Paynter, M., & Koçak, T. (2008). Fully pipelined bloom filter architecture. IEEE Communications Letters, 12(11), 855 - 857. 10.1109/LCOMM.2008.081176

Fully Pipelined Bloom Filter Architecture

Michael Paynter and Taskin Kocak

***Abstract*—Recently, we proposed a two-stage pipelined Bloom filter architecture to save power for network security applications. In this letter, we generalize the pipelined Bloom filter architecture to k-stage and show that significant power savings can be achieved by employing one hash function per stage. We analytically show that the expected power consumption and latency of the fully pipelined Bloom filter architecture will not be greater than that of the two hash functions and two clock cycles, respectively, however large the number of hash functions is. Furthermore, we discuss the worst-case performance of the proposed architecture.**

***Index Terms*—Bloom filters, network intrusion detection, low-power design.**

I. Introduction

BLOOM filters have recently received considerable attention in the networking research community for many applications such as network intrusion detection, peer-to-peer networking, resource routing, traffic management [1]. Bloom filters were first proposed by B. Bloom in 1970 [2] and were originally popular in database applications. They trade off a small, known false positive rate (FPR) for high space efficiency and are suitable for string matching applications where the effects of false positives can be mitigated.

A Bloom filter is a simple way of representing a set and testing strings for membership of that set. For example, in network intrusion detection systems this set will be the Snort rules [6]. A Bloom filter represents the set of n signatures in an m-bit array. The elements in this array are set to '0' before programming. Each signature is hashed k times by the independent hash functions, $h_1, h_2, ..., h_k$. It is assumed that each hash function maps uniformly to a random number in the range $\{0, 1, ..., m-1\}$. This number indicates a bit location in the m-bit array which is then set to '1'. If a bit is set to '1' more than once there is no additional effect and the bit remains set to '1'. To test a string for membership of the programmed Bloom filter, the query string, y, is hashed k times as before. If all of the hashes point to the bit locations that are set to '1' (i.e., match) then this indicates that the query string is a member of the set. If any of the hashes point to a bit location that is set to '0' (i.e., mismatch) then the query string is definitely not a member of the set.

There is a possibility that a non-member string might select all '1's from the programmed array. This scenario yields a *false positive*. This might be acceptable if the probability is small enough or if the effects of false positives can be mitigated. Following the analysis of [3], the probability of a false positive, f, can be estimated from the Bloom filter parameters:

$$f = \left(1 - e^{\frac{-kn}{m}}\right)^k \tag{1}$$

And the number of hash functions that minimizes f is given by

$$k = \left(\frac{m}{n}\right) \ln 2 \tag{2}$$

Regular Bloom filters are usually implemented in hardware, as shown in Fig. 1, with multiple, single-ported lookup memories due to the lack of k-port memory devices [7]. The m-bit array is split such that each hash function accesses its own array of m/k bits, separate from the others to minimize the access latency, which would otherwise increases to k times for a single-port m-bit lookup memory. This architecture suffers from a greater FPR than the regular architecture with m-bit array; however, this is shown to be negligible in the asymptotic case and in practice [1].

The number of hash functions required to minimize the false positive probability of a Bloom filter is typically large, for example k=35. In some applications, such as network intrusion detection due to very low rate of malicious traffic (e.g., 0.1% [8]), there is no need to compute all hash functions to get a result of non-membership. Recently, we introduced *pipelined Bloom filters* to exploit this [4]. Basically, a pipelined Bloom filter consists of several groups of hash functions that are utilized in different stages. The first stage always computes the hash values. The second and further stages are used only if there is a match in the previous stage(s). In [4], we demonstrated that a two-stage pipelined Bloom filter can yield significant power savings. In this paper, we generalize

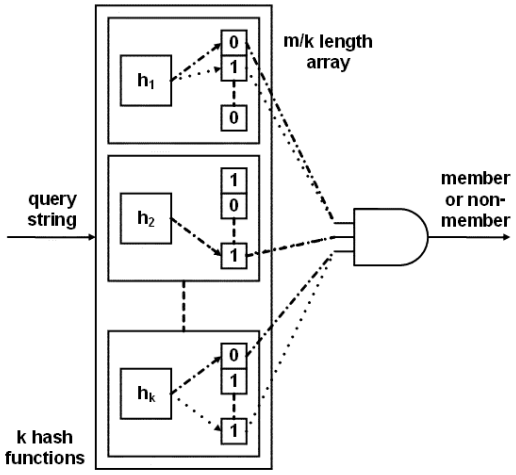

Fig. 1. Hardware implementation of a regular Bloom filter.

Manuscript received July 23, 2008. The associate editor coordinating the review of this letter and approving it for publication was A. Shami.

The authors are with the Department of Electrical and Electronic Engineering, University of Bristol, Bristol, BS8 1UB, United Kingdom (e-mail: {mp3379, t.kocak}@bristol.ac.uk).

Digital Object Identifier 10.1109/LCOMM.2008.081176

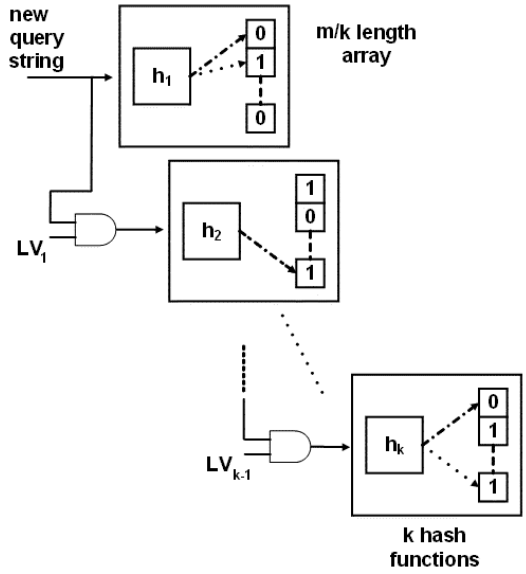

Fig. 2. Hardware implementation for the fully pipelined Bloom filter architecture.

the pipelined Bloom filters and introduce the fully pipelined architecture for Bloom filters.

II. Fully Pipelined Bloom Filter Architecture

The architecture of the fully pipelined Bloom filter is shown in Fig. 2. By fully pipelined, it is meant that each stage has only one hash function unlike the two-stage version in [4] where there are many hash functions per stage. The fully pipelined Bloom filter has the same number of hash functions as the regular Bloom filter. Hence, the false positive probability is the same. In the query phase, the first hash function, h_1, is fed by a new query string every cycle. A query string is progressed to the next stage only when the previous hash function produces a match (i.e., lookup value, LV_i=1). The programming phase is the same as the regular Bloom filter.

A. Power Consumption Analysis

The main benefit of pipelining the Bloom filter is saving power. In [4], we showed that two-stage pipelining helps to reduce power consumption up to 90 percent if there is a fixed combination of, say, 5 hash functions in the first stage and the rest of the hash functions in the second stage. However, the power saving ratio diminishes when there are high number of matches in the first stage and the second stage is utilized more. Thus, the fully pipelined architecture is a remedy to this problem. In the following, we will derive the power savings achieved by the fully pipelined architecture. By using the same notation in [4], the power consumption of a regular Bloom filter is given by

$$P_{regular} = \sum_{i=1}^{k} (P_{H_i} + P_L) = k.(P_H + P_L) \tag{3}$$

where P_H is the power dissipated by a hash function and P_L is the power required to perform lookup on the array. Without loss of generality, the power consumption of each hash function, P_{H_i}, is assumed to be equal. The power dissipated by the AND gate is also considered to be negligible.

The average power dissipation of the fully pipelined Bloom filter can be calculated as follows:

$$P_{pipelined} = P + P \cdot s + ... + P \cdot s^{k-1} \tag{4}$$

where $P = P_H + P_L$ is the power dissipated by a single stage and $s = 1 - e^{\frac{-kn}{m}}$ is the probability of a hash function selecting a '1' in the lookup array [4]. Here it is assumed that the input stage of the pipeline is continuously filled with another query string. Eq. (4) can be rearranged to

$$P_{pipelined} = P \cdot \sum_{i=1}^{k} s^{i-1} \tag{5}$$

This can be expanded using the values for P and s

$$P_{pipelined} = (P_H + P_L) \cdot \sum_{i=1}^{k} \left[\left(1 - e^{\frac{-kn}{m}} \right)^{i-1} \right] \tag{6}$$

It is interesting to examine the average power dissipated by an optimal Bloom filter (i.e., the FPR is minimized). This can be shown by examining the series generated by the summation. Following the analysis in [1], an optimal Bloom filter exists when $1 - e^{\frac{-kn}{m}} = 0.5$. The Eq. 6 becomes

$$P_{pipelined} = (P_H + P_L) \cdot \sum_{i=1}^{k} 0.5^{i-1} = (P_H + P_L) \cdot \sum_{i=0}^{k-1} \left[\frac{1}{2^i} \right] \tag{7}$$

Now, let us examine the behavior in the limit as $k \to \infty$.

$$P_{pipelined} = (P_H + P_L) \cdot \left[1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... + \frac{1}{2^{k-1}} \right] \tag{8}$$

By using the geometric series given in Eq. (9) [5]

$$\frac{1}{1-x} = \sum_{j=0}^{\infty} x^j = 1 + x + x^2 + ... \quad \text{for} \quad |x| < 1 \tag{9}$$

and setting $x = s = \frac{1}{2}$, the expected power dissipation of the fully pipelined Bloom filter is at most

$$P_{pipelined} = (P_H + P_L) \cdot 2 \tag{10}$$

The exciting conclusion from this is that however large the number of hash functions, k, there are in a fully pipelined Bloom filter, the expected power dissipated will not be greater than that of two hash functions.

B. Latency Analysis

A similar argument can be applied to calculate the average latency of the fully pipelined Bloom filter architecture:

$$\tau_{pipelined} = \tau + \tau \cdot s + ... + \tau \cdot s^{k-1} = \tau \cdot \sum_{i=1}^{k} s^{i-1} \tag{11}$$

where τ is the clock period.

As before if it is assumed that the Bloom filter is optimal then

$$\tau_{pipelined} = \tau \cdot \sum_{i=1}^{k} 0.5^{i-1} = \tau \cdot \sum_{i=0}^{k-1} \left[\frac{1}{2^i} \right] \tag{12}$$

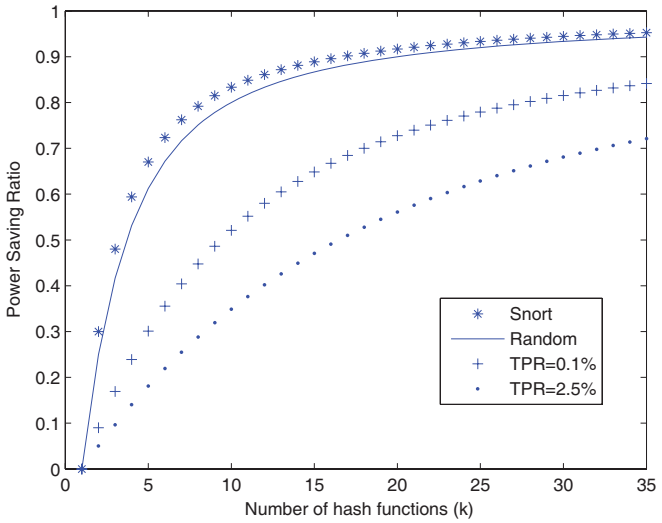

Fig. 3. Power saving ratio for a fully pipelined Bloom filter.

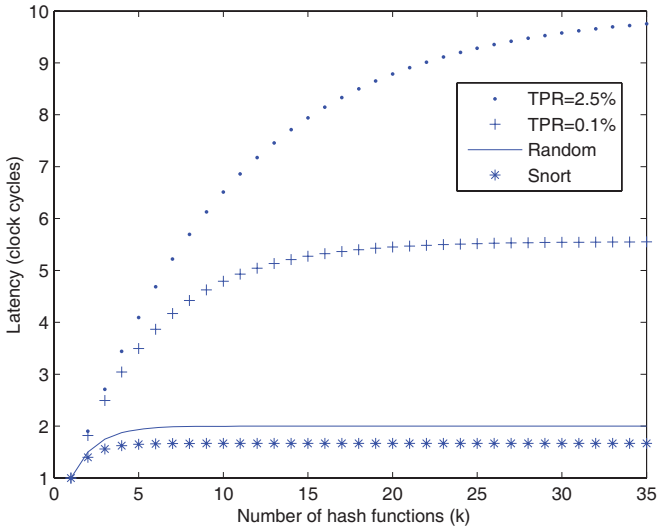

Fig. 4. Latency of a fully pipelined Bloom filter.

This result is similar to the result for expected power dissipation given by Eq. (7). It can be shown in the same way that the expected latency of a fully pipelined Bloom filter will tend towards 2 clock cycles in the limiting case, as $k \to \infty$. This is also exciting because it means that the expected latency of a fully pipelined Bloom filter will never exceed 2 clock cycles however large the number of hash functions, k, is.

III. Discussion

The false positive rate is dependent on the number of '1's or '0's in the lookup array after the programming. The number of '1's will be concentrated tightly around the mean (i.e. 50% or s=0.5) with independent uniform hash functions and a large programming input set [1]. This is the value used for random programming and query, and follows the expected results given in the previous section. We implemented three hash functions, H_3, XOR and Bit Extraction [9], and programmed Snort signatures [6]. The number of '1's as a percentage of the lookup array escalates from 38% to 42%. Hence, we use an average of s=0.4. Another important factor is the true positive rate (TPR), which could be higher than FPR. The majority of the network traffic is still very clean (i.e. malicious free), hence the alert rate of a NIDS system is very low such as 0.1% [8]. We take this and another, but a higher value, say, up to 2.5% malicious traffic to compare the power and latency results for the fully pipelined architecture. The corresponding s values are obtained by setting s^{35}=0.001, which yields s=0.82 for TPR=0.1% and similarly s=0.9 for TPR=2.5%.

In order to assess the improvements in power consumption, we calculate the power saving ratio (PSR)

$$PSR = \frac{(P_{regular} - P_{pipelined})}{P_{regular}} \tag{13}$$

Using Eqs. (3) and (5), and simplifying with P

$$PSR = \frac{\left(P \cdot k - P \cdot \sum_{i=1}^{k} s^{i-1}\right)}{P \cdot k} = 1 - \frac{1}{k} \cdot \sum_{i=1}^{k} s^{i-1} \tag{14}$$

As illustrated in Fig. 3, the PSR increases for increasing k. For large values of k, over 95% power savings can be achieved by the fully pipelined version. In the worst case, the power consumption of the fully pipelined architecture will be as high as that of the regular Bloom filter.

Normally, due to the very low probability of hitting malicious network traffic, the expected latency will be confined within two clock cycles. However, in the case of high TPR, the average latency will increase as shown in Fig. 4. And in the worst case, the query latency could be k times longer than that of the regular Bloom filter.

IV. Conclusions

In this letter, we proposed a fully pipelined Bloom filter architecture. It is targeted for network security applications, owing to the non-malicious input characteristics most of the time; however, it could be used for other applications. Analytically, we demonstrated that on average the fully pipelined Bloom filter will consume power no more than that of two hash functions and its latency will never exceed two clock cycles. Over 95% power savings can be achieved, though, the query latency could be longer with high TPR.

References

[1] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: a survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485-509, 2005.
[2] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422-426, 1970.
[3] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, "Deep packet inspection using parallel Bloom filters," *IEEE Micro*, vol. 24, no. 1, pp. 52-61, 2004.
[4] T. Kocak and I. Kaya, "Low-power Bloom filter architecture for deep packet inspection," *IEEE Commun. Lett.*, vol. 10, no. 3, pp. 210-212, 2006.
[5] E. Kreyszig, *Advanced Engineering Mathematics*, 8th Edition. New York: John Wiley & Sons, Inc., 1999.
[6] Sourcefire, Inc., "Official Snort ruleset," Columbia, MD (web: http://www.snort.ort/pub-bin/downloads.cgi).
[7] D. Sanchez, L. Yen, M. Hill, and K. Sankaralingam, "Implementing signatures for transactional memory," in *Proc. 40th IEEE/ACM Int'l Symp. on Microarchitecture*, Chicago, IL, 2007.
[8] K. Hwang, Y. Chen, and H. Liu, "Defending distributed systems against malicious intrusions and network anomalies," in *Proc. IEEE Parallel and Distributed Processing Symp.*, Denver, CO, 2005.
[9] I. Kaya and T. Kocak, "A low power lookup technique for multi-hashing network applications," in *Proc. IEEE Annual Symp. on VLSI (ISVLSI)*, Karlsruhe, Germany, 2006.