traffic, ACM Sigmetrics 2005, June 6-10, 2005, Banff, Alberta, Canada.

[6] CHENG Guang, GONG Jian, DING Wei, WU Hua, Adaptive sampling algorithm for detection of superpoints, Science in China Series F: Information Sciences, vol. 51, no. 11: 1804-1821, 2008.

[7] C Estan, K Keys, D Moore, G Varghese, Building a better NetFlow, ACM Sigcomm 2004, Aug. 2004.