Optimization of Network Flow Monitoring

Martin Žádník[*]
Faculty of Information Technologies
Brno University of Technology
Božetěchova 2, 612 00 Brno, Czech Republic
izadnik@fit.vutbr.cz

Abstract

A flow cache is a fundamental building block for flow-based traffic processing. Its efficiency is critical for the overall performance of a number of networked devices and systems. The efficiency is mainly dependent on a utilized replacement policy. This work proposes an approach based on Genetic Algorithm. The proposed approach starts from recorded traffic traces and uses Genetic Algorithm to evolve innovative replacement policies tailored for the flow cache management in particular deployments. An extension of the replacement policy is proposed to improve the already optimized policy even further. The extension is based on an evolution of a replacement policy and a classifier of packet-header fields. The results show a consistent decrease in an eviction ratio in case of two considered problems – reduction of overall number of evictions and reduction of eviction in case of heavy-hitting flows.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques, Performance attribute; D.2.8 [**Design Styles**]: Cache memories

Keywords

Replacement policy, flow cache, Genetic Algorithm, heavy-hitter

1. Introduction

Flow-based network traffic processing, that is, processing packets based on some state information associated to the flows which the packets belong to, is a key enabler for a variety of network services and applications. This form of stateful traffic processing is used in modern switches and routers that contain flow tables to implement forwarding, firewalls, NAT, QoS, and collect measurements. The number of simultaneous flows increases with the growing number of new services and applications. It has become a challenge to keep a state per each flow in a network device processing high speed traffic. Therefore, a flow table (i.e., a structure with flow states) must be stored in a memory hierarchy. The memory closest to the processing is known as a flow cache. Flow cache management plays an important role in terms of its effective utilization, which affects the performance of the whole system.

Realistic policies (without a knowledge of the future) are based on heuristics. The heuristics aim at achieving results of the optimal policy (with the knowledge of the future access pattern) by estimating the future access based on the past access pattern. A large number of the heuristics was proposed in various areas (processor architecture, web objects, databases). But network traffic exhibits specific characteristics, moreover, each network and link may differ in its characteristics. Also each network application may install specific requirements on stateful traffic processing. These specifics and requirements may render standard replacement policies less effective.

To this end, this paper proposes an automated design of cache replacement policy optimized to a deployment on particular networks. Genetic Algorithm is proposed to automate the design and the optimization process. Genetic Algorithm generates and evaluates evolved replacement policies by a simulation on the traffic traces until it finds a fitting replacement policy. The proposed algorithm is evaluated by designing replacement policies for two variations of the cache management problem. The first variation is an evolution of the replacement policy with an overall low number of state evictions from the flow cache. The second variation represents an evolution of the replacement policy with a low number of evictions belonging to heavy-hitting flows only. Optimized replacement policies for both variations are found while experimenting with various encoding of the replacement policy and genetic operators. An extension of the replacement policy is also proposed. The extension complements the replacement policy with an additional information extracted from packet headers.

The paper is divided into the following sections: Section 2 summarizes related work, utilized data sets and the flow cache management problem, Section 3 proposes Genetic Algorithm as a mean to design optimized replacement policy. Replacement policy is extended with classifier in Section 4. The experiments are presented in Section 5 and the comparison with other policies in Section 6. Section 7 concludes the work and suggests future research.

[*]Recommended by thesis supervisor: Prof. Lukáš Sekanina. Defended at Faculty of Information Technologies, Brno University of Technology on May 23, 2013.

Žádník, M.: Optimization of Network Flow Monitoring. Information Sciences and Technologies Bulletin of the ACM Slovakia, Vol. 5, No. 2 (2013) 11-16

2. Background

2.1 IP flows

A flow F is a set of packets sharing a common identifier [3]. A commonly used identifier is a 5-tuple composed of IP addresses, port numbers and protocol. A lifetime of a flow is labeled as $|F|_t$, the number of packets in a flow is labeled as $|F|_p$ and the number of bytes in a flow is labeled as $|F|_b$.

We define a heavy-hitting flow (heavy-hitter) as the flow that utilizes more than a certain percentage of the link bandwidth. In order to avoid bias to short-lived flows which overall do not carry significant amount of traffic, we require the heavy-hitter to exist for at least five seconds (based on experiments). Therefore, we compute a flow's link utilization as $|F|_b/max(5, |F|_t)$.

Throughout this paper, we group flows into three reference heavy-hitting categories based on their link utilization: L_1 flows ($> 0.1\%$ of the link capacity), L_2 (between 0.1% and 0.01%), L_3 (between 0.01% and 0.001%) and the rest L_4.

2.2 Replacement policies

Replacement policies may be divided into several categories: recency, frequency and adaptive. To the best of our knowledge none of the following policies except LRU (Least Recently Used) has been tested for flow cache management. LRU is a well-known example of recency based replacement policy that is widely used for managing virtual memory, file buffer caches, and data buffers in database systems. However, LRU caches are susceptible to the eviction of frequently used items during a burst of new items. Many efforts have been made to address the inability to cope with access patterns with weak locality. For example, SLRU (Segmented LRU) keeps the most utilized items in a protected segment which stays intact during a burst of new access. Another example is LRU-2 [7] which considers last two accesses rather than the last access only.

LIRS (Low Inter-reference Recency Set) [4] policy manages heavily utilized and weakly utilized items in two separate LRU stacks. Authors of LIRS suggest to allocate 99% of cache size to the heavily utilized items (we use this setup in our experiments).

LFU (Least Frequently Used) is a well-known example of frequency based replacement policy. But LFU*-aging [8] is often used rather than LFU. LFU*-aging overcomes shortcomings of LFU. It prevents prioritization of very popular items by limiting the access count. LFU*-aging also periodically decreases the access count. This leads to aging of very popular items which are no longer popular. Lastly, it does not insert a new item in the cache if there are items with more than one access only.

The adaptive policies [5, 2, 6] recognize deviance from a normal patterns such as linear or cyclic access patterns. If a specific pattern is recognized the adaptive policy may change a strategy, for example to switch from LRU to MRU (Most Recently Used) [6]. The characteristics of network traffic renders it very hard to utilize these adaptive approaches since there are no truly obvious patterns except large attacks which are hard to predict or detect promptly.

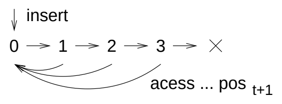

Figure 1: Graphical representation of LRU replacement policy. $LRU = (0, (0, 0, 0, 0))$.

Table 1: Traffic decomposition.

Mawi-14:00	L_1	L_2	L_3	Total
Flows	0,23%	0,93%	9,43%	10 mil.
Packets	31,97%	18,92%	20,71%	44 mil.
Bytes	68,35%	17,13%	9,22%	32 mil.
Snjc-13:00	L_1	L_2	L_3	Total
Flows	0,00%	0,02%	0,35%	8 mil.
Packets	0,36%	10,15%	17,07%	137 mil.
Bytes	0,85%	24,01%	36,48%	83 mil.
Vut-15:00	L_1	L_2	L_3	Total
Flows	0,10%	0,49%	3,2%	2,5 mil.
Packets	55,64%	15,22%	13,38%	58 mil.
Bytes	76,38%	14,74%	5,75%	47 mil.

2.3 Flow cache and replacement policy

In our work, the flow cache is regarded as a list of up to N flow states. 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 the replacement policy is to reorder flow states based on their access pattern. Each packet causes one cache access and one execution of the policy. If the current packet causes a cache miss (i.e., a new flow arrives) and the cache is full then the flow at the end of the list is evicted.

Formally, we can express a RP that is based on the access pattern as a pair (s, U) where s is a scalar representing an insert position in the list where new flow states are inserted 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. As an example, in Figure 1 we provide a graphical representation of the LRU policy for a cache of size 4, which in our formulation is expressed as $LRU = (0, (0, 0, 0, 0))$.

In a theoretical situation, the replacement policy keeps all items globally ordered by their probability of being evicted. But a practical implementations usually divide the cache into equal-sized lists (of size R) which are managed independently. Such a scheme may be viewed as a semi-associative cache or Naive Hash Table (NHT) if the concept is based on using a hash to address a list. The task of the replacement policy is to order flow states in each list independently on others.

2.4 Data set

We use three traces of Internet traffic: a 15-min trace from the Mawi archive collected at the 155 Mbps WIDE backbone link (samplepoint-F on March 4th 2010 at 14:00) [1], and an anonymized, unidirectional 5-min trace from the Caida archive collected at the 10 Gbps Equinix San Jose link (dirA on July 7th 2009 at 13:00 UTC) [10] and VUT 15-min bidirectional campus trace collected on October 18th 2011 at 15:00.

Table 1 summarizes the working dimensions of our traces and show a breakdown of the composition of the three flow categories.

3. Design by Genetic Algorithm

Finding a highly optimized RP for a particular network application can be difficult. The design must consider available cache size, flow size distribution, flow rate, and other traffic dynamics. We propose using GA to explore the space of possible RPs to evolve the most effective.

3.1 Fitness

A fitness function expresses a quality of the candidate policies with respect to a given application, cache size and traffic characteristics. Throughout this work we consider two cache management problems – to achieve low number of total evictions and to achieve low number of evictions of states belonging to heavy-hitters. The fitness function representing the first problem is expressed as:

$$\sum_{F\in\{F_0,\dots,F_M\}} \nu_F(S) + M, \tag{1}$$

where M is the number of flows and $\nu_F(S)$ is the number of cache misses of flow F under policy S. GA tries to minimize the expression (1), i.e., the number of evictions.

In case of lowering the number of evicted flow states belonging to heavy-hitters, the fitness function captures differentiation by the heavy-hitter categories as:

$$\begin{aligned} 100\cdot(\sum_{F\in L_1} \nu_F(S) + |L_1|)+ \\ +10\cdot(\sum_{F\in L_2} \nu_F(S) + |L_2|)+ \\ +\sum_{F\in L_3} \nu_F(S) + |L_3|, \end{aligned} \tag{2}$$

where the number of evictions is weighted by the link utilization in each category. This assigns higher importance to track the very large flows over the large flows which themselves have higher importance than the medium flows.

3.2 Representation

We propose the vector-based definition to encode the representation of the candidate solution. A search space represented by such encoding contains R^{R+1} candidate solutions. We observe that without imposing any constraint on the vector-based definition of RP, we allow candidate solutions that perform very poorly or are very similar in their structure. We introduce optimization to avoid generating bad policies and policies with similar structure.

The optimization Opt_1 focuses on candidate policies that do not utilize full length of the list due to unreachable positions in the vector U. The optimization restricts a value of each item u_i in the vector U:

$$u_0 = 0, u_i = (0,\dots,i-1), i = 1,\dots,R-1. \tag{3}$$

The optimization allows for policies advancng a flow state to a head of the list. Since a packet-arrival means an access to the flow state such optimization is legitimate and does not remove promising candidate solutions from the search space of size $R(R-2)!$.

The optimization Opt_2 focuses on merging similar candidate policies into a single policy. The optimization is based on a modification of the vector U encoding. A new encoding is represented by a vector of triples $Q = ((c_0, w_0, q_0), \dots, (c_{Z-1}, w_{Z-1}, q_{Z-1}))$ where c_j defines the number of items in a sequence, each sequence starts with the value w_j and $q_j = \{0, 1\}$ is incremented to the preceding value in the sequence, $j = 0,\dots,Z-1$. Z is the number of the sequences. For example, vector $U = (0, 0, 0, 1, 2, 3)$ may be represented as $Q = ((2, 0, 0), (4, 0, 1))$ – the first sequence contains two items with initial value 0 and increment 0, the second sequence contains 4 items with starting value zero and increment one. At the same time it holds that $\sum_{j=0,\dots,Z-1} c_j = R$ and for each $c_j : 0 < c_j \leq 8$. The first optimization is enforced by constraining values w_j to ranges $0,\dots,\sum_{k=1,\dots,j-1} c_k$ for $j > 0$.

The vector Q may be transformed to vector U by the following algorithm. For $i = 0,\dots,R-1$:

$$u_i = w_0 + i * q_0 \text{ for } 0 \leq i < c_1, \tag{4}$$

$$\begin{aligned} u_i = w_j + (i - \sum_{k=0,\dots,j-1} c_k) * q_j \\ \text{for } \sum_{k=0,\dots,j-1} c_k \leq i < \sum_{k=0,\dots,j} c_k. \end{aligned} \tag{5}$$

This leads to a reduction of the search space by several orders of magnitude. For line length $R = 32$ and $Z = 8$ the upper estimate is $3.3 \cdot 10^{20}$. The candidate solution (s, Q) is decoded by (4), (5) into (s, U) and evaluated by the simulation of the replacement policy.

3.3 Genetic operators

Three types of mutation operators are defined for the representation (s, U). The operator O_{mut} alters the chromosome values to a random value with probability p_{mut}. The operator O_{w-mut} alters the chromosome values to a random value with an increasing probability $p_{mut}(u)$ according to its index i. The operator O_{mut-1} increases or decreases the value by a random increment from a set $\{-1,\dots,1\}$ with an increasing probability $p_{mut}(u)$ according to its index i. In order to satisfy Opt_1 the range for items in vector U is constrained to $0,\dots,i-1$ for all operators.

For the representation (s, Q) the only defined operator is O_{mut} implemented with constraints of Opt_2.

A one-point crossover and two-point crossover have been tested on both representations. The effects of crossover in comparison to mutations are negligible and are not discussed further.

4. Replacement Policy Extension

So far, only the access pattern (via a position in the list) was utilized to perform replacement decisions. Specifically, a cache hit would always cause the searched flows state to advance its position toward the head of the list. This concept is extended with an ability to exploit information from header fields of the packets that cause a cache hit. First, a set of packet fields that partially predict arrival of the next packet or the heavy-hitter must be identified.

The analysis of relevant fields from a network and transport layer are performed. Based on this analysis following fields seems to be the most relevant for prediction:

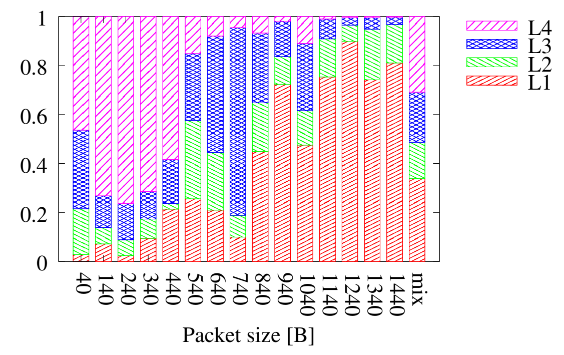

Figure 2: Normalized histograms of packet sizes with respect to the heavy-hitter category.

IP protocol, TCP Flags, TCP Window and Total Length. Figure 2 shows normalized histograms of flow sizes for several packet lengths. It is evident that certain packet sizes (larger than 1000 B) are more likely to appear when the packet belongs to the heavy-hitter.

Only a combination of these fields may achieve a reasonably precise prediction.The combination is done by a decision tree which is trained on a trace annotated by flow size or next packet arrival.

The decision tree should ideally complement the access-based replacement policy. To this end, the replacement policy is evolved together with the decision tree. It is up to GA how the result of classification is utilized. The extended replacement policy is defined as $S_{ext} = (s, U, A)$ (respectively (s, Q, A)), where A is a vector of four values. Each value corresponds to a length of the next packet arrival or to the flow category. The position of flow state after an update is computed as:

$$pos_{t+1}(F) = u_{pos_t(F)} + a_k, \qquad (6)$$

where k is the result of classifier that is an index of a particular category.

5. Experiments

In our approach, we start with a population of C = 6 candidate solutions 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 replacement 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 potentially adapted to ongoing traffic. During each step of evolution, 10 parents out of 5 candidates are selected using tournament selection to produce offspring. Then, mutation operators are applied and the resulting offspring is evaluated by the fitness function. In each generation, the candidates are replaced by the offspring except if the current generation contains the best candidate which is preserved (so called elitism). The experiments are done for various cache sizes. Throughout the experiments we use the flow cache of 8 K states simulated on *Mawi-2010/04/14-14:00* trace.

Many experiments with various genetic operators and representations have been done. Only the most important

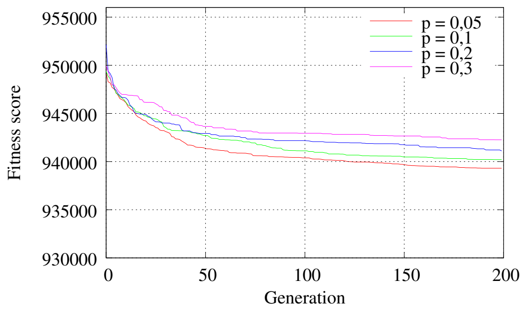

Figure 3: Evolution for Opt_1 and O_{mut} with $p_{mut}(u) = 0.05, 0.1, 0.2, 0.3$ (average from ten runs).

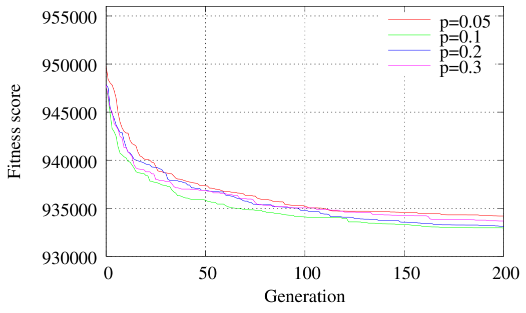

Figure 4: Evolution for Opt_1 and O_{mut} with $p_{mut} = 0.05$, $0.1, 0.2, 0.3$ (an average from ten runs).

ones are presented. Figures 3 and 4 demonstrate the difference between representations Opt_1 and Opt_2. The figures depict evolution of a replacement policy reducing the total number of evictions. The graph plots the fitness score of the best candidate in each generation averaged over ten runs. Although representation Opt_2 is more constraining in terms of the structure of the update vector better solutions are evolved. This is due to a significant reduction of the search space to the promising solutions. The best replacement policies are found using Opt_2 representation and O_{mut} with p_{mut}= 0.2. The best evolved replacement policy lowering the total number of evictions is called GARP (Genetic Algorithm Replacement Policy). The policy is depicted on Figure 5. The best evolved replacement policy lowering the number of evictions for heavy-hitters is called GARP-Large (GARP-Large). The policy is depicted on Figure 6.

Figure 7 depicts the capability of GA to evolve replacement policy together with a classifier. Minimum, maximum and average fitness values are averaged over ten runs. During the first 50 generations GA finds a sufficiently good candidate (see the minimum) but still new and possibly bad candidates are generated (see the maximum). For the rest of the evolution GA is focused on optimizing candidates from a local set of solutions. Based on the observation of the classification vector A, GA decides to employ classification results to identify infinite and long intervals between subsequent packets of the same flow and to identify L_1 and L_4 flows in case of the heavy-hitters.

6. Results

We have compared the performance of GARP and GARP-Large with that of LRU, LRU-2, SLRU, LIRS, EXP1, LFU*-aging, S^3-LRU-7 [9] and the optimal policy OPT. OPT uses the knowledge about future packet arrivals or knowledge about heavy-hitters and their remaining du-

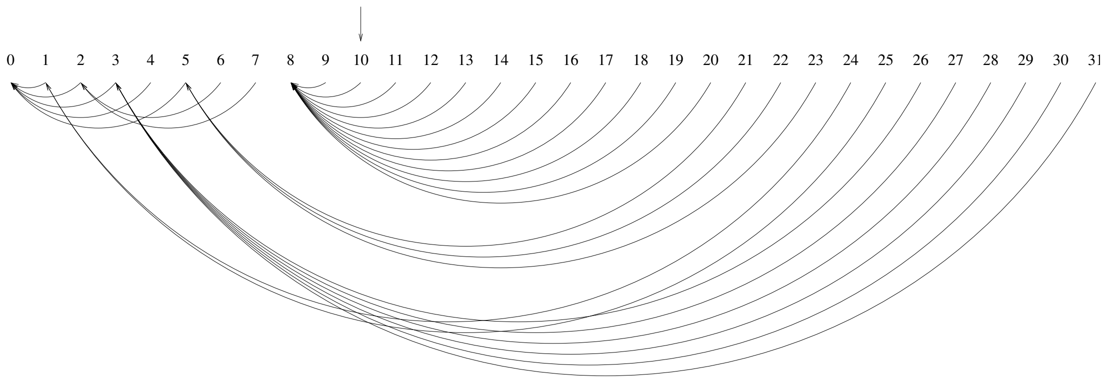

Figure 5: Evolved replacement policy for the flow cache reducing the total number of evictions; GARP = (10, (0, 0, 0, 0, 0, 0, 2, 2, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 5, 5, 5, 1, 1, 3, 3, 3, 3, 3, 3)).

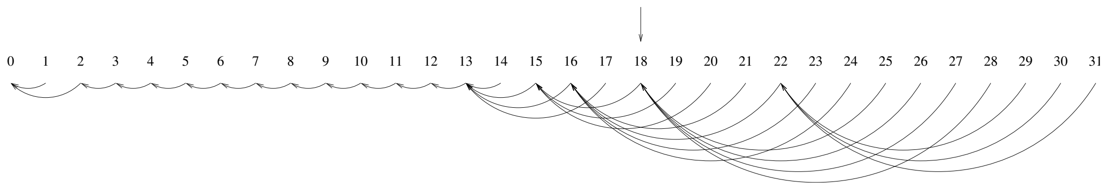

Figure 6: Evolved replacement policy for the heavy-hitter flow cache; GARP-Large = (18, (0, 0, 0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 13, 13, 13, 15, 15, 15, 16, 16, 16, 16, 18, 18, 18, 18, 22, 22, 22)).

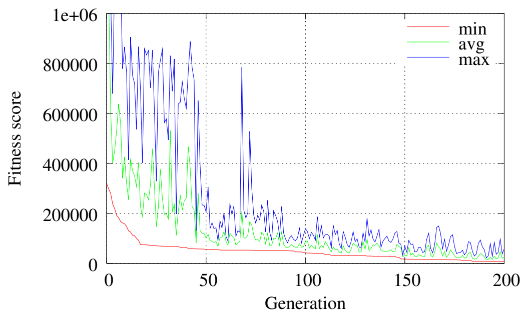

Figure 7: Evolution for Opt_1 and O_{mut} with $p_{mut} = 0.15$ (an average from ten runs).

ration. The evaluation is performed with cache sizes of 64K, 96K, 128K, 160K flow states for Snjc data set, and with 4K, 8K, 12K and 16K flow states for Mawi and Vut. The line size is 32 states.

Table 2 captures the results when considering the problem of reducing the total number of evictions. For each policy, the table contains the ratio of evictions to the total number of packets (M_1) and the increase in the number of reported flows (M_2); if a flow is evicted before its end then two or more flows are reported. The results in Table 2 show that the frequency based policies such as EXP1 or LFU*-aging are not designed to cope with the specific characteristics of the network traffic. Recency based policies such as LIRS, LRU, SLRU perform better. Interestingly, LRU and LRU-2 provide the same results. The LRU-2 reduces to a plain LRU if there are states with no more than a single packet received. Since the traffic contains many of such flows both policies select the same flow states for eviction most of the time. SLRU performs close the optimized GARP. The size of the protected segment was set up to 7 flow states according to the experiments with various sizes of protected segment.

Table 2: Comparison of replacement policies in the ability to reduce the eviction rate (the flow cache size is 64K flow states for Snjc trace and 8K for Mawi and Vut trace).

	Mawi-14:15		*Snjc-13:05*		*Vut-15:05*	
Policy	M_1	M_2	M_1	M_2	M_1	M_2
EXP1	18,8	257,1	17,7	298,3	19,9	600,5
LFU*-aging	8,3	169,1	5,8	163,8	2,5	162,6
LIRS	7,8	164,2	6,8	174,3	3,1	177,1
LRU-3	6,1	150,8	4,5	148,6	2,4	159,2
LRU-2	6,1	150,8	4,5	148,6	2,4	159,2
LRU	6,1	150,8	4,5	148,6	2,4	159,2
SLRU-7	6	149,1	4,3	146,4	2,3	157,7
GARP	5,3	144,8	4,1	144,1	2	151,6
GARP + + Class.	4,4	139,8	3,8	139,2	1,9	150
OPT	1,9	115,6	0,9	109,3	0,9	122,7

GARP and GARP with classifier of next packet arrivals operate closest to the achievable minimum represented by the OPT.

Table 3 captures the results when considering the problem of reducing the number of evicted heavy-hitters on the Mawi trace. The results display the ratio of evicted heavy-hitter states to the total number of heavy-hitter packets for each category (M_3) and the ratio of heavy-hitters that witness at least one miss to the total number of heavy-hitters (M_4).

The results show that GARP-Large outperforms other RPs and even the S^3-LRU-7 policy which is specifically designed to prioritize heavy-hitter flow states, or SLRU-21 where a large protected segment for 21 flow states allows keeping heavy-hitters protected against small flows. Most of the heavy-hitters do not witness any cache miss

Table 3: Comparison of heavy-hitter replacement policies on the *Mawi-14:15* data set.

	M3			M4		
Policy	L_1	L_2	L_3	L_1	L_2	L_3
LFU*-aging	29%	37%	21%	79%	91%	85%
LIRS	21%	33%	27%	50%	88%	71%
LRU-2	19%	30%	10%	79%	95%	48%
LRU	19%	30%	10%	79%	95%	48%
SLRU-21	14%	17%	8%	33%	47%	22%
S^3-LRU-7	5%	14%	21%	26%	68%	55%
GARP-L.	4%	8%	12%	9%	21%	34%
GARP-L. + + Class.	2%	3%	8%	6%	6%	22%
OPT-L.	0%	1%	6%	0%	2%	36%

(see the low number of M4). Moreover, if a heavy-hitter witnesses the cache miss it is the only one in most of the cases (75%).

7. Conclusion

This work was focused on the optimization of network flow monitoring by designing the customized replacement policy for the flow cache. To this end, the GA was proposed to evolve the replacement policies that would fit a particular utilization. The evolved replacement policies were compared with other policies in terms of reducing the number of total evictions and evictions of heavy-hitters. The results showed that GA is capable to evolve the optimized replacement policies which work close to the optimal solution. The replacement policy extension was proposed to complement replacement policy with predictive information from the packet headers. The extension brought the replacement policies even closer to the optimum. The approach proposed in this work may find its utilization in various applications, such as route caching, software defined networking and NetFlow monitoring.

The future work will be focused on improving the classification procedure. If the results of classification or certain statistics were stored in the flow state the classification would have been more precise. Another challenge is to evolve the policy in an online manner.

Acknowledgments. The author would like to thank his supervisor Prof. Ing. Lukáš Sekanina, Ph.D. and a consultant Ing. Jan Kořenek, Ph.D. for their advice and valuable comments.

References

[1] The mawi archive, 2010.

[2] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. An implementation study of a detection-based adaptive block replacement scheme. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 18–18, Berkeley, CA, USA, 1999. USENIX Association.

[3] B. Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101 (Proposed Standard), Jan. 2008.

[4] S. Jiang and X. Zhang. Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, 30:31–42, June 2002.

[5] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 9–9, Berkeley, CA, USA, 2000. USENIX Association.

[6] N. Megiddo and D. S. Modha. Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, Apr. 2004.

[7] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. *SIGMOD Rec.*, 22:297–306, June 1993.

[8] U. Vallamsetty, P. Mohapatra, R. Iyer, and K. Kant. Improving cache performance of network intensive workloads. In *Proceedings of the International Conference on Parallel Processing*, pages 87 –94, Washington, DC, USA, 2001. IEEE Computer Society.

[9] M. Žádník, M. Canini, A. W. Moore, D. J. Miller, and W. Li. Tracking elephant flows in internet backbone traffic with an fpga-based cache. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 640 –644, 2009.

[10] C. Walsworth, E. Aben, kc claffy, and D. Andersen. The caida anonymized 2009 internet traces, 2009.

Selected Papers by the Author

M. Žádník, M. Canini. Evaluation and Design of Cache Replacement Policies under Flooding Attacks. In: Proceedings of the 7th International Wireless Communications and Mobile Computing Conference, Istanbul, TR, IEEE CS, 2011, s. 1292-1297

M. Žádník, M. Canini. Evolution of Cache Replacement Policies to Track Heavy-hitter Flows. In: Passive and Active Measurement, Atlanta, US, Springer, 2011, s. 21-31, ISBN 978-3-642-19259-3, ISSN 0302-9743

M. Žádník, M. Canini. Evolution of Cache Replacement Policies to Track Heavy-hitter Flows (poster). In: Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, La Jolla, US, ACM, 2010, s. 2, ISBN 978-1-4503-0379-8

M. Canini, W. Li, M. Žádník, and A. W. Moore. Experience with high-speed automated application-identification for network-management. In Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS'09, pages 209-218, New York, NY, USA, 2009.

M. Žádník. Flow Measurement Extension for Application Identification In Networking Studies IV, Selected Technical Reports, Prague, CZ, CESNET, 2010, s. 57-70, ISBN 978-80-904173-8-0

M. Žádník, et al. Tracking Elephant Flows in Internet Backbone Traffic with an FPGA-based Cache. In: 19th International Conference on Field Programmable Logic and Applications, Prague, CZ, IEEE, 2009, s. 640-644, ISBN 978-1-4244-3892-1

M. Žádník, J. Kořenek, O. Lengál, P. Kobierský Network Probe for Flexible Flow Monitoring. In: Proc. of 2008 IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop, Bratislava, SK, IEEE CS, 2008, s. 213-218, ISBN 978-1-4244-2276-0