International Journal of Applied Engineering Research ISSN 0973-4562 Volume 12, Number 21 (2017) pp. 10541-10545

A Robust Bloom Filter

Yoon-Hwa Choi
Department of Computer Engineering, Hongik University,
Seoul, Korea.
Orcid: 0000-0003-4585-2875

Abstract

A Bloom filter is a space-efficient randomized data structure representing a set for membership queries. Faults in Bloom filters, however, cannot guarantee no false negatives. In this paper, we present a simple redundancy scheme for detecting false negatives and tolerating false positives induced by faults in Bloom filters during normal operation. A spare hashing unit with a simple coding technique is employed in the robust Bloom filter to tolerate faults without significant hardware overhead. Re-computing with shifted operands is used to locate the faulty unit upon detecting a fault. Both permanent and transient faults are considered in developing the fault diagnosis scheme. In the event of a permanent fault, the proposed Bloom filter is to operate in a degraded mode with some increase in false positive probability.

Keywords: Robust Bloom filter; faults; fault detection; fault location

INTRODUCTION

Bloom filters have been used for various applications, such as packet filtering and routing, web cache sharing, and string matching [1][2]. Some other applications include error detection and correction or defect tolerance in various memory arrays or data sets [3-5]. A Bloom filter is a compact data structure representing a set of *n* elements to support membership queries. The advantage of Bloom filters is that it provides a tradeoff between the space requirement and the false positive probability [6-8]. In addition, no false negatives can be guaranteed.

Although Bloom filters can greatly reduce the memory space required for representing a set, they are desired to be robust and fault-tolerant in some critical applications. In most applications of Bloom filters, a small false positive probability can be tolerated. Hence faults in Bloom filters do not cause a problem as long as the resulting false positive probability still lies in the acceptable range. False negatives, induced by faults, however, need to be detected and corrected.

Bloom filters are often implemented in hardware to achieve high performance [9][10]. In those cases, both permanent and transient faults might occur in the components, such as hashing units and bit vectors, composing the Bloom filters, during normal operation. Faults in Bloom filters might cause a fatal error since no false negatives cannot be guaranteed. Hence it is desirable to achieve robustness to faults unless it requires significant hardware and time overhead.

Fault detection and tolerance in Bloom filters has been considered in [11][12]. In [11] a hardware-based scheme for tolerating faults in Bloom filters for deep packet inspection was presented. In [12] a scheme to protect the contents of Bloom filters for soft errors was proposed. They focused on tolerating soft errors in a bit-vector storing signatures, while achieving a protection level similar to that of single error correction (SEC) code.

In this paper, we present an efficient scheme for designing hardware Bloom filters robust to faults. It detects false negatives due to faulty hashing units using a single spare hashing unit, while employing some existing coding technique for the bit-vector data structure. To locate the faulty hashing unit upon detecting a fault the scheme uses re-computing with shifted operands. The proposed Bloom filter resumes normal operation by logically isolating the identified faulty hashing unit with some increase in false positive probability.

BLOOM FILTER

A Bloom filter for representing a set $S=\{s_1,s_2,....,s_n\}$ of *n* elements is a bit vector M of *m* bits, initially all set to 0 to make it empty, with *k* independent hash functions h_1, h_2,...,h_k that map each element of the set S to the set {0,1,2,...,*m*-1}. A typical Bloom filter is shown in Fig. 1.

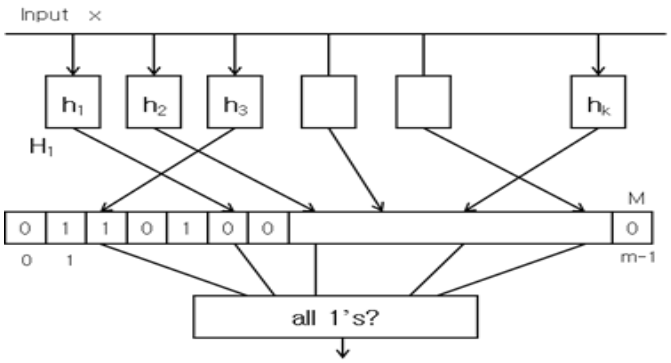

Figure 1: A Bloom Filter

For each element sϵS, the Bloom filter produces *k* hash values $h_i(s)$, $1 \leqq i \leqq k$, ranging from 0 to *m*-1, and sets the corresponding bits in the *m*-bit vector M to 1. To see if a given input *x* belongs to S, it checks whether all the bits (in M) at the locations corresponding to the *k* hash values for *x* are set to 1. If the condition is satisfied, it accepts *x* as a member of S with some

false positive probability. On the other hand, if any of the mapped locations is zero, *x* is determined to be a non-member with certainty. Thus a Bloom filter can lead to false positives. However, it will never return a false negative, where an item is rejected while it actually belongs to the set.

Faults in a Bloom filter might produce a wrong hash value such that the decision made on a membership query can be different from the expected one. In Fig. 1, for example, the hash value $h_3(x)=2$ and the resulting M[2] is equal to 1. If the hash unit for h_3 is faulty and thus returns an incorrect hash value, say 3, as opposed to 2, the resulting lookup ends up with a 0. Hence no-false-negatives cannot be guaranteed unless some provisions are made to detect the fault. Faults in a Bloom filter may also affect false positive probability. However, the changes in the probability are manageably small, and hence cannot be of concern.

In the following section, we define our fault model for the proposed robust Bloom filter and present how to deal with the faults to prevent false negatives.

ROBUST BLOOM FILTER

In order to tolerate faults in Bloom filters, we modify the design with *k* hash units as shown in Fig. 2, where a spare hash unit, H_s, some parity bits for error detection/correction in the bit-vector M, not shown in the dotted block, and a selection and diagnostic unit (SD) for false-negative detection are added. The *k* hashing units, H_1, H_2,..., H_k, normally perform the hash functions, $h_1,h_2,...,h_k$, respectively. For fault location, however, they can be arranged to also perform hash functions, $h_2,h_3,....,h_k,h_1$, respectively (i.e., left-shifted by one hashing unit) for recomputing, to be explained shortly.

The role of the selection unit in SD is to select a hashing unit H_i with $M[v_i]=0$ for a membership query, where v_i is the hash value returned from H_i. The hash function h_i of the selected hashing unit H_i is then realized in the spare hashing unit, H_s, for comparison. A mismatch in the comparison indicates that there is a fault in the Bloom filter, in either H_i or H_s, if we assume that there is a single fault. If H_s is faulty, the decision made on a membership query by the Bloom filter is correct. If the selected hashing unit H_i is faulty, on the other hand, the decision on the query might be wrong, depending on the number of 0's in the lookups on the bit vector M.

This comparison test, however, will be shown to be sufficient to detect all false negatives in the Bloom filter in Fig. 1. In the case where there is a single 0 (i.e., all the remaining *k*-1 hash functions point to 1 bits) and the corresponding hashing unit is faulty, the robust Bloom filter readily detects the false negative, and it thus can correct the fatal error right away. If multiple hash functions point to 0 bits in the bit vector M, although the robust Bloom filter does not check it, the mismatch in comparison has nothing to do with false negative, but indicates the presence of a fault in the Bloom filter. The Bloom filter, in that case, functions correctly even with the faulty hashing unit until it causes a false negative to occur.

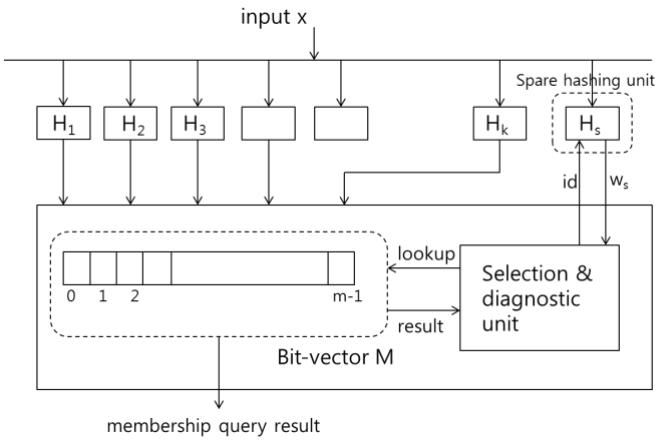

Figure 2: A Robust Bloom Filter

A. Fault Model

The fault model we use here is functional and thus independent of the internal structure of the hashing units. We assume that there is a single fault in the Bloom filter. That is, if a hashing unit is faulty, all other hashing units, including the spare hashing unit, are fault-free during the diagnosis process. This assumption is based on the fact that the mean-time-between-failure (MTBF) is expected to be much larger than the membership query interval of the Bloom filter. The bit-vector M may fail but some well-known coding techniques, such as parity codes, can be used locally and independently for error detection and correction [13]. In [12] a scheme for protecting the bit-vector M from soft errors has been proposed, and it can also be employed for tolerating soft errors. Hence in this paper we focus on faults in hardware hashing units, assuming that faults in the bit vector M can be detected and corrected locally. Comparators are assumed to be fault-free, since they are relatively simple to implement. Self-checking comparators may be used instead.

Both transient and permanent faults in hashing units are considered in achieving robustness. In the case of a transient fault, the corresponding hashing unit might generate an incorrect hash value, possibly leading to a wrong decision. It, however, has only transient effects, and can thus behave normally after that without any recovery action being taken. On the other hand, hashing units with some permanent faults need to be isolated from the rest of the Bloom filter to avoid unnecessary time delays to deal with any subsequent errors caused by the faults. Moreover, the potential occurrence of multiple faults due to delayed isolation of faulty hash units makes the diagnosis much more difficult, if not impossible.

B. Fault Detection

Let S be a set of n elements for the robust Bloom filter and h_i be the i-th ($1 \leqq i \leqq k$) hash function. For a given input x the hash value of a faulty hashing unit H_i is denoted by $g_i(x)$ ($\neq h_i(x)$). Then we need to consider the following two cases depending on the input x.

(1) Input x is an element of S

The faulty hashing unit H_i produces a random hash value $g_i(x)$ between 0 and m-1. If $M[g_i(x)]=0$, as opposed to 1, a false negative occurs.

(2) Input x is not an element of S

(a) The faulty hashing unit H_i, where $M[h_i(x)]=0$, might return an incorrect hash value $g_i(x)$ such that $M[g_i(x)]=1$ (as opposed to 0). If $M[h_j(x)]=1$ for ($1 \leqq j \leqq k$ and $j \neq i$), a false positive occurs due to the faulty hashing unit H_i.

(b) The faulty hashing unit H_i, where $M[h_i(x)]=1$, might produce an incorrect hash value $g_i(x)$ such that $M[g_i(x)]=0$ (as opposed to 1). If $M[h_j(x)]=1$ for ($1 \leqq j \leqq k$ and $j \neq i$), a false positive is avoided due to the faulty H_i.

In order to guarantee no false negatives, the above case(1) must always be detected. On the other hand, case(2) does not cause a significant problem unless it noticeably increases false positive probability. Due to the cancellation effect of 2(a) and 2(b), changes in false positive probability cannot be of concern, under the assumption that all the m hash values are equally-likely for each faulty hashing unit.

Now we present our fault diagnosis scheme for the robust Bloom filter in Fig. 2. The primary role of the spare hashing unit, H_s, is to detect false negatives. Each time an input x is applied, the k hashing units return k hash values, $v_1, v_2, .., v_k$, ranging from 0 to m-1. The Bloom filter then checks the bits in M at locations corresponding to the k hash values to see if they are all 1's. At the same time the SD in Fig. 2 finds the *id i* of a hashing unit H_i with $M(v_i)=0$, if it exists.

The spare hashing unit H_s will then perform the same hash function h_i for comparison and produce the resulting hash value w_s. In any case, if there is a mismatch between $M[v_i]$ and $M[w_s]$, it is clear that there is a fault, but we do not know which one is faulty. In order to identify the faulty hashing unit, we then recompute with shifted operands, (i.e., H_i performs h_{i+1} instead of h_i, except for H_k) as illustrated in Fig. 3, where $M[v_3]$ is assumed to be 0 at time t and H_3 is thus selected by SD. At time t+1 the hash function h_3 is again performed by H_s, returning a hash value w_s. In the case of a mismatch in comparison, recomputing is done with shifted operands at time t+2.

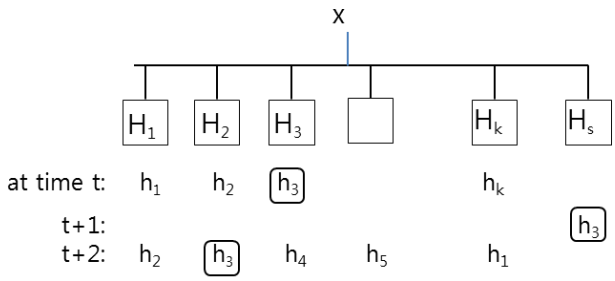

Figure 3: Re-computing for Fault Location

Let w_i represent the hash value produced by recomputing with shifted operands. The faulty hashing unit can then be located using the Table 1, where two possible combinations of lookup results, the second and third rows, and the decision made for each of them are shown. The first row in Table 1 is included simply to show the lookup results in the error-free case, although recomputing with shifted operands is not actually performed.

Table 1: Comparison Diagnosis

$M[v_i]$	$M[w_i]$	$M[w_s]$	decision
0	–	0	fault-free
0	0	1	H_s is faulty
0	1	1	H_i is faulty

False negatives induced by a faulty hashing unit can be detected by the last row in the table, although non-false-negatives can also show the same bit combination. To further differentiate between them we simply check to see if all the remaining k-1 hash functions point to 1 bits. If the condition is satisfied, we can conclude that the false negative induced by the faulty hashing unit H_i is correctly detected. In the case of the second row, the spare hashing unit is faulty.

The action to be taken after the fault diagnosis differs depending on the type of the fault. In the case of a transient fault, the corresponding hashing unit might behave correctly thereafter until another fault occurs. Hence the Bloom filter can resume normal operation with the same fault tolerance capability. In the case of a permanent fault, on the other hand, the faulty hashing unit needs to be removed from the rest of the Bloom filter.

The table for comparison diagnosis correctly locates the faulty unit, but it does not provide any information on the type of the fault. Hence additional testings to check to see if the fault is transient or permanent need to be done. An early decision can be made by checking the faulty hashing unit with the same input for several times to see if it still produces the same wrong hash value. Once a hashing unit is determined to have a permanent fault, the unit can be removed from the rest of the Bloom filter simply by masking its effect.

Our fault diagnosis scheme for the Bloom filter with an *m*-bit vector M and *k* hash functions can be formally described as follows.

Fault Diagnosis of Bloom filters

1. Obtain *k* hash values $v_1,v_2,..,v_k$ from $H_1,H_2,\ldots H_k$.

2. Look up the bit vector M to get $M[v_1],M[v_2],\ldots,M[v_k]$.

3. If $M[v_j]=1$ for $1 \leqq j \leqq k$, determine that $x \epsilon$ S

else find the *id* of a hashing unit H_i with $M[v_i]=0$

4. Perform the hash function h_i using the spare hashing unit H_s to get the hash value w_s and check to see if $M[w_s]=0$

5. If $M[w_s]=1$

5(a) Re-compute to obtain the hash value w_i with shifted operand

5(b) Look up the bit-vector M to get $M[w_i]$

5(c) The faulty unit is determined using majority voting based on $M[v_j],M[w_s]$, and $M[w_i]$.

In the above description, step 4 can be performed in parallel with normal operation. That is, any subsequent membership query can proceed along with the step 4. In addition, step 5 is only needed when the condition $M[w_s]=1$ is satisfied. Since the MTBF is expected to be sufficiently larger than the cycle time of each membership query, the probability that there is a mismatch in comparison in step 4 is negligibly small. In addition, the wrong hash value due to a fault does not cause a problem in membership query as long as no false negative can be guaranteed. Hence we claim that the time overhead for implementing the proposed scheme is negligibly small. The extra time for recomputing and majority voting is only needed to locate the faulty hashing unit upon detecting a fault.

The proposed robust Bloom filter is not intended to detect faulty hashing units at the time they return a wrong hash value. It performs normal function even in the presence of faults and the resulting wrong hash values, as long as the faults do not cause a false negative to occur. It, however, guarantees to detect all false negatives immediately under the single fault assumption. The wrong hash value due to a faulty hashing unit is masked if the corresponding lookup result is the same as the correct one. Even if the lookup returns a different value, the final decision on a membership query is acceptable with some negligible false positive probability unless a false negative occurs.

C. Performance Degradation

A faulty hashing unit may cause false negatives to occur, as observed in the previous section. If we assume that faulty hashing units generate a random hash value, all the *m* possible hash values are equally-likely to occur with a probability of 1/*m*. After inserting all the *n* elements into the bit-vector *M* of size *m*, with *k* hash functions, the probability that a particular bit of M is still 0, P_0, can be written as

$$P_0 = (1 - \frac{1}{m})^{kn} \approx e^{-kn/m}$$

The false positive probability, P_{fp}, is, then, given by

$$P_{fp} = (1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{-kn/m})^k$$

Suppose that there is a fault in the *k* hashing units. Then for a membership query with x ϵ S the probability that a false negative occurs due to the faulty hashing unit is P_0.

It is known that P_{fp} can be minimized when *k* is equal to ln2 $(\frac{m}{n})$ [8]. For given values of *m*, *n*, and *k*, satisfying the above condition, the probability that a false negative occurs due to the faulty hashing unit is 0.5, unless some provisions are made to detect it. Hence a single faulty hashing unit makes the Bloom filter unusable due to extremely low reliability. The false positive probability, P_{fp}, under the assumption that all hashing units are faulty-free, is $(1/2)^k$. The probability does not change noticeably in the presence of faults due to the cancellation effect of (2)a and (2)b on the previous page. In fact, if *k*= ln2 $(\frac{m}{n})$, they contribute equally to the change of false positive probability. Hence the changes in false positive probability in general are likely to be negligible and cannot be of concern in most applications.

In the proposed Bloom filter, once a hashing unit with permanent faults is identified, we can isolate the unit from the rest of the Bloom filter to resume normal operation. If the spare hashing unit is faulty, the Bloom filter works as before, but robustness to faults cannot be maintained. If one of the *k* regular hashing units is faulty, on the other hand, there are two options we can choose from. One is to isolate the faulty unit and replace it with spare hashing unit. The Bloom filter works as before, but without provision for fault detection. The other one is to reduce the number of hashing units to *k*-1, while retaining the fault detection capability, using the same bit-vector M. In this degraded mode, the false positive probability will increase accordingly. The false positive probability for the reduced Bloom filter with *k*-1 hashing units, is $P^r_{fp} \approx (1 - e^{-kn/m})^{k-1}$. The increase in the false positive probability is $\frac{P^r_{fp}}{P_{fp}} \approx \frac{1}{1-e^{-kn/m}}$. If *k*=*ln*2 $(\frac{m}{n})$, the ratio is approximately 2. That is, the false positive probability is doubled each time a hashing unit is removed due to some permanent faults.

CONCLUSION

This paper presented a simple technique for achieving robustness to faults in hardware Bloom filters. It employs a spare hashing unit and some error detection/correction code for the bit-vector data structure to detect false negatives due to faults in the Bloom filters. Fault detection is done with the spare

hashing unit during normal operation. Upon detecting a fault recomputing with shifted operands is used to locate the faulty unit. The time overhead for fault detection and location is expected to be negligibly small since recomputing with shifted operands is performed only when there is a mismatch in comparison for fault detection. The proposed scheme can possibly be extended to detect false negatives even in the presence of multiple faults unless the spare hashing unit is faulty.

ACKNOWLEDGMENTS

This work was supported by 2015 Hongik University Research Fund.

REFERENCES

[1] S. Geravand and M. Ahmadi, "Bloom filter applications in network security: a state-of-the-art survey," Computer Networks, Vol. 57, Dec. 2013, pp. 4047-4064.

[2] T. Kocak and I. Kaya, "Low-power Bloom filter architecture for deep packet inspection," IEEE Communications Letters, vol.10, no. 3, Mar. 2006.

[3] G. Wang, W. Gong, and R. Kastner, "On the use of Bloom filters for defect maps in nanocomputing," IEEE ICCAD, Nov. 2006, pp. 743-746.

[4] M. Mitzenmacher and G. Varghese, "Biff(Bloom filter) codes: Fast error correction for large data sets," IEEE Int. Symp. Information Theory, 2012.

[5] P. Reviriego, S. Pontarelli, J.A. Maestro, and M. Ottavi, "A synergetic use of Bloom filters for error detection and correction," IEEE Trans. VLSI Systems, Vol. 23, No. 3, 2015, pp. 584-587.

[6] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors," Communications of the ACM 13:7, 1970, pp. 422-426.

[7] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, J.W. Lockwood, "Deep packet inspection using parallel Bloom filters," IEEE Micro, Jan-Feb. 2004, pp. 52-61.

[8] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," Internet Mathematics, Vol.1, No. 4, 2003, pp. 485-509.

[9] M.V. Ramakrishna, E. Fu, and E. Bahcekapilli, "Efficient hardware hashing functions for high performance computers," IEEE Trans. Computers, Vol.46, No. 12, Dec. 1997, pp. 1378-1381.

[10] M.J. Lyons and D. Brooks, "The design of a Bloom filter hardware accelerator for ultra low power systems," ISLPED-14, 2009, pp. 371-376.

[11] M.-H. Lee and Y.-H. Choi, "A fault-tolerant Bloom filter for deep packet inspection," IEEE 23rd PACRIM Int. Symp. Dependable Computing, Dec. 2007, pp. 389-396.

[12] P. Reviriego, S. Pontarelli, J.A. Maestro, and M. Ottavi, "A method to protect Bloom filters from soft errors," IEEE Int. Symp DFT, 2015, pp. 80-84.

[13] C.L Chen and M.Y. Hsiao, "Error-correcting codes for semiconductor memory application: a state of art review," IBM Journal of Research and Development, vol. 28, no.2, Mar 1984, pp. 124-134.