Evolution of Cache Replacement Policies to Track Heavy-hitter Flows

Martin Zadnik[1] and Marco Canini[2]

[1] Brno University of Technology, Czech Republic. izadnik@fit.vutbr.cz
[2] EPFL, Switzerland. marco.canini@epfl.ch

Abstract. Several important network applications cannot easily scale to higher data rates without requiring focusing just on the large traffic flows. Recent works have discussed algorithmic solutions that trade-off accuracy to gain efficiency for filtering and tracking the so-called "heavy-hitters". However, a major limit is that flows must initially go through a filtering process, making it impossible to track state associated with the first few packets of the flow.
In this paper, we propose a different paradigm in tracking the large flows which overcomes this limit. We view the problem as that of managing a small flow cache with a finely tuned replacement policy that strives to avoid evicting the heavy-hitters. Our scheme starts from recorded traffic traces and uses Genetic Algorithms to evolve a replacement policy tailored for supporting seamless, stateful traffic-processing. We evaluate our scheme in terms of missed heavy-hitters: it performs close to the optimal, oracle-based policy, and when compared to other standard policies, it consistently outperforms them, even by a factor of two in most cases.

1 Introduction

Flow-based network traffic processing, that is, processing packets based on some state information associated to the flows to which the packets belong, is a key enabler for a variety of network services and applications. For example, this form of stateful traffic processing is used in modern switches and routers that contain flow tables to implement firewalls, NAT, QoS, and collect statistics.

Flow-based traffic processing faces scaling challenges in that it potentially requires tracking and managing the state of millions of concurrent flows while keeping up with ever increasing data rates. In a number of cases, it is not necessary to track the state of each individual flow. Based on the generally known observation that a small number of flows account for a large amount of network traffic (e.g., see [1]), it has been suggested that scalable traffic measurement and accounting can be done by accurately measuring only the few large flows [2]. This can be generalized to other applications where the application goals can be met well enough by just focusing on the so called "heavy-hitters". For example, a traffic shaping system may focus on rate-limiting the large flows while the low-rate flows can utilize a small share of bandwidth at their will.

In [2], a memory-efficient structure called the Multistage filter has been introduced to define a scalable and efficient algorithm for *identifying* heavy-hitters. The limit of this approach is that a flow will only be accounted for once its traffic volume has passed the filter and until this time no state can be assigned to that flow. As this limit is intrinsic to the filtering approach, the works that have extended the method above (e.g., [3, 4]) have inherited this limit.

However, associating flow state since a flow's first packet is critical for certain applications. For example, classifying traffic based on application identification (e.g., [5]) require statistics or payload data collected from the first few packets of a flow. In addition, network security schemes implement stateful processing for the initial packets of each flow. Further, OpenFlow switches [6] are managed by a controller that acts upon the first packet of each flow and installs flow-specific rules into the switch flow table. Therefore, filtering approaches are not always applicable and other approaches must be utilized despite their higher costs.

In this paper, we treat the problem of *identifying and tracking* heavy-hitters as that of finding a cache replacement policy that strives to avoid evicting the heavy-hitters from the flow table (from now flow cache). The intuition is that, if in the presence of a full cache and a new flow starting (causing a cache miss) the policy only chooses to evict flows that are not heavy-hitters (or unlikely), then the state of heavy-hitters is definitely preserved in the cache since their first packet. Effectively, compared to filtering, we trade-off the absence of false negatives and, partially, memory efficiency to support *tracking with state* the heavy-hitters from their initial packets.

In order to find such a replacement policy we utilize Genetic Algorithms (GA). GA explore the space of possible solutions in search for a solution that exploits characteristics learned from recorded traffic traces and tailor the replacement policy to traffic patterns which could hardly be considered when manually designing a policy. We compare the evolved policies with other standard replacement policies. In our trace-driven evaluation, our scheme performs the best, even by a factor of two in most cases. The results demonstrate that our approach is promising in supporting stateful traffic processing focused on the heavy-hitters.

2 Background

Genetic Algorithms. GA are widely used in various areas of science and engineering to find solutions to optimization and search problems [7]. The main idea is to evolve a set (a population) of candidate solutions to find better replacement policy. A candidate solution is encoded as a genome which is an abstract representation (e.g., a binary string) that can be modified with standard genetic operators such as mutation and crossover. Starting from a population of randomly generated candidate solutions the evolution happens in generations. In each generation, some highly-scored solutions are selected to produce offspring. The offspring are evaluated in terms of their fitness to the problem and form a new generation. The evolution stops once a maximum number of generations has been produced or a satisfactory fitness level has been reached.

In their recent work [8], Kaufmann *et al.* described the usage of GA to minimize data collisions in a CPU cache line by tuning the address mapping in an application-specific way. We regard this work as orthogonal to ours in that they optimize the selection of a cache line to avoid collisions, but maintain the original replacement policy while we are concerned with the optimization of the cache replacement policy within a single cache line.

Cache Replacement Policies. Least Recently Used (LRU) is a widely used replacement policy for managing caches. However, LRU caches are susceptible to the eviction of frequently used items during burst of new items. Many efforts have been made to address the inability to cope with access patterns with weak locality. For example, Segmented LRU (SLRU) [9] seeks to combine both locality and frequency to achieve better hit ratios. An SLRU cache is divided into two segments: a probationary segment and a protected segment. When the cache is full and a miss occurs, the new item is added to the probationary segment and the least recently used item of this segment is removed. If a cache hit corresponds to an item in the probationary segment, the item is moved to the protected segment taking the place of the least recently used item in that segment.

We proposed a minor variation of SLRU for tracking large flows called Single-Step SLRU (S^3-LRU) [10]. Compared to SLRU, S^3-LRU does not order the items within each segment by their last access, but on each cache hit it advances the hit item of a single step toward the front of the cache (protected segment) by swapping its position with that of the adjacent item. However, S^3-LRU is only marginally better than SLRU in certain cases as our evaluation demonstrates.

Molina [11] proposed an algorithm for evicting small flows from the flow table using forecasts of the future flow volume based on the current volume and recent flow growth rate. Statically partitioning the flow cache in several subsets makes the approach efficient for identifying heavy-hitters. However, in this approach heavy-hitters can be evicted before having a chance to significantly increase their growth rate. For example, in our datasets we found that with this strategy 80% of the heavy-hitters witness a cache miss.

Filtering. Estan and Varghese [2] suggested a scalable traffic accounting scheme which focuses upon the identification and monitoring of heavy-hitter flows. In this scheme, only the packets which belong to flows identified as heavy-hitters are recorded by the flow table. The identification algorithm takes advantage of a memory-efficient data structure called Multistage filter. However, the limit of this approach is that a flow will only be accounted for once its volume has passed the identification stage, and no state in the flow table can be assigned to this flow until that time. We consider their approach as complementary to our scheme (also it would not be straight-forward to compare fairly).

3 Datasets

The definition of a flow changes based on the application. One that is commonly used identifies a flow based on the 5-tuple composed of its IP addresses, port numbers and protocol. In our work, a flow is a unidirectional stream of packets

	Mawi				Equinix			
	v.large	large	medium	Total	v.large	large	medium	Total
Flows	0.23%	0.93%	9.43%	4.0M	0.00%	0.02%	0.35%	21.8M
Packets	31.97%	18.92%	20.71%	53.0M	0.36%	10.15%	17.07%	344.1M
Bytes	68.35%	17.13%	9.22%	33.5G	0.85%	24.01%	36.48%	207.7G

Table 1: Mawi dataset. (1 hour, 155 Mbps link, avg/min/max active flows: 67.3K/ 56.5K/250.1K) Equinix dataset. (15 min, 10 Gbps link, avg/min/max active flows: 1.7M/179K/1.8M, only 160 instances of very large flows)

sharing the same 5-tuple, but our approach can be easily generalized to allow the flow identifier to be a function of the header field values. A flow ends based on an inactivity timeout of 60 s or based on the TCP connection tear down.

We define a *heavy-hitter* to be a flow that utilizes more than a certain percentage of a link bandwidth during its lifetime. Also, we only consider a flow as heavy-hitter if it exceeds the threshold utilization for at least 5 s. Therefore, we compute a flow's link utilization as $\frac{bytes}{\max(5, lifetime)}$. This excludes short-lived flows with intensive bursts of packets that do not carry a significant amount of traffic overall. Lowering the 5 s interval significantly increases the number of heavy-hitters which potentially causes the GA to focus on short-lived flows at the expenses of long-lived heavy-hitters, although the traffic volume due to these short-lived flows is just a small fraction (e.g., with 1 s interval the number of heavy-hitters increases by 50% while the number of additional bytes due to heavy-hitters increases by less than 1%).

We group flows into three reference categories based on their link utilization: very large flows ($> 0.1\%$ of the link capacity), large flows (between 0.1% and 0.01%), medium flows (between 0.01% and 0.001%). We then report how well our approach performs for each category.

We use two traces of Internet backbone traffic: a 1-hour trace from the Mawi archive collected at the 155 Mbps WIDE backbone link (samplepoint-F on March 20th 2008 at 14:00)[3], and an anonymized, unidirectional 15-min trace from the Caida archive collected at the 10 Gbps Equinix San Jose link (dirA on July 17th 2008 at 13:00 UTC) [12]. Table 1 summarizes the working dimensions of our traces and shows a breakdown of the composition of the three flow categories.

4 Approach

Definitions. We regard the flow cache of size N as a list of up to N flow states (or simply flows) F. This allows us to treat the cache management problem as keeping the list of flows ordered by their probability of being evicted (highest goes last). Then, the role of a replacement policy (RP) is to reorder flow states based on their access pattern. Each packet causes one cache access and one execution of the RP. If the current packet causes a cache miss (i.e., a new flow arrives) and the cache is full, the flow at the end of the list is evicted.

[3] `http://mawi.wide.ad.jp/mawi`

Formally, we can express a RP that is based solely on the access pattern as a pair $\langle s, U \rangle$ where s is a scalar representing the zero-based position for inserting new flow states and U is a vector $(u_1, u_2, \ldots, u_N)$ which defines how the flows are reordered. Specifically, when a flow F stored at position $pos_t(F)$ is accessed at time t, its new position is chosen as $pos_{t+1}(F) = u_{pos_t(F)}$, while all flows stored in between $pos_{t+1}(F)$ and $pos_t(F)$ see their position increased by one. For example, the LRU policy for a cache of size 4 is expressed with $LRU = \langle 0, (0, 0, 0, 0) \rangle$.

Evolution of Replacement Policies. Our goal is to find a RP that has the least number of evicted heavy-hitters or, using caching terminology, minimizes the miss rate for heavy-hitters. We use the number of heavy-hitters that witness a cache miss as a metric to capture the effectiveness of a RP—the objective is to reduce this number. Finding such a RP is difficult due to a large number of factors including flow size distribution, flow rate, and other traffic dynamics. We propose using GA to explore the space of possible RPs to identify the most effective. We chose GA for its ability to infer useful discriminators from traffic characteristics and to be easily customized to accommodate changes in the problem specification, e.g., different flow definitions, different traffic subpopulations of interest [4], etc.

The vector-based definition of a RP is a good fit to encode the candidate solution. It supports the standard genetic operators for mutation and crossover. Mutation modifies a particular value in the vector with given probability p_{mut} while crossover swaps parts of the vector between two solutions with probability p_{cross}. The RP evolution is performed offline using network traces. The following pseudo-code illustrates the evolution process:

```
population = GenerateRandomPopulation();
Evaluate(population); best = SelectBestIndividual(population)
while (not endcondition):
   newpopulation = SelectNewPopulation(population + best, fitness);
   CrossoverIndividuals(newpopulation, p_cross);
   MutateIndividuals(newpopulation, p_mut);
   FixInviableIndividuals(newpopulation);
   Evaluate(newpopulation);
   best = SelectBestIndividual(newpopulation + best);
   population = newpopulation;
result = best;
```

We start with a population of $C = 5$ candidates generated at random. The population size is a trade-off between evolution progress and population diversity. A large population means having a long time between replacements of generations due to lengthy evaluation of all candidate solutions. On the other hand, a small population cannot afford preserving currently low-scored solutions which could become good solutions. We use a relatively small population so the evolution process can progress faster allowing the RP to be adapted to ongoing traffic. We will study adaptation mechanisms in future work. During each step of evolution, 5 candidates are selected using tournament selection from a parent population and the best individual so far. Then, crossover and, subsequently, mutation operators are applied and the resulting offspring are evaluated with

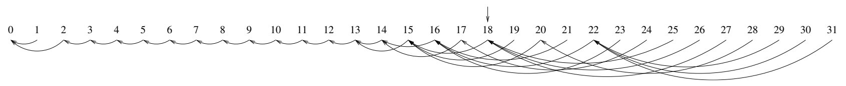

Fig. 1: An example of a RP produced by GA using the Mawi dataset. The arrows represent where to move a flow state when it is accessed. $RP = \langle 18, (0,0,0,2,3,4,5,6,7,8,9,10,11,12,13,13,14,14,15,15,15,16,16,16,17,18,18,18,20,22,22,22)\rangle$.

a fitness function. The fitness function is the sum of cache misses for the flows in the three reference groups weighted by the link utilization thresholds: 0.1% for the first group, 0.01% for the second and 0.001% for the third[4]. Effectively, the fitness function simulates the cache behavior with a candidate RP. To lower the evolution time, we evaluate the fitness using only a small part of a traffic trace, namely 5 min for Mawi and 1 min for Equinix. This has negligible impact on the results because we use a small cache size which becomes full within few seconds of simulation time. In each generation, the candidates are replaced by the offspring and the best candidate so far is preserved (so called elitism).

Search Optimizations. Without imposing any constraint on the vector-based definition of RP, we allow undesired candidate solutions: those that (*i*) do not utilize the entire list due to unreachable positions in the update vector U, or (*ii*) worsen the position of a flow despite it being accessed. Excluding these solutions reduces the search space, which helps GA to perform better and faster. Using simple heuristics we ensure the reachability of all positions and that any access improves the position of a flow. In our experiments, we observe that the GA converges to a promising solution faster if we split the run of GA into two consecutive phases, each with a different setup. The first phase is intended to search through the space to quickly find various viable solutions. Therefore, mutation changes the values in the vector to new, randomly-generated values. While experimenting with GA, we found that $p_{mut} = 0.3$ works well during this phase but close values work well too. The probability of one-point crossover is set to $p_{cross} = 0.3$ which allows to exchange information (parts of vector) between the selected parents. If there is no significant fitness improvement in the population, we enter the second phase which focuses on optimization. The crossover operator is no longer utilized as the possible solutions either differ significantly (crossover would produce a hybrid that would quickly be discarded) or are very similar. We modify the mutation operator to increase/decrease each vector value by one with probability $p_{mut} = 0.5$. In total, we produce 50 generations and we make 10 independent runs of GA, from which we select the best solution. Figure 1 presents an example of a GA-produced RP while Figure 2 shows the fitness evolution in a typical run of GA.

Discussion. So far we have considered a flow cache as an ordered list of flow states. In such a simplistic model, the complexity of our scheme is $O(n)$ in the case of a hit and $O(2 \cdot n)$ in the case of a miss where n is the cache size. However, the applications we target typically already have a certain hardware

[4] This assigns higher importance to track true heavy-hitters.

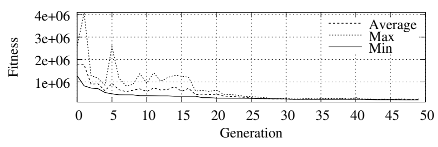

Fig. 2: A typical run of GA. "Min" represents the best solution.

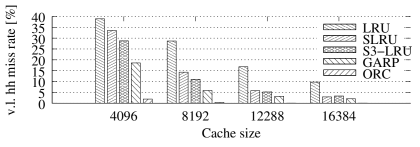

Fig. 3: Missed very large heavy-hitters vs. cache size (Mawi trace).

support for stateful traffic processing at wire speed. Our scheme is meant to be implemented in hardware and integrated with the existing support for stateful processing. A practical hardware implementation usually divides the cache into a number of equally-sized lines which are managed independently. Each line is able to accommodate multiple flow states and the lookup of a flow state is performed in parallel by a set of comparators. Thus, the scheme runs with complexity O(1). An hash value of the flow identifier is used to address a line in the cache. In such a basic scheme called Naïve Hash Table (NHT), each line in the cache executes the same RP. We evaluate our approach with these realistic settings. Our previous work in [10] shows that a line size with up to 64 items can be implemented in an FPGA (Field Programmable Gate Array). We choose to evolve RPs for a line size of 32 items as we consider it a good trade-off between what can be easily implemented in hardware and accuracy performance.

Finally, as our scheme operates online, the question arises how to maintain low false negatives (cache misses for heavy-hitters) despite changing traffic conditions. A plausible solution is to run our scheme in parallel with a Multistage filter to estimate the number of flows evicted from the cache which are identified as heavy-hitters by the filter. If this count exceeds a given threshold, we could trigger the creation of a new RP based on recently recorded traffic traces. Due to space limitations, we do not further discuss a complete solution and leave a thorough study for future work.

5 Evaluation

In this section, we present the results of our evaluation with a software implementation of the flow cache which allows us to easily report on the cache misses.

We compare the performance of a genetically evolved RP (referred to as GARP) with that of LRU, SLRU [9], S^3-LRU [10] and the best possible policy, which is based on an oracle (ORC). ORC uses the knowledge about active heavy-hitters and their remaining duration to evict a flow that is not a heavy-hitter if possible, otherwise the heavy-hitter that will end soonest. For SLRU and S^3-LRU, we select the insert position that performs the best across our datasets. The values are 21 and 7 for SLRU and S^3-LRU, respectively.

We experiment with cache size of 64K, 96K, 128K, 160K flow states for Equinix, and 4K, 8K, 12K and 16K flow states for Mawi. We use line size of 32

RP	Mawi			Equinix		
	v.large	large	medium	v.large	large	medium
LRU	19%	30%	10%	5%	25%	28%
SLRU	9%	19%	7%	3%	19%	18%
S^3-LRU	7%	21%	11%	2%	16%	22%
GARP	3%	8%	6%	2%	9%	10%
ORC	0.3%	1%	9%	0%	0%	0%
GARP-Eq.	5%	9%	8%			
GARP-Ma.				2%	11%	12%

Table 2: Comparison of cache misses for heavy-hitters between GARP and other RPs. Cross-evaluation of GARP trained on different datasets. (Mawi – cache size: 8K, line size: 32; Equinix – cache size: 128K, line size: 32)

states. In our experiments, the flow cache is approximately one order of magnitude smaller than the number of concurrently active flows.

Table 2 presents the number of heavy-hitters that experience a cache miss, normalized by the total number of heavy-hitters in each category obtained on the Mawi and Equinix datasets. In the case of Equinix dataset, the cache size is large enough to accommodate all heavy-hitters and so ORC does not cause any cache miss. However, using smaller cache sizes quickly deteriorates the cache misses for any real RP because of the large number of non heavy-hitters (99%) present in the Equinix dataset.

The results show that GARP consistently outperforms LRU and in most instances performs at least two times better than the other RPs which already have an ability to cope with access patterns with weak locality. Most of the heavy-hitters witness just one cache miss. We note that when experimenting with larger caches (see Figure 3) the difference in performance between policies decreases as the cache itself can store a significant share of all concurrent flows. However, it is often prohibitively expensive to have a large cache.

We perform a cross-evaluation to assess whether a GARP produced for one network link is applicable to another or whether the performance are unsatisfactory due to GA over-fitting for a particular training dataset. Second part of Table 2 demonstrates that the difference between two GARPs evolved on different datasets is quite modest. The suffixes *-Equinix* and *-Mawi* indicate the dataset the GARP was evolved on. These results indicate that our approach is promising and might find more general applicability, e.g., with different definitions of flows of interest [4]. Finally, to gain insight on the temporal stability of GARP, we test the performance of RPs evolved on our datasets and applied to traffic traces collected one year later than the training datasets (at the same links). We find that the performance does not significantly decay (on average less than 1.5% for Mawi and 2% for Equinix). Moreover, we evolve RPs on these newer traces and we find that, for both Mawi and Equinix, the newly obtained GARP is very similar to the GARP produced from the corresponding older dataset: quantitatively, the differences between the RPs' update vectors expressed as mean squared error are 0.42 for Mawi and 0.57 for Equinix.

6 Replacement Policy Extension

We now extend the RP with the ability to exploit information from the header fields of the packet that causes a cache hit. We consider two fields: packet size and TCP flags – chosen based on the analysis (omitted for a lack of space) of the statistical characteristics across the flow groups of the field values in our datasets. One intriguing approach would be to replace the update vector U with a matrix in which each row corresponds to a particular update vector for a given set of input field values. For example, the first row could be the update vector corresponding to the packet size 0 and the FIN TCP flag while the second row could be for packet size 1, etc. However, this quickly brings to the well-known problem of search space explosion.

We avoid this problem by maintaining a single update vector, but we complement the selection of the new position with a decision tree that uses the field values to increase by one, decrease by one, or maintain the position selected by the update vector. Based on our experiments, increases/decreases of two or more give worse results.

We only consider TCP flags (FIN, RST to decrease) and packet sizes of these ranges: [0 - 359] to decrease, [360 - 1000] to maintain and [1001 - max. size] to increase the update position. We determined these values from the analysis of the difference in the distribution of packet sizes between heavy-hitters and other flows in our datasets.

We tried using the current flow size (packets or bytes) as another parameter of the decision tree, and found that it does not bring further improvements. This is not entirely unexpected because the flow state's current position is determined by the history of all cache accesses, therefore the information from the current number of packets is implicitly already used.

As this extension is agnostic of the specific way in which the update vector is determined, we can apply it to all the considered RPs. Table 3 presents the number of cache misses normalized as before obtained on the Mawi and Equinix datasets with a cache size of 8K and 128K flow states, respectively. Each cache line stores 32 flow states. The extension works well only for the policies that do not progress the flow state right to the first position in the line. The GARP still achieves the best performance but it sees a smaller improvement than S^3-LRU. This is because the GARP itself has already been optimized to the observed traffic patterns (e.g., network scans), and applying a decision tree provides only a little additional information. We leave it as future work to evolve the replacement policy together with the decision tree.

7 Conclusions

We proposed a paradigm shift for scalable traffic processing focused on the large flows, regarded as the problem of managing a small flow cache. By design, our scheme allows to identify and track the heavy-hitters since their first packets. We demonstrated that Genetic Algorithms can evolve cache replacement policies

Extended RP	Mawi			Equinix		
	v.large	large	medium	v.large	large	medium
LRU	19%	29%	10%	5%	24%	28%
SLRU	9%	18%	7%	3%	18%	17%
S^3-LRU	4%	16%	17%	0%	11%	14%
GARP	3%	7%	7%	0%	8%	9%
ORC	0.3%	1%	9%	0%	0%	0%

Table 3: Comparison of cache misses for heavy-hitters between extended GARP and other extended RPs. (Mawi – cache: 8K, line size: 32; Equi. – cache: 128K, line: 32)

that obtain results close to optimal while consistently outperforming standard policies. Finally, we believe that our approach can find more general applicability in other network-based applications where performance critically depends upon cache performance such as route caching.

Acknowledgements

This work was partially supported by the BUT FIT grant FIT-10-S-1 and the research plan MSM0021630528. We thank the anonymous reviewers and our shepherd, Alan Mislove, for their many helpful comments and suggestions.

References

[1] Feldmann, A., et al.: Deriving traffic demands for operational ip networks: methodology and experience. IEEE/ACM Trans. Netw. **9**(3) (2001) 265–280
[2] Estan, C., Varghese, G.: New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. Trans. Comp. Syst. **21**(3) (2003)
[3] Bu, T., Chen, A., Lee, P.P.C.: A Fast and Compact Method for Unveiling Significant Patterns in High Speed Networks. In: Proceedings of INFOCOM'07. (2007)
[4] Ramachandran, A., Seetharaman, S., Feamster, N., Vazirani, V.: Fast Monitoring of Traffic Subpopulations. In: IMC '08. (2008)
[5] Canini, M., Li, W., Zadnik, M., Moore, A.: Experience with High-Speed Automated Application-Identification for Network-Management. In: ANCS '09. (2009)
[6] McKeown, N., et al.: Openflow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. **38**(2) (2008)
[7] Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1989)
[8] Kaufmann, P., Plessl, C., Platzner, M.: EvoCaches: Application-specific Adaptation of Cache Mappings. In: NASA/ESA Conference on AHS. (2009) 11–18
[9] Karedla, R., Love, J.S., Wherry, B.G.: Caching strategies to improve disk system performance. Computer **27**(3) (1994) 38–46
[10] Zadnik, M., Canini, M., Moore, A., Miller, D., Li, W.: Tracking elephant flows in internet backbone traffic with an fpga-based cache. In: FPL'09. (2009) 640–644
[11] Molina, M.: A Scalable and Efficient Methodology for Flow Monitoring in the Internet. In: Proceedings of the 18th ITC-18. (2003)
[12] Shannon, C., et al.: The caida anonymized 2008 internet traces (2008) `http://www.caida.org/data/passive/passive_2008_dataset.xml`.