This paper was presented as part of the High-Speed Networks 2011 (HSN 2011) Workshop at IEEE INFOCOM 2011

Adaptive Bloom Filters for Multicast Addressing

Zalan Heszberger*, János Tapolcai*, András Gulyás*, Jozsef Biro*, András Zahemszky*, Pin-Han Ho†
* Budapest University of Technology and Economics, Dept. of Telecommunications and Media Informatics
1117 Budapest, Magyar tudósok körútja 2. Hungary, Email: {heszberger,tapolcai,gulyas,biro,zahemszky}@tmit.bme.hu
† Dept. of Electrical and Computer Engineering, University of Waterloo, Canada, p4ho@uwaterloo.ca

***Abstract*—In-packet Bloom filters are recently proposed as a possible building block of future Internet architectures replacing IP or MPLS addressing that solves efficient multicast routing, security and other functions in a stateless manner. In such frameworks a bloom filter is placed in the header which stores the addresses of the destination nodes or the traversed links. In contrast to the standard Bloom filter, the length of the in-packet Bloom filter must be highly adaptive to the number of stored elements to achieve low communication overhead. In this paper we propose a novel type of Bloom filter called Adaptive Bloom filter, which can adapt its length to the number of elements to be represented with a very fine granularity. The novel filter can significantly reduce the header size for in-packet bloom filter architecture, by eliminating the wasting effect experienced in existing "block-based" approaches which rely on concatenating several standard Bloom filters. Nevertheless, it requires slightly more calculations when adding and removing elements.**

I. Introduction

Seeking for efficient multicast solutions, the application of Bloom filters is becoming increasingly popular in future internet architectures [1], [2] designed with a goal to replace IP or MPLS addressing. The Bloom filter [3] is a simple yet efficient tool for deciding whether an element belongs to a set or not. Placed in packet headers, the *in-packet* Bloom filters can effectively address a set of nodes or links, hereby qualifying themselves as a strong candidate solution for efficient stateless multicast addressing.

The in-packet Bloom filter concept implements source routing, where each network element (link or host) is assigned with a (random) address, which is a bit array (codeword), and the header contains a Bloom filter for the set of elements (links, hosts) in the multicast tree. The Bloom filter is a bit array equals to the bitwise OR of the codewords of the elements involved, also called as zFilter. When a packet with in-packet Bloom filter arrives to a router, membership testing is performed. The operation is extremely fast when implemented on digital signal processor, which leads to simpler router architecture compared to current IP routers [2]. Basically, a bitwise AND operation is performed on the Bloom filter placed in the header and the address of the outgoing link or node. Moreover, bloom filters are favored for their space efficiency since the filter requires much less space than listing the identifiers for each links/hosts in the multicast tree. Furthermore, this allows routers to stay quasi-stateless, because the routers only need to know the address of the neighboring nodes and links, and no global topology information is required for forwarding. As a price for the space efficiency false positives may occur, which means there are some elements not contained by the set for which the membership test gives positive answers.

In such a networking context the number of elements (e.g. the number of participating hosts in a multicast communication) is not known in advance and may lie within a wide range, from few to a few tens of thousands. Therefore using the original Bloom filter framework one has to design the filter size for the worst-case, thus wasting storage space when smaller number of elements are actually in the filter. However for in-packet Bloom filters the storage space efficiency is the key requirement to keep the communication overhead at a low level. Note that, in traditional Bloom filter applications like in database systems, the *storage space of membership testing* and the *storage space of adding (or removing) elements* is not distinguished. However, for in-packet Bloom filters a network management entity may store some additional information needed to add and remove elements from the filter. As a consequence we identify two performance metrics for Bloom filters: storage space for membership testing and storage space for adding or removing elements. The main requirement for in-packet filters is to keep storage space for testing as small as possible.

Numerous variants of the framework aimed at providing a scalable solution that keeps the simplicity and efficiency of the original Bloom filter while not relying on the explicit knowledge of the number of elements [4]–[8]. A common feature of these studies is that they use original Bloom filters as building blocks often with varying sizes. To meet the false positive requirement and also scale with the number of stored elements several standard Bloom filters are concatenated, and a single set element is stored in a given Bloom filter block. This approach eliminates the low performance of the standard filters requiring huge space because of the conservative assumptions regarding the set size; however raises the question at a lower level, in the level of the filter blocks concatenated. Since it is not known in advance how much elements there will be in a given block, such "block" based approaches still suffer from wasting of storage space for membership testing.

In this paper we propose novel filter variants that can significantly reduce the header size for in-packet bloom filter architecture. It is achieved by fully eliminating the wasting effect stemming from uncertainty of the number of elements to be stored in the future. We have to sacrifice some of the simplicity of the original Bloom filter construction, and slightly increase the storage space required for adding elements.

In our construction instead of using standard Bloom filters as building blocks, a special large filter is dimensioned for the worst-case scenario (based on estimation on the maximal set size) into which the elements are inserted and a fine grained truncating mechanism is defined, that provides an optimal sized filter for the actual number of members meeting the false positive requirement without wasting space because of the ”blocking” effect. We also show that due to performance reasons it is more reasonable to use non-uniform hash functions for generating the codewords for the investigated elements.

The rest of the paper is organized as follows. In Section II we briefly overview the main characteristics of Bloom filters and the current scalable implementations. As a main result of the paper we define our novel Adaptive Bloom filter with fine-grained truncating mechanisms and provide formulae for determining the truncation point for a given number of elements and false positive probability in Section III. Since the required parameters cannot be expressed in a closed form we formulate an optimization problem, discuss its solutions, and present the performance of the adaptive filter with the optimized settings in Section IV and V.

II. Background and Related Work

In the original Bloom filter design [3], a set of elements chosen from a universe are represented by an array of m bits (codeword), among of which for each element to be included in the filter a maximum of k bits are set in positions selected with k different *hash functions*. Each of the k hash functions maps the given element onto one of the m bit positions. The hash functions are assumed to be independent and each position is selected with equal probability. The Bloom filter is an m bit long binary array representing a set of elements. It consists of the bitwise OR of the codewords of the elements in the set. As the main feature membership test can simply be performed by checking 1's in all bit positions that are set in the codeword of the underlying element.

The performance of the filter is measured by the false positive rate that is the probability that an element appears to be included (by test) but actually it is never added. After the inclusion of n elements into the filter false positive probability is usually calculated as:[1]

$$P_f \approx \left(1-(1-p)^{nk}\right)^k \tag{1}$$

where $p = 1/m$ is the probability of choosing uniformly one element among m. Using the above formula the size of the filter can be determined that ensures a given false positive probability.

Several modification of the original framework has been introduced since. In [10] Counting Bloom filters are presented extending the original framework with the possibility of eliminating elements from the filter by introducing multiple-bit counters in place of bit positions. The aim of Compressed Bloom filters in [11] is to optimize space requirement using coding theory, while Spectral Bloom filters [12] address the problem of the multi-set storing. In Weighted Bloom Filter [13] the uneven membership query rates and membership probability information of different elements have been taken into account under reasonable frequency models.

More recent modifications of the Bloom filter framework aims at applications where the number of set elements is not known in advance. In [7] smaller standard Bloom filters are concatenated to ensure scaling with the number of elements introducing a false positive probability growing linearly with the number of filters. Independently Scalable Bloom filters were introduced in [4], [6] which are built in phases with each phase the size of the newly added filter grows exponentially guaranteeing the false positive rate remaining under a predefined threshold. The Incremental Bloom filter in [5] applies fill factor threshold technique instead of the false positive probability defining the Restricted Fill Bloom Filter. The paper provides optimal solution for the memory requirement in cases when certain information about the distribution of the number of elements is known in advance.

A common property of the above dynamically increasing filters is that with inserting new elements into the filter the size quickly grows with large quanta of integral multiples of the bit-length representing the elements. These procedures usually require more memory than necessary since the last block is almost always partially filled. In the following we introduce a new construction where filter size can be finely tuned to the number of stored elements with an adaptive fine-grained truncating mechanisms.

III. Adaptive Bloom filters

To ensure a moderate communication overhead in-packet Bloom filters must be highly adaptive to the number of stored elements (links, hosts, etc.) satisfying a predefined false positive threshold f_p. Throughout this paper we treat this requirement in the following way: Let m_i be the size of a filter, when representing i elements. Our main target hereafter is to minimize the cost function given in the form

$$C = \sum_{i=1}^{n} c_i \cdot m_i \tag{2}$$

where c_i is the part of the input and denotes the relative frequency of including exactly i elements into the filter. For in-packet Bloom filters c_i is the probability of having a multicast demand with exactly i links and hosts. This cost function expresses the average amount of memory the filter consumes when used in networking applications.

Note that for in-packet Bloom filters m_i is the size of the header when i link or host is included in the multicast tree. Clearly, the length of the header depends on the false positive threshold and on the actual link/host addresses, which is further elaborated in the next section.

To achieve a minimal cost according to (2), in what follows we introduce an Adaptive Bloom filter construction that relies on non-uniform hash functions setting bits in a codeword with probability p_i depending on their position i. From this an M-bit long filter is built up, however depending on the maximum

[1]A more rigorous calculation of the false positive probability can be found in [9].

tolerable false positive probability f_p only the first m_j bits are deployed in the packet header where j is the current number of elements in the set. Clearly $M \geq m_n$ must hold, where n is the maximum number of element that the filter may contain. For ensuring optimal filter size for a given distribution of element set sizes, membership test parameters should be carefully adjusted at distinct parts of the filter.

Theorem 1: The false positive probability of the heterogeneous m-bit Bloom filter with bit probabilities $p_1, p_2, \dots p_m$ where $\sum_{i=1}^{m} p_i$ can be calculated as:

$$P_f^H = \left(1 - \sum_{i=1}^{m}\left(p_i\left(1-p_i\right)^{nk}\right)\right)^k \tag{3}$$

Proof: Let the set of link elements to be checked against our filter denoted by S. We generate the bits of a codeword for a given element with probabilities depending on their position: $p_1, p_2, \dots p_m$ where $\sum_{i=1}^{m} p_i$. The probability that the adaptive Bloom filter at position i, denoted by B_i, is not set after the insertion of n elements using k hash functions per element is clearly computed by:

$$P(B_i = 0) = (1 - p_i)^{nk} \tag{4}$$

By the law of Total Probability the probability that the jth hash function generating a bit position in the codeword of element s (where $s \in S$ is not included in the Bloom filter) selects position i (or more formally $h_j(s \in S) = i$), when the ith bit in the filter is set is given by:

$$\begin{aligned} P_{h_j} &= \sum_{i=1}^{m} P(B_i = 1 \mid h_j(s) = i) P(h_j(s) = i) \\ &= \sum_{i=1}^{m} P(B_i = 1) P(h_j(s) = i), \end{aligned} \tag{5}$$

as hash functions are independent of the state of the Bloom filter. Putting (4) into (5) and using that $P(h_j(s) = i) = p_i$ we get:

$$P_{h_j} = \sum_{i=1}^{m} (1 - (1 - p_i)^{nk}) p_i \tag{6}$$

After applying all k hash functions and assuming the bit-false events to be independent [9] the probability that all the bits of the codeword of element s is included in the Bloom filter (that is we get a false positive answer for the membership query of element s) is given by the formula:

$$P_f \approx (\sum_{i=1}^{m} (p_i - p_i(1 - p_i)^{nk}))^k \tag{7}$$

Applying that $\sum p_i = 1$ immediately gives the statement of the theorem. ■

It is easy to see that the special setting of $p_1 = p_2 = \dots = p_m = 1/m$ recovers the result for the original Bloom filter. The heterogeneity of the bit set probabilities at different parts of the element codewords ensures that the information is concentrated at predefined areas in the filter carefully adjusted according to the relative frequency c_i of membership cardinality i. E.g. when the number of set members stored in the future is expected to be relatively small, bit set probabilities optimally adjusted at the beginning of the filter ensures that most of the time only a small truncation of the filter will be enough for satisfying a given false positive requirement. This property provides us the possibility to reduce C the weighted average filter size for membership testing.

With respect to the efficient computation of formula (3), in the homogeneous case of $p_1 = p_2 = \dots = p_m = 1/m$ the optimal number of hash functions can easily be given in a closed form [3]. The heterogeneous problem, on the other hand leads to a optimization task that can only be numerically dealt with. Although in many cases the establishment of the set membership filter is not a time constrained process.For this purpose, in the next section, we present an interesting heuristic alternative for the Adaptive Bloom filter where near-optimal solution can more easily be found in a closed form, and its performance is comparable to that of the original Bloom filter (3). For fast computation a numerical optimization process is also proposed along with some comparisons of the results to related scalable type Bloom filters derived from relevant literature.

IV. "BLOOM-LIKE" MEMBERSHIP SET FILTERS

The performance of the proposed adaptive filter in the previous section mainly depends on the values of p_i and k (i.e. the number of hash functions), therefore their quasi optimal setting is crucial. In this section a recursive heuristic approach is presented that is considered being an approximation of the original heterogeneous Bloom filter (3), for which a near-optimal closed form solution is proved to be obtainable by moderate effort. Then in the next section we formulate a numerical optimization problem which fine tunes the heuristic solution.

The idea comes from the observation that in (3) the i^{th} bit is set with probability approximately $q_i = kp_i$ when $kp_i \ll 1$ which considering that M, the (almost never used) total length of the filter, is typically large compared to k can be accepted as a reasonable assumption. Let us call q_i as *bit probabilities*. The behavior of the above Bloom filter can be simulated by simply applying Bernoulli random variables for each i index with parameter kp_i. To establish the false positive probability of such a Bloom filter "imitation" we have the following Theorem:

Theorem 2: The false positive probability of a m bit long filter established by bitwise ORing n codewords generated by m number of independent Bernoulli random variables applied to the ith position with set probabilities q_i:

$$P_f^L = \prod_{i=1}^{m} (1 - q_i(1 - q_i)^n) \tag{8}$$

Proof: In the described algorithm the probability that a bit at the ith position in the filter containing n elements is not set can clearly be computed by:

$$q_i^b = (1 - q_i)^n. \tag{9}$$

The probability that the bit of the codeword of a new element at the ith position is set while the filter at the same bit position is not (which is the only case when a codeword is considered to be not included in the filter due to the ith bit position) is:

$$q_i^{ni} = q_i q_i^b = q_i(1 - q_i)^n. \tag{10}$$

Applying the above for all m bit positions and exploiting full independence among them we get to the statement of the Theorem. ∎

To shed some light on the connection between formulae (8) and (3) we can use the following line of thoughts:

Taking the well-known approximation

$$\prod_{i=1}^{m}(1 - x_i) \approx 1 - \sum_{i=1}^{m} x_i, \tag{11}$$

for positive m and x_i when $mx_i \ll 1$, it can be applied to (8) and so we get:

$$P_f^L \approx 1 - \sum_{i=1}^{m} q_i(1 - q_i)^n. \tag{12}$$

Now applying (11) in the form of:

$$(1 - x)^m \approx 1 - mx, \tag{13}$$

and substitute $q_i = kp_i$ we get

$$P_f^L \approx (1 - \sum_{i=1}^{m} p_i(1 - kp_i)^n)^k. \tag{14}$$

Finally using that $kp_i \ll 1$, applying again (13) in the same direction we arrive to

$$P_f^L \approx (1 - \sum_{i=1}^{m} p_i(1 - p_i)^{kn})^k. \tag{15}$$

which is exactly (3).

A. Heuristic Recursive Closed Formula

In the following relying on a simple recursive heuristic we establish a sub-optimal q_i constellation in a closed form which is also used as an initial point for further numerical optimization in the next section.

According to our problem formulation in Section III we search for $m_1, \ldots, m_n$ and $q_1, \ldots, q_{m_n}$ in case of given f_p and n, which minimize C cost function (2) under the constraints:

$$\prod_{j=1}^{m_i} \left(1 - q_j(1 - q_j)^i\right) \le f_p \qquad i = 1, \ldots, n. \tag{16}$$

To minimize m_i for any given i separately under the constraint (16) is recursively performed by minimizing $\left(1 - q_j(1 - q_j)^i\right)$, which has the minimum at $q_j = 1/(i + 1)$ for all j. Now as a heuristic decision let us choose:

$$q_{m_{i-1}+1} = q_{m_{i-1}+2} \cdots = q_{m_i} = \frac{1}{1 + i}. \tag{17}$$

Next m_i is calculated accoridng to q_j by the recursive process:

$$m_1 = \left\lceil \frac{\log f_p}{\log(3/4)} \right\rceil \tag{18}$$

and

$$m_i = m_{i-1} + \left\lceil \frac{\log f_p - \sum_{j=1}^{m_{i-1}} \log(1 - q_j(1 - q_j)^i)}{\log(1 - i^i(i + 1)^{-i-1})} \right\rceil. \tag{19}$$

Using the above formulation a suboptimal solution based on the filter construction described in Section IV can be calculated.

V. Greedy Search for Fine Tuning Bit Probabilities

In this section we define a non-linear optimization problem which seeks for the optimal values for q_j.

A. Optimization Problem for Adaptive Bloom-like filters

For Adaptive Bloom-like filters of Section IV we have Eq. (16) which has n constraint to fulfill. Let M be a constant definately at least $M \ge m_n$ If $q_1, \ldots, q_M$ are known according to the recursive process of Eqs. (18) and (19) m_i can be evaluated for $i = 0, \ldots, n$. Let us denote this recursive process by $m_i = \alpha(i, f_p, q_1, \ldots, q_M)$. Note that, evaluating α for a given $q_1, \ldots, q_M$ values can be done in linear time.

The task of the optimization process for Adaptive Bloom-like filters is to find the $q_1, \ldots, q_M$ vector, for a given f_p threshold such that

$$\min_{q_1,\ldots,q_m} \sum_{i=1}^{n} c_i \cdot \alpha(i, f_p, q_1, \ldots, q_M) \tag{20}$$

is minimal. Note that, according to Eq. (20) the total cost of a given $q_1, \ldots, q_m$ vector can be evaluated in $O(nm)$.

B. Optimization Problem for Adaptive Bloom filters

Simlarly for Adaptive Bloom filters of Section III we have Eq. (3) which defines the false positive rate. By assigning $k := \lceil \sum_{j=1}^{M} c_j \rceil$ we define the following n constraint to fulfill

$$\left(1 - \sum_{i=1}^{m} \left(\frac{q_i}{k}\left(1 - \frac{q_i}{k}\right)^{nk}\right)\right)^k \le f_p \tag{21}$$

Similarly, according to the recursive process of Eqs. (18) and (19) m_i are evaluated for $i = 0, \ldots, n$, and the objective remains Eq. (20).

C. Optimization process

As an initial solution we take results of the heuristic closed formula presented in Section IV-A. Inspired by the fast cost evaluation method we propose simple greedy random search to obtain a near optimal solution for $q_1, \ldots, q_M$. As shown on Algorithm 1, in each random operation we generate a random index $1 \le j \le m_n$, and a random small value $\delta \ll 1$, and δ is rapidly decreasing during the search process. Next we perform a randomly selected operation, which can be (1) $q_k := q_k + \delta$ for $k = 1, \ldots, j$, and (2) $q_k := q_k - \delta$ for $k = 1, \ldots, j$, and

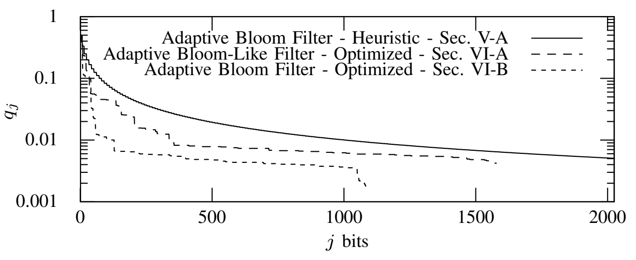

Fig. 1. The value of q_j for different j when the $f_p = 0.1$ and $n = 200$.

(3) $q_k := q_k + \delta$ for $k = j+1, \ldots, M$, and (4) $q_k := q_k - \delta$ for $k = j+1, \ldots, M$. If the modified solution has a larger C, the last random operation is reverted. The greedy random process is repeated until 300 consecutive random operations cannot decrease the cost of the solution C. See also Fig. 1 for the q_j values for the heuristic closed formula and after optimization with objectives of the two cost functions defined in Sections V-A and V-B.

Algorithm 1: Optimize $q_1, \ldots, q_M$

```
Set $q_j$ according to the recursive approach in Sec. V.;
$\delta = 0.1$;   $t = 0$;   $C_b = \infty$;
while $\delta > 0.0001$ do
    $k = rand(m_n)$ a random index;
    switch rand(4) do
        case 1  for $k = 0, \ldots, jj$ do  $q_k = q_k + \delta$;
        case 2  for $k = 0, \ldots, jj$ do  $q_k = q_k - \delta$;
        case 3  for $k = jj+1, \ldots, M$ do  $q_k = q_k + \delta$;
        case 4  for $k = jj+1, \ldots, M$ do  $q_k = q_k - \delta$;
    end
    Evaluate cost $C$;
    if $C \geq C_b$ then where $C_b$ is the best save solution
        revert the values of $q_1, \ldots, q_M$;
        $t = t + 1$;
    else
        $C_b = C$;   $t = 0$;
    end
    if $t > 300$ then $\delta = \delta/10$;   $t = 0$;
end
return $q_1, \ldots, q_M$;
```

D. Evaluation

Five methods have been implemented for evaluation. The three methods proposed in the paper are the recursive heuristic formula presented in Section IV-A, the optimization of Section V-A, and the optimization of Section V-B. Besides two more methods the Incremental Bloom filter presented in [5] and the Scalable Bloom filter of [6]. Fig. 2 shows the values of m_i for $i = 1, \ldots, n$ obtained by the five methods. Both Incremental and Scalable Bloom filters add large blocks to the filters, which is indicated by large steps in the increase of storage requirement for membership testing. At the same time all Adaptive Bloom filters provide a smooth increase in the size of storage space as the number of elements in the filter grows, which saves an average of 40%. The averages are shown on Fig. 3 for different false positive values calculated in the homogeneous setting $c_i = 1/n$ (see equation (2)).

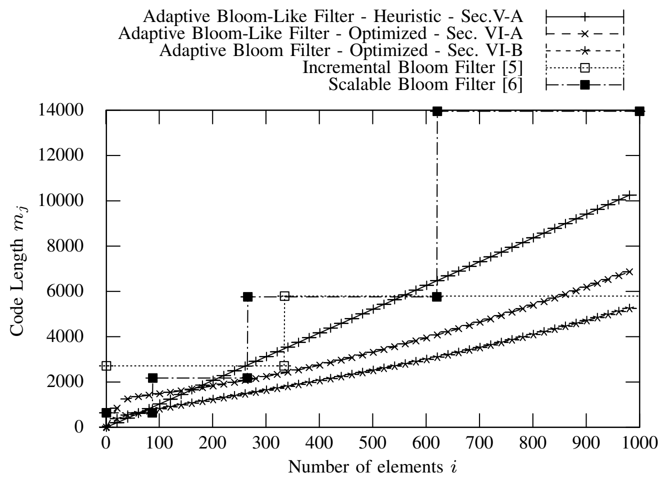

Fig. 2. Comparison of Incremental Scalable and Adaptive Bloom Filter in terms of code length for different number of elements in the Bloom filter when $f_p = 0.1$. The settings of the Scalable filter: $r = 0.5, s = 2, m_0 = 128$

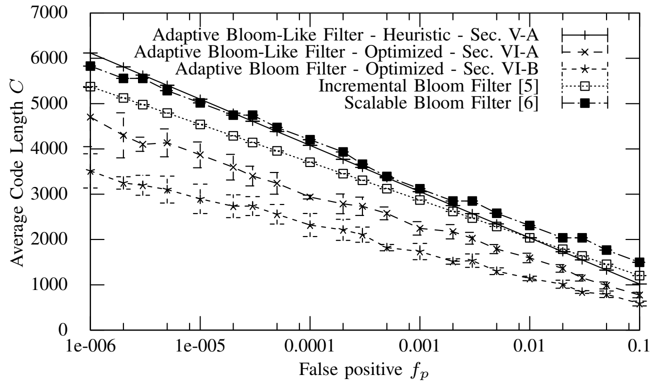

Fig. 3. Comparison of Incremental Scalable and Adaptive Bloom Filter in terms of average storage size for membership testing for different false positive values, when the maximum number of elements is fixed to 200. The settings of the Scalable filter: $r = 0.5, s = 2, m_0 = 128$

The performance of the Adaptive Bloom filters after the optimization of Section V-B outperform all the counterparts for every false positive threshold investigated. In our experience the optimization process of Section V can save up to 50% of the storage space for membership testing compared to the recursive heuristic solution.

Acknowledgements

This work is connected to HSNlab and the scientific program of the ”Development of quality-oriented and cooperative R+D+I strategy and functional model at BME” project. This project is supported by the New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002). J.T. was supported by the Magyary Zoltán post-doctoral program.

VI. Conclusions

Our paper extends the palette of in-packet Bloom filters by introducing a concept called Adaptive Bloom filter. As a main feature it can reduce the storage size required for membership testing compared to alternative scalable Bloom filter designs. This property is particularly important for implementing stateless multicast addressing in future Internet proposals, as the filter is placed in the header of each packet. Non-uniform hash functions lie in the heart of the novel filter achieving a desired distribution of the information coded by the bits set. For simple evaluation purposes a heuristic recursive closed formula is given along with an optimization approach for finding near optimal parameter settings. The numerical results support that an average of 40% save is achieved by the Adaptive Bloom filter in the membership testing storage requirement against its counterparts.

REFERENCES

[1] S. Ratnasamy, A. Ermolinskiy, and S. Shenker, “Revisiting IP multicast,” *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4, p. 26, 2006.

[2] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander, “LIPSIN: Line speed publish/subscribe inter-networking,” *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 195–206, 2009.

[3] B. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[4] K. Xie, Y. Min, D. Zhang, J. Wen, and G. Xie, “A scalable Bloom filter for membership queries,” in *IEEE Global Telecommunications Conference, 2007. GLOBECOM’07*, 2007, pp. 543–547.

[5] F. Hao, M. Kodialam, and T. Lakshman, “Incremental bloom filters,” in *Proc. IEEE INFOCOM*, 2008, pp. 1741–1749.

[6] P. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, “Scalable bloom filters,” *Information Processing Letters*, vol. 101, no. 6, pp. 255–261, 2007.

[7] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, “The Dynamic Bloom Filters,” *IEEE Transactions on Knowledge and Data Engineering*, pp. 120–133, 2009.

[8] D. Guo, J. Wu, H. Chen, and X. Luo, “Theory and network applications of dynamic bloom filters,” in *Proc. 25th IEEE INFOCOM*. Citeseer, 2006.

[9] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang, “On the false-positive rate of Bloom filters,” *Information Processing Letters*, vol. 108, no. 4, pp. 210–213, 2008.

[10] L. Fan, P. Cao, J. Almeida, and A. Broder, “Summary cache: a scalable wide-area web cache sharing protocol,” *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, p. 293, 2000.

[11] M. Mitzenmacher, “Compressed bloom filters,” *IEEE/ACM Transactions on Networking (TON)*, vol. 10, no. 5, pp. 604–612, 2002.

[12] S. Cohen and Y. Matias, “Spectral bloom filters,” in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, p. 252.

[13] J. Bruck, J. Gao, J. C. A., and Y. Matias, “Weighted Bloom Filter,” in *Information Theory, 2006 IEEE International Symposium on*. IEEE, 2006, pp. 2304–2308.