SRAM Arrays with Built-in Parity Computation for Real-Time Error Detection in Cache Tag Arrays

Ramon Canal
Universitat Politècnica de Catalunya
Barcelona, Spain
rcanal@ac.upc.edu

Yiannakis Sazeides
University of Cyprus
Nicosia, Cyprus
yanos@cs.ucy.ac.cy

Arkady Bramnik
Intel Corp.
Haifa, Israel
arkady.bramnik@intel.com

***Abstract*— This work proposes an SRAM array with built-in real-time error detection (RTD) capabilities. Each cell in the new RTD-SRAM array computes its part of the real-time parity of an SRAM array column on-the-fly. RTD based arrays detect a fault right away after it occurs, rather than when it is read. RTD, therefore, breaks the serialization between data access and error detection and, thus, it can speed-up the access-time of arrays that use on-the-fly error detection and correction. The paper presents an analysis and optimization of an RTD-SRAM and its application to a tag array. Compared to a state-of-the-art tag array protection, the evaluated scheme has comparable error detection and correction strength and, depending on the array dimensions, the access time is reduced by 5% to 18%, energy by 20% to 40% and area up to 30%.**

Keywords—reliability, SRAM array, tag array, error detection and correction, real-time error detection (RTD).

I. Introduction

Memory arrays used in modern processors are vulnerable to various errors in the field (e.g., soft-errors [1]). Designers of processors for high-availability and mission-critical systems virtually always employ error detection and correction codes [2], such as SECDED, to protect from errors the data in memory arrays. In general, a code adds one or more parity bits to each word in an array to encode redundant information about the stored data. When a codeword (data plus parity) is read from the array, an error is detected if the codeword is illegal. The coding algorithm and the number of parity bits used, determine the strength of the code: how many corrupted bits it can detect and how many, if any, it can correct.

Until recently, all memory protection coding schemes require reading an array entry first to detect if it has been corrupted. Real-time Error Detection (RTD) [3] is a newly proposed error detection approach that can detect a fault in an array instantaneously after it occurs. RTD, consequently, breaks the dependence between data access and error detection and, thus, it can reduce the access time of arrays that use on-the-fly error detection and correction. RTD's key requirement is a built-in logic to track in real-time the parity of the cells in each array column.

The previous RTD work, proposed real-time error detection for flip-flop-based arrays by adding dedicated separate columns, in a bit-sliced array design, for parity computation. Also, it showed how to use RTD to build a 2D ECC code that provides error detection and correction for an array. The benefits of RTD were demonstrated with the help of a gate level analytical model.

In this paper, for the first time, we present how to integrate the real-time column parity computation in an SRAM array with a modified SRAM cell design that inserts the parity computation into the cell. This reduces the area footprint of the solution and at the same time the compact design minimizes the computation delay and power. We perform a detailed timing analysis and optimization of RTD-SRAM using SPICE simulation (instead of an analytical approach). Furthermore, we show how to use RTD-SRAM to protect a Tag Array [4] (the previous RTD work focused on arrays containing data). Tag arrays are different from data arrays in that what is needed during an address lookup is a correct hit/miss definition not the correct tag [5][6][7][8]. This opens up an opportunity for a different RTD design optimized for tag arrays. We compare the RTD-SRAM based Tag array against the state of the art Fast-Tag Hit ECC design [8] and show that the RTD design provides superior delay and energy across the board and area reduction for small arrays.

RTD-SRAM can also be used to track the real-time parity of cells in a row (horizontal) and, in addition to reliability, it can be used for post-silicon validation [9]. In particular, RTD can be very effective in reducing the time needed to root-cause bugs that manifest as SRAM array-content corruptions for both test and production chips. We do not discuss these other uses of RTD-SRAM due to space limitations.

In the remaining of the paper, we provide background on RTD (Section II), the RTD SRAM array proposal (Section III), an evaluation of the RTD SRAM array (Section IV), a review of cache-tag array protection schemes (Section V), the implementation of RTD-SRAM cache-tag array and its evaluation (Section VI), and conclusions (Section VII).

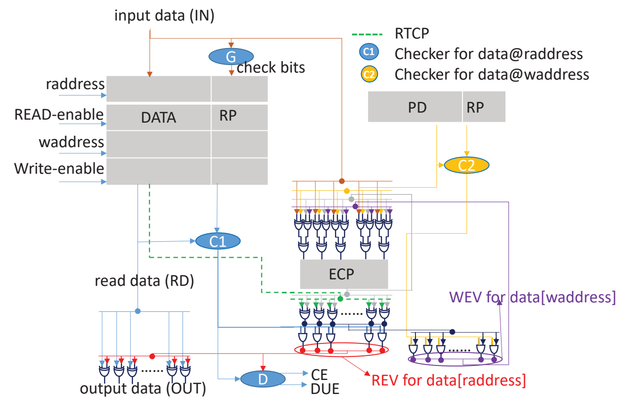


Fig. 1: 2D ECC RTD Architecture [3]

II. RTD Background

The high level RTD 2D ECC design proposed in [3] is shown in Fig. 1. An array with RTD needs to include combinational logic that determines in real-time the parity of

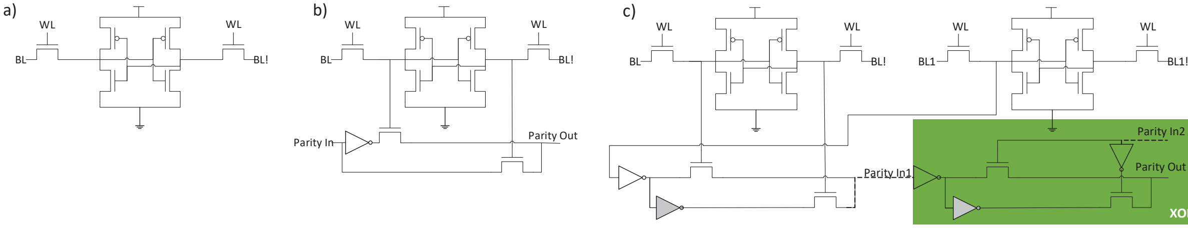


Fig. 2: a) Baseline SRAM cell, b) SRAM cell with RTD. c) Tree-based 2-bit RTD SRAM array macro

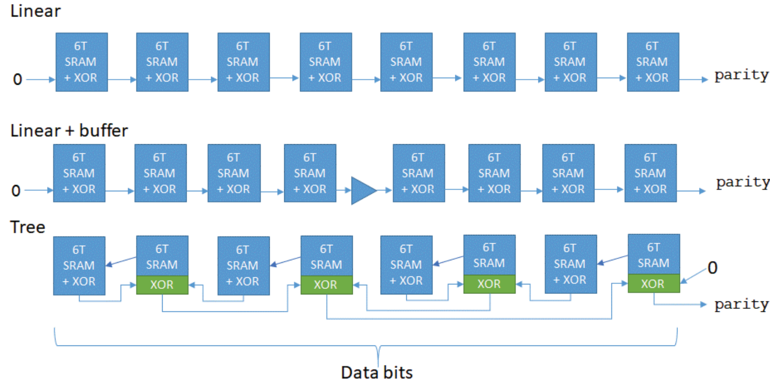


Fig. 3: Schematic of the linear and tree connections

the cells in each array column (real-time column parity or RTCP). Additionally, it is required to maintain the expected parity for each column (expected column parity or ECP) and on initialization to set the ECP according to the array contents. The RTCP is calculated all the time whereas the ECP is updated each time the array is written. Specifically, ECP is updated with the bitwise-xor of the current ECP value, the new value (IN) about to be written in a location, the previous value (PD) in the location about to be overwritten and the write error-vector (WEV). The WEV is zero when the PD does not contain an error, otherwise, when the PD contains an error, it is equal to the bitwise-xor of the ECP and the RTCP. We detect errors in the PD by adding a traditional row parity (RP) per entry and using a parity checker (C2) on reads. The traditional RP is also used during a read cycle to determine whether the data read (RD) from the array contains an error or not with the help of another checker (C1). The array output (OUT) is produced by performing a bitwise-xor of the RD with a read error-vector (REV). If there is no error, the REV is set to zero, otherwise, it is set to the bitwise-xor of the ECP and RTCP.

The above is a 2D ECC scheme since it uses both row and column parity to detect and correct errors. RTD 2D ECC does not require a very expensive (in terms of cycle count) correction procedure to determine the columns with errors (as previous 2D ECC works did [10]). RTD circumvents costly correction by employing a dedicated array port to track the RTCP.

To keep background brief, we note that the discussion considers the case of one error at any given time in the array and we do not discuss how the decoder works (D) and how the PD is obtained. We discuss multi-bit errors, decoding and reading the PD in Section VI.

III. RTD-SRAM Array

The conventional SRAM cell is depicted in Fig. 2.a. The 6T SRAM is connected to the Wordline (*WL*) and the Bitlines (*BL* and *BL!*). The 1-bit RTD SRAM array (Fig. 2.b) is composed of a conventional 6T SRAM cell plus an XOR built with transmission gates that are connected to the SRAM internal nodes (i.e., a static port). Both PMOS and NMOS transistors are used in the transmission gates for better output signal quality. This XOR will propagate the parity computation across the column. This gate XORs the value of *Parity In* to the cell value. The output is the *Parity Out*.

A. Fast Real-Time Column Parity (RTCP) Computation

We can connect all the parity XORs in a column to compute the RTCP. The simplest and more compact way is to connect one row to the next. Thus, all the XORs are connected in series. A first approximation for the delay of this solution is the number of XOR gates traversed to compute the column parity. The delay of such approach is O(n) where *n* is the number of rows. Alternatively, we can add inverters/buffers (a.k.a. repeaters/drivers) to avoid significant RC delay and parity signal quality degradation (i.e., slope growth). The third option is a tree-like structure as we do not need intermediate results. In the tree structure, the delay will be $O(\log_2 n)$. Evidently, for arrays with a large number of rows, the tree structure will result in better delays as compared to the linear.

Fig. 2.c shows a 2-bit RTD SRAM array. The leftmost SRAM cell computes the XOR of the two cells. The rightmost XOR has 2 “programmable” inputs: *Parity In1* and *Parity In2*. These inputs will be used to make the appropriate connections for the tree structure. In Fig. 2.c, it computes the parity resulting from the previous cell parity and an external input (*Parity In2*), which could be, for instance, the next set of 2 RTD SRAM cells (as shown in the first 2 cells in the tree structure in Fig. 3). The XORs in Fig. 2.c make use of one extra inverter (shaded inverter in Fig. 2.c). This inverter is needed to isolate the inputs of the XOR from the output. By isolating the inputs, we ensure that, in the worst case, an inverter’s load is the next XOR’s inverter plus one pass gate

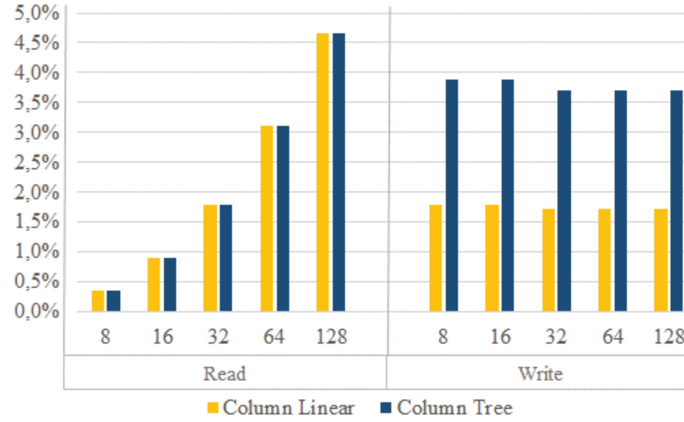


Fig. 4: Read (decoder + WL activation + BL development) and Write (decoder + cell write) cell access time increase for different row counts.

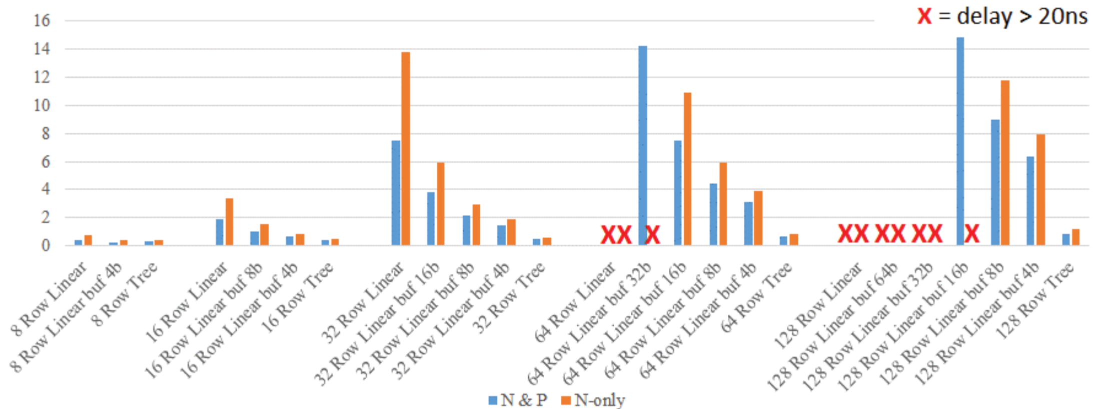


Fig. 5: Column parity computation delays for linear, linear+buffer and tree for different row numbers and different buffer positions. Blue columns indicate pass gates with NMOS and PMOS. Orange columns indicate NMOS-only pass gates.

(i.e., *Parity In2*) after traversing one pass gate. Otherwise, one inverter could be driving the signal through all the levels of pass gates in the tree.

Fig. 3 shows a schematic of the linear, linear with buffers and the tree connections. The linear design uses Fig. 2.b cells and the tree design uses Fig. 2.c cells. For ease of understanding, the green XOR box in Fig. 3 refers to the green boxed XOR in Fig. 2.c. For illustration ease, Fig. 3 depicts a row (8-data bits). Nevertheless, this organization can be performed at the column level as well, which is what is used for the remaining paper.

IV. Evaluation

A. Simulation Environment

We implemented the RTD-SRAM arrays in NGSPICE using the 28nm bulk planar model (BSIMv4.0 MOSFET). Vdd is 1V. Wire capacitance is included by scaling down to 28nm (matching ITRS trends) the nominal values available in the FreePDK45 for metal 1,2 (same value) and 3. Intracell connections are assumed to be metal 1 and 2 whereas intercell connections are metal 3. The 6T cell size is 0.127um2 [11] and the cell aspect ratio 1.5 (pull-up transistors are 3λx3λ, pull-down transistors are 2λx8λ, tgates are 2λx4λ). The XOR size is 50% of the 6T size (for the linear designs) and 100% for the tree design (due to the extra inverters). To accommodate the bitlines and wordlines, we assume an equivalent cell spacing of 10% of the cell size (vertically and horizontally). Cells' size and spacing is used to compute wire length for wordlines, bitlines, and RTCP. All configurations have one conventional R/W port.

B. Evaluation of Cell Access Time

In this section, we measure the impact on read and write SRAM array access time when RTD is implemented and compare it to a conventional SRAM array without RTD. This analysis does not consider any detection/correction logic outside the array. It solely analyzes the array access impact of the RTD proposal. The analysis with detection/correction logic is performed in Section VI.B.

Adding more connections to the storage node of the SRAM cell – a static port for RTCP computation - increases the cell capacitance and, thus, it may slow down read and write operations. Fig. 4 shows the impact on the array access time for read and write operations. The extra cell port affects the array access delay. Read timing includes address decoding, WL activation and BL development. The read delay increases up to 4.5% for 128 rows. The impact is small as the new cell port only affects the cell access time (i.e. WL activation to BL development).

Fig. 4 also shows a 4% increase in write access time (decoding + cell write) for the tree configuration. The increase reduces to 1.5% for the linear configuration. The higher increase in the tree configuration is caused by the extra inverter connected to the cell node. Larger arrays show a smaller impact. This is caused by the bigger decoder delay and the constant cell write delay. In our design, address decoding and BL setup work in parallel during writes (being address decoding the slowest of the two).

C. *Evaluation of RTD SRAM Array Organization*

Fig. 5 shows the parity computation delay for different row counts. These numbers include both the wire delay and the individual XOR delays for RTCP computation after a write. We have evaluated both NMOS+PMOS pass gates; as well as, NMOS only pass gates. NMOS+PMOS design is comparatively better than NMOS only, despite the extra transistor needs. It generates a high-quality signal that is able to propagate faster.

The linear design is comparatively very slow for large number of rows and it cannot produce a result in 20ns for 64 and 128 row designs. Even organizations with a buffer every 64 rows or every 32 rows do not produce the RTCP even after 20ns. Only a buffer every 4 or 8 bits guarantees a delay below 20ns for both NMOS-only and PMOS and NMOS pass gates for large rows.

On the other hand, tree designs are able to complete the RTCP computation below 1.2ns for all cases. Clearly, the reduction of XORs to traverse (e.g., for 128 rows: 8 (tree) vs 128 (linear)) has a much bigger impact on delay than the larger wiring in the tree structure. We will use the tree structure from here on.

V. Cache Tag array, Tag Errors, Tag Protection

A. *Cache Tag Array*

Caches are a key performance feature for the modern processor. In general, a cache is organized as a set associative multi-way memory structure that is implemented with tag and data arrays. The tag array contains entries with memory addresses and the data array contains entries with lines of memory data. The tag and data arrays of a cache have the same organization and each tag entry holds the line address of the corresponding entry in the data array.

A cache is searched by splitting a lookup address into the tag, set and offset fields. The offset specifies the index within a line, the set corresponds to the index of a cache set and the tag is the field stored in the tag array.

During a cache read or write cycle, all the valid tags in the cache set of the lookup address are compared with the Tag of the lookup address and a hit/miss signal is generated per way. On a read cycle, when a way hit signal is activated, the data in its corresponding way are transferred to the cache output. On a write cycle, a way hit signal controls in which way the input data are written. Consequently, the timing of hit/miss signal is performance critical since it influences the cache access latency.

Besides the read and write flow, the third major cache flow is a replacement after a cache miss. Replacement writes a new cache line into a way selected by a replacement algorithm. In the case of a writeback cache, if the selected way contains modified data then the data line needs to be evicted before filling in the new line.

Figs. 6.a-b show how the 4-bit tags of a selected set in a tag array are used to produce per way hit/miss signals by comparing the lookup tag against the stored tags in a two-way set. In the first example (6.a), the lookup tag is not in the set (both ways return a miss), whereas in Fig. 6.b there is a hit in way-1.

B. *Errors in the Tag Array and Tag Array Protection*

In the presence of errors in the tag array, the following problematic behaviors can occur during lookup: false-hit and false-miss. The false-hit and false miss are illustrated in Figs. 6.c-d that show for the corresponding examples in Figs. 6.a-b

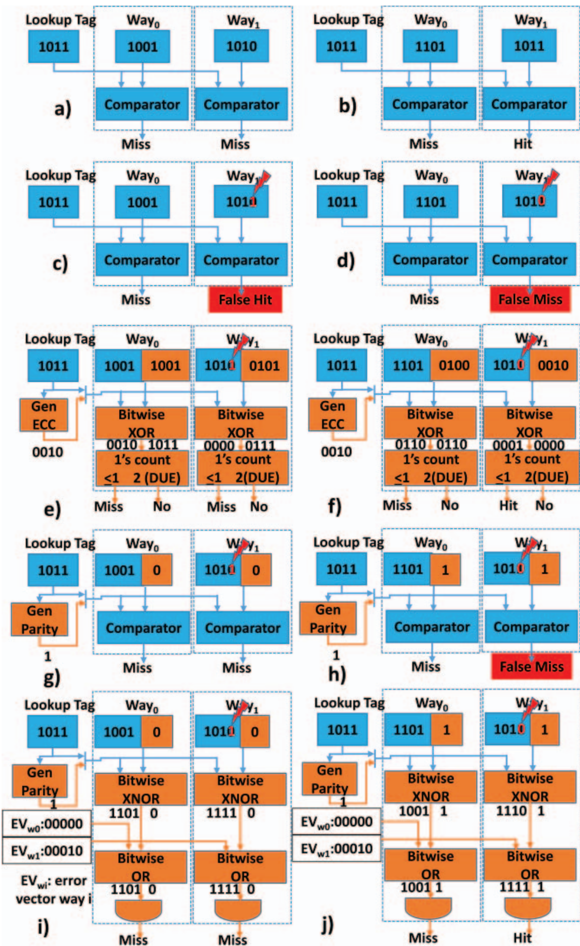
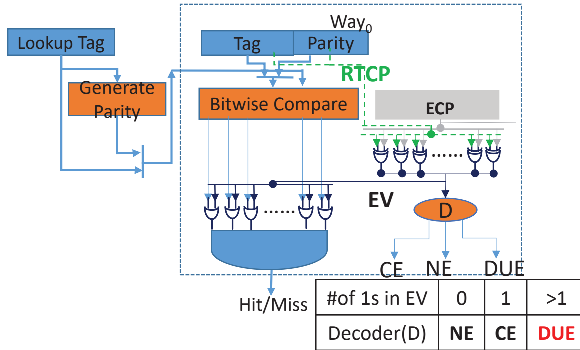


Fig. 6: Tag Array Miss and Hit Behavior; without Error and No Protection (a,b); with Error: No Protection (c,d), Fast-Tag Hit ECC (e,f), Fast-Tag Hit Parity(g,h), Fast-Tag Hit RTD (i,j)

how a corruption in the tag of way-1 can lead to false-hit and false-miss. Both errors can have grave consequences since with a false-hit wrong data is forwarded for use whereas with a false-miss, if the faulty-way is modified, a stale copy of the line will be fetched and used.

A tag array can be protected against corruption with a conventional ECC coding scheme such as those used to protect a cache data array (e.g., a SECDED code). Specifically, a tag entry can be augmented with check bits that are used during lookup to detect an error in the tag and, if possible, correct it before it is used for hit/miss definition. Unfortunately, a conventional ECC scheme functionality lies in the read critical path for hit/miss definition and may increase cache access latency. However, previous work [5][6][7][8] observed that a stored tag protected with an ECC code, can determine if it is a hit or a miss by only checking its Hamming Distance (HD) [2] from the lookup tag codeword. For instance, when using a SECDED code per tag, the hit signal is activated as long as the HD between the lookup codeword and a stored tag is one or less. When the distance is two, a DUE is signaled since SECDED, with minimum HD 4, cannot resolve whether the faulty tag corresponds to a miss or a hit. If the distance is three or more then there is a miss. This fast-tag hit approach eliminates the need for costly and slow hardware to correct the cache tags before checking for a hit or



#of 1s in EV	0	1	>1
Decoder(D)	NE	CE	DUE

Fig. 7: Fast-Tag Hit RTD

a miss. Figs. 6.e-f illustrate the use of fast-tag hit approach (SECDED code is from [13]) and how it eliminates false-hit and false-miss. Figs. 6.g-h show that a fast-tag hit with parity (minimum HD 2) approach is sufficient to avoid false-hits but it cannot avoid false-misses.

VI. An RTD Use Case: Cache Tag Array ECC using Fast-Tag Hit RTD

Although fast-tag hit ECC is an improvement over a conventional ECC it still introduces the ECC logic (see the 1's count in Figs. 6.e-f) in the critical path of the hit/miss definition. In this section, we propose to use the SRAM-RTD design (Section III), to protect effectively a tag array against errors while speeding up its access time as compared to the fast-tag hit ECC. We refer to this new design as fast-tag hit RTD.

Fig. 7 illustrates how to use an RTD-SRAM to provide a tag array with a novel single-bit error correction capability. The figure shows the flow during a cache lookup. To keep the explanation brief, we assume at most a single bit error in a column at any given time (we discuss subsequently the case with multiple column errors).

The proposed scheme does a bitwise comparison of codewords (as in the fast-tag hit ECC Figs. 6.e-f) but it only uses a single parity bit to protect each tag instead of several bits needed by a conventional ECC code to provide single bit correction (e.g., SECDED requires 7 check bits for a 31-bit tag). Single bit correction is realized by leveraging the SRAM-RTD built-in RTCP. Specifically, on a read cycle we use the bitwise-XOR of the RTCP and the array ECP (the result of this operation is referred to as read-error-vector (REV)). The REV indicates which columns contain an error and it is bitwised-OR with the result of the bitwise-comparison. More specifically:

- If no column contains an error (all REV bits are zero) then the comparison result is used as is for hit/miss definition.
- If there is a single column with error, that column's comparison result is excluded (by forcing it to indicate match) and the hit/miss definition is determined by the remaining comparison bits. This is correct, because a 1-bit parity is a minimum HD 2 and when excluding (erasing) a bit from the comparison, two different codewords, with no errors in their remaining bits, are guaranteed to have at least HD 1 and, thus, cannot match.
- When the number of columns with errors are more than one a DUE is signaled by the decoder (denoted as D in Fig. 7, a table defines its behavior) because there is a chance for a false-hit. Erasing two or more bits in a codeword with minimum HD two allows two different tags to match.

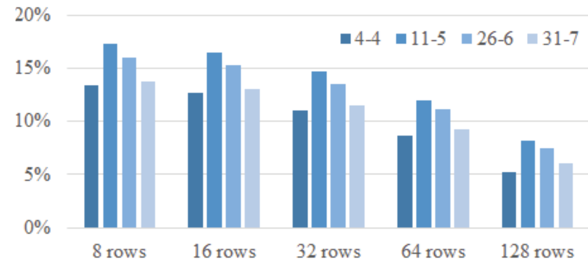


Fig. 8: RTD vs SECDED Fast-Tag Hit access time reduction.

A subtle but important difference of the tag RTD design in Fig. 7 versus the RTD design for data in Fig. 1, is that for tags we do not need to check whether the accessed Tag has an error but only whether any column in the array has an error. This helps simplify the RTD tag design and remove one of the checkers (C1) found in Fig. 1. This, also, reduces both the delay and area as well as the decoder energy which becomes independent of the lookup set.

In the case of a cache replacement, the ECP needs to be updated (as in Fig. 1). This update requires the previous tag (PD) of the line that gets replaced. This tag can be stored during the lookup, that turned out to be a miss, by using the replacement policy at that time to determine which way will get replaced. Therefore, no extra read is needed by Fast-Tag Hit RTD to obtain the PD.

The decoder circuitry (not shown due to space limitations) uses a 1's count function [8] to determine whether there are 0,1, or >1 columns with errors (number of bits set in REV). This circuitry's output is not in the critical path of the hit/miss definition because it only matters in the case of a DUE (Detected Unrecoverable Error). That is to say, it is acceptable to have speculative hit/miss definition that may turn out to be a DUE [8]. This does not compromise functionality as long as the >1 error signal is resolved before forwarding a wrong but valid line. In general, such time windows exist in caches because additional time after the hit definition is needed either to multiplex the data of a selected way or to access the selected way. We illustrate how Fast-Tag Hit RTD provides correct hit/miss definition in the presence of errors in Figs. 6.i-j.

A. *Multiple Errors*

The Fast-Tag Hit RTD can be augmented to handle cases with multiple columns with error and multiple errors in a column using horizontal and vertical logical interleaving [12][10][3]. For example, two-way horizontal interleaving uses two parity bits per tag to protect with separate parity the even and odd bit positions. This way a horizontal burst of two horizontal will belong to different codewords and can be corrected. We do not discuss interleaving issues further due to space limitations.

B. *Performance comparison fast-tag hit RTD vs ECC*

Fig. 8 compares the tag-array access time of Fast-Tag Hit SECDED [8] vs Fast-Tag Hit RTD (this proposal). In particular, we analyzed four configurations of the tag array described as (#tag bits, #ECC bits): 4-4, 11-5, 26-6 and 31-7. Note that in RTD, just one parity bit is needed for ECC; whereas SECDED needs the number listed. Unless indicated otherwise all configurations are 1-way associative.

The measured delay includes the address decoder, the SRAM array access plus the error detection and correction mechanisms particular to each proposal. As previously described in the beginning of Section VI, both mechanisms

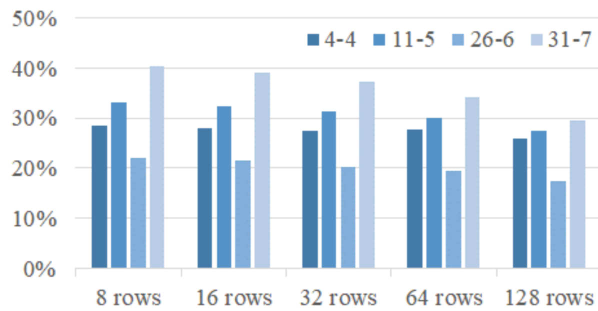


Fig. 9 RTD vs SECDED Fast-Tag Hit array average read energy reduction.

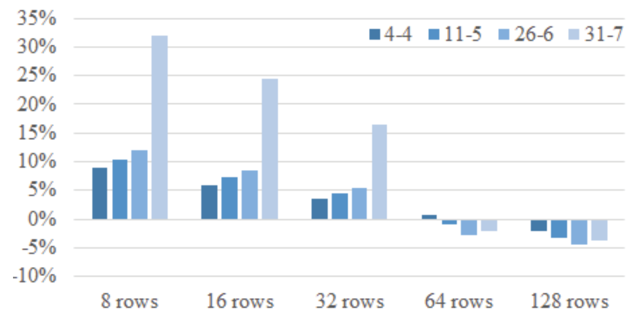


Fig. 10: RTD vs SECDED Fast-Tag Hit array area reduction (negative numbers mean area increase).

have comparable detection and correction capabilities. In other words, both mechanisms can correct single-bit errors and detect two-bit error when the errors are in a single tag.

RTD clearly reduces the access time over ECC. Delay reduction is between 5% and 18%. Despite RTDs initial increase of the read access time (see Fig. 4), when we consider the error detection and correction circuitry, RTD is much faster than the SECDED version of Fast-Tag Hit. RTD's advantage reduces as the number of rows increases. This is caused by the higher SRAM array access time over the baseline (without RTD, the one used by the SECDED version) as seen in Fig. 4.

Finally, the different configurations offer similar savings for a given number of rows (within 4%). The minor differences are caused by the different gate types and number (depth) used for SECDED and RTD detection and correction.

Fig. 9 shows the read energy reduction of Fast-Tag RTD when compared to the SECDED version. Energy reduction is between 18 and 40% depending on the number of rows and the protection level. Energy and area wise, SECDED computation requires a large amount of power and gates. By replacing it with parity, we remove a lot of gates; and hence reduce energy. On top of that, parity needs just one extra bit stored, whereas SECDED ranges between 4 and 7 in the configurations analyzed. Write operations achieve almost identical savings as they also benefit from the smaller number of gates and storage.

Fig. 10 shows the area comparison for the arrays analyzed following the methodology in [3]. While RTD can save a significant amount of area (~30%) for small highly protected arrays (i.e., an 8-row 31-7configuration), the benefits disappear as arrays become bigger. The area cost of the extra (pass-gate based) XOR for each SRAM cell dominates the reduction in the SECDED computation and Tag hit-miss logic.

When considering caches with multiple ways, access and energy benefits stay similar (as the detection and correction logic is replicated) but area benefits are reduced heavily. For instance, for a 128 row 4-4 configuration, 4-way provides a 7% area reduction but 8-way provides none and 16-way suffers a 4% area increase (data not shown in a figure due to space limitations).

VII. Conclusions

This work shows how to provide a built-in RTD functionality in an SRAM array that enables to track in real time the parity of the cells in each array column. An analysis of an optimized RTD-SRAM design shows that the use of a tree reduction for the RTCP calculation results in minimal impact on read and write access delay. The proposed RTD-SRAM is then applied to a tag array for error protection by using the RTCP to perform error erasure. The proposed Fast-Tag Hit RTD is compared against a state of the art Fast-Tag Hit ECC and shown to provide faster latency, lower energy and lower or similar area while having a comparable error protection.

References

[1] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," in *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305-316, Sept. 2005, doi: 10.1109/TDMR.2005.853449.

[2] R. W. Hamming, "Error detecting and error correcting codes," in *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147-160, April 1950, doi: 10.1002/j.1538-7305.1950.tb00463.x.

[3] Y. Sazeides et al., "2D Error Correction for F/F based Arrays using In-Situ Real-Time Error Detection (RTD)," in the *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Frascati, Italy, 2020, pp. 1-4, doi: 10.1109/DFT50435.2020.9250878.

[4] Luong Dinh Hun et al., "Mitigating soft errors in highly associative cache with CAM-based tag," IEEE International Conference on Computer Design, 2005, doi: 10.1109/ICCD.2005.76.

[5] W. Wu, D. Somasekhar and S. Lu, "Direct Compare of Information Coded With Error-Correcting Codes," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 11, pp. 2147-2151, Nov. 2012, doi: 10.1109/TVLSI.2011.2169094.

[6] P. Reviriego, S. Pontarelli, M. Ottavi and J. A. Maestro, "FastTag: A Technique to Protect Cache Tags Against Soft Errors," in *IEEE Transactions on Device and Materials Reliability*, vol. 14, no. 3, pp. 935-937, Sept. 2014, doi: 10.1109/TDMR.2014.2332616.

[7] Byeong Yong Kong et al., "Low-Complexity Low-Latency Architecture for Matching of Data Encoded With Hard Systematic Error-Correcting Codes." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 22, no. 7 (2014): 1648-1652 doi: 10.1109/TVLSI.2013.2276076.

[8] A. Gendler, A. Bramnik, A. Szapiro and Y. Sazeides, "Don't Correct the Tags in a Cache, Just Check Their Hamming Distance from the Lookup Tag," in the *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Vienna, 2018, pp. 571-582, doi: 10.1109/HPCA.2018.00055.

[9] S. Mitra, S. A. Seshia and N. Nicolici, "Post-silicon validation opportunities, challenges and recent advances," in *Design Automation Conference (DATE)*, Anaheim, CA, 2010, pp. 12-17, doi: 10.1145/1837274.1837280.

[10] J. Kim, N. Hardavellas, K. Mai, B. Falsafi and J. Hoe, "Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding," in the *40th Annual IEEE/ACM International Symposium on Microarchitecture*, Chicago, IL, 2007, pp. 197-209, doi: 10.1109/MICRO.2007.19.

[11] Shien-Yang Wu et al., "A highly manufacturable 28nm CMOS low power platform technology with fully functional 64Mb SRAM using dual/tripe gate oxide process," in *2009 Symposium on VLSI Technology*, Honolulu, HI, 2009, pp. 210-211.

[12] J. Maiz, S. Hareland, K. Zhang and P. Armstrong, "Characterization of multi-bit soft error events in advanced SRAMs," in *IEEE International Electron Devices Meeting* 2003, Washington, DC, USA, 2003, pp. 21.4.1-21.4.4, doi: 10.1109/IEDM.2003.1269335.

[13] M. Y. Hsiao, "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes," in *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395-401, July 1970, doi: 10.1147/rd.144.0395.