Handling errors in i-TLBs using a simple parity method

***Abstract*—The instruction Translation Lookaside Buffer (i-TLB) stores the most accessed addresses. Single Event Upsets (SEUs) and Multiple Cell Upsets (MCUs) can affect i-TLB's memory cells, reducing the system performance. Error Correction Codes (ECC) might protect i-TLBs from MCUs, but these solutions consume more area and power. This paper presents a simple parity method that protects i-TLBs using the principle of locality and the Hamming distance. The proposed method does not require extra bits and only calculates the parity of the least significant bits (LSBs). The results point out that it is possible to protect i-TLBs from SEUs and MCUs by protecting only the LSBs.**

Keywords— translation lookaside buffer, multiple bit upset, fault protection, false positives.

I. INTRODUCTION

Operating systems allocate running programs in pages on the virtual memory address space. A Memory Management Unit (MMU) translates virtual memory addresses into physical ones using page tables, stored in RAM. The translation process is slow due to the time required to access the page table. To speed-up memory access, a dedicated circuit called Translation Lookaside Buffer (TLB) stores the most referenced pages. A TLB explores the principle of locality, which is the tendency of a program to access the same set of addresses and nearby addresses over a short period. Figure 1 depicts the relationships of the main components of a memory system.

As the microelectronics technology continuously scales down, the probability of single event upsets (SEUs) and multiple cell upsets (MCUs) induced by radiation in memory devices like TLBs increases. A TLB is a content address memory (CAM) which relates virtual page numbers (VPNs) to physical page addresses. For each memory request, the TLB checks if the requested address matches one of the stored VPNs. When there is a hit, the associated physical page address is returned. Otherwise, the system must retrieve the physical address from the page table in the RAM [1]. Usually a memory system has two TLBs, one for the data pages and another for the instruction pages. In this paper, we target the problem of protecting an instruction TLB (i-TLB) from SEUs and MCUs.

SEUs and MCUs may generate false negative and false positive errors in i-TLBs. A false negative happens when a cell upset modifies a VPN, and the TLB fails to translate that VPN, causing a page fault, which forces the operating system to recover the physical address from RAM memory.

Figure 2 depicts a false negative error on an 8-bit TLB (a). An external interference changes the value of two bits of

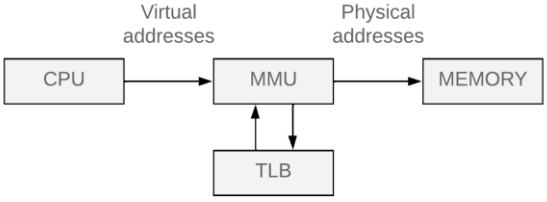

Fig. 1. Architecture representation of a memory system. MMU and TLB performing virtual address translation to physical address.

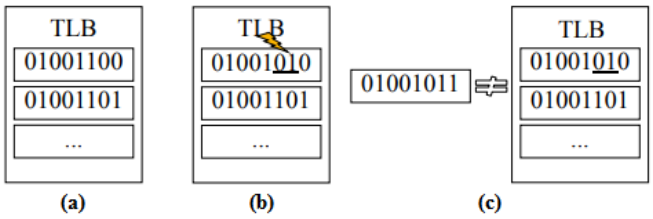

Fig. 2. False negative example.

a VPN (b). When the system tries to translate that address, it does not match any addresses stored in the TLB, causing a page fault and loss of performance (c).

In false-positive cases, the value pointed by the TLB corresponds to a valid, however, incorrect page. The memory system fetches a wrong instruction from RAM, which may cause serious failures such as system crashes or data corruption. Figure 3 depicts a TLB similar to the previous example (a), in which there are two errors on a VPN. In this example, the address requested by the CPU matches the value stored in the TLB, after the cells upset (b). Thus, the system recovers one instruction from an incorrect code region, which may cause severe errors (c).

The Figure 4 depicts an example, considering the 8-bit virtual address 11011001, which is fetched in a simple 4-line TLB. In this TLB are the addresses 11011011, 11011010, 11011101, 11011000. They are all close addresses, but none equal to the requested one. A TLB miss (a) occurs and the processor looks up the address in the page table. However, a fault reaches the least significant bit of the address 11011000, changing its value to 11011001. Thus, in (b), the searched word coincides with the one that was changed. As originally the address pointed to was another one, the wrong instruction will be executed.

There are different methods to protect TLBs from SEUs and MCUs, including: parity techniques [1], spatial redundancy [2] and error correction techniques [3][4][5]. However, many of the previous solutions have memory and

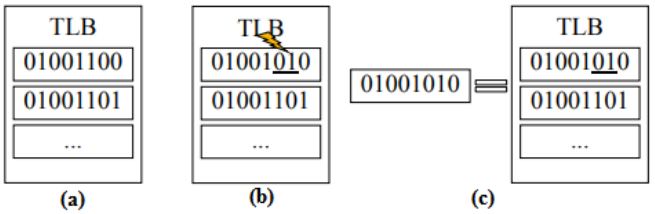

Fig. 3. False positive example.

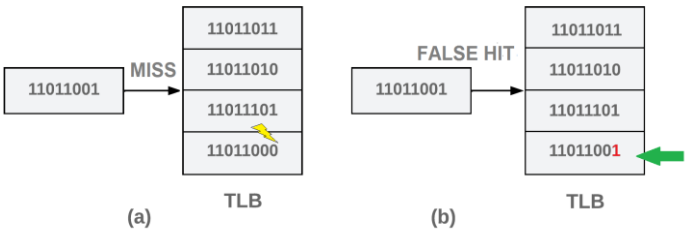

Fig. 4. A TLB miss (a) and false hit (b) representation.

time overhead issues. In [6], the authors propose a method that does not require extra bits by codifying parity in one or two VPN data bits. This method reduces the overhead while keeping low false-positive rates.

We extend the strategy proposed in [6] by calculating the parity with smaller groups of bits from VPN, thus, reducing the area and time overhead. We evaluate our proposition with different quantities of bits by comparing it with a non-protected TLB and the original code [6].

II. Related Work

Error correction and error detection codes are widely used in computer systems to protect memory devices from data corruption. Parity is a technique to detect simple errors or multiple even errors in data memories. Usually, each entry of a TLB has an additional parity bit to detect errors [1]. When an error is detected, the TLB entry is invalidated. However, this method has poor performance for double adjacent errors.

Reed-Muller [7] is a code based on majority logic capable to correct multiple error patterns, however its cost is not affordable to protect TLBs. Hamming code is a linear code capable to correct simple errors and detect double errors.

Some works [8], [9], [10], [11] codify the data word into a matrix to check errors on the columns and rows using Hamming, extended, Hamming, or parity. The Matrix code [8] is a linear code based on Hamming, which codifies a word into a matrix, achieving less area overhead than Reed-Muller for a slightly smaller correction performance. CLC [9] is a matrix code which outperforms Matrix and Reed-Muller. An extended version of CLC is presented in [10], this version improves multiple errors correction performance. PHICC [11] is another matrix code targeted to low-cost computational applications by using Hamming and interleaving. The previous codes effectively handle MCUs but they add a considerable area overhead due to the redundancy bits and codification circuit.

Henceforth, we present propositions target to cope SEUs and MCUs in TLB. In [2], a copy of TLB content is stored as back-up. When an error is detected, the content of TLB is overwritten by the back-up. This scheme doubles the memory space to implement a TLB. In [3], a matrix SEC-DED (single error correction, double error detection) based on parity and Hamming is used to protect the CAM and RAM of a TLB. In [4], Hamming distance and matchline schemes are used to increase the distance between words to correct single errors in TLBs. A matrix code based on Hamming [5] protects TLBs from single, double and triple adjacent errors.

In [12] a linear code handles simple and double adjacent errors in TLBs, but it also requires extra bits. [13] proposes a matrix code to correct simple, double, and triple adjacent errors in TLBs. The code has poor performance for triple errors. All previous methods are based on adding extra bits to handle errors on TLBs.

A different method to deal with errors in i-TLBs is to explore the principle of locality [6]. The pages stored in an i-TLB are usually located nearby for a short period, thus the goal is to propagate the errors to the most significant bits of the VPN. Thereby, whenever an error occurs it will result in an address far from the current set of pages stored in the i-TLB and it will be handled as a false negative. Basically, the method replaces the most significant bit with the parity of the VPN. This method does not require an extra bit to store the parity and handle simple and double errors effectively. It also can be extended by replacing the MSB and the MSB-1 bits with the odd and even parity of the VPN.

In this work, we extend the method presented in [6] by evaluating it using just the least significant bits (LSBs) of a VPN. Thus, we further reduce the overhead of area and time by reducing the number of XOR gates used to calculate the parities.

III. Parity Encoding in VPN

We present a simple example to explain the method sequence approached in this work. As can be seen in Fig. 5, the VPN is encoded and presented to the TLB with the new configuration. In this method, it is not needed to decode VPN'. When the CPU requests a VPN, that VPN is coded and the TLB tries to match it with the values stored in the CAM. If there is a match, the physical address is returned. If there isn't a match, a page fault is issued.

The encode method proposed by Sanchez [6] replaces the two MSBs with the odd and even parity of a VPN in an i-TLB, as described in Fig. 6 for 32-bit in VPN.

VPN[MSB] is the result of parity between all odd-valued positions, while VPN[MSB-1] is the result of parity between even positions. The others bit are not affected by the codification.

We present the encoding method for an 8-bit VPN. Consider a VPN equal to 10100011, the MSB = 1 and MSB-1 = 0. The encoder replaces the MSB and MSB-1 by the values in (1) and (2):

$$VPN'[MSB] = 1 \oplus 1 \oplus 0 \oplus 1 \quad (1)$$

$$VPN'[MSB-1] = 0 \oplus 0 \oplus 0 \oplus 1 \quad (2)$$

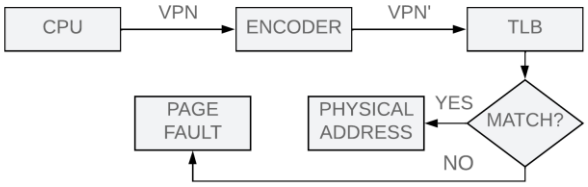

Fig. 5. Encoding and match verification for the method described in this work.

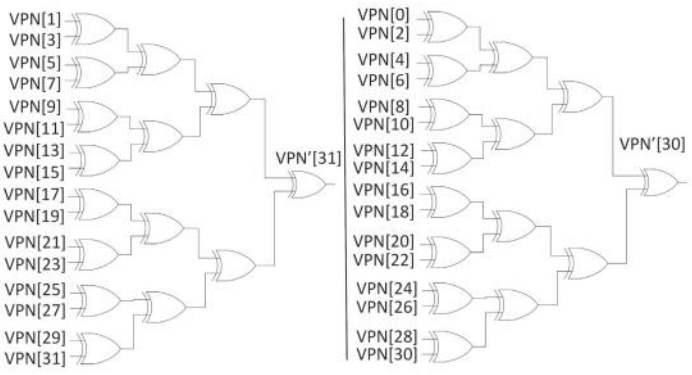

Fig. 6. XOR gates circuit of the scenario with 32-bit in VPN [6].

In this example, the new value of MSB is 1, while the new value of MSB-1 is also 1. The coded word VPN' is 11100011 and it will be stored in the i-TLB.

The parity encoding in VPN increases the Hamming distance between neighboring addresses to two bits. An example of this increment is described: the Hamming distance between the addresses 10111010 and 10111011 is equal to 1, but, when these addresses are encoded, the new values are 01111010 and 00111011, with the distance increasing to 2.

This ensures that if the TLB memory cells are hit by a single-bit error, no single error will be produced. In the same way, double adjacent errors are also avoided, due to the process of calculating separating odd and even bits.

IV. Proposed Method

In this article, we extend the code presented in the previous section, however, we use smaller groups of bits to calculate the parities. The main idea is to further explore the principle of locality. The VPNs stored in the i-TLB tend to be in the same region, so, in theory, we should protect only the LSBs from SEUs and MCUs, because errors in the upper bits would result in false negatives.

In our proposal, the Hamming distance is also increased, even using fewer bits in the calculation of parity. Using the same example as the previous section, if we use parity calculation with only the 4 less significant bits, the addresses 10111010 and 10111011 when encoded are equal to 00111010 and 01111011, also increasing from 1 to 2.

We propose four scenarios using 4, 8, 12, and 16 LSBs. The hypothesis is that errors in these bits are more likely to cause false positives than errors in the upper bits. The main goal is to evaluate if the proposed codes are capable to assure the same rate of false positives as the original code, that uses all the VPN bits. The proposed codes require fewer XOR gates as depicted in Table I and in Fig. 7.

Table I points out that the 4-LSB scenario requires only 6.67% of the gates of the original code [6]. If we use the 16-LSB scenario, the percentage of required XOR gates is 46.67%.

Figure 7 depicts the number of XOR gates required in the 8-LSB scenario, which only uses 20% of the gates of the 32 bits approach, depicted in Fig. 6.

TABLE I. Number of XOR Gates For Each Code For A 32-Bit Example

Code	Original	4-LSB	8-LSB	12-LSB	16-LSB
XOR gates	30	2	6	10	14
%	100	6.67	20	33.33	46.67

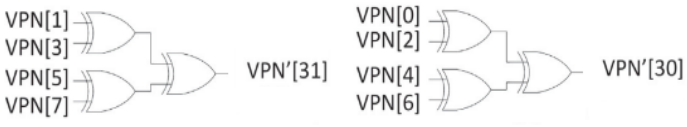

Fig. 7. XOR gates circuit of the 8-LSB scenario.

V. Evaluation

To evaluate the proposed scheme, we have conducted some experiments using memory traces of real applications: merge sort, compression algorithm, FFT, and bloom filter.

Figure 8 depicts the algorithm of the simulation. When the simulator runs, the algorithm verifies the running counter. If the running number is <= 10,000, the fault injection algorithm runs. If the simulation 10,000 executions, the simulation ends. The fault injection can cause a false positive. At the moment that the simulator found a false positive, the simulator stops and the false positive counter is increased. If a fault is not found, the simulator runs until the word with error injected is replaced. After the false positive counter is increased or word error replaced, the running counter increases and the algorithm is restarted.

The tool Intel PIN [14] extracted the memory traces from the execution of those applications on a personal computer. We have used the first 500,000 instruction addresses to evaluate the codes. We have simulated the execution of the memory traces on an i-TLB with the following configurations: 8 lines, fully associative, LRU, and 32 bits VPN.

We have evaluated the following scenarios: no error protection code, codification proposed in [6], and the schemes proposed in this article, which use 4, 8, 12, and 16 bits to calculate the parities.

The simulation injects single error (SE), double adjacent error (DAE), and triple adjacent error (TAE). The fault injection is pseudo-random, occurring in different positions for each execution. The simulator specifies the address of the injected error in the TLB, the time to inject the error, as well as in which bit of the VPN the error is added.

Injecting a failure into a bit, having all parameters already drawn, it is about changing the value of that bit from 0 to 1 or 1 to 0. If the type of error is double adjacent, will reach the drawn bit and the neighbor on the right. If it's triple adjacent error, it hits both neighbors. Figure 9 depicts an example for each type of failure: SE (a) reaching a single bit,

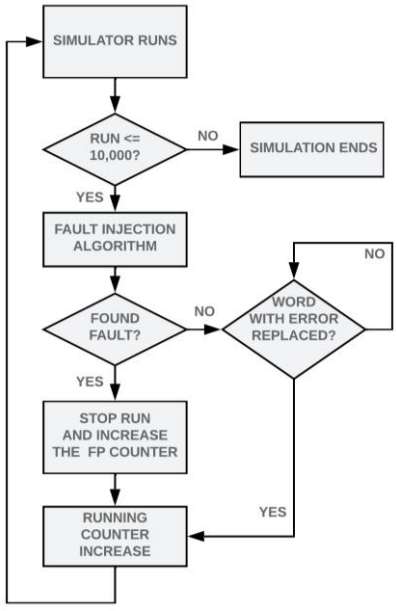

Fig. 8. Representation of the simulation algorithm.

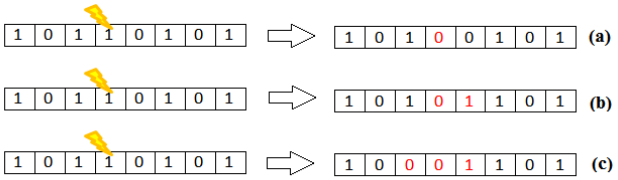

Fig. 9. Single error (a), double adjacent error (b) and triple adjacent error (c) examples.

DAE hitting the right neighbor (b), and TAE reaching both neighbors (c).

The simulator is capable to detect false positives and false negatives, but this paper focuses on false positives. The simulation runs until a false positive is detected, or the word with error injected is replaced. For each scenario, the simulator runs 10,000 executions and registers the number of false positives.

Figure 10 depicts that the simulator inputs are the memory traces and the fault injection algorithm, while the output is the number of false positives obtained in that simulation. The traces simulate real application addresses, which will occupy the TLB entries: merge sort, compression, FFT and Bloom Filter. Fault injection inserts three types of faults into the bits of TLB addresses: single errors, double adjacent errors, and triple adjacent errors. At the end of each execution, the simulator returns 1, in false positive case, or 0, in case it didn't. This output is forwarded to a counter that accumulates this result for all executions of the same type (fault, code and trace equals).

VI. Results and Discussion

In this section, we present the results of the experiments detailed in the previous section. Table II presents the percentage of false positives obtained in the experiments for the scenario without error correction code. The DAE was the most aggressive type of fault, it points out that changing two adjacent bits increases the probability of false positives. For the TAE scenario, there is a decrease of false positives, probably the page addresses with errors were pointing to distant addresses and resulted in false negatives.

The results in Table III point out that the previous method [6] is very effective to handle false positives, it avoided all the three types of errors. Table IV presents the results when only the LSBs are used to calculate the parities, which is the proposition of this paper. From 8- to 16-LSB the results are similar to the ones achieved by the original method. For 4-LSB, there is a considerable reduction in the number of false positives.

TABLE II. Experiment Results in Percentage of False Positives in Non-Protected Scenario

Application	SE(%)	DAE(%)	TAE(%)
Merge Sort	1.37	2.14	1.79
Compression	1.42	2.16	1.73
FFT	1.26	1.98	1.40
Bloom Filter	1.22	2.37	1.75

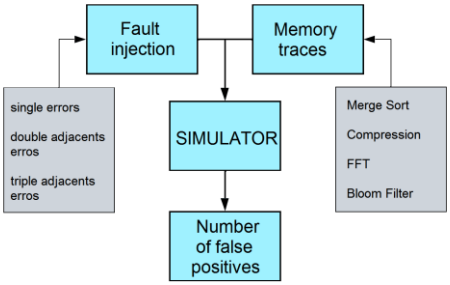

Fig. 10. Representation of inputs and outputs from the simulator.

TABLE III. Experiment Results in Percentage of False Positives in Original Code Scenario

Application	SE(%)	DAE(%)	TAE(%)
Merge Sort	0	0	0
Compression	0	0	0
FFT	0	0	0
Bloom Filter	0	0	0

For the merge sort benchmark, the number of false positives is reduced to 17% for SE and 10% for DAE and TAE. In compression case, this reduction was about 15% in the three scenarios. For the FFT benchmark reduced to 24% for SE, 10% for DAE and 17% for TAE. Lastly, for the Bloom Filter scenario reduced to 28% for SE, 12% for DAE and 10% TAE.

For all the applications on the benchmark, the results are very similar, no false positive is detected using 8-, 12-, 16-LSBs. These results point out that errors in the LSBs are more probable to cause false positives. Thus, it was verified that it is possible to protect i-TLBs using the parity of the LSBs, reducing the number of XOR gates used in the circuit.

TABLE IV. Experiment Results in Percentage of False Positives For Each Proposed Scenario

Application	Scenario	SE(%)	DAE(%)	TAE(%)
Merge Sort	4-LSB	0.24	0.21	0.17
	8-LSB	0	0	0
	12-LSB	0	0	0
	16-LSB	0	0	0
Compression	4-LSB	0.21	0.31	0.27
	8-LSB	0	0	0
	12-LSB	0	0	0
	16-LSB	0	0	0
FFT	4-LSB	0.31	0.19	0.24
	8-LSB	0	0	0
	12-LSB	0	0	0
	16-LSB	0	0	0
Bloom Filter	4-LSB	0.34	0.28	0.17
	8-LSB	0	0	0
	12-LSB	0	0	0
	16-LSB	0	0	0

VII. Conclusion

SEUs and MCUs may cause false positives in i-TLBs generating severe failures such as system freezing and silent data corruption. Methods to protect i-TLBs are usually based on redundancy bits and add area overhead. A new code [6] based solely on parity and with no extra bits is capable to reduce the number of false positives. In this work, we extend this code by further exploring the principle of locality. We propose to calculate the parity of the LSBs, because errors in these bits are more likely to cause false positives than errors in the upper bits. The results point out that the scenarios with 8, 12, and 16 LSBs reach protection levels equivalent to the original code. Furthermore, the scenario with 4 LSBs have slightly less performance.

References

[1] T. Griffith Jr and L. E. Thatcher, "Tlb parity error recovery," May 31 2005, uS Patent 6,901,540.

[2] LANG, S. M. Processor fault tolerance through translation lookaside buffer refresh.[S.l.]: Google Patents, 2013. US Patent 8,429,135

[3] SANCHEZ-MACIAN, A.; REVIRIEGO, P.; MAESTRO, J. A. Combined modular key and data error protection for content-addressable memories.IEEE Transactions onComputers, IEEE, v. 66, n. 6, p. 1085–1090, 2016

[4] K. Pagiamtzis, N. Azizi, and F. N. Najm, "A soft-error tolerant content-addressable memory (cam) using an error-correcting-matchscheme," in IEEE Custom Integrated Circuits Conference 2006.IEEE, 2006, pp. 301–304

[5] A. Sanchez-Macian, P. Reviriego, and J. A. Maestro, "Hamming sec-daed and extended hamming sec-ded-taed codes through selective shortening and bit placement", IEEE Transactions on Device and Materials Reliability, vol. 14, no. 1, pp. 574–576, 2012.

[6] A. Sanchez-Macian, L. Aranda, P. Reviriego, and J. Maestro, "Reducing false positives due to double adjacent errors in instruction tlbs", Microelectronics Reliability, vol. 102, p. 113494, 2019.

[7] B. Varghese, S. Sreelal, P. Vinod, and A. Krishnan, "Multiple bit errorcorrection for high data rate aerospace applications," in 2013 IEEE Conference on Information & Communication Technologies. IEEE,2013, pp. 1086–1090.

[8] C. Argyrides, H. R. Zarandi, and D. K. Pradhan, "Matrix codes: Multiple bit upsets tolerant method for sram memories," in 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007). IEEE, 2007, pp. 340–348.

[9] H. d. S. Castro, J. A. da Silveira, A. A. Coelho, F. G. e Silva, P. d. S.Magalhães, and O. A. de Lima, "A correction code for multiple cells upsets in memory devices for space applications," in 2016 14th IEEE International New Circuits and Systems Conference (NEWCAS).IEEE, 2016, pp. 1–4.

[10] F. Silva, A. Muniz, J. Silveira, and C. Marcon, "Clc-a: An adaptive implementation of the column line code (clc) ecc," in 2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI). IEEE, 2020, pp. 1–6.

[11] P. Magalhães, O. Alcântara, and J. Silveira, "Phicc: an error correction code for memory devices," in 2019 32nd Symposium on Integrated Circuits and Systems Design (SBCCI). IEEE, 2019, pp. 1–6.

[12] J. Li, L. Xiao, P. Reviriego, and R. Zhang, "Efficient implementations of 4-bit burst error correction for memories,"IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 65, no. 12, pp. 2037–2041, 2018.

[13] V. Kiani and P. Reviriego, "Improving instruction tlb reliability with efficient multi-bit soft error protection", Microelectronics Reliability,vol. 93, pp. 29–38, 2019.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation", Acm sigplan notices, vol. 40, no. 6, pp. 190–200, 2005.