This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Error Detection and Correction in SRAM Emulated TCAMs

Pedro Reviriego, Salvatore Pontarelli, and Anees Ullah

***Abstract*— Ternary content addressable memories (TCAMs) are widely used in network devices to implement packet classification. They are used, for example, for packet forwarding, for security, and to implement software-defined networks (SDNs). TCAMs are commonly implemented as standalone devices or as an intellectual property block that is integrated on networking application-specific integrated circuits. On the other hand, field-programmable gate arrays (FPGAs) do not include TCAM blocks. However, the flexibility of FPGAs makes them attractive for SDN implementations, and most FPGA vendors provide development kits for SDN. Those need to support TCAM functionality and, therefore, there is a need to emulate TCAMs using the logic blocks available in the FPGA. In recent years, a number of schemes to emulate TCAMs on FPGAs have been proposed. Some of them take advantage of the large number of memory blocks available inside modern FPGAs to use them to implement TCAMs. A problem when using memories is that they can be affected by soft errors that corrupt the stored bits. The memories can be protected with a parity check to detect errors or with an error correction code to correct them, but this requires additional memory bits per word. In this brief, the protection of the memories used to emulate TCAMs is considered. In particular, it is shown that by exploiting the fact that only a subset of the possible memory contents are valid, most single-bit errors can be corrected when the memories are protected with a parity bit.**

***Index Terms*— Field-programmable gate arrays (FPGA), soft errors, ternary content addressable memories (TCAM).**

I. Introduction

Soft errors are a major concern for modern electronic circuits and, in particular, for memories [1]. A soft error can change the contents of the bits stored in a memory and cause a system failure. The soft error rate in terrestrial applications is low. For example, in [2], it was estimated that for a 65-nm static random access memory (SRAM) memory, the bit error rate was on the order of 10^{-9} errors per year. That would translate to only one error per year for a system that uses 1 Gbit of memory. However, even such a low error rate is a big concern for critical applications such as communication networks on which the network elements such as routers have to provide a high level of reliability and availability. Therefore, soft errors are an important issue when designing routers or other network elements, and manufacturers take them into account and incorporate error mitigation techniques [3], [4]. For example, error detection and correction codes are commonly used to protect memories [5]. A parity bit can be added to each memory word to detect single-bit errors, or a single-error correction (SEC) code can be used to correct them. These codes require additional bits per word thus, increasing the memory size and also some logic to write and read from the memory. For example, for a 16-bit word, an SEC code requires 5 bits while a parity check requires only one.

Manuscript received April 4, 2018; revised July 14, 2018 and September 7, 2018; accepted October 12, 2018. *(Corresponding author: Pedro Reviriego.)*

P. Reviriego was with Universidad Antonio de Nebrija, 28015 Madrid, Spain. He is now with Universidad Carlos III de Madrid, 28911 Madrid, Spain (e-mail: revirieg@it.uc3m.es).

S. Pontarelli is with Consorzio Nazionale Interuniversitario per le Telecomunicazioni, 00133 Rome, Italy (e-mail: salvatore.pontarelli@uniroma2.it).

A. Ullah is with the Sir Syed CASE Institute of Technology, Islamabad 44000, Pakistan (e-mail: anees.ullah@case.edu.pk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2018.2877131

Ternary content addressable memories (TCAMs) are a special kind of content addressable memories [6] that support do not care bits (commonly denoted as “x”) that match both a zero and a one. TCAMs are widely used in networking applications to perform packet classification [7]. They can be implemented as standalone devices or integrated as part of networking application specific integrated circuits (ASICs) [8]. The TCAM memory cells different from normal SRAM cells, in which they check the incoming value for a match to the stored value that can be for each bit 0, 1, or x. The results from all the words are then sent to a priority encoder that returns the match with the highest priority. This comparison and selection logic introduces a significant overhead in terms of area and power consumption relative to that of an SRAM memory. Protecting TCAMs against soft errors is challenging as error correction codes (ECCs) are not easily applicable because all words are checked in parallel. This means that a decoder per word would be needed, which would lead to a very large area and power overhead. A number of schemes have been proposed to protect TCAMs that are based, for example, on replicating part of the rules or on using other structures such as Bloom filters to check the results of the TCAM searches [9], [10].

Field-programmable gate arrays (FPGAs) provide a flexible platform to implement systems. In particular, they provide a vast amount of logic and memory resources that can be configured to implement a given functionality. This makes them attractive for networking applications [11]. However, they do not include CAM or TCAM blocks because FPGAs are also used in many applications that are not related to networking. For binary CAMs that is not an issue as they can be easily emulated using cuckoo hashing and RAM memories with a small cost overhead [8]. TCAMs are also emulated using the logic and memory resources, but in this case, the overheads are much larger (making emulation not competitive for ASIC implementations). To emulate the TCAMs in FPGAs, a number of schemes have been proposed in the literature [12]–[16]. Some of them implement the TCAM memory cells with FPGA flip-flops and logic [12]. This approach has limited scalability in terms of the TCAM size, and therefore, schemes based on using the SRAM memories embedded in the FPGA [13]–[16] are preferred and implemented by FPGA vendors [17].

When SRAM memories are used to implement a TCAM, a large number of bits are used for each TCAM cell. For example, in [13], it has been shown that more than 55 bits of the Xilinx FPGAs block RAMs (BRAMs) are needed for each single TCAM bit. In case of distributed RAMs, 6 bits are needed for each TCAM bit.

This means that a large number of memory bits are used and thus the probability of suffering soft errors increases. To protect them, ECCs can be used, but as discussed before, they add additional memory overhead [18]. For TCAMs that are emulated using logic and flip-flops, protection can be implemented by using triple modular redundancy that triplicates the flip–flops and adds voting logic to correct errors, thus requiring a large resource overhead.

In this brief, it is shown that the specificity of the contents stored in memories used to emulate TCAMs can be exploited to implement an efficient error correction method. In particular, when memories are protected with a parity bit to detect single-bit errors, the proposed scheme will be able to correct most of the single-bit errors. This makes the technique attractive to improve the

1063-8210 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

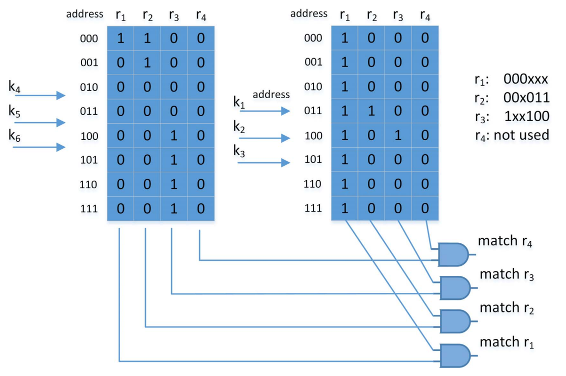

Fig. 1. Example of a TCAM with 6-bit keys and four rules emulated using two SRAMs.

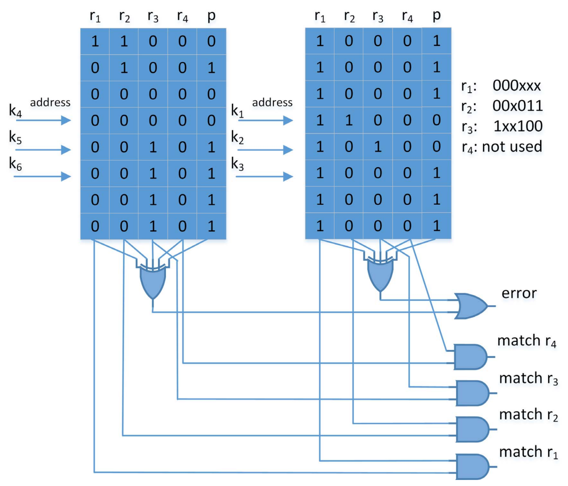

Fig. 2. Parity protected TCAM with 6-bit keys and four rules emulated using two SRAMs.

reliability of FPGA-based TCAM implementations without incurring large overheads in resource usage. In more detail, the proposed implementation reduces the FPGA slices required for protection by at least 50% when compared to an SEC protection for common TCAM sizes.

This brief is organized as follows. Section II discusses FPGA-based TCAM implementations. The proposed error correction scheme is presented in Section III and evaluated in Section IV. Finally, the conclusions are summarized in Section V.

II. FPGA-Based TCAM Implementations

There are two main alternatives to implement TCAMs on FPGAs. The first one is to use the FPGA logic resources and flip-flops to implement the TCAM cells and match lines. The second is to use the block memories inside the FPGA [13].

In the first alternative, the bits of the rules are stored in flip-flops. As discussed before, each bit can take three possible values: 0, 1, and x. For example, a flip-flop can be used to store if the bit is 0 or 1 and another flip-flop that acts as a mask and is set when the bit is do not care [12]. Then, the programmable logic can be used to implement the comparison against the key. This alternative uses many resources per rule and, therefore, cannot be used to implement large TCAMs with tens of thousands of rules of more than 100 bits that operate at high speed.

The second alternative is based on the use of the embedded memories available in the FPGA. To do so, the key is divided into smaller blocks of b bits. Then, a rule can be emulated using a 1-bit memory of 2^b positions for each block. When searching for a key, all the memories are accessed using the corresponding key bits and if all the positions read have a one, a match is detected. In general, k rules can be implemented by using a $k - bit$ memory of 2^b positions for each block. This is best illustrated with an example. Let us consider a key of 6 bits that is divided into two blocks of 3 bits. Then, a TCAM with four rules can be implemented as shown in Fig. 1. It can be observed that the memories have $2^3 = 8$ positions and a width of 4 bits. The leftmost memory is accessed using the upper 3 bits of the key and the other with the lower three. Those bits are used to determine the address of the position read from the memory. The rules stored in each bit are also shown in Fig. 1. Let us consider a search for key: 000011. We would access the first position (address 000) on the leftmost memory reading 1100 and the four position (address 011) on the other memory reading 1100. After performing AND there would be a match only for rules r_1 and r_2. Looking now at the rules, it can be observed that rules that are not used (r_4) have zeros in all the memories and positions. For the rest of the rules, the number of ones in a given memory depends on the number of x bits that the rule has on the key bits used as addresses on that memory. When there are no x bits, only one position has a one, when there is one x bit, two positions have a one, when there are two x bits, four positions have a one, and so on. In general, if there are n_x bits that are x, there will be 2^{n_x} ones on the memory.

Considering now the implementation cost, since each block stores b bits of a rule and requires 2^b bits of SRAM memory. The cost in SRAM bits per TCAM bit in this scheme is $2^b/b$ [13]. Therefore, it would seem that smaller values of b are more efficient. However, this is not entirely true as the logic needed to combine the blocks together increases with the number of blocks. It should also be mentioned that a large physical memory can be split into several blocks so that each one implements b bits of a rule. Then, several memory accesses are needed to complete a search operation, this can be mitigated by operating the memory at a larger speed or using multiport memories [16].

For Xilinx FPGAs, there are two types of memory resources: lookup table random access memories (LUTRAMs) and BRAMs. The first ones are built with the same lookup tables (LUTs) that are used to implement the logic and are generally small with 32 or 64 positions. On the other hand, BRAMs are larger having 36 K bits that can be configured with different word sizes, the largest being 72 bits that corresponds to 512 positions. Therefore, LUTRAMs have a much lower cost per bit ($2^5/5$) than BRAMs ($2^9/9$). However, the total number of memory bits available is larger for BRAMs than for LUTRAMs.

A key observation for the protection of the SRAM-based TCAM implementations is that the contents of the SRAMs are determined by the rules stored and that only a few combinations of all possible values are used. This means that the SRAM contents have an intrinsic redundancy that could potentially be used to protect the memories. This idea is explored in the rest of this brief.

III. Error Detection and Correction in SRAM-Based TCAMs

The scheme proposed to protect the memories used to emulate the TCAM uses a per word parity bit to detect single-bit errors. Then, once an error is detected, the intrinsic redundancy of the memory contents is used to try to correct the error. The implementation of the parity protection is shown in Fig. 2 where p corresponds to the parity bit. It can be seen that in addition to the match signal, an error signal is generated when there is a mismatch between the

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

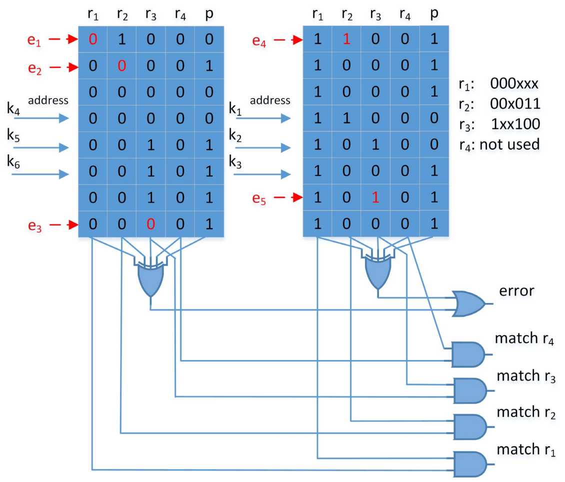

Fig. 3. Examples of single-bit errors on a parity protected TCAM with 6-bit keys and four rules emulated using two SRAMs.

stored parity and the recomputed one. This is a standard parity protection that can detect all single-bit errors [5]. Detecting the error on every access is crucial to avoid incorrect results on search operations.

Let us now assume that a single-bit error has occurred on a given word and that it is detected with the parity check. Upon error detection, we can check the contents of the memory to try to correct the error. A first attempt could be to read all the words in the memory and count the number of positions that have a one for each rule. Let us denote that number as the weight of the rule in that memory. For example, in the leftmost memory of Fig. 2, r_1 would have a weight of 1, r_2 of 2, and r_3 of 4. This can help us identify the erroneous bit as the weight for an error-free rule can only be 0, 1, 2, 4, and 8 for an 8-position memory. To further discuss the error correction process, let us focus on the examples of single-bit errors shown in Fig. 3. For example, e_3 affects r_3 on the leftmost memory by changing its weight from 4 to 3. Since 3 is not a valid value, after detecting the parity error, we would identify that the erroneous bit is that in r_3 and we would correct it. This approach would be effective for rules that have a weight larger than two, i.e., they have two or more "x" bits on the key bits that correspond to that memory. On the other hand, for rules with a lower weight, checking the weight alone may not be enough. Let us now consider a rule with weight two. Then, an error that changes a zero to a one will change the weight to three and the error will be corrected. However, when a one is changed to a zero (as in e_2), then the new weight would be one that is a valid value and the error cannot be corrected. This, however, is less likely to occur as only 2 positions have a one. If we now consider a weight one rule, an error that sets another bit to one would produce a weight of two that is also valid. However, not all weight two combinations are possible. This is clearly seen when looking at e_4. In that case, the values of r_2 that are one would correspond to key values 000 and 011 and those do not correspond to a valid rule. In general, only positions that correspond to key values that are at distance one from the original value will not be detected. On the other hand, an error that sets to zero the position that was one in a weight one rule can be corrected by checking if the rule has zero weight on the other memories. If that is the case, then the rule is disabled and the bit is not in error. Otherwise, the rule had a weight of one and the error is corrected. Finally, an error in a rule that had a weight of zero can also be corrected by checking the weight of the rule on the other memories.

TABLE I
PERCENTAGE OF CORRECTABLE SINGLE-ERROR PATTERNS FOR COLUMNS OF DIFFERENT WEIGHTS

Weight	Coverage
0	100
1	$100 \cdot (1 - b/2^b)$
2	$100 \cdot (1 - 2/2^b)$
≥ 4	100

TABLE II
PERCENTAGE OF CORRECTABLE SINGLE-ERROR PATTERNS FOR BLOCK MEMORIES

Weight / b	5	6	7	8	9
0	100	100	100	100	100
1	84.4	90.6	94.5	96.9	98.2
2	93.8	96.9	98.4	99.2	99.6
≥ 4	100	100	100	100	100

The previous discussion shows that by using the intrinsic redundancy of the memory contents, many single-bit error patterns could be corrected. Let us now quantify the fraction of single-bit error patterns that can be corrected for each weight in a memory of 2^b positions.

1) Weight zero: all patterns can be corrected.
2) Weight one: all except those that set a bit to one for a position with an address at distance one, this corresponds to $1 - b/2^b$.
3) Weight two: all patterns can be corrected except the two that set a position with a one to a zero, this corresponds to $1-2/2^b$.
4) Weight four or larger: all patterns can be corrected.

It can be seen that most of the error patterns are corrected. This is better seen in Table I that illustrates the percentage of correctable patterns for columns of different weights. The only cases where not all errors can be corrected are weight one and two, and for those, the percentage will approach 100% when b is large. The percentage of errors that can be corrected for different values of b is shown in Table II. It can be seen that even for small memories ($b = 5$ corresponds to 32 positions), the error coverage is close to 90% in the worst case. For larger memories, the coverage is over 95% and gets close to 100%. For example, for $b = 9$, the coverage is over 98% in the worst case. This shows the effectiveness of the proposed scheme in correcting single bit errors when the memories are protected with a parity bit.

The pseudocode of the proposed correction algorithm is shown in Algorithm 1. The process starts when a parity error is detected when reading a word from a block memory. To correct the error, we need to identify the bit (or column) affected by the error. To do so, in the first phase, all the positions in the block are read and the column weights are computed by adding the ones seen in each column. Then, the second phase checks different cases for the column weight to try to identify the erroneous column. If that occurs, the bit of that column in the word that had the parity error is the erroneous bit and it is corrected. In the algorithm, this second phase starts by checking if there is a column that has an illegal weight. As discussed before, the only valid column weights are: $0, 1, 2^i$ for $i = 1, 2, \ldots, b$. Therefore, if a column has, for example, weight three, then it is the erroneous one. If the erroneous bit is found, it is corrected and the process ends. Otherwise, we proceed to check columns that have zero weight. Those should correspond to TCAM entries that are not used and should have zero weight on all the other memory blocks. Therefore, we check if they have also zero weight on another block. If not, the error has been found and it is corrected. If all the columns with zero weight have also weight zero on another block, we proceed

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 1 Proposed Algorithm for Error Correction

```
Require: Parity error detected in a memory word
 1: Read memory and compute column weights
 2: if there is a column with illegal weight then
 3:     Correct that bit on the erroneous word
 4:     return error corrected
 5: end if
 6: if there are columns with zero weight then
 7:     Read another memory
 8:     Compute the weights of those columns
 9:     if any has a non zero weight on the other memory then
10:         Correct that bit on the erroneous word
11:         return error corrected
12:     end if
13: end if
14: if there are columns with weight one then
15:     Read another memory
16:     Compute the weights of those columns
17:     if any has zero weight on the other memory then
18:         Correct that bit on the erroneous word
19:         return error corrected
20:     end if
21: end if
22: if there are columns with weight two then
23:     Read the memory
24:     Check the patterns of those columns
25:     if any has an illegal pattern then
26:         Correct that bit on the erroneous word
27:         return error corrected
28:     end if
29: end if
30: return uncorrected error
```

to check columns of weight one. For that, we check if they have zero weight on another block. If that is the case, that column is the one that suffered the error and we correct it. If not, we proceed to the last step in which columns of weight two are checked. To do that, the two addresses of the two positions that contain a one are XOR*ed*. If the result has more than a one, the column has suffered an error and we correct it. If that does not happen, then we have suffered one of the few errors that are not correctable. The overall process can easily be implemented in a soft processor that is present in many FPGA-based networking applications to manage the control functions [11].

IV. Evaluation

To evaluate the benefits of the proposed scheme, it has been implemented using Vivado Design Suite 2016.3 in a Xilinx Artix-7 xc7a100tcsg324 FPGA that is part of a Nexys4 double data rate board. Both distributed memory (LUTRAMs) and BRAMs implementations are considered. In both cases, the memory sizes that minimize the memory bits needed per TCAM bit are used. This corresponds to a configuration of 32 positions LUTRAMs corresponding to five key bits per memory, and for the second option, the BRAM is configured as 512 positions of 72 bits. The soft error coverage of the proposed technique for each option corresponds to the first and last columns of Table II. Similarly, if the memory blocks were split into several smaller logical blocks the soft error coverage could be checked in Table II. The unprotected memory (Fig. 1) and the memory protected with an SEC code have also been implemented for comparison. The protection with an SEC code is illustrated in Fig. 4.

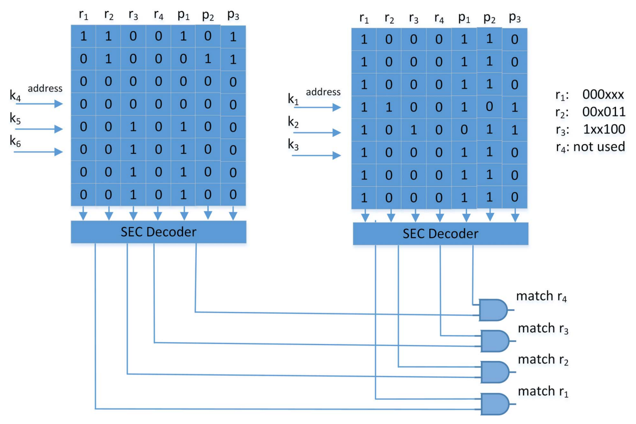

Fig. 4. SEC protected TCAM with 6-bit keys and four rules emulated using two SRAMs.

TABLE III
Resource Utilization and Delay of a TCAM Emulated Using Distributed Memory on Xilinx Artix-7 FPGA

Rules x Width	Technique	LUTs	LUTRAMs	Slices	Delay
64x40	Unprotected	704	512	179	19.26
64x40	Proposed	786	520	255	19.92
64x40	SEC	1232	568	498	21.70
512x40	Unprotected	5613	4096	1500	28.37
512x40	Proposed	6449	4104	1830	31.03
512x40	SEC	13077	4176	3656	47.34
1024x40	Unprotected	11149	8192	2997	41.03
1024x40	Proposed	12837	8200	3595	47.75
1024x40	SEC	27308	8280	7258	57.32
2048x40	Unprotected	17692	16384	4439	72.67
2048x40	Proposed	21258	16392	6811	73.40
2048x40	SEC	46811	16472	12666	78.88

TABLE IV
Resource Utilization and Delay of a TCAM Emulated Using BRAM Memory on Xilinx Artix-7 FPGA

Rules x Width	Technique	LUTs	BRAMs	Slices	Delay
64x40	Unprotected	128	5	40	10.33
64x40	Proposed	219	5	71	14.46
64x40	SEC	1218	5	342	18.40
512x40	Unprotected	512	38	165	17.20
512x40	Proposed	1028	38	342	19.83
512x40	SEC	4807	38	1534	29.25
1024x40	Unprotected	1027	73	339	36.88
1024x40	Proposed	2055	73	694	38.68
1024x40	SEC	11083	73	3533	40.05
2048x40	Unprotected	8193	135	2470	73.06
2048x40	Proposed	10253	135	3174	74.98
2048x40	SEC	23548	135	6951	73.00

It can be observed that additional parity bits and an SEC decoder are needed on each memory block to correct the errors. The number of additional bits will depend on the number of rules, for example, 6 bits are needed for a 32-bit word (rules) and 7 bits for a 64-bit word (rules).

The FPGA resources needed and the delay in nanoseconds for the three options are shown in Table III for the LUTRAM implementation and in Table IV for the BRAM implementation. In both cases, TCAMs of with 40-bit keys and 64, 512, 1024, and 2048 rules are considered. As discussed before, the LUTRAMs correspond to a 32×1 memory that is accessed using 5 bits of the keys. Therefore, 8 memories are required to process the 40 bits of the key.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

For BRAMs, each memory has 512 positions that can cover 9 bits of the key, and therefore, only five memories are needed. From the results in Table III, it can be seen that the proposed scheme reduces by more than half the amount of resources needed both in terms of LUTs used for logic and slices compared to an SEC protection, The same observation holds for LUTs and slices in Table IV with even larger reductions. In the case of LUTRAMs and BRAMs, the reductions are lower. This can be explained as the number of parity check bits needed to implement SEC is not large, and additionally, for BRAMs, they have a 72-bit width so that in all cases there are spare bits that can be used for the parity check bits. Another advantage of the proposed scheme is that it has a lower impact on delay than the use of SEC protection. Finally, it is interesting to relate the resources used with those available in the FPGA. In more detail, the device used has 63 400 LUTs of which only 19 000 can be configured as LUTRAMs and 135 BRAMs. This means that a 2048 $\times$ 40 TCAM uses almost all LUTRAMs in the LUTRAM implementation whereas in the case of BRAMs, all BRAMs are used and part of the memory blocks need to be emulated using LUTRAMs. In both cases, an SEC protected implementation leaves much fewer LUTs to implement the rest of the system than the proposed scheme.

In summary, the proposed scheme can be an interesting option to improve the protection of the memories used to emulate TCAMs on FPGAs. However, it should be noted that the proposed technique does not correct all single-bit errors and that it requires some overhead in the software controller of the TCAM to perform correction once an error is detected. Therefore, whether to use it or not will depend on the design requirements in terms of reliability and on the resources available.

V. Conclusion

In this brief, a technique to protect the SRAMs used to emulate TCAMs on FPGAs has been proposed. The scheme is based on the observation that not all values are possible in those SRAMs, and thus, there is some intrinsic redundancy of the memory contents. This redundancy is used to correct most single-bit error patterns when the memories are protected with a parity bit to detect errors. The proposed technique reduces significantly the resources needed to protect the memories and can be an interesting option for designs on which reliability is a concern but resources are limited.

The idea presented in this brief can be extended to other memory configurations. For example, it can be used to detect errors on an unprotected memory by periodically scrubbing the contents to check their correctness. It could also be used when the memory is protected with a more powerful code that can detect several bit errors to correct multiple bit errors. For example, for a memory protected with an SEC code, double-bit error patterns could be detected and then use the intrinsic redundancy of the memory contents to correct them.

References

[1] N. Kanekawa, E. H. Ibe, T. Suga, and Y. Uematsu, *Dependability in Electronic Systems: Mitigation of Hardware Failures, Soft Errors, and Electro-Magnetic Disturbances*. New York, NY, USA: Springer-Verlag, 2010.

[2] J. L. Autran *et al.*, "Soft-errors induced by terrestrial neutrons and natural alpha-particle emitters in advanced memory circuits at ground level," *Microelectron. Rel.*, vol. 50, no. 9, pp. 1822–1831, Sep. 2010.

[3] A. L. Silburt, A. Evans, I. Perryman, S. J. Wen, and D. Alexandrescu, "Design for soft error resiliency in Internet core routers," *IEEE Trans. Nucl. Sci.*, vol. 56, no. 6, pp. 3551–3555, Dec. 2009.

[4] A. Evans, S.-J. Wen, and M. Nicolaidis, "Case study of SEU effects in a network processor," in *Proc. IEEE Workshop Silicon Errors Logic-Syst. Effects (SELSE)*, Mar. 2012, pp. 1–7.

[5] 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, Mar. 1984.

[6] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE J. Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, Mar. 2006.

[7] F. Yu, R. H. Katz, and T. V. Lakshman, "Efficient multimatch packet classification and lookup with TCAM," *IEEE Micro*, vol. 25, no. 1, pp. 50–59, Jan./Feb. 2005.

[8] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proc. ACM SIGCOMM*, 2013, pp. 99–110.

[9] I. Syafalni, T. Sasao, and X. Wen, "A method to detect bit flips in a soft-error resilient TCAM," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 6, pp. 1185–1196, Jun. 2018.

[10] S. Pontarelli, M. Ottavi, A. Evans, and S. Wen, "Error detection in ternary CAMs using Bloom filters," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2013, pp. 1474–1479.

[11] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sep./Oct. 2014.

[12] M. Irfan and Z. Ullah, "G-AETCAM: Gate-based area-efficient ternary content-addressable memory on FPGA," *IEEE Access*, vol. 5, pp. 20785–20790, 2017.

[13] W. Jiang, "Scalable ternary content addressable memory implementation using FPGAs," in *Proc. ACM ANCS*, San Jose, CA, USA, Oct. 2013, pp. 71–82.

[14] Z. Ullah, M. K. Jaiswal, and R. C. C. Cheung, "E-TCAM: An efficient SRAM-based architecture for TCAM," *Circuits, Syst., Signal Process.*, vol. 33, no. 10, pp. 3123–3144, Oct. 2014.

[15] A. Ahmed, K. Park, and S. Baeg, "Resource-efficient SRAM-based ternary content addressable memory," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 4, pp. 1583–1587, Apr. 2017.

[16] I. Ullah, Z. Ullah, and J.-A. Lee, "Efficient TCAM design based on multipumping-enabled multiported SRAM on FPGA," *IEEE Access*, vol. 6, pp. 19940–19947, 2018.

[17] *Ternary Content Addressable Memory (TCAM) Search IP for SDNet: SmartCORE IP Product Guide, PG190 (v1.0)*, Xilinx, San Jose, CA, USA, Nov. 2017.

[18] V. Gherman and M. Cartron, "Soft-error protection of TCAMs based on ECCs and asymmetric SRAM cells," *Electron. Lett.*, vol. 50, no. 24, pp. 1823–1824, 2014.