Error Detection in Ternary CAMs Using Bloom Filters

Salvatore Pontarelli, Marco Ottavi
University of Rome ”Tor Vergata”
Rome Italy
{pontarelli,ottavi}@ing.uniroma2.it

Adrian Evans
iRoC Technologies,
Grenoble, France
adrian.evans@iroctech.com

Shi-Jie Wen
Cisco Systems Inc.
San Jose, CA, USA
shwen@cisco.com

***Abstract*—This paper presents an innovative approach to detect soft errors in Ternary Content Addressable Memories (TCAMs) based on the use of Bloom Filters. The proposed approach is described in detail and its performance results are presented. The advantages of the proposed method are that no modifications to the TCAM device are required, the checking is done on-line and the approach has low power and area overheads.**

I. Introduction

A Content Addressable Memory (CAM) is memory capable of comparing a word of input data against all the data words stored in the memory, providing as result the address of the matching data [1], [2]. A binary CAM is the simplest implementation of a CAM where search words are composed only of 0s and 1s. A ternary CAM (TCAM) also allows for the matching of a third value “X” which represents a don’t care value and consequently allows a highly compacted means of storing patterns to match. For example an entry “10XX1” in a ternary CAM matches the following four patterns “10001”, “10011”, “10101”, “10111”. The typical implementation of the don’t care bits is obtained by defining a mask string for each entry of the Ternary CAM. In the previous example the mask string would be “00110” while the actual entry is “10–1” where “-” can be either 0 or 1. TCAMs are widely used in high speed network systems to implement features such as classification and access control [3]. In fact, their ability to perform massive comparisons with $\mathbf{O}(1)$ complexity makes them extremely appealing for searching large tables with very low latency. The use of the Xs makes it possible to apply admission or quality of service (QoS) policies to classes of traffic using a modest number of table entries [4].

Currently, the use of nanometer scale technology, the reduction in operating voltages and the increase in the overall number of stored bits have caused a consequent increase in the error rate due to the occurrence of Single Event Upsets (SEUs). SEUs occur when sub-atomic particles strike a sensitive area of a circuit and the interaction between the silicon and particles creates free charge that can be collected by the sensitive circuit nodes close to the location of the particle impact. The collected charge can change the state of a circuit, for example, by flipping the value of a bit from 0 to 1 or $viceversa$.

Many strategies have already been proposed to mitigate the effects of SEUs in random access memories, based on information redundancy or on technology/circuit level solutions. Error detection and correction codes [5] are a typical application of information redundancy. Depending on the type of memory to protect, and on the required data integrity level, different codes can be used [6], [7]. Instead, technology and circuit solutions are aimed at increasing the minimum charge (the so-called critical charge) value needed to flip the value of a node [8].

It is difficult to directly apply protection techniques based on information redundancy to TCAMs because all entries are accessed simultaneously. However, networking systems based on TCAMs require high reliability and new approaches are required to mitigate the effects of SEUs.

In the literature different techniques have been proposed to mitigate the effects of SEUs in TCAMs. Almost all the proposed techniques require modifications to the CAM architecture [14] performed at circuit or architectural level or are based on a background parity scan which has a long detection latency.

Instead, we propose a method that does not require modifications to the internal structure of the TCAM, since this is often designed in a full custom design flow to minimize power and area consumption of the core cell. The proposed technique can be implemented as a stand-alone module that operates in parallel with the TCAM and checks the correctness of the results of each access. Error detection is immediate, unlike techniques based on background parity scans. Moreover, the proposed solution is particularly well suited to TCAM applications with wide word sizes, like the ones used in modern network systems, since the overhead is independent on the TCAM word size.

This paper extends the method proposed in [9] for binary CAMs, to Ternary CAMs, managing the presence of don’t care bits by using suitable hashing mechanisms.

The method presented in [9] proposes to add in parallel to the CAM a well known data structure, called a Bloom Filter, to efficiently detect if the TCAM has provided a correct result or if it was affected by an error. A Bloom Filter is a structure that can be realized efficiently with limited hardware resources, or with efficient software algorithms.

In a Bloom Filter when data has to be stored (or queried)

This work supported in part by Cisco Systems Inc. via a Cisco Research Award. Marco Ottavi is supported by the Italian Ministry for University and Research Program “Incentivazione alla mobilità di studiosi stranieri e italiani residenti all’estero”, D.M. n.96, 23.04.2001

978-3-9815370-0-0/DATE13/ © 2013 EDAA

it is hashed with multiple hash functions, and at the output of each hash a corresponding memory location is written (read). A Bloom Filter performs a limited number of memory accesses, one for each hash output, and therefore its memory can be easily protected against SEUs using standard ECC techniques. Therefore we assume that the Bloom Filter is error-free.

A Bloom Filter performs two tasks: 1) stores a set of items in its memory, and 2) quickly responds to a query about the presence of an item. The drawback of using such a structure lies in its probabilistic nature which suffers from false positives. A false positive occurs when the hash functions alias. With a certain probability (called the false positive rate), the Bloom Filter can report a data element as being present when in fact it was not inserted into the Bloom Filter. However, when a Bloom Filter signals that a data is not present this is always the case (i.e. a false negative never occurs in a Bloom Filter).

The rest of the paper is structured as follows: Section II presents a survey of previously proposed methods for protecting CAMs and TCAMs against the occurrence of SEU. Section III gives a background on Bloom Filters, while Section IV describe how to manage the don't care bit by using suitable hashing functions. Section V presents the proposed architecture of a TCAM with error detection and error correction capabilities. Section VI presents the results obtained by using both a synthetic data set and an actual industrial data set. Finally, Section VII draws the conclusions and discusses further work that can be done to extend this method.

II. Related works

In industry, the standard approach to error detection in TCAMs is to protect the entries with a parity code. It is not practical to simultaneously check the parity of all entries and it is not sufficient to only check the parity of the matching entry, since an error on a higher priority entry can create an incorrect match. Therefore, a parity scan engine runs in the background and sequentially reads through the entries, checking the parity and triggering an error indication if there is a mismatch. Software then intervenes and re-writes the corrupted TCAM entry. The drawback to this method is that there can be a significant latency between when an error occurs and when it is detected and then fixed by software. During this window of time, incorrect decisions are occurring. Multiple interleaved parity bits are used to protect against upset events which corrupt adjacent cells which are an increasing concern.

In [15], error correcting codes for TCAMs are proposed, however, these require a 200% overhead in order to correct single bit errors. In [14], a hardened TCAM cell with cross-coupled feedback loops is proposed. A SER reduction of about 30% is achieved with an area penalty of 15%. Others [16], have proposed using a duplicate TCAM for error detection and correction.

None of the existing solutions provide strong, on-line error detection with a modest area overhead.

III. Overview of Bloom Filters

In this section we present an overview of the well-know Bloom Filter [11] structure.

A Bloom Filter [11] is a probabilistic data structure used to check the membership of an element in a set. The structure allows the occurrence of false positives ($i.e.$ the Filter signals an element as present even if it is not true), but false negatives are not possible ($i.e.$ if an element is present the Filter will never signal the opposite). Elements can be added to the set, but not removed and the more elements are added to the set, the larger the probability of false positives. In this paragraph we report the equations needed to correctly dimension the Filter with respect to the required false positive probability and the expected number of elements to be stored. A Bloom Filter is implemented as a bit array of m bits accessed via k hash functions $H_1(x)...H_k(x)$, each of which maps a set member x to one of the m bits within the bit array. We denote as $v(i)$ the value of bit i within the bit array.

Two operations are possible with a Bloom Filter:

1) **Insertion:** an element x is inserted into the Filter by setting to one all the indexes of the bit array addressed by the k hash functions. In a mathematical notation this corresponds to:
$\forall i \in \{1..k\}, v(H_i(x)) \leftarrow 1$.
2) **Querying:** An element is present in the Filter if all the values of the bit array addressed by the k hash functions are equal to 1.
$result \leftarrow \min\{v(H_i(x))\},\ i \in \{1..k\}$

For a Bloom Filter in which n elements are stored, the probability $\rho(n)$ that a given bit in the Filter is zero is given by:

$$\rho(n) = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-n\frac{k}{m}}. \tag{1}$$

If we test membership of an element that is not in the set, each of the k bit array values indexed by the hash is 1 with probability $1 - \rho(n)$. The probability of all of them being 1, which would cause the false positive, is then

$$P_{fp}(n) = (1 - \rho(n))^k \approx (1 - e^{-\frac{kn}{m}})^k \tag{2}$$

The probability of false positives decreases as m increases, and increases as n increases. For a given m and n, the value of k (the number of hash functions) that minimizes the probability is: $k = m/n \ln 2 \approx 0.7 \cdot m/n$

Using the optimal value of k we obtain

$$P_{fp}(n) = 2^{-k} \approx 0.61^{m/n} \tag{3}$$

Using equation (3) we can size the Bloom Filter according to a required number of elements to be stored and to a required minimum false positive rate. Figure 1 illustrates the structure of a Bloom Filter.

IV. Application-Aware Hashing

The trivial method of inserting don't care bits in a BF is to fully expand each of the masked values to its unmasked values. This is not scalable as the number of insert operations and the size of the Bloom filter would grow exponentially with the number of don't care bits. From a hardware perspective

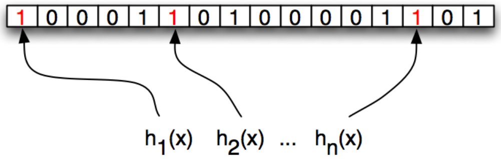


Fig. 1. Scheme of a Bloom Filter. The data x to be stored is hashed n times and the bits addressed by the hashes are set.

a TCAM can have dont cares in any bit of any entry. Some practical considerations can be used to identify patterns within the entries based on TCAM usage in industrial applications.

In networking systems, a single large TCAM is used to support multiple features or applications. Examples of such applications are packet classification, access control lists (ACLs), traffic sampling (Netflow) and Quality of Service (QoS), to cite the most common. The entries in the TCAM are divided based on the applications and a small number of bits within the lookup word are used to select which set of entries to match when a lookup is performed for a specific application. For a given feature, a specific set of fields from the packet header are looked up. The usage of fields within an application is regular and we discern patterns in where don't care bits appear for each application. This regularity of the don't care bits can be exploited to perform *application aware hashing* and thus avoid the explosion that would result from a naive expansion of don't care bits.

The fields that compose a TCAM entry can be classified into three types:

1) **Fully Unmasked Type.** This represents any field of size N bits where the mask for all N bits is such that the corresponding lookup bit must match exactly. These entries correspond to a binary CAM.
2) **Prefix Type.** This represents any field, of size N bits, where the most significant M bits are unmasked (must match) while the least significant (N - M) bits are masked (don't care).
3) **Generic Type.** This represents an arbitrary field of N bits where any combination of bits within the field may be masked or unmasked, and this combination could differ across TCAM entries.

Frequently generic fields can be decomposed into a portion which is fully unmasked, thus minimizing the number and size of generic fields. In fact, TCAM configurations are usually computed by suitable algorithms that transform classification rules based on port ranges, IP address and so on, into a set of TCAM entries [12]. These algorithms produce for each rule, a set of very similar TCAM rows, that therefore are regular enough to be managed by suitable algorithms. Since these algorithms differ from one feature to the next, and therefore different kinds of similarities occurs, changing the feature under observation, the application aware approach must define some managing rules for each different feature.

The key idea to avoid state explosion is to select a different set of hash functions for each feature in order minimize the don't care bits. We call this approach Application-aware hashing.

In particular, for prefix fields we use the approach proposed in [13] for longest prefix matching, while for generic field we propose a hashing methodology aimed at compacting the number of dont care bits. The method in [13] partially masks the least significant bits, hashing only a variable prefix part of the field. The hash functions are grouped with respect to an interval of the prefix length (see Fig. 2). During the insertion phase, the field is masked following the given prefix don't care configuration, and is inserted in the Bloom Filters using the whole set of hash functions. During the search phase, the item to be searched, is progressively masked and hashed, like depicted in Fig.2).

The regularity of TCAM configurations allows us to define some rules, that can be applied to perform the application-aware hashing. since the most significant bits of the TCAM define the feature that define the subsequent fields of the TCAM, depending on the values of this bits, different hashing rules are applied to the other fields of the feature. We can define the following rules:

1) Unmasked fields can be used as input of the hash functions,
2) Fields that are always masked can be ignored in all the hash functions,
3) Fields of prefix type can be managed using the the longest prefix match algorithm,
4) Otherwise mask bits are compressed in order to reduce the number of don't care bits (e.g. XORing contiguous bits).

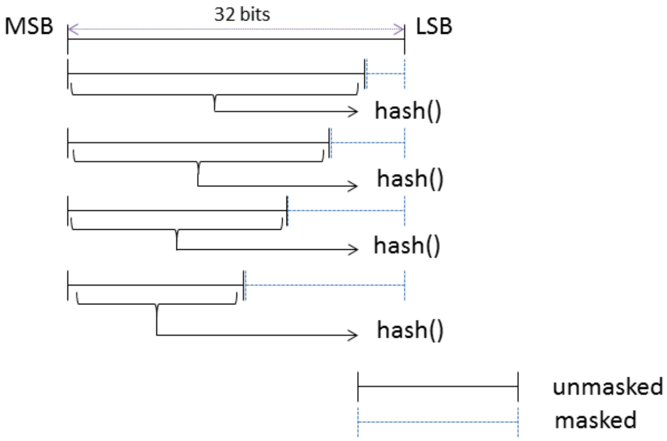
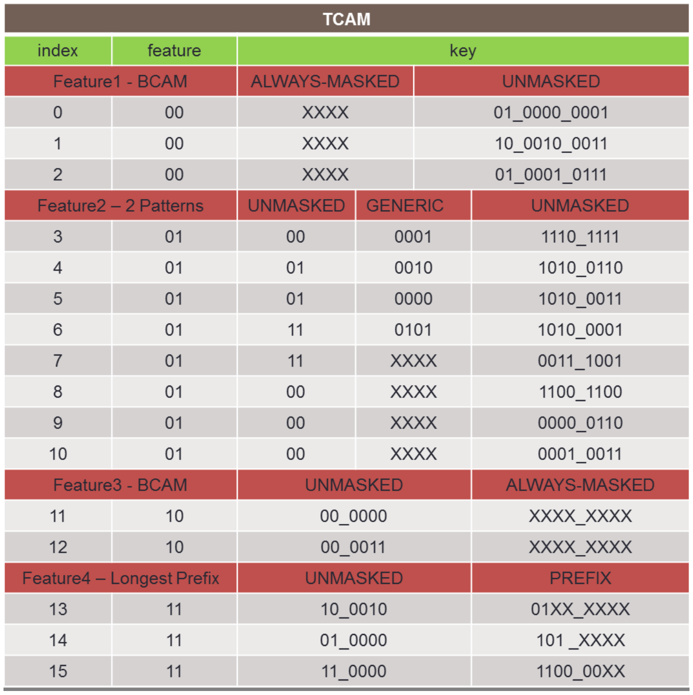


Fig. 2. Masking of bits for Longest Prefix Match.

In order to describe our method, a sample 16-bit wide TCAM configuration is shown in Fig. 3. In this example the TCAM contains 4 features:

- Feature 1: In this feature the TCAM is configured to act as a binary CAM. The feature is composed of three fields. The first field (00) identify the feature, the second is always masked, the third is always unmasked. This feature



TCAM				
index	feature	key		
Feature1 - BCAM		ALWAYS-MASKED	UNMASKED	
0	00	XXXX	01_0000_0001	
1	00	XXXX	10_0010_0011	
2	00	XXXX	01_0001_0111	
Feature2 – 2 Patterns		UNMASKED	GENERIC	UNMASKED
3	01	00	0001	1110_1111
4	01	01	0010	1010_0110
5	01	01	0000	1010_0011
6	01	11	0101	1010_0001
7	01	11	XXXX	0011_1001
8	01	00	XXXX	1100_1100
9	01	00	XXXX	0000_0110
10	01	00	XXXX	0001_0011
Feature3 - BCAM		UNMASKED	ALWAYS-MASKED	
11	10	00_0000	XXXX_XXXX	
12	10	00_0011	XXXX_XXXX	
Feature4 – Longest Prefix		UNMASKED	PREFIX	
13	11	10_0010	01XX_XXXX	
14	11	01_0000	101 _XXXX	
15	11	11_0000	1100_00XX	

Fig. 3. Example of content of a TCAM with multiple features: the two Most significan bits define which kind of feature is stored in the row.

acts as a 10-bits width binary CAM. Following the above defined rules, the application-aware hashing can be done using as inputs for the Bloom Filter only the bits $in[9\ldots0]$ of the input vector $in[15\ldots0]$.

- Feature 2: The feature is composed of four fields. The first two bits (01) identify the feature, the second field, composed of two bits, are always unmasked, the third field is generic (*i.e.* can be masked or unmasked), while the last field is always unmasked. For this feature we define as an input to the hash function the concatenation of three bit vectors:

$$in[13..12],\ (in[11]\oplus in[10]\oplus in[9]\oplus in[8]),\ in[7..0]$$

  Using this vector as input for the Bloom Filter, when the third field presents unmasked entries, a regular item is inserted in the Bloom Filter. Instead, for the masked entries (*e.g.* the entries with the Xs) the xor compression reduces the number of Xs in the input to the Bloom Filter to a single bit. At this stage we can expand the don't care, inserting the 2 items corresponding to 0 and 1 in the BF. This kind of compression works well under the condition that, even if the field is defined as generic, the values of this fields often present some regularities and are repeated among the different rows of the TCAM.
- Feature 3: This feature is similar to feature 1, configuring this part of the TCAM as a binary CAM. The first field (10), identifies this feature, the second field is unmasked, the third field is aways masked. These rows act as a 6-bit wide binary CAM. The application-aware hashing approach is applied checking if the value of the first field is 10. In this case, the input for the Bloom Filter are the bits $in[13\ldots8]$ of the original input vector.
- Feature 4 : This feature is composed of three fields. The first field identifies the feature (11), the second one is a 6 bits unmasked field, while the third one uses Longest Prefix Match. Applying the rules for application-aware hashing, the whole input vector is given to the Longest prefix matching algorithm, setting the range for bit masking for the bits from 0 to 7.

V. Architecture

In this section two architectures for detecting errors in TCAM are shown. Both architectures use the application-aware approach to insert and query items in a set of Bloom Filters. The first architecture (Fig. 4) uses two Bloom Filters, one that take as input the same key that is applied to the TCAM, while the other takes as input the concatenation of the couple $\{key,entry\}$, where $entry$ is the response of the TCAM to the key input. The first BF detects the occurrence of a SEU causing a pseudo-MISS [10], *i.e.* the TCAM says that the searched key is not present. If such type of error occurs, the output of the first BF and of the TCAM are discordant, since the TCAM gives a MISS, while the BF gives a HIT. The second BF detects the occurrence of pseudo-HIT or multi-HIT [10] errors, *i.e.* the TCAM gives as output an erroneous output $entry$, while the correct value should be another value or a MISS condition. This error is detected by the second BF, which signals that the couple $\{key,entry\}$ in not present in the TCAM. If, due to hash collision, a false positive occurs in the BF the error is undetected. The experimental data presented in the next section shows that the occurrence of both an error in the TCAM and a false positive in the BF are very low, even with very small size BF. The experiments we performed show up to 100% error detection, when the BFs are large enough to avoid many false positives. The two BFs detect disjoint types of errors and the fraction of errors detected by the two BFs strictly depends on the TCAM configuration and on the input keys applied to the TCAM. In particular, when the TCAM is configured to be used as a full classifier [12], the first BF becomes useless, since the MISS condition is never present for the error-free TCAM.

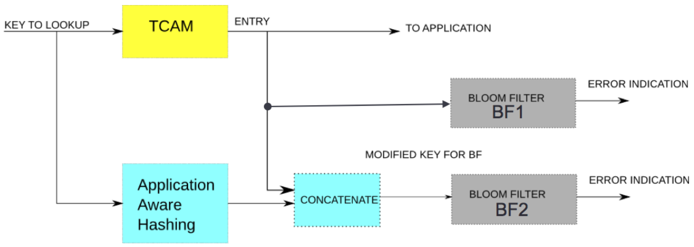


Fig. 4. Two Bloom Filters approach.

The second architecture shown in Fig. 5 uses M Bloom Filters and splits the content stored in the TCAM into M different subsets. Each item is inserted in only in the i-th Bloom Filter, the one corresponding to the condition $i = entry \bmod M$. We remark that, since the false positive rate of Bloom

Filters depends on the ratio between the number of inserted items and the size of the Bloom Filter, the false positive rate is independent by M, the number M in which the TCAM configuration set is divided. Unlike the first architecture, this architecture presents a lower error detection latency, because the query to the Bloom Filters can be done in parallel with the TCAM lookup. The TCAM output is only used to select which response of the Bloom Filter should be taken into account. The drawback of this architecture is that undetected errors can be caused both by hash collision, like in the previous architecture, or when an erroneous entry ($i.e.$ the response of a TCAM affected by a SEU) and the correct one ($i.e.$ the one stored in the Bloom Filter set) have the same value modulo M. The use of $M = 16$ or $M = 32$ allows us to reduce this aliasing probability to 6% or 3%. [1] When no false positive occurs, the set of Bloom Filters provides a one-hot encoded word as output, while with the occurrence of a false positive in some of the Bloom Filters that does not correspond to the requested key, the output word has multiple bits to 1. If a false positive occurs simultaneously with an error in the TCAM, the error is undetected if the i index given by the modular operation selects one of the Bloom Filters affected by the false positive.

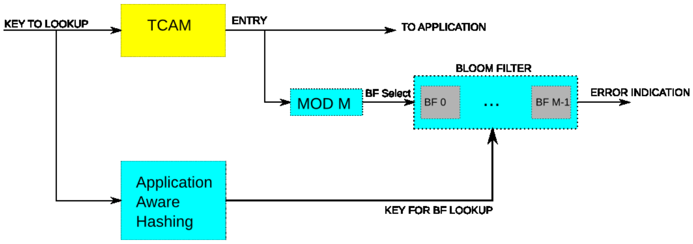


Fig. 5. Modular detection approach.

The algorithm for error detection using the second architecture is given in Fig. 6. First the algorithm checks if all the Bloom Filters are 0. This correspond to the condition of a MISS in the TCAM. If the TCAM gives MISS as a response, then we assume that no error has occurred. Otherwise, if the TCAM responds with an $entry$ we assume that a pseudo HIT error is occurred. Instead, if some of the Bloom Filters give 1 as a response, we select the Bloom Filter with index $i = entry$ mod M and check if it is equal to 1. In this case we assume that no errors have occurred, otherwise we signal the presence of an error in the TCAM. This check condition also detects a pseudo MISS in the TCAM (not shown in Fig. 6 for clarity), since the MISS signal is not compatible with some $BF(\cdot) = 1$. The condition of $BF(i) = 1$ and $i = entry$ mod M can also be due to the concurrent occurrence of a false positive in the Bloom Filter and to an error in the TCAM. The experimental result shown in the next section prove that this is very unlikely.

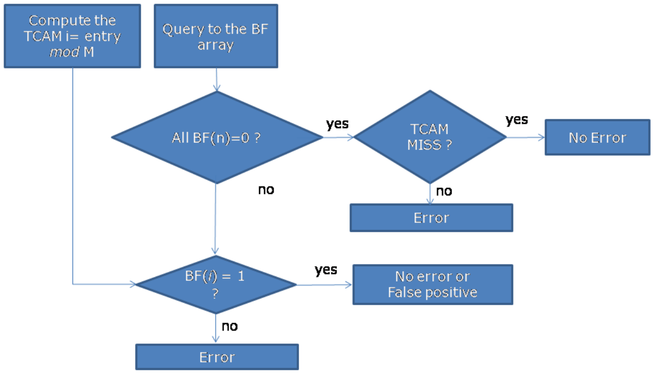


Fig. 6. Algorithm for detection.

VI. Experimental results

In order to evaluate the effectiveness of the proposed methods, a set of simulations have been performed. A C++ software model of TCAM and Bloom Filters has been designed and both the TCAM and the Bloom Filters have been configured using a real data set provided by Cisco Systems. Ten thousand simulations were run where a fault was injected into the TCAM for each run. The activation of the fault was determined by comparing the output of the TCAM under test, with the output from a golden TCAM. For the activated faults, the number of faults detected by our architectures is reported. We compute the overhead as the number of bits of storage in the Bloom Filter with respect to the number of configuration bits of the TCAM. This is a highly pessimistic, since the real silicon area required to implement a TCAM cell is roughly twice the area of an SRAM cell, since the TCAM cell also contains matching logic [1]. We do not evaluate the logic required to implement the hash function of the Bloom Filters, since it is composed of only a modest number of XOR gates [17], and therefore is negligible with repect to the area required for the memory. The two architectures presented in the previous section have been simulated, and the size of the Bloom Filters has been varied, in order to study the impact of false positives on the efficiency of the proposed methods. For the second architecture we limit our analysis to $M = 4$, since the TCAM configuration being used was too small to benefit from a larger value of M.

Size of one BF (bits)	Error detected by BF1	Error detected by BF2	Total Error coverage	Total overhead
2K	2013	1307	100%	32%
1K	2332	1306	100%	16%
512	2343	1242	100%	8%
256	2068	1342	98.7%	4%
128	1911	1168	89.1%	2%

TABLE I
Results of fault injection experiments for the first architecture

[1] This effect can be totally eliminated using a suitable constraint in the TCAM configuration. The key corresponding to the erroneous entry and the key corresponding to the correct one differ for a maximum of t contiguous bits, where t is the maximum length of a multiple bit upset. The constraint of avoiding that these couples of $keys$ have the same value of $entry$ modulo M can be easily fulfilled when $M = 16$ or $M = 32$.

The results reported in Table I show that the errors in the TCAM are detected by one of the two Bloom Filters used in the first architecture (BF1 checking $value$ and BF2 checking $\{key,entry\}$) . As expected, the set of errors detected by the two Filters is disjoint, and the two Filters are able to detect 100% of injected faults when the size of the Bloom Filters is sufficiently large. Even without considering that SRAM memory cells are smaller than TCAM cells, the ratio 8% is less than the nominal 12.5% overhead that would be required for byte-wise parity. When the overhead is further reduced to 4% or 2% the effect of false positives in the Bloom Filters produces some undetected errors, decreasing the error coverage to 98.7% and 89.1% respectively.

In Table II the results for the second architecture are reported.

size of one BF (bits)	error detected	total number of errors	Error Coverage	Total overhead
1K	3211	3662	87.6%	32%
512	2908	3326	87.4%	16%
256	3010	3461	86.9%	8%
128	3130	3669	85.3%	4%
64	4276	3507	82%	2%

TABLE II
RESULTS OF FAULT INJECTION EXPERIMENTS FOR THE SECOND ARCHITECTURE

As expected, even with a large Bloom Filter, the error coverage is less than 100%, due to the aliasing of the modular operation. However, the ratio of undetected error is less than expected, (should be 25%, since $M = 4$), since the aliasing does not occur when the fault inside the TCAM produces a pseudo-HIT. In fact, in this case all the Bloom Filters give 0 as response,while the TCAM signals the presence of the key in an erroneous $entry$. The condition that all BFs are equal to 0 allows detecting the error without any modular operation, and therefore without the issue of aliasing. Aliasing still remains an issue for multi-HIT errors [10]. When the size of the Filters is sufficiently large, no false positives occur, and the error coverage is slightly independent of the size of the Filters. It can be noticed that the overheads have the same values of the previous architecture, as expected.

When the overhead is further reduced to 4% or 2% the effect of false positive in the Bloom Filters is less pronounced that in the first architecture. A further analysis of the data for these cases shows that with an overhead of 4%, the ratio of false positive is 2%, to bits of the Bloom Filter set are set to 1 and the other 2 bits are still 0. Therefore, the concurrent event of a error in the TCAM and of a false positive, can be detected with a 50% probability. Instead, with 2% overhead, the false positive rate increases up to 30%, but only in few cases (about 3%) three bits are set to one, while for 27% only two bit are set to one. This behavior allows us still to detect some errors even if the Bloom Filters are affected by false positive.

VII. CONCLUSIONS AND FUTURE WORK

This paper presents a method for on-line error detection in TCAMs. Unlike existing TCAM error detection techniques, the proposed method does not require any modification to the TCAM structure and can thus be added to an existing system. Unlike approaches based on background scrubbing to identify errors, the proposed technique identifies errors immediately when the lookup is performed. in the TCAM, therefore allows a low detection latency. The expansion of the don't care bits in the TCAM configuration has been managed by using a set of rules that exploit the types of applications that rely on TCAMs. Based on the application, different hash function are used to insert and query the items in the Bloom Filters - an approach that is called Application Aware Hashing. Two different architectures for error detection have been proposed. The efficiency of the two architectures has been evaluated with a real data set provided by Cisco Systems. Extension to dynamically insert and evict entries in the Bloom Filter is part of on-going work. Similarly to the proposal of [9] the use of Counting Bloom Filters enables the insertion and deletion of items in Bloom Filters. Techniques to minimize the area overhead impact of implementing counting Bloom Filters are also under investigation.

REFERENCES

[1] K. Pagiamtzis, A. Sheikholeslami, “Content Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey”, IEEE Journal of Solid-State Circuits, vol. 41, n. 3, pp. 712–727, Mar. 2006.

[2] L. Chisvin, R. J. Duckworth, “Content-addressable and associative memory: alternatives to the ubiquitous RAM”, IEEE Computer, vol.22, no.7, pp. 51–64, 1989.

[3] H.Chao, ”Next generation routers”, Proc. IEEE , vol. 90, no.9, pp. 1518–1558, Sep. 2002.

[4] Cisco Catalyst 6500 Series Switches Data Sheet, available Online

[5] W.W. Peterson, E.J. Weldon, “Error-correcting codes”, The MIT Press, 1972.

[6] G.C. Cardarilli, M. Ottavi, S. Pontarelli, M. Re, A. Salsano, “A Fault-Tolerant Solid State Mass Memory for Space Applications,” IEEE Transactions on Aerospace and Electronic Systems, vol. 41, n. 4, pp. 1353–1372, October 2005.

[7] G.C. Cardarilli, M. Ottavi, S. Pontarelli, M. Re, A. Salsano “Data integrity evaluations of Reed Solomon codes for storage systems, ” Defect and Fault Tolerance in VLSI Systems, 2004. DFT 2004.

[8] N. Kanekawa, E. H. Ibe, T. Suga, Y. Uematsu, “Dependability in Electronic Systems: Mitigation of Hardware Failures, Soft Errors, and Electro-Magnetic Disturbances,” ISBN 978-1-4419-6714-5, Springer Verlag, 2010.

[9] S. Pontarelli, M. Ottavi, Error Detection and Correction in Content Addressable Memories by Using Bloom Filters, accepted for publication on IEEE Transactions on Computers.

[10] S. Pontarelli, M. Ottavi, A. Salsano, “Error Detection and Correction in Content Addressable Memories”, in Proc. of the 25th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'10), October 2010.

[11] B. Bloom. “Space/Time Tradeoffs in Hash Coding with Allowable Errors”. Communication of ACM, no, 13, issue 7, pp. 422-426, 1970.

[12] K. Lakshminarayanan, A. Rangarajan and S. Venkatachary, “Algorithms for advanced packet classification with ternary CAMs,” ACM SIGCOMM Computer Communication Review, vol.35, no. 4, pp. 193-204, 2005.

[13] S. Dharmapurikar, P. Krishnamurthy and D.E. Taylor, “Longest prefix matching using Bloom Filters,” IEEE/ACM Transactions on Networking, vol.14, no. 2,pp. 397-409, 2006.

[14] N. Azizi, F. N. Najm, “A family of cells to reduce the soft-error-rate in ternary-CAM,” Design Automation Conference, 2006 43rd ACM/IEEE, pp. 779-784, 2006.

[15] S. Krishnan, R. Panigrahy, S. Parthasarathy, “Error-Correcting Codes for Ternary Content Addressable Memories” IEEE Transactionson Computers, vol.58, no. 2, pp. 275-279, 2009.

[16] Wright et al, US Patent 7,254,748.

[17] M. V.Ramakrishna, E. Fu, E. Bahcekapili, “Efficient hardware hashing functions for high performance computers” Computers, IEEE Transactions on, Vol. 46 no. 12 pp:1378-1381, 1997.