# FlowMon Cache Simulation and Analysis of Inter-Packet Gaps

Miroslava Kramáreková[1], Daniel Jakubík[1], Martin Žádník[2], and David Šafránek[1]

[1] Faculty of Informatics, Masaryk University Brno, Czech Republic,
{mirka, deny, dawe}@liberouter.org
[2] Faculty of Information Technology, Brno University of Technology, Czech Republic,
xzadni00@stud.fit.vutbr.cz

**Abstract.** Precise monitoring of network traffic can help a lot of network applications to improve their functionality. By network monitoring, applications can obtain a better conception concerning flows and packets on the network. In consequence, execution of network applications can be realised more effectively with such knowledge. To satisfy needs of high-speed network monitoring, we have developed a hardware accelerated netflow probe FlowMon which collects data about the network traffic and provides them to the application layer. FlowMon hardware design contains memories configured in a hierarchy which is similar to common computer architectures. In order to find the optimal configuration of such a memory hierarchy in FlowMon, we have analysed information about temporal and spacial localisation of packets in real network traffic. Especially, we developed a simulation model of the cache memory, which, if employed appropriately, can speed up the netflow processing in FlowMon.
In this report we present our results regarding simulation of the cache memory. These results make a crucial framework which helps hardware designers to justify the FlowMon design.

## 1 Introduction

### 1.1 FlowMon

FlowMon [2] processes and works with data in terms of *flows*. A flow is a set of packets each having the same properties (e.g., IP addresses). When FlowMon receives a packet, it has to update the respective monitoring information record for the relevant flow. If the received packet belongs to a flow that is not monitored yet, it has to create a new record for the respective new flow. Owing to this functionality, FlowMon has to manipulate its internal memory very frequently. The faster the manipulation with the memory is, the more detailed monitoring FlowMon can provide.

Hardware designers of the Liberouter team [1] have decided to implement FlowMon in the environment of memories organised in a hierarchy which is similar to that we can find in present computers [3]. Especially, the most speed-critical memory in such a hierarchy is the cache memory. Cache memory is a very expensive piece of hardware, thus it is important to know its required capacity which is sufficient for the required speed-up of the design. Information about the currently monitored flows can be managed in such a cache memory accelerating the relevant memory operations. If the flow of a currently received packet is already present in the cache, we say there is a *hit*. Otherwise, if the flow of the packet is not managed in the cache, we say there is a *miss*. To verify potential choices of hardware designers to employ a cache of a particular size, we have simulated the required functionality of the cache on a real network traffic and we have analysed the respective number of hits and misses. In general, we have established a framework for analyses of the FlowMon's possible speed-up for given sizes of the cache memory.

## 1.2 Flow

In our model the *flow* consists of packets, which have the same *key*. Each packet sent through the network has a source and a destination IP address and a source and a destination port, from which its *key* is generated. As the main aim of our simulation is to analyse potential usage of cache memory inside the FlowMon design, we do not employ traditional time-based analysis of network traffic but we rather consider a metric based on inter-packet distance. In particular, we consider the number of foreign packets which can be received until the key of a particular flow is removed from the cache. The resulting measuring technique is based on the notion of inter-packet gaps. A *gap* between the packet *A* and the packet *B*, both considered related to the same flow (both having the key *K*), is defined as the number of those packets having a key different from *K* which have been received between *A* and *B*.

The main reason for not including timing information of packet reception is independence of cache usage on throughput of the line. What is most crucial with respect to the cache usage analysis is the distance of two immediately following packets related to the same flow. Such information is not affected as much by timing aspects as by structure of the flow. Therefore our analysis employs inter-packet gaps.

## 1.3 Input and output of the simulation and analyses

As the input for the simulation and analyses we have taken a file that contained headers of packets from a real traffic sample obtained by FlowMon from 10Gb CESNET backbone mirrored to 1Gb line. The header information we have processed during the analyses contains the following items:

– for each key (hashed representation of a particular flow):
  - number of packets with the same key
  - average gap between packets of the key
  - minimum and maximum gap between packets of the key
  - standard deviation from sizes of the gaps of the key
  - histogram of the gaps of the key
– for all packets of all flows:
  - maximum and minimum gap
  - average, weighted average and standard deviation from the averages of the keys
  - histogram of all averages, maximums and minimums of the keys

From histograms created for all packets we have generated graphs which emphasise the most crucial statistical information.

Finally, concerning the simulation of the FlowMon cache memory, we have analysed number of hits and misses with respect to different sizes of the cache.

We have processed inputs and established outputs by using the scripts described in Figure 1.

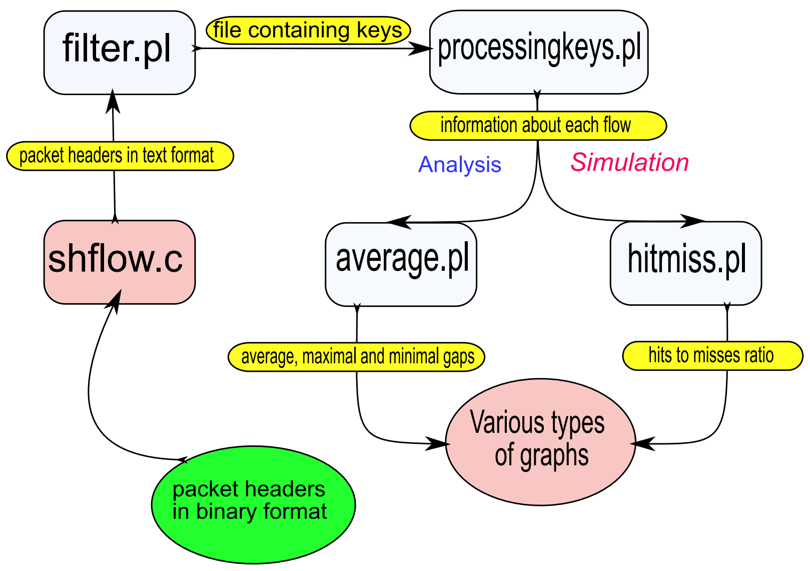

**Fig. 1.** Scheme of scripts for netmix simulation and analyses

For explanation of the scripts, see the following sections.

## 2 Analyses

The file that we have processed has the binary UHN format and contains headers of packets. As we needed to work with this information and we found out that a program for transforming a binary UHN file to a text file already exists, we decided to use it. After using this program we got the file with headers in textual form. We have implemented scripts which manipulate the data in this textual format. We have chosen perl for encoding of this scripts because it has a good implementation of regular expressions and it provides dynamic structures, i.e., an array of unknown size. We have created few perl scripts which search in the input text, analyses the data and generates files with statistics results. These scripts cooperate by inter-connecting them by pipes.

## 2.1 Keys

First script that we implemented is `filter.pl`. This script is looking for the destination and the source IP address and the destination and the source port of packets in the input file. Then it creates keys of the packets from this information. For example, the keys from the addresses in IPv4 looks like `0.0.0.1:0,0.0.0.0:2`, where `0.0.0.1` is the source IP address, `0` is the source port, `0.0.0.0` is the destination IP address and `2` is the destination port. Each key, created in this way, is managed as a single line of the file $Fk$ which is sent to the standard output. More particularly, this line represents a respective key of the currently processed packet. We needed to work with the list of keys more times, thus we decided to create it once at the beginning.

`Filter.pl` is capable of working with IPv4 as well as with IPv6 addresses. The script can be simply modified for looking for the IPs and the ports of packets in the input file that is in different format.

## 2.2 Flow Statistics

Another script is `processingkeys.pl`. This script processes the output of `filter.pl`. Each of the lines of $Fk$ containing the key of a particular packet is numbered by a unique number. This number represents the order of the incoming packet. Main part of the script manages two associative hash arrays $A_1$ and $A_2$.

Each item of the first array ($A_1$) consists of the flow's key and the number of the last flow's packet. When the key is read from the input, the script checks whether the array $A_1$ contains this key. If it does so, the inter-packet gap is counted from the number of the currently processed packet and the previously processed packet (saved in the array $A_1$). Hence, in this way we get the gap between two last packets of the flow. After that, the number of the last packet, that was already saved, is replaced by number of processed packet. Otherwise the new item of $A_1$ is created with the key of the processed packet and its number.

Each item of the second array ($A_2$) consists of the flow's key, as in $A_1$, and of a list. Lists in $A_2$ contain inter-packet gaps, counted for the packets of the flow, as mentioned in the previous paragraph. In fact, if there is too many foreign packets between two packets of one particular flow, the flow is deleted from the cache. In our analyses we do not delete this flow from the array $A_2$, but if the gap is too long, we set it to zero (maximal length of the gap is set by the respective script parameter).

From the values saved in the array $A_2$, characteristics of the each single flow are counted. We are interested in the number of packets in the flow, minimal, maximal, and average inter-packet gaps, and standard deviation from the gaps of the flow. For each flow a histogram of inter-packet gaps is constructed. This information along with information about the number of all processed packets and the number of all non-zero gaps is sent to the standard output. To a specific file, the name of which can be provided as a parameter of the script, we can save information about all gaps — i.e., the respective histogram, the number of zero gaps bigger than allowed maximum, average standard deviation, and again the number of all non-zero gaps. If the output file name is not provided, all this information is also sent to the standard output.

### 2.3 Statistics of Network Traffic

Concerning the network traffic analysis, we focus on the data from which a potential speed-up caused by increasing the cache memory can be computed. We compute such a speed-up from cumulative amount of gaps measured w.r.t. their size. Enumeration of the speed-up is achieved by employing the Amdahl's Law [4]. The general Amdahl's Law puts the upper limit on the theoretic speed-up of a parallel processing computation with respect to acceleration achieved due to increasing number of processors. A variation of Amdahl's Law can be written in terms of the ratio of cycle counts required to complete data fetch operations from memory and cache. In our situation, the speed-up introduced by the cache depends on the amount of small inter-packet gaps. The smaller are the gaps – the higher is the speed-up. Precisely, assuming that $f$ is a cumulative amount of gaps w.r.t. their size and $s$ the expected speed-up, we adapt the Amdahl's equation in the following way:

$$s \leq \frac{1}{(1-f)+\frac{f}{2.5}} \tag{1}$$

where the constant 2.5 is taken w.r.t. the caching mechanism used in FlowMon. This equation is implemented in the script to compute the speed-up for each line of the particular histogram.

Technically, there is a lot of records containing information about the flows obtained by the `processing.pl` script, especially, the output is too long for direct analysis. Thus, the data can be further processed by another script — `average.pl`. This script browses the output of `processingkeys.pl` and reads the information such as the average, maximum, minimum gap of the flows, etc.

The main part of the script is responsible for generation of three histograms (histogram of average, histogram of maximum, and histogram of the minimum gaps). During browsing the output of the `processingkeys.pl` three arrays are created by `average.pl`, and the information regarding the averages, maximums and minimums, is consequently added there. Finally, the script generates histograms from the values saved in the arrays. Each histogram has five columns each one having the following meaning:

**from** – value, from which the gaps are bigger
**to** – value, from which the gaps are smaller or equal
**amount** – amount of the gaps, whose size is between the respective values of **from** and **to**
**cumulative** – value containing the information of how many gaps have size lower than the respective value of **to**
**speed-up** – potential speed up of the system, counted according to the Amdahl's law (1) for the respective value of **cumulative**

Besides the above-mentioned histograms, `average.pl` also counts and returns different information about the flows. In particular, it also returns average and weighted average from the average, maximum and minimum gaps of the analysed flows. It includes also standard deviation computed from averages and from weighted averages.

For counting of the weighted average it is important to know the weight of the average of each flow. In particular, when the number of all processed packets is $A$ and the number of the packets of a particular flow is $N$, the weight $W$ is $W = N/A$. Finally, at the end of the output stream, information about the average, maximum and minimum number of the packets for each particular flow is returned.

## 3 Results of the Analyses

We have performed four different analyses, each of which is set up for a different value of the maximal allowed gap between two packets of a particular flow. The maximal allowed gap characterises the limit of time of keeping the flow information in the cache. We have subsequently analysed the traffic with setting the maximal allowed gap to the following values — 100000, 200000, 300000 and 20000000. Because of the observation that no gap in the analysis has reached the size greater or equal to 20 000 000, we can interpret some results for this value as if there were no limits for the gap size. However, some of the results could not be interpreted without consideration of the gap size limit. We have chosen the above mentioned maximal gap numbers to establish a uniform overview of the cache usage while not keeping the resulting information overburdened. To give a rough presumption of how the inter-packet gap number is related with inactive timeout in netflow, we have realised in our experiments that 200000 gaps agree with cca 1 second long inactive timeout. As the reasons for removing a flow key from the cache can be in general independent of time, e.g., insufficient cache space or hash collisions, maximal allowed gap is more relevant measure for our simulation purposes than exact netflow timeout.

### 3.1 Results independent of the limit of the gap

The input file has contained 14210876 packet headers. All of them have been processed during analyses. These packets have generated 1573132 different keys. For the setting of our analyses the average number of the packets with the same key (packets related to the same flow) is 9, the maximal number is 58567 and the minimal number is 1.

### 3.2 Differences in results caused by different maximal allowed gaps

There are differences between, e.g., the sizes of average gaps of the flows. With respect to state limitations we do not mention information about all 1 573 132 flows, but we present average of averages and other symmetrisation results regarding the flows. The results differ with different maximal allowed gaps.

**Overall gaps** From the script `processingkeys.pl` we get two outputs. The first one contains information about each single flow, second one contains a histogram of all gaps and the summarising information regarding all gaps. In particular, these results abstract from information about the relation of the gap to the particular flow. The summarising information is the following:

**Zero gaps, which are bigger than the allowed maximum** – number of all gaps, whose size is bigger than the allowed maximum, so their size is set to zero (”Zero gaps” in Table 1)

**Average standard deviation** – average from standard deviations of all flows

**Number of all non-zero gaps** – gaps, which are not first gaps (first gaps have still zero size) or bigger than the allowed maximum (”Non-zero gaps” in Table 1)

For the real values see the table:

| | **100 000** | **200 000** | **300 000** | **20 000 000** |
|---|---|---|---|---|
| **Zero gaps** | 653286 | 454332 | 365983 | 0 |
| **Average standard deviation** | 7497.16 | 12459.75 | 17408.91 | 172625.94 |
| **Non-zero gaps** | 11984458 | 12183412 | 12271761 | 12637744 |

**Table 1.** Summary information regarding all gaps

**Average gaps** We have counted the average inter-packet gap for each flow. Then, from these averages, we have counted its average, standard deviation, weighted average and weighted standard deviation. All these statistical data are summarised in Table 2.

| | **100 000** | **200 000** | **300 000** | **20 000 000** |
|---|---|---|---|---|
| **average from averages** | 4697.135 | 7205.769 | 9546.225 | 135910.321 |
| **standard deviation from averages** | 1013901.366 | 25063.049 | 36029.837 | 908411.376 |
| **weighted average** | 16207.764 | 8535.365 | 10403.703 | 58982.495 |
| **standard deviation from weighted averages** | 10.047 | 0.055 | 0.060 | 0.181 |
| **maximum average** | 199996 | 199996 | 299980 | 14090749 |

**Table 2.** Table of averages taken from statistic information of flows

Next we have created a histogram containing all the values presented above. We have constructed graphs from the histograms established for different maximal allowed gaps. The graph in Figure 2 is a cut graph of the cumulative amount of gaps, the size of which has been smaller then the value of the item *to* in histogram (column “cumulative” in the histogram). The graph in Figure 3 is a cut graph of speed-up of the system for different setting of the average gap. E.g., the speed-up for the value 2000 is counted as the average gap of the saved flows in the cache (which is 2000 in this particular case).

**Maximum and minimum gaps** For each flow we also found the maximum and minimum inter-packet gap. Again, we have constructed a histogram of all these maximum and minimum gaps. In consequence, we have counted the average and the weighted

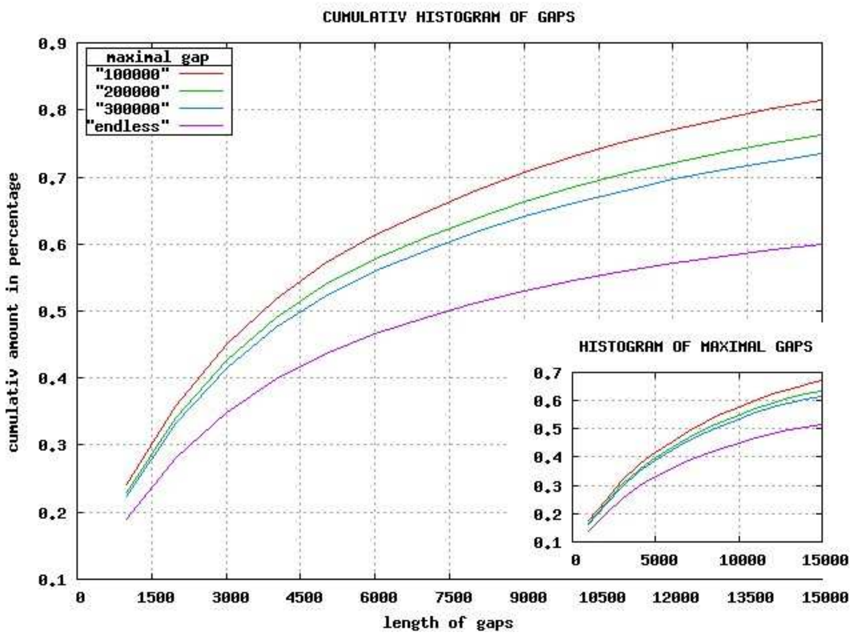

**Fig. 2.** Cumulative histogram of gaps

average of the maximum and minimum gaps. Finally, we have found maximal maximum gap and maximal minimum gap. It is worth noting that maximal maximum gap and minimum gap cannot be bigger than maximum allowed gap for the corresponding analysis.

| | **100 000** | **200 000** | **300 000** | **20 000 000** |
|---|---|---|---|---|
| **maximum gaps averages** | 9412.004 | 15563.977 | 21589.499 | 252499.558 |
| **maximum gaps weighted averages** | 44751.034 | 65662.288 | 82708.748 | 379858.459 |
| **maximal maximum gap** | 100000 | 200000 | 299999 | 14128346 |
| **minimum gaps averages** | 2561.761 | 3754.443 | 4823.760 | 87308.533 |
| **minimum gaps weighted averages** | 916.812 | 1274.784 | 1583.018 | 21086.952 |
| **maximal minimum gap** | 99996 | 1999960 | 299980 | 14090749 |

**Table 3.** Table of maximum and minimum gaps

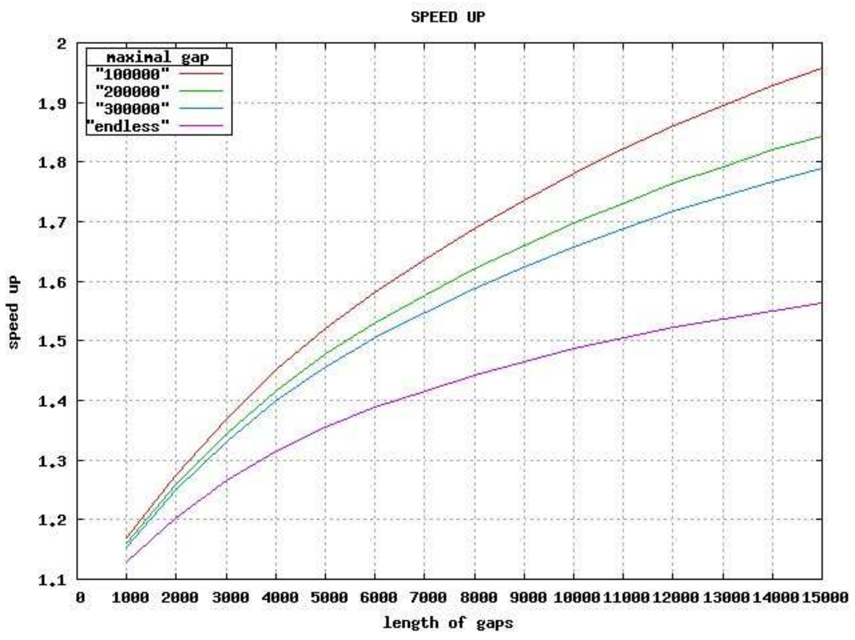

**Fig. 3.** Speed up

## 4 Run of the Simulation

### 4.1 Hits and Misses

The aim of FlowMon designers has been to implement in the hardware part a hierarchy of memories (cache and DRAM) in order to speed up the work with the processed data. Our script `hitmiss.pl` simulates the work of the cache memory. The first version of this simulation script has been developed as a very simple program. There has been defined an associative hash array with a constant size of its elements — a natural representation of the cache memory. The size of this array (the capacity of the cache memory) can be set by the script parameter. The script reads the line from the standard input step by step (this script has been developed in such a way that it fits the pipe-style linking with the script `filter.pl`). After loading of the key, script checks if the key is already saved in the array. The reason why this script has been developed was to find out how big the cache should be in order to find the optimal number of cache hits. There are two global variables in the script – `hit` and `miss`. If the currently processed key is found in the array, `hit` is increased, otherwise, `miss` is increased provided that the key is added to the cache. If the cache is full, the first value of the array is deleted additionally before adding the key to the cache. After successful running of the script the resulting values of these two variables are printed to the standard output.

During experiments with the above mentioned script we have found out that the script needs to employ a more sophisticated method of organisation of the array. Especially, when the script was started with a reasonably high parameter value (the length of the cache), its computation has been too slow. Such a high time complexity was caused by frequent reorganisation of the hash structure to which the values were being saved. To overcome this problem, we have created a large array of pointers (containing 999983 items) in which each pointer refers to a smaller array. Access to the array of pointers is realised by a hash function. The respective sub-arrays are processed sequentially.

## 5 Results of the Simulation

We have made a simple model of the cache and we have simulated its function. The model is represented by the script `hitmiss.pl`.

### 5.1 Hits and misses

We have run the script *hitmiss.pl* for different parameter values representing different setting of the cache capacity. In other words, this means different numbers of the flows that can be stored in the cache in the same time. Table 4 shows the results obtained for particular settings of the cache capacity. The right-most column represents ratio of number of hits with respect to number of all packets. The results are visualised by graphs in Figure 4 and Figure 5.

| **Table of the hits and the misses** | | | |
|---|---|---|---|
| | **hits** | **misses** | **ratio** |
| **100** | 857700 | 13353176 | 6,04% |
| **200** | 866652 | 13344224 | 6,10% |
| **500** | 5328302 | 8882574 | 37,49% |
| **1000** | 6474444 | 7736432 | 45,56% |
| **2000** | 7873891 | 6336985 | 55,41% |
| **5000** | 9729892 | 4480984 | 68,47% |
| **10 000** | 10803954 | 3406922 | 76,03% |
| **15 000** | 11265142 | 2945734 | 79,27% |
| **20 000** | 11524476 | 2686400 | 81,10% |

**Table 4.** Table of the hits and the misses

## 6 Conclusion

### 6.1 Summary of the Results

Although we have based our analyses on just one sampled network flow, the obtained results show that potential investment in a higher-capacity cache memory does not bring the expected overall speed-up of the FlowMon system. However, what capacity of the

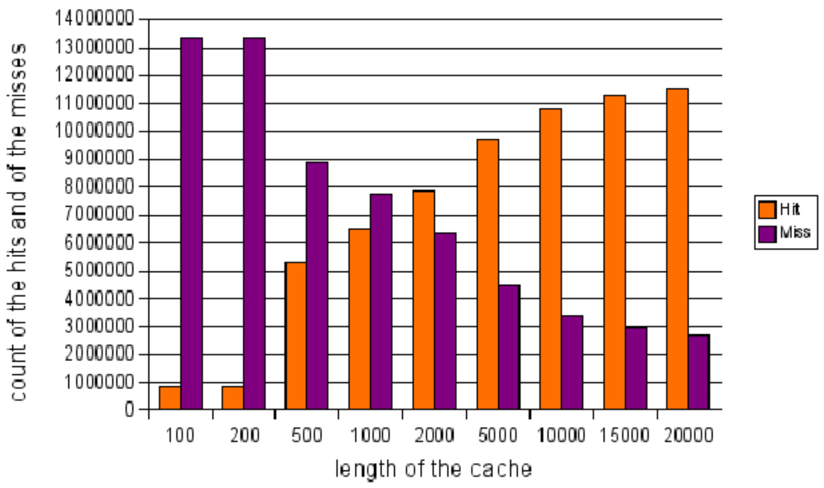

**Fig. 4.** Results of the FlowMon cache simulation

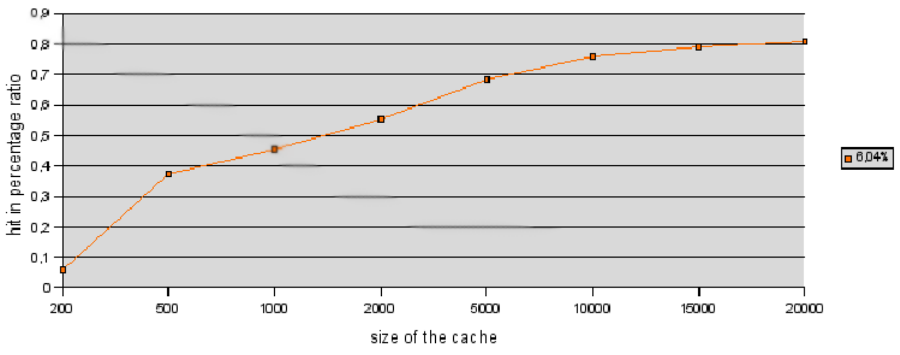

**Fig. 5.** Distribution of different cache sizes

cache memory should be exactly used in FlowMon will be more clear from more experiments obtained from various samples of real network flows. This remains for future work.

## 6.2 Future Work

In future, we plan to change the scripts in order to enable collection of some more information which currently seems to be needed. However, our aim is to develop a robust conception about the network traffic. In order to realise that we need to analyse more input data, i.e., various high-rated samples of real network traffic). We currently encounter problems with catching such detailed netflow data. Especially, using tcpdump for getting some new inputs currently does not allow us to catch full gigabyte traffic. E.g., when we run tcpdump with parameters `tcpdump i eth1 -s 40 -w traffic.dump` we encountered 2/3 of all packets lost. It is obvious, that it is impossible to monitor the traffic with current software tools. This deficiency itself calls for a hardware acceleration of the netflow monitoring process. We believe that FlowMon with its HW acceleration will be able to satisfy the needs as our first experiments show. Any way, analyses of such incomplete inputs like those presented in this report can be also very useful for making a statistical picture about real network traffic.

## References

1. Liberouter Project. Description of COMBO cards. http://www.liberouter.org/hardware.php
2. Žádník, M. Overview of NetFlow Monitoring Adapter. CESNET Technical Report 8/2004.
3. Žádník, M. NetFlow probe firmware design. http://www.liberouter.org/netflow/design.php
4. Amdahl, G. Validity of the single-processor approach to achieving large-scale computational capabilities. In Proceedings of AFIPS Conference, volume 30, page 483, AFIPS Press, 1967.