ISSN: 2278 – 909X
International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE)
Volume 6, Issue 1, January 2017

Error Detection and Correction by using Bloom Filters

R. Prem Kumar, Smt. V. Annapurna

Abstract---Bloom filters (BFs) provide a fast and efficient way to check whether a given element belongs to a set. BFs are widely used in various applications like networking, computer architectures and communications. At present these bloom filters are also extended to various scenarios. In advanced electronic circuits, the reliability becomes a challenge due to the increase in radiation, manufacturing variations and reduction of noise margins as technology improves. Here, the BFs are used to detect and correct errors in the associated data set. This gives a reuse of existing BFs to also detect and correct errors such that there is no need to add extra error correction methods which reduces the area of the proposed method.

Index Term: Bloom filters, soft errors, error detection and correction.

I. Introduction

Bloom filter checks whether an element belongs to a set in a simple and efficient way [1]. It is a probabilistic data structure. It is used in various computer architectures and networking applications [2]. The BFs are also used to reduce data lookups in the large data bases for example in Google Bigtable [3].

The extension of the BF basic structures are implementing over years for example counting BFs are implemented in order to remove the elements in the BF [4]. Another extension in order to enhance the transmission in network, compressed BFs was proposed [5]. To achieve error correction in large data sets, Bloom filter (Biff) codes are proposed recently that are based on Bloom filter [6].

Bloom filters are mostly implemented using electronic circuits [7], [8]. In order to get high performance, Bloom filters contents are generally stored in high speed memory and the processing is implemented in a processor or in separate circuitry. The element set which are used to construct Bloom filter is generally stored in low speed memory [9].

As technology scales, the challenging thing for electronic circuits is to maintain reliability. The most common errors occurred due to radiation, interferences and other effects. So, in order to operate the circuits reliably at various levels the mitigation methods are implemented. The important element in implementation of Bloom filters are memories. By using spare rows and columns methods, the permanent errors are generally corrected for memories. But, soft errors which are occurred due to radiation can affect the memory cell by changing the value during circuit operation. Soft errors are one which doesn't damage the memory device but produces a wrong value in the memory cell i.e., by converting zeros to ones or ones to zeros but operates correctly [10]. Generally in memories, the per word parity bit of different higher error correction codes (ECCs) are used to deal soft errors [11].

In electronic circuits to reduce errors the Bloom filters are proposed. For example, to detect faulty words in nanomemory the Bloom filters is used [12]. In [13], the CBF is used to detect and correct errors in content addressable memories (CAMs). Here, the purpose is to find errors in CAM entries where the CBF and CAM are used in parallel. This can be obtained by studying the results of CAM and CBF such that they are uniform. If an error is found, the correction method is started to get the correct value in the modified CAM entry by using the additional copy of its contents. The BFs are included externally in both the cases to detect and correct errors but it doesn't exist in the actual design. In Biff codes, the same method applies where the error correction is done by using extended Bloom filters. Therefore, in all the above cases the bloom filters are added externally but actually not present in the original design. This is distinct to the reuse of existing BFs which already exists in system for error correction.

Here, a method to exploit present CBFs to also perform the error detection and correction to the elements of the set associated with CBF is implemented. The method based on concept of algorithm based fault tolerance (ABFT), which introduces the reuse of existing elements of the design to implement fault tolerance with lower cost [14]. As same as ABFT, the proposed method gives a reuse of existing CBFs to detect and correct the errors. This method considers the per word parity bit to protect the elements of set in the memory and the single bit errors are corrected by using the CBFs. The operative of this method is represented using traffic classification.

The base idea in proposed method can be implemented when the elements of the set are stored in memory protected with advanced ECCs. In BFs the simplified version of proposed method can be applied but in such situation the percentage of errors that to be corrected becomes much lower.

The remaining of this brief is arranged as follows. In section II, an analysis of BFs and CBFs is presented. In section III, the implementation is provided. In section IV, traffic Classification case study is presented to study effectiveness and benefits of this method. At last, the conclusion is presented in section V.

II. Overview of BFs

The construction of Bloom filter is done by using a set of k hash functions h_1, h_2, . . . h_k which maps the input element x to one of m bits. The BF consists of two operations which are defined below.

1*) Insertion:* To insert an element x in BF, the bits in array that matches to positions $h_1(x)$, $h_2(x)$, . . . $h_k(x)$ are set to one.

2*) Query:* To query an element x in BF, the bits in any that matches to positions $h_1(x)$, $h_2(x)$, . . . $h_k(x)$ are read and the element is assumed to be in BF only if all of them which are read is one.

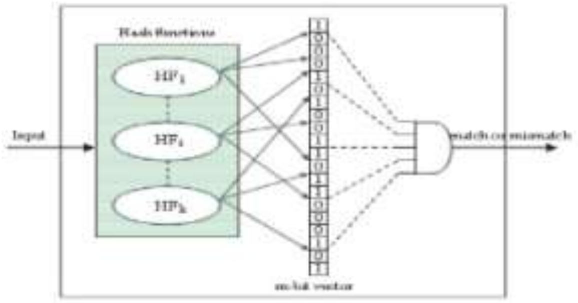

Fig.1. Bloom filter with k hash functions.

This operation assures that if an element is added in bloom filter, when a query is done the element will be found. But, the BF can make false positives when a query is made for an element which is not added to BF. That is the element shows incorrectly as being stored in BF, where the element is not present in the set. This can happen when the positions of the other elements are set to one that corresponds to hash values of that element. An example of a Bloom filters representing a set {x, y, z}. The element w is not in the set {x, y, z}. The element v represents the false positive where m=18 and k=3 is represented in figure 2.

Considering that hash functions are uniformly distributed, after inserting 'n' elements in BF, the probability $P_0(n)$ that a given bit in the array is zero is

$$p_o(n) \cong \left(1-\frac{1}{m}\right)^{kn} \cong e^{-\frac{kn}{m}} \qquad (1)$$

Therefore, the probability of a false positive can be approximated as

$$p_{fp}(n) \cong (1-p_o(n))^k \cong \left(1-e^{-\frac{kn}{m}}\right)^k \qquad (2)$$

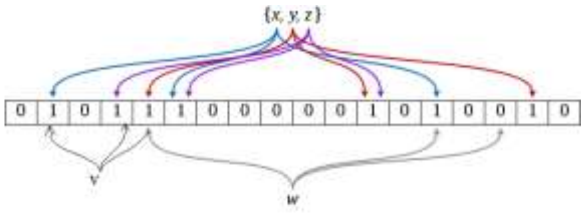

Fig.2. Bloom filter example.

The false positive probability (p_{fp}) depends on ($1-p_0(n)$) and k. The first expression gives the probability that an element in CBF has a value other than zero which is known as load factor. The load factor represents the false positive probability and the number of elements is stored in the CBF. The load faster which is used in experiments is represented as

$$lf \cong (1-p_o) \qquad (3)$$

In BFs, the problem is that elements cannot be removed. This is due to a position with one in array might be shared by various elements and thus clearing $h_1(x)$, $h_2(x)$, . . . $_k(x)$ positions for element x can also affect other elements in BF. To address this problem, the CBFs are introduced. In CBFs, the array of m bits is replaced with array of integer of b bits and the following operations are performed.

1) *Insertion:* To insert an element *x* in CBF, the integers in array that correspond to positions $h_1(x)$, $h_2(x)$,…, $h_k(x)$ are incremented by one.
2) *Query:* To query for an element *x* in CBF the integers in array that correspond to positions $h_1(x)$, $h_2(x)$,…, $h_k(x)$ are read and if and only if all of them are larger than zero the element is considered to be in CBF.
3) *Removal:* To remove an element *x* from CBF, the integers in array that correspond to positions $h_1(x)$, $h_2(x)$,…, $h_k(x)$ are decremented by one.

By using the integer instead of bits gives the removal of elements as each position in array stores the number of elements that share that position. The false positive rate of CBF is same as BF.

III. Implementation

The proposed method represents that CBF in addition to structure that performs fast membership check to element set, also provide a redundant representation of the element set. Hence, this redundancy can be used for error detection and correction.

To analyze this method, general implementations of CBFs where the slow memory is stored with elements of set and faster memory is used to store the CBF. Generally, it is considered that elements of set are stored in DRAM and CBF is stored in cache [9]. The reason behind this is that elements of set are accessed only when elements are read, added or removed hence access time is not an issue but CBF needs to be accessed frequently hence it requires fast access time to increase its performance. Since entire element set is stored in slow memory, no incorrect deletion will be present as it would be found when the element is getting removed from slow memory. Hence, false negative in CBF is not an issue in our method.

Generally, memories are protected with per word parity bit or with single bit error correction code [11]. This is considered based on observation that most errors affect single bit or even if they affect various bits, the errors can be divided among various words by use of interleaving [16]. The time between the soft errors is mostly large because they are rare events [10]. Generally these errors are considered as isolated due to arrival rate of terrestrial applications is of at least days or weeks. Hence, if the soft error arrives by a time any previous error can be detected or corrected. This is the consideration needed when single bit error correction codes are used.

In this method, it is considered that both DRAM and cache are protected with per word parity bit which can detect single errors. Hence, by using single bit error correction codes it is considered that error are isolated.

The objective for this method implementation is to correct single bit errors using CBF. Hence, without having the cost of adding an ECC to memories, the CBF will give single bit error correction.

The initial step to perform error correction is to detect errors. It is performed by checking parity bit while accessing either DRAM or caches. To have earlier detection of errors, the use of scrubbing to periodically read memories is considered [15]. If the error gets detected, then the correction procedure is implemented.

Hence, the error can occur either in CBF itself or the element set which is stored in memory. If error occurs in CBF, then it is corrected by clearing entire CBF and reconstructed it using element set. If error occurs in element set, then the procedure is complex and it is divided in two phases that are represented in the following methods. First the fast and simple procedure is used and if it is unable to correct error then and advanced and complex procedure for error correction is used.

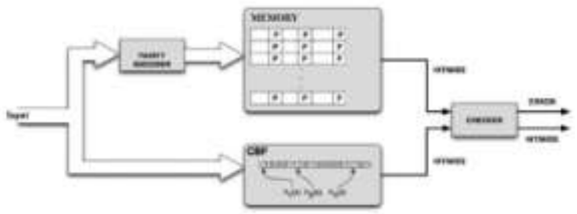

Fig.3. Proposed error correction scheme for CBF.

A) *Simple procedure for correction of errors in element set:*

To implement simple correction procedure, let us consider that a single bit error affects element x which is detected using parity bit. Hence, x_e is read from memory. The correct value has to be x_e if error affects the parity bit. If error affects ith data bit, the correct value is $x_{em}(i)$, where $x_{em}(i)$ is value read (x_e) with i^{th} bit inverted. To find which of those is correct value of x, the candidates [x_e and all $x_{em}(i)$] is tested for membership to CBF. If only one of the candidates is found in CBF then the value is correct and no false positives are occurred. If more than one value is found, then there is an occurrence of false positive, hence it can't find the correct value so, the advanced procedure should be followed which is described in section III-B. The present method requires only l+1 queries to CBF, where l represents the number of bits in each element of set. But, the correction rate defends on false positive rate of CBF. The probability that an error is corrected by using the method is

$$P_{correction} \cong (1 - p_{fp})^{l} \qquad (4)$$

Which is the probability that none of l candidates are not x return a false positive on query. Some elements in set may only differ in one or two bits from another element in set, for these the above formula couldn't be considered. In that case, the proposed correction method may fail as one of candidates may also be valid one and hence, the advanced procedure to be used. This effect is heavily dependent on properties of elements in the set hence it is application dependent. In any case the probability of equation (4) is to be used as upper rather than an approximation.

B) *Advanced Procedure for correction of errors in element set:*

When the simple method fails to correct the error, then the advanced method is used for the correction of errors. The correction method starts by making a copy of CBF in DRAM memory. Then all elements in set except the incorrect one are removed from CBF. This will leave CBF with only the values that relates to the original values of the elements x. Now all the candidates [x_e and all $x_{em}(i)$] is queried in CBF which has only x as entry. As in previous method, if only one of candidate matches the CBF, it is the correct one. If more than one candidate matches the CBF, then the error cannot be corrected.

The probability that a given x and another value y gives the same values of hash function h_1, h_2 . . . h_k is

$$P_{CBF(x)=CBF(y)} \cong \frac{k!}{m^k} \qquad (5)$$

Therefore, the correction probability for this advanced procedure can be approximated as

$$P_{correction} \cong \left(1 - \frac{k!}{m^k}\right)^l \qquad (6)$$

Which will be very close to 100% in many practical scenarios as *m* is typically large.

The advanced correction rate comes at the cost of more complex procedure, which needs replication of CBF, the removal of all entries except the erroneous one (n-1), and at last the query of l+1 candidates. However, soft errors are rare events where this method is used only when simple method is unable to correct the errors.

Finally, it must be considered that when CBF experiences overflow in counters, the second method could not be used. This is not a major problem as the overflow occurrence. Probability is very low when counter with four bits are considered. In any method, the over flows are detected once it occurs, the second method will be disabled.

The same method can also be applied to memory protected with single error correction double error detection (SEC-DED) Code to correct double errors. In such case, the simple procedure would be of less use as the number of candidates to test is $\binom{l+1}{2}$ and hence, it is unlike that none of them gives false positive. On the other hand advanced correction procedure will be used to correct the errors with probability close to one.

IV. Evaluation

The proposed method is evaluated using a example of CBF used to speed-up traffic classification. The different IP addresses are stored in CBF which allows knowing whether the element is present or not. Here IP version four is taken which consists of 32 bits of each element. To test this method, CBF is inserted with 24 random IP addresses and the single parity bit is added to the input elements. The three random input elements are taken from the existing elements where even parity is considered to protect the input elements in memory. By checking the parity bits of input elements we get the incorrect element and the correction procedures which are described in section III were applied. This process is repeated many times such that more number of random single bit errors is tested. First the simple error correction method is used and if it is unable to correct error, then the advanced error correction procedure is used. In all these procedures the single bit errors are corrected. The effectiveness of simple correction method mainly depends upon the load of CBF. It is observed that for low load this method can correct most errors and if the load increases then the number of hash functions also to be increased to get better error correction. When the load is high, the number of entries can be removed and when low load is achieved, simple correction method is used to correct errors. Thus, the entire advanced procedure is used only for small fraction of errors.

Table: Comparison of memory size

Input bit width	SEC additional bits	Proposed additional bits	Memory size reduction
08	4	1	25%
16	5	1	19%
32	6	1	13%

The cost savings obtained by using the proposed method is estimated as the implementation of SEC code on 32 bit requires 6 bits. But, by using only a single parity bit the single error correction is achieved by CBF. Therefore, the savings is 5 bits per element or roughly 13% of memory storage is saved for the element set. A different way to get benefits is that SEC can be implemented in element set when the memory is protected with per word parity bit. Thus, the reliability can be increased without adding new hardware resources.

V. Conclusion

In this brief, a new application of BFs is proposed. The idea is to use BFs in existing applications to also detect and correct errors in their associated element set. In particular, it shows that CBFs can be used to correct errors in associated element set. This gives a cost efficient solution to reduce soft errors in applications which use CBFs.

The configuration considered here is that the memory protected with per word parity bit for which the CBF is used to get single bit error correction. This show how existing CBFs can be used to get error correction in addition to perform the membership checking function.

The general idea can also be used when memory is protected with more advanced codes. For example, if a SEC-DED code is used, the CBF is used to correct double errors. In addition, the simplest part of error correction method can be applied to BFs to get some degree of error detection and correction.

References

[1] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors, "*Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[2] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Proc. 40th Annu. Allerton Conf.*, Oct. 2002, pp. 636–646.
[3] C. Fay *et al.*, "Bigtable: A distributed storage system for structured data," *ACM TOCS*, vol. 26, no. 2, pp. 1–4, 2008.
[4] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *Proc. 14th Annu. ESA*, 2006, pp. 1–12.
[5] M. Mitzenmacher, "Compressed bloom filters," in *Proc. 12th Annu. ACM Symp. PODC*, 2001, pp. 144–150.
[6] M. Mitzenmacher and G. Varghese, "Biff (Bloom Filter) codes: Fast error correction for large data sets," in *Proc. IEEE ISIT*, Jun. 2012, pp. 1–32.
[7] S. Elham, A. Moshovos, and A. Veneris, "L-CBF: A low-power, fast counting Bloom filter architecture," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 6, pp. 628–638, Jun. 2008.
[8] T. Kocak and I. Kaya, "Low-power bloom filter architecture for deep packet inspection," *IEEE Commun. Lett.*, vol. 10, no. 3, pp. 210–212, Mar. 2006.
[9] S. Dharmapurikar, H. Song, J. Turner, and J. W. Lockwood, "Fast hash table lookup using extended bloom filter: An aid to network processing," in *Proc. ACM/SIGCOMM*, 2005, pp. 181–192.
[10] M. Nicolaidis, "Design for soft error mitigation," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 405–418, Sep. 2005.
[11] C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM J. Res. Develop.*, vol. 28, no. 2, pp. 124–134, 1984.
[12] G. Wang, W. Gong, and R. Kastner, "On the use of bloom filters for defect maps in nanocomputing," in *Proc. IEEE/ACM ICCAD*, Nov. 2006, pp. 743–746.
[13] S. Pontarelli and M. Ottavi, "Error detection and correction in content addressable memories by using bloom filters," *IEEE Trans. Comput.*,vol. 62, no. 6, pp. 1111–1126, Jun. 2013.
[14] A. Reddy and P. Banarjee, "Algorithm-based fault detection for signal processing applications," *IEEE Trans. Comput.*, vol. 39, no. 10, pp. 1304–1308, Oct. 1990.
[15] A. M. Saleh, J. J. Serrano, and J. H. Patel, "Reliability of scrubbing recovery-techniques for memory systems," *IEEE Trans. Rel.*, vol. 39, no. 1, pp. 114–122, Apr. 1990.
[16] P. Reviriego, J. A. Maestro, S. Baeg, S. J. Wen, and R. Wong, "Protection of memories suffering MCUs through the selection of the optimal interleaving distance," *IEEE Trans. Nucl. Sci.*, vol. 57, no. 4,pp. 2124–2128, Aug. 2010.
[17] Pedro Reviriego, Salvatore Pontarelli, "A synergetic use of bloom filters for error detection and correction", IEEE Transactions on VLSI systems, VOL. 23, NO. 3, March 2015.

Mr. R. Prem Kumar received B.Tech degree in (ECE) from MITS, Madanapalle in 2013. Currently he is pursuing M.Tech in JNTU College of engineering Anantapur. His areas of interest are Digital Systems and Computer Electronics.

Smt. V. Annapurna completed M.Tech and working as Lecturer, ECE Department, JNTU College of Engineering, Anantapur & pursuing Ph.D in Image Processing. Fields of interest are Image Processing and VLSI.