2009 International Conference on Computational Science and Engineering

K-Stage Pipelined Bloom Filter for Packet Classification

Mahmood Ahmadi and Stephan Wong
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
{ M.ahmadi,j.s.s.m.wong }@tudelft.nl

***Abstract*— A Bloom filter is a simple space-efficient randomized data structure for representing a set in order to support membership queries. In recent years, Bloom filters have increased in popularity in database and networking applications. In this paper, we introduce a k-stage pipelined Bloom filter architecture to decrease power consumption. In the bit-array of a Bloom filter, bits corresponding to the index pointed to by hashing functions are checked and a "match"/"mismatch" is determined. The match/mismatch determination process can be organized in a k-stage pipelined Bloom filter architecture. We present a k-stage pipelined Bloom filter, the power consumption analysis and utilize a software packet classifier to customize the k-stage pipelined Bloom filter architecture in packet classification. The results of the software packet classifier with real packet traces show that more than 75% of mismatched packets can be detected by the first three stages of the pipelined Bloom filter architecture (the remaining 25% comprises 17% matched and 8% mismatched packets). Therefore, a 4-stage pipelined Bloom filter architecture with one hashing function in the first three stages and $k-3$ parallel hashing functions in the last stage is more appropriate for power consumption optimization in packet classification.**

Keywords: Pipelined Bloom filter, power consumption, packet classification, hashing functions.

I. Introduction

Demand for portable network applications and devices is constantly driving the need for low-power solutions at the chip, system, and algorithm levels. These low-power management solutions are key in the industry's continuing quest to become smaller, cheaper, and more portable. Due to the large-scale integration and high speed circuitry, network processors deployed in typical network equipment can consume more power than other components. As a key attached component to the network processor, the packet classifier must definitely to be designed with power efficiency in mind.

Packet classification is increasingly being used by networking devices such as routers, switches, and firewalls to support of many different services such as quality of service (QoS), virtual private network (VPN), policy-based routing, traffic shaping, firewalls, and network security [8][12][16]. In order to provide these services, the router must categorize the incoming packets according to different criteria. These criteria are determined based on one or more fields in the packet header. Packet header fields include destination and source IP addresses, the protocol type, and the destination and source port numbers. Essentially, packet classification can be seen as the categorization of incoming packets based on their headers according to specific criteria that examine specific fields within a packet header. The criteria are comprised of a set of rules that specify the content of specific packet header fields to result in a match. Traditionally, hashing is utilized in packet classification to speed up the process of determining whether an incoming packet matches a certain rule (that in turn determine the action to take on the packet). A solution to decrease the processing time in packet classification is the utilization of Bloom filters [1][5][7][16]. Packet classification using Bloom filter includes two stages: Bloom filter programming and membership checking. In the programming stage, for the given item (rules in packet classification) a Bloom filter computes k hash values using hashing functions. After that, it utilizes these values to set the bits in an m-bit bit-array. For each incoming packet, membership checking is performed. In the standard Bloom filter, k hashing functions are utilized and the m-bit bit-array is checked. If it finds any unset bit at those addresses, it declares that incoming packet is not a member of the set. This is called a *mismatch*. Contrary, if all bits are set then it is likely that the incoming packet is a member of the set. This is called a *match*. In the membership checking stage in a standard Bloom filter all the bits in the bit-array will be checked, but when the first unset bit is encountered (considering a sequential checking) membership checking can be stopped. Consequently, this process can decrease the power consumed by the hashing functions and look up operations in the Bloom filter.

In this paper, we present a k-stage pipelined Bloom filter architecture and the analysis of power consumption. Subsequently, we utilize a software packet classifier to customize the k-stage pipelined Bloom filter architecture in packet classification and analyze the average number of '0's in the bit-array. Finally, we determine the packet mismatch rate for the different packet traces. Our results show that more than 75% of mismatched packets are detected by the first three stages (25% of the remaining packets includes 17% matched packets and 8% mismatched packets). Therefore, we can utilize a 4-stage pipelined Bloom filter for packet classification where in the first three stages each stage includes one hashing function and last stage includes $k-3$ hashing functions that operate in parallel. This paper is organized as follows. Section II presents related work. Section III describes a Bloom filter, its power model and

978-0-7695-3823-5/09 $26.00 © 2009 IEEE
DOI 10.1109/CSE.2009.408

packet classification concepts. Section IV describes our k-stage pipelined Bloom filter architecture and its power model. Section V presents our results and performance evaluation. In Section VI, we draw the overall conclusions.

II. Related Work

In this section, we take a brief look at the previous work regarding power-efficient Bloom filters. In [9][10][11], a two-stage pipelined Bloom filter was proposed for network intrusion detection. It was shown that the lesser number of hashing functions implemented in the first stage of a pipelined Bloom filter, the more the power saving is. The authors also examined three type of hashing functions to observe the effect of power saving. The challenge in the two-stage pipelined Bloom filter is related to determining the number of hashing functions in each stage. For each incoming key, the configuration (number of hashing functions) in each stage should be changed or the number of hashing functions can be fixed. In the first case, some configuration overhead is the result and in the second case the configuration does not operate efficiently. In [14], the power, latency, and area characteristics for two counting Bloom filter implementations using full custom layouts in a commercial 0.13 μm technology was investigated. Their first implementation is based on a SRAM array of counts and the second is based on an array of linear feedback shift register counters. In our approach, we propose a k-stage pipelined Bloom filter for packet classification where each stage only includes one hashing function. Additionally, we perform a power analysis of our proposed Bloom filter. Subsequently, we investigate a software packet classifier and analyze the average number of '0's and mismatched packet rate. Based on these observations, we conclude that a 4-stage pipelined Bloom filter is more beneficial in packet classification.

III. Bloom Filter Concept

In this section, we present the standard Bloom filter and its power consumption model.

A. Standard Bloom Filter

A Bloom filter is a simple space efficient randomized data structure for representing a set in order to support membership queries. Burton Bloom introduced Bloom filters in the 1970s [4]. A set S $(x_1, x_2, ..., x_n)$ of n elements is represented by an array V of m bits that are initially all set to 0. A set of k independent hash functions h_1, h_2, ..., h_k (each with an output range between 1 and m) is utilized to set k bits in array V at positions $h_1(x)$, $h_2(x)$, ..., $h_k(x)$ for all x in set S. More precisely, for each element $x \in S$, the bits at positions $h_i(x)$ are set to 1 for $1 \leq i \leq k$. Moreover, a location can be set to 1 multiple times. To verify whether an item y is a member of the set S, the same set of hash functions is utilized to determine $h_i(y)$ (for $1 < i < k$) indicating the locations in array V to be checked whether their content is a 1. If one of these location yields a 0, y is certainly not a member of the set S. If all locations yield a 1, there is a high probability that y is a member of the set S (*positive*). However, as increasingly more bits in array V are set to 1, one can imagine that the probability of a false positive will increase. It must be clear now that there is an inverse relation between the number of bits in the array and the false positive rate. In the extreme case, when all bits in the array are set, every search will yield a (false) positive. The false positive probability is given as follows [16]:

$$p_f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{\frac{-kn}{m}}\right)^k \qquad (1)$$

In this equation, n represents the number of elements, m represents the number of bits in the bit array and k represents the number of hashing functions. For a given m and n, the value of k (the number of hash functions) that minimizes the probability is as follows:

$$k = \frac{m}{n} ln2 \approx 0.7\frac{m}{n} \qquad (2)$$

B. Hash Based Packet Classification

A high-level approach for multiple field search employs tuple spaces with a tuple representing information in each field specified by the rules. Srinivasan, et al. [17][18][19] introduced the tuple space approach and the collection of tuple search algorithms. We provide a simplified example rule classification on five fields in Table I. Address prefixes cover 32-bit addresses and port ranges cover 16-bit port numbers. For address prefix fields, the number of specified bits is simply the number of non-wildcard bits in the prefix. For the protocol fields, the value is simply a boolean: '1' if a protocol is specified, '0' if a wildcard is specified [19][20]. The number of specified bits in a port range are less straightforward to define. The authors introduced the concept of nesting levels and Range-IDs to define the tuple value for port ranges. The nesting level specifies the layer of the hierarchy and the Range-ID uniquely labels the range within its layer. In this manner, all port ranges can be converted to a (Nesting level, Range-ID) pair. We present in the following an example to illustrate Range-IDs. The full range, in this example (0-65535) always has the id 0. The two ranges at level 1 namely, (0, 1023) and (1024, 65535) in our example receive id 0, and 1, respectively. The example of mapping a port range to a nesting level and a Range-ID for Table I is depicted in Figure 1.

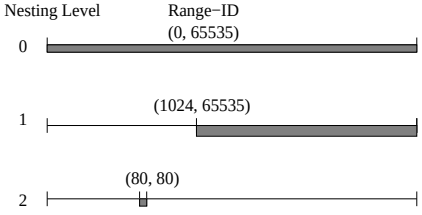

Fig. 1. Assigning values for ranges, based on the nesting level and the Range-ID.

Rules	Destination IP (address mask)	Source IP (address mask)	Port No.	Protocol No.	Tuple space
R1	192.168.190.69 (255.255.255.255)	192.168.80.11 (255.255.255.0)	*	*	[32,24,0,0]
R2	192.168.3.0 (255.255.255.0)	192.168.200.157 (255.255.255.255)	eq www	tcp	[24,32,2,1]
R3	192.168.198.4 (255.255.255.255)	192.168.160.0 (255.255.255.0)	gt 1023	udp	[32,24,1,1]
R4	193.164.0.0 (255.255.0.0)	193.0.0.0 (255.0.0.0)	eq www	udp	[16,8,2,1]
R5	192.168.0.0 (255.255.0.0)	192.0.0.0 (255.0.0.0)	eq www	tcp	[16,8,2,1]
R6	0.0.0.0 (0.0.0.0)	0.0.0.0 (0.0.0.0)	*	*	[0,0,0,0]

TABLE I
SIMPLIFIED EXAMPLE OF RULE CLASSIFICATION.

In the following, we illustrate how a search key is constructed from a packet based on a tuple. A search key for the tuple [8, 24, 2, 0, 1] is constructed by concatenating the first octet of the packet source address, the first three octets of the packet destination address, the Range-ID of the source port, the range at nesting level 2 covering the packet source port number, the Range-ID of the destination port range at nesting level 0 covering the packet destination port number, and the protocol field. Finally, all algorithms using the tuple space approach involve a search of the tuple space or a subset of the tuples in the tuple space.

C. Power Model for Standard Bloom Filter

A Bloom filter consists of a set of hashing functions and a bit-array to lookup. Therefore, the consumed power by the Bloom filter is a summation of consumed power by the hashing functions and the bit-array lookup. The architecture of a standard Bloom filter is depicted in Figure 2.

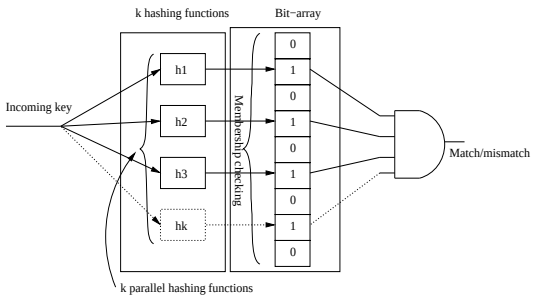

Fig. 2. A standard Bloom filter with k hashing functions.

In Figure 2, a Bloom filter includes k hashing functions that are working simultaneously. In a Bloom filter, all k-bits pointed to by hashing functions are set in the bit-array (m bits) in the programming stage and are checked in the membership checking stage. Universal hashing functions are generally utilized in the Bloom filter. We utilized a class of universal hashing functions that is called H3 hashing functions [2][3][6][13].

A standard Bloom filter utilizes k hashing functions in order to make a decision on the input. Hence, the consumed power by a standard Bloom filter depicted in Figure 2 is a summation of the consumed power by the hashing functions that is represented by Po_{hash}, and the consumed power by bit-array lookup operations that is represented by Po_{lookup}, plus the consumed power by k-input *'and'* operation that is represented by $Po_{Gate_{And_k}}$.

$$Po_{BF_{Standard}} = \sum_{i=1}^{k} (Po_{hash_i} + Po_{Lookup}) + Po_{Gate_{And_k}} \quad (3)$$

In Eq. 3, the consumed power by all hashing functions is equal, therefore, Eq. 3 is rewritten as follows:

$$Po_{BF_{Standard}} = k\,(Po_{hash} + Po_{Lookup}) + Po_{Gate_{And_k}} \quad (4)$$

IV. K-STAGE PIPELINED BLOOM FILTER

In this section, we present the concept of the k-stage pipelined Bloom filter and its analysis. The k-stage pipelined Bloom filter is a Bloom filter that implements its hashing functions in a pipelined manner. A k-stage pipelined Bloom filter architecture is depicted in Figure 3.

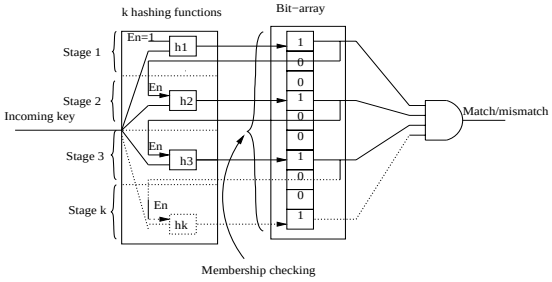

Fig. 3. k-stage pipelined Bloom filter architecture.

Basically, a k-stage pipelined Bloom filter as depicted in Figure 3 consists of k group of hashing functions. Each stage always computes the hash values and the next stage only compute the hash values if in the previous stage there is a match between the input item and the bit-array sought. In Figure 3, *'En'* represents matching in the previous stage and enables the next stage of the pipeline. The advantage of using a k-stage pipelined Bloom filter is if the current stage produces a mismatch, there is no need to use the next stages in order to decide whether the input item is a member of the bit-array. At worst, it will operate like a standard Bloom filter, which utilizes all of the hashing functions before making a decision on the type of the input. The k-stage pipelined Bloom filter is utilized in membership

checking stage, since in the programming stage all of hashing functions are utilized and pipeline stages are permanently full. In a Bloom filter with m bits (bit-array size), n items and k hashing functions, $1/m$ represents the probability any one of the m bits set by a single hashing function operating on a single input item. $(1 - 1/m)$ is the probability that the bit is unset after a single hash value computation with a single item. The probability that a bit is still unset after all the items are programmed into the pipelined Bloom filter by using k independent hashing functions is as follows:

$$P_{unset\ bit} = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{\frac{-kn}{m}} \tag{5}$$

Consequently, the probability that any one of the bits is set is as follows:

$$p_{set} = (1 - p_{unset\ bit}) = 1 - e^{\frac{-kn}{m}} \tag{6}$$

From Figure 3, we can write the consumed power by pipelined Bloom filter as follows:

$$\begin{aligned} Po_{BF_{pipelined}} = \; & Po_{hash_{h_1}} + Po_{lookup_{h_1}} + \\ & p_{set_{h_1}}\left(Po_{hash_{h_2}} + Po_{lookup_{h_2}}\right) \\ & + p_{set_{h_1}} p_{set_{h_2}}\left(Po_{hash_{h_3}} + Po_{lookup_{h_3}}\right) \\ & + \ldots + p_{set_{h_1}} \cdots p_{set_{h_{k-1}}}\left(Po_{hash_{h_k}} + Po_{lookup_{h_k}}\right) \\ & + Po_{Gate_{and_k}} \\ = \; & Po_{hash_{h_1}} + Po_{lookup_{h_1}} + \\ & A\left(Po_{hash_{h_2}} + Po_{lookup_{h_2}}\right) + \\ & A^2\left(Po_{hash_{h_3}} + Po_{lookup_{h_3}}\right) + \ldots + \\ & A^{k-1}\left(Po_{hash_{h_k}} + Po_{lookup_{h_k}}\right) + Po_{Gate_{and_k}} \\ = \; & \sum_{i=1}^{k} A^{i-1}\left(Po_{hash_{h_i}} + Po_{lookup_{h_i}}\right) + \\ & Po_{Gate_{And_k}} \quad with\ A = \left(1 - e^{\frac{-kn}{m}}\right) \end{aligned} \tag{7}$$

In Eq. 7, $p_{set_{h_i}}$ represents the probability of the bit pointed to by the hashing function with index i is set. The consumed power by pipelined Bloom filter is represented as follows:

$$\begin{aligned} Po_{BF_{pipelined}} = \; & \sum_{i=1}^{k}\left(1 - e^{\frac{-kn}{m}}\right)^{i-1}\left(Po_{hash_{h_i}} + Po_{lookup_{h_i}}\right) \\ & + Po_{Gate_{And_k}} \end{aligned} \tag{8}$$

From Eqs. 4 and 8, we can observe that the difference between these equations are related to their coefficients, the coefficient of Eq. 4 is k and coefficient of Eq. 8 is $\sum_{i=1}^{k}\left(1 - e^{\frac{-kn}{m}}\right)^{i-1}$. In other words, the consumed power by the k-stage pipelined Bloom filter is less than the consumed power by the standard Bloom filter if $\sum_{i=1}^{k}\left(1 - e^{\frac{-kn}{m}}\right)^{i-1} < k$ or $\frac{\sum_{i=1}^{k}\left(1 - e^{\frac{-kn}{m}}\right)^{i-1}}{k} \leq 1$. Therefore, we will investigate graph of the following equation in next section.

$$Coefficient\ rate = \frac{\sum_{i=1}^{k}\left(1 - e^{\frac{-kn}{m}}\right)^{i-1}}{k} \tag{9}$$

It should be noted that Eq. 9 (coefficient rate) represents the consumed power by a k-stage pipelined Bloom filter that is normalized to the consumed power by a standard Bloom filter.

V. Results and Performance Evaluation

In this section, we present the simulation and experimental results of our k-stage pipelined Bloom filter architecture. The simulation results were generated based on a mathematical analysis of our architecture using Maple v.12.0 and the implementation results were generated using a software packet classifier[2].

The coefficient rate for configurations $k = ln(2)m/n$ that generates a minimum false positive probability is depicted in Figure 4.

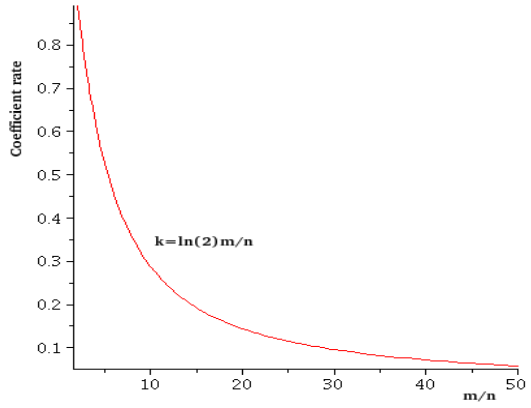

Fig. 4. Coefficient rate in k-stage pipelined Bloom filter for configuration $k = ln(2)m/n$.

From Figure 4, we can observe that the growth of the number of hashing functions increases the power saving rate. The simulation results show that the utilization of pipelining in the Bloom filter decreases power consumption in compared to a standard Bloom filter. To realize the idea, we implemented a software packet classifier and analyzed real packet traces to examine the capability of the proposed solution. For testing purposes, we utilize different rule-set databases and packet traces that have been used by the Applied Research Laboratory in Washington University in St. Louis [15]. The specification of the rule-set databases and packet traces is presented in Table II.

Table II includes seven rule-set databases and correspondent packet traces. The rule-sets FW1, ACL1, and IPC1 are extracted from real rule-sets and others generated by the Classbench benchmark. More details on Classbench, rule-set databases, and packet traces can be found in[15]. We utilize a software packet classifier and determine the average number of '0's in the bit-array of the Bloom filter that belonged to the biggest tuple. Our observations in the execution of the software packet classifier show that when the tuples are created the biggest tuple contains more than

Rule Database	FW1-100	FW1-1k	FW1-5k	FW1-10k	FW1	IPC1	ACL1
Number of rules	92	971	4653	9311	266	1550	752
Number of tuples	26	42	52	57	36	179	44
Packet trace	**FW1-100**	**FW1-1k**	**FW1-5k**	**FW1-10k**	**FW1**	**IPC1**	**ACL1**
Number of Packets	920	8050	46700	93250	2830	17020	8140

TABLE II

RULE SET DATABASE AND PACKET TRACE SPECIFICATION.

half of the rules. It should be noted that the software packet classifier was tested and results were generated for different packet traces and their correspondent rule-set databases. The average number of '0's pointed to by hashing functions in the bit-array for different packet traces is depicted in Figure 5.

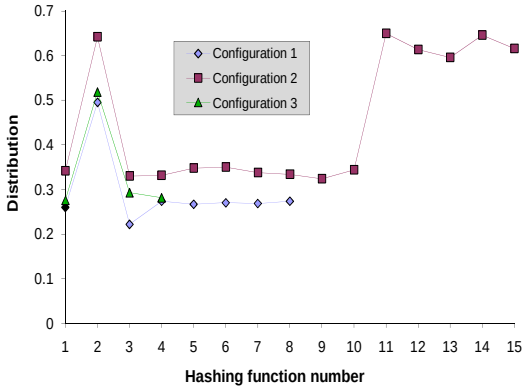

Fig. 5. Average number of '0's in the bit-array of Bloom filter of biggest tuple using software packet classifier for three different Bloom filter configurations in the membership checking stage ('*configuration 1*' with 8 hashing functions, '*configuration 2*' with 15 hashing functions and '*configuration 3*' with 4 hashing functions).

In this figure, three configurations are represented. In all of the configurations, the size of bit-array to have a minimum false positive probability is computed from Eq. 2. Since for the given number of hashing functions and the number of rules, the size of bit-array in Bloom filter is computed. '*configuration 1*' represents a Bloom filter with 8 hashing functions and bit-array size $m = 11.2n$ (n is the number of rules). '*configuration 2*' represents a Bloom filter with 15 hashing functions and bit-array size $m = 21.6n$ and '*configuration 3*' represents a Bloom filter with 4 hashing functions and bit-array size $m = 5.1n$. Based on the graphs in Figure 5, we can observe that for different configurations there are some zero's in the bit-array of Bloom filter that pointed to by hashing functions in the membership checking stage. Since after the bit-array creation in the Bloom filter programming stage using rule-set databases, we check the different bits in the bit-array pointed to by hashing functions in the membership checking stage. After the utilization of all packet traces for each configuration, we normalize the number of zero's for each packet trace to the number of packets in a packet trace and compute the average number of zero's in all packet traces. Therefore, we conclude that in the Bloom filter in membership checking stage there are some zero bits in the bit-array that are pointed to by hashing functions. We can decrease the power consumption in the Bloom filter when it is utilized in packet classification by controlling hashing functions. A question that should be addressed is related to designing of the pipelined Bloom filter architecture. How can the number of zero's be utilized in the design of the pipeline architecture? We investigate the number of mismatched packets in the different bits (pipeline stages) in the bit-array of Bloom filter in the membership checking stage. The mismatched packet detection rate is depicted in Figure 6.

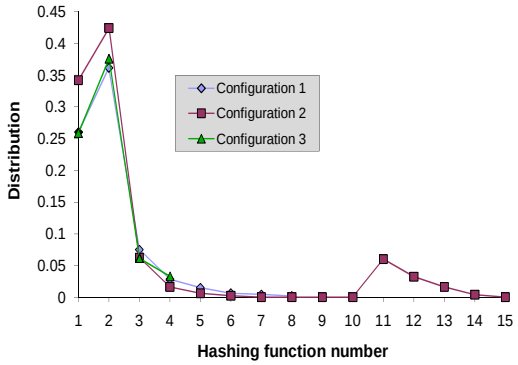

Fig. 6. Average mismatched packets rate for all packet traces in the biggest tuple using software packet classifier for three different Bloom filter configurations.

Based on Figure 6, we can observe that more than 75% of mismatched packets are detected by the first three stages and 17% of packets are matched against of rules and 8% reminder are mismatched that are detected by other pipeline stages. In the second configuration, we can observe that when the hashing function number is changed between 11 and 13 some mismatched are detected. To overcome the problem, we can utilize an additional pipeline stage in the architecture to include other hashing functions which detects the rest of mismatched packets. In the other words, a 4-stage pipelined Bloom filter architecture is useful for packet classification where the first three stages includes first three hashing functions and last stage includes last $k - 3$ hashing functions. The architecture of a 4-stage pipelined Bloom filter is depicted in Figure 7.

In this figure, we can observe that the first three stages

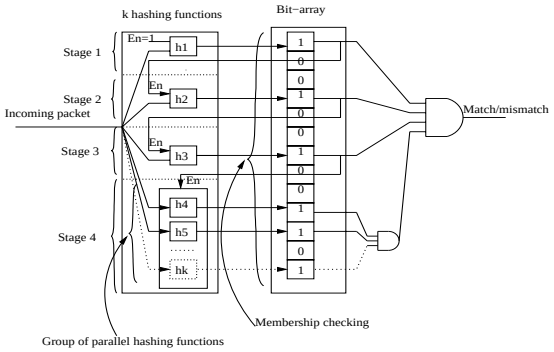

Fig. 7. Our 4-stage pipelined Bloom filter architecture where the first three stages contains one hashing function and the forth stage contains $k-3$ hashing functions that operate in a parallel manner.

that are most frequently used includes only one hashing function and last stage includes other $k-3$ hashing functions. This fact decreases power consumption in comparison to a standard Bloom filter and pipeline latency in comparison to k-stage pipelined Bloom filter architecture. Similar to the power model of the k-stage pipelined Bloom filter the power model for the 4-stage pipelined Bloom filter is as follows:

$$
\begin{aligned}
Po_{BF_{pipelined_4}} = \quad & Po_{hash_{h_1}} + Po_{lookup_{h_1}} + \\
& p_{set_{h_1}} \left(Po_{hash_{h_2}} + Po_{lookup_{h_2}} \right) + \\
& p_{set_{h_1}} p_{set_{h_2}} \left(Po_{hash_{h_3}} + Po_{lookup_{h_3}} \right) + \\
& p_{set_{h_1}} p_{set_{h_2}} p_{set_{h_3}} \left(Po_{hash_{h_4}} + Po_{lookup_{h_4}} \right) + \cdots \\
& + p_{set_{h_1}} p_{set_{h_2}} p_{set_{h_3}} \left(Po_{hash_{h_k}} + Po_{lookup_{h_k}} \right) + \\
& Po_{Gate_{and_4}} + Po_{Gate_{and_{k-3}}} \\
= \; & Po_{hash_{h_1}} + Po_{lookup_{h_1}} \\
& + A \left(Po_{hash_{h_2}} + Po_{lookup_{h_2}} \right) \\
& + A^2 \left(Po_{hash_{h_3}} + Po_{lookup_{h_3}} \right) \\
& + A^3 \left(Po_{hash_{h_4}} + Po_{lookup_{h_4}} \right) + \ldots + \\
& A^3 \left(Po_{hash_{h_k}} + Po_{lookup_{h_k}} \right) + \\
& Po_{Gate_{and_4}} + Po_{Gate_{and_{k-3}}} \\
= \; & \sum_{i=1}^{3} A^{i-1} \left(Po_{hash_{h_i}} + Po_{lookup_{h_i}} \right) + \\
& \sum_{i=4}^{k} A^3 \left(Po_{hash_{h_i}} + Po_{lookup_{h_i}} \right) + \\
& Po_{Gate_{and_4}} + Po_{Gate_{and_{k-3}}} \\
& with \; A = \left(1 - e^{\frac{-kn}{m}} \right)
\end{aligned}
\tag{10}
$$

In Eq. 10, if we the consumed power by '$Gate_{and_4}$' and '$Gate_{and_{k-3}}$' (the consumed power by 4-input and (k-3)-input'*and*' gates) is equal to the consumed power by '$Gate_{and_k}$' then Eq. 10 is rewritten as follows:

$$
\begin{aligned}
& Po_{BF_{pipelined_4}} = \left(\sum_{i=1}^{3} \left(1 - e^{\frac{-kn}{m}} \right)^{i-1} + \sum_{i=4}^{k} \left(1 - e^{\frac{-kn}{m}} \right)^{3} \right) \\
& \left(Po_{hash_{h_i}} + Po_{lookup_{h_i}} \right) + Po_{Gate_{and_k}}
\end{aligned}
\tag{11}
$$

From Eqs. 4, 8, and 11, that represent the consumed power for the standard, k-stage, and 4-stage Bloom filters, respectively, we can observe that the difference between them is related to their coefficients. The graph of coefficients of the k-stage pipelined Bloom filter and 4-stage pipelined Bloom filter that are normalized to the coefficients of a standard Bloom filter is depicted in Figure 8.

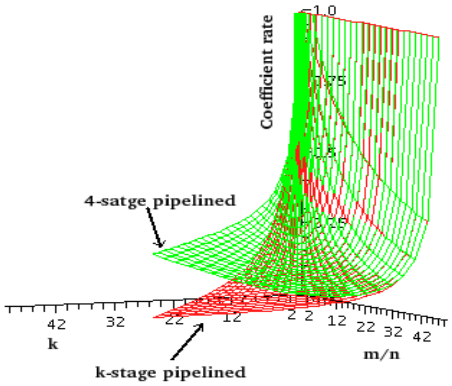

Fig. 8. Coefficient rate in k-stage pipelined Bloom filter and 4-stage pipelined Bloom filter for different configurations in terms of k and m/n ($k \leq m/n$) that are normalized to the coefficients of a standard Bloom filter.

From Figure 8, we can observe that the consumed power of the 4-stage pipelined Bloom filter is more than the k-stage pipelined Bloom filter but less than the standard Bloom filter.

VI. Overall Conclusions

In this paper, we presented a k-stage pipelined Bloom filter architecture to decrease the power consumption in packet classification. The presented results show that the pipelined Bloom filter architecture decreases power consumption in comparison to the standard Bloom filter. Our observation of the software packet classifier for real packet trace s shows that the first three stages of the pipelined Bloom filter detects most of the mismatch packets, therefore, a 4-stage pipelined Bloom filter is sufficient to classify packets. The 4-stage pipelined Bloom filter is more appropriate than standard Bloom filter when the power consumption is critical. We expect this approach to be useful in the design of low power memory and packet classifier architectures utilized in network processors. These low power network processors can be utilized in the ad-hoc networks.

References

[1] M. Ahmadi and S. Wong. "A Cache Architecture for Counting Bloom Filters". In *Proc. of 15'th IEEE Int. Conf. on Networks (ICON2007)*, pages 218–223, November 2007.

[2] M. Ahmadi and S. Wong. "Modified Collision Packet Classification Using Counting Bloom Filter in Tuple Space". In *Proc. of the 25'th IASTED Int. Conf. on Parallel and Distributed Computing and Networks (PDCN 2007)*, pages 70–76, February 2007.

[3] M. Ahmadi and S. Wong. "A Memory-optimized Bloom Filter Using An Additional Hashing Function". In *Proc. of IEEE Globecom 2008 Next Generation Networks, Protocols, and Services Symposium*, pages 2479–2483, December 2008.

[4] B. H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". *Communication of the ACM*, 13(7):422–426, July 1970.

[5] A. Broder and M. Mitzenmacher. “Network Applications of Bloom Filters: A Survey”. In *Proc. of 14’th Annual Allerton Conf. on Communication, Control, and Computing*, pages 636–646, October 2002.

[6] J. Lawrence Carter and Mark N. Wegman. “Universal Classes of Hash Functions”. In *Proc. of the 9’th annual ACM symp. on Theory of Computing*, pages 106–112. ACM Press, 1977.

[7] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood. “Fast Packet Classification Using Bloom Filters”. In *Proc. of the ACM/IEEE Symp. on Architecture for Networking and Communications Systems*, pages 61–70. ACM, 2006.

[8] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. “Summary Cache: A Scalable Wide-Area (WEB) Cache Sharing Protocol”. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[9] I. Kaya and T. Kocak. “A Low Power Lookup Technique for Multi-Hashing Network Applications”. In *Proc. of the IEEE Computer Society Annual Symp. on Emerging VLSI Technologies and Architectures*, page 179, 2006.

[10] I. Kaya and T. Kocak. “Energy-efficient Pipelined Bloom Filters for Network Intrusion Detection”. In *IEEE Int. Conf. on Communications (ICC06)*, pages 2382–2387, June 2006.

[11] I. Kaya and T. Kocak. “Low-power Bloom Filter Architecture for Deep Packet Inspection”. *IEEE Communications Letters*, 10(3):210–212, March 2006.

[12] S. Kumar and P. Crowley. “Segmented Hash: An Efficient Hash Table Implementation for High Performance Networking Subsystems”. In *Proc. of the Symp. on Architecture for Networking and Communications Systems (ANCS05)*, pages 91–103, October 2005.

[13] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. “Efficient Hardware Hashing Functions for High Performance Computers”. *IEEE Transaction on Computer*, 46(12):1378–1381, 1997.

[14] E. Safi, A. Moshovos, and A. Veneris. “L-CBF: a Low-power, Fast Counting Bloom Filter Architecture”. In *In Proc. of the Int. Symp. on Low Power Electronics and Design*, pages 250–255, 2006.

[15] H. Song. “Evaluation of Packet Classification Algorithms”. http:// www.arl.wustl.edu/ hs1/PClassEval.html, 2006.

[16] H. Song, J. Turner, S. Dharmapurikar, and J. Lockwood. “Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing”. In *Proc. of the Int. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 181–192, August 2005.

[17] V. Srinivasan. “*IP Lookup and Packet Classification*”. PhD thesis, Washington University, August 1999.

[18] V. Srinivasan. “A Packet Classification and Filter Management System”. In *Proc. of the Int. IEEE Conf. INFOCOM*, pages 1464–1473, 2001.

[19] V. Srinivasan, S. Suri, and G. Varghese. “Packet Classification Using Tuple Space Search”. In *Proc. of the Int. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 135–146, 1999.

[20] D. E. Taylor. “*Models, Algorithms, and Architectures for Scalable Packet Classification*”. PhD thesis, Department of Computer Science and Engineering Washington University, August 2004.