CBFD: A Count-Based Fault Detection Scheme for Memory Arrays

Yiannakis Sazeides*, Bushra Ahsan*, Isidoros Sideris*, Lorena Ndreu*, Sachin Idgunji† and Emre Özer†
*University of Cyprus †ARM

Abstract—[1] **The performance-cost benefits enjoyed for decades due to the scaling of device area are challenged by power and reliability constraints. Fixed power envelopes and increases in static and dynamic variations have lead to higher probability of parametric and wear-out failures. This is particularly true for processor memory arrays, such as caches, that dominate the area of modern processors and are built with minimum sized SRAM cells that are prone to failure. It is, therefore, becoming essential to develop scalable cost-effective fault-tolerant techniques for processor memory arrays.**

Our attempt, presented in this paper, to address scalable memory array fault tolerance is a novel error detection scheme that relies on a count-based method, called *CBFD* (count-based-fault-detection), that provides the number of 1s(0s) in an array at any given time. The state overhead of the method is $(log_2k) + 1$ bits for an array with k bits. For fault-free array operation a basic invariance is maintained by the method: the number of 1s(0s) never exceeds the array size or becomes negative. This invariance can be violated when there is a fault. The paper introduces the count-based method and explain its detection latency, overheads, and fault coverage. It also discusses how it can be used to detect soft and hard errors both in array cells and peripheral logic.

I. Introduction

For the past several decades, technological developments have facilitated the continuous miniaturization of devices on silicon chips. The resulting increase in device density has been offering designers the opportunity to place more functionality per unit area and recently has allowed the integration of large caches and many cores into the same chip. Unfortunately, the scaling of other key design parameters has not followed suit. In particular, voltage [1], [2] and silicon circuits probability of failure [3] are scaling slower and faster rate respectively than area scaling.

These distressing trends are leading to projections that the semiconductor industry may hit a scaling wall unless cost-effective techniques are developed to address the power and reliability challenges. In particular, it will be impossible to operate all on-chip resources, even at the minimum voltage for safe operation, due to power constraints, and the growing design and operational margins used to provide silicon primitives with resiliency against static [4] and dynamic [5] variations will consume the scaling benefits.

A recently published resilience roadmap underlines the magnitude of the reliability problem we are facing [3]. Table I shows the p_{fail} (probability of failure) predicted in [3] for inverters, latches and SRAM cells due to random dopant fluctuations as a function of technology node (the trends for negative-bias-temperature- instability [6] are similar). The trends clearly show that for all types of circuits the p_{fail} increases at a much faster rate than the scaling rate. However, not all circuits are equally vulnerable, SRAM cells that are usually built with minimum sized devices are highly more likely to fail. What is more is that if we resort to voltage operation below safety margins the SRAM p_{fail} increases exponentially [7].

TABLE I
PREDICTED p_{fail} FOR DIFFERENT TYPES OF CIRCUITS AND TECHNOLOGIES [3].

Technology	Inverter	Latch	SRAM
45nm	≈ 0	≈ 0	6.1e-13
32nm	≈ 0	1.8e-44	7.3e-09
22nm	≈ 0	5.5e-18	1.5e-06
16nm	2.4e-58	5.4e-10	5.5e-05
12nm	1.2e-39	3.6e-07	2.6e-04

These trends render paramount the development of reliability techniques that are both scalable and performance effective. This is especially important for processor memory arrays, such as caches, that take most of the real-estate in processors and contain numerous vulnerable to failure SRAM cells.

Existing techniques used in current processors such as ECC codes [8], sparing [9], larger more resilient cell [10] can help mitigate the problem with typical area costs in the range of 12-25% but are not scalable solutions because with higher parametric and wearout p_{fail} stronger and more expensive codes, additional spares and larger cells will be needed.

In this paper we present a scalable memory array fault-detection scheme called *CBFD* (count-based-fault-detection). The state overhead of this scheme is a counter with $(log_2k)+1$ bits for a memory array with size k bits. The proposed scheme requires *the read-write memory array invariance*: every write to a location is preceded by a read. This enables the CBFD counter to keep track of the number of 1s(or 0s) in the array at any given time. For a fault-free memory array the number of 1s(0s) can not exceed the array size or become negative. However, this invariance can be violated when there is faulty operation that CBFD detects to report a faulty memory array. The paper presents the operation as well the properties of the count-based method. It also discusses how CBFD can be used to detect soft and hard-errors in array cells and peripheral logic.

The remainder of the paper is organized as follows: the functionality and properties of the CBFD scheme are discussed in Section II. Section III examines the CBFD's fault coverage

[1]The research leading to this paper is supported by the European Commission FP7 project "Energy-conscious 3D Server-on-Chip for Green Cloud Services (Project No:247779 "EuroCloud")"

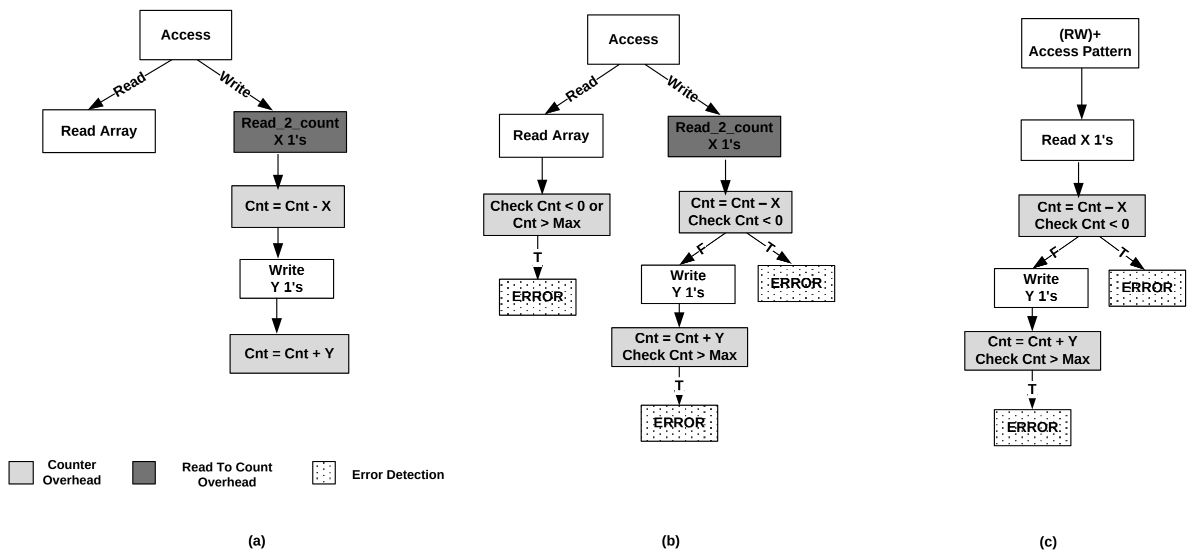

Fig. 1. (a) CBFD Mechanism Flow without error detection (b) CBFD Mechanism Flow with error detection (c)(RW)+ flow with error detection

and compares it to other protection schemes. We explain the potential uses of this scheme in Section IV. Section V discusses related work. Finally, we conclude in Section VI.

II. CBFD: Counter Based Fault Detection Method

This section presents discussion and analysis for the: (i) functionality of the proposed method, (ii) its detection latency, and (iii) overheads and ways to mitigate them depending on access patterns.

Unless indicated otherwise, a faulty array is assumed to have a single faulty cell to simplify presentation.

A. Method

Lets assume that it is desirable to track at any given time the number of 1's(or 0's) stored in a memory array, array henceforth, with N entries and B bits per entry using a counter. Further, lets assume that always at initialization the number of 1's in the array is known and stored in the counter.

A generic method that achieves the above is to perform before every write to an array entry a read from the entry and increment the counter by the difference between the number of bits with a 1 in the new and the old entry value. The counter value is not updated during normal reads. We refer to a read performed only for counter updating as *read-to-count*. The flow diagram on how this method works is shown in Fig. 1.a.

The above method relies on *the read-write array invariance*: before each write to an entry there is a read-to-count from that entry. Consequently, the number of 1's in different writes to the same entry are not accumulated. Only the 1's in the most recent write to each entry contribute to the counter's value. This means that the counter can only take values in the range of 0 to NB and the counter needs $(log_2 NB) + 1$ bits. When the counter value is 0 it means all the cells in the array are 0, and when the counter is NB all the cells are 1. In general when the counter has value x, it means there are x 1s and $NB - x$ 0s in the array. The method can be used to detect faults in a memory array by performing the checks illustrated in Fig. 1.b with an error detected (i) on a write whenever a counter gets outside the correct range, and (ii) on a read when the counter indicates the array contains all 0s(1s) and the output value contains a 1(0).

It is important to note that the proposed method does not require additional state to track which and the order of the locations that are accessed.

We illustrate the CBFD operation in Fig. 2 assuming a 4x1 array that contains initially all zeros that is indicated in a CBFD counter of 3 bits. Each write access is denoted by a $W_i[j]$ with i representing the write value and j the entry. Similarly $R[j]$ denotes reading from entry *j*. In order for the counter to keep track of the values, a $R2C[j]$ (read-to-count from entry j) are added before every write.

The original access pattern in Fig. 2.a with five accesses, 2 reads and 3 writes, results in a fault-free CBFD value of '0'. Fig. 2.b and Fig. 2.c show how faults can be detected on a write and a read respectively.

Fig. 2 highlights a subtle but important property of CBFD: it can detect faults with a delay. For example, when an entry with a faulty value is overwritten it causes the counter to be updated incorrectly, but the value of the counter at this point may be in the correct range and the fault is not detected. The fault will be detected if subsequent array updates cause the counter value to become <0(>maximum) or we read a 1(0) from an array that, as indicated by the counter, contains all 0s(1s). This non-determinism in detection delay is discussed next.

B. Fault Detection Latency

One important attribute of a fault detection method is its detection latency. For protection schemes such as parity and SECDED [8] the detection latency is mainly a function of the logic used to generate syndrome for an information word. For CBFD, however, the detection latency is non-deterministic and it can even be unbounded. Several parameters can influence CBFD's detection latency: (i) the array size: the larger the

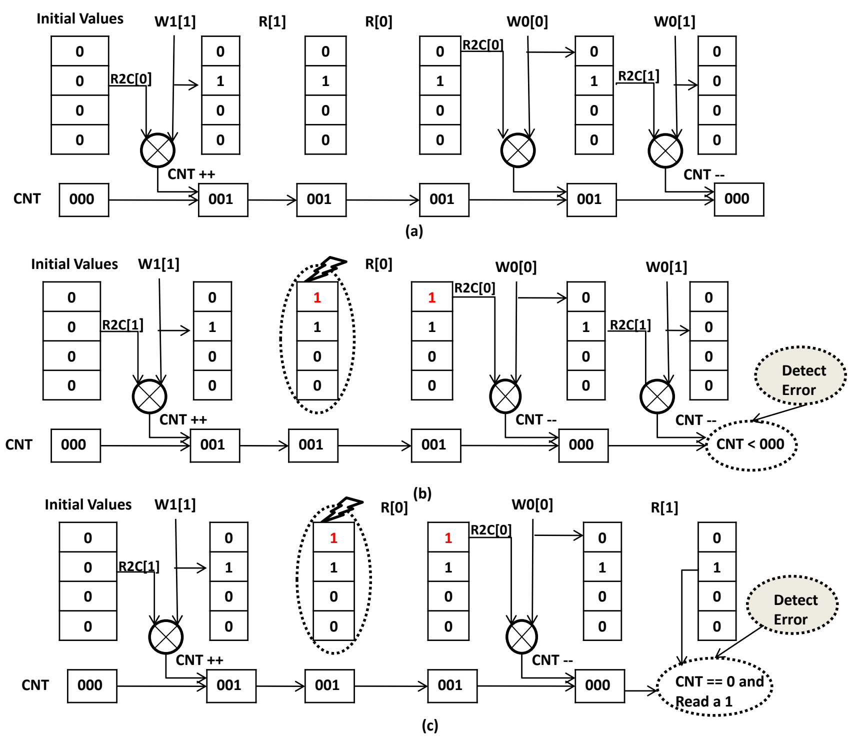

Fig. 2. (a) Fault free accesses, (b) Fault and Write detection, and (c) Fault and Read detection

array size the larger the range of the CBFD counter and the less likely for the counter to be at 0 or maximum values where faults are more likely to be detected, (ii) the dynamic program information used to access and write an array: this influences whether and how often a CBFD counter returns to 0 or the maximum value where faults are more likely to be detected, and (iii) the type of error: an entry with a soft-error when it is written contributes only once to an incorrect update of the CBFD counter. On the other hand, an entry with a hard error can potentially contribute every time the entry is written thus moving faster the counter towards the edges of the correct range and fault detection.

The implications of these observations is that the CBFD counter can be updated incorrectly and this may eventually lead to the detection of a fault but as long as the value of the counter remains in the correct range the fault can not be detected and its detection latency can be unbounded.

One approach to address the latency due to large counter range is to reduce the size of the array for which a CBFD counter is used for. One way to accomplish this is with array partitioning. Assume that we divide a NxB bit array in P partitions. P counters will need to track the number of 1's in a P smaller arrays each with size NxB/P bits. This is expected to help reduce detection latency but still it does not address completely unbounded detection latency.

A cursory analysis of the effects of dynamic program behavior and partitioning on the CBFD counters is shown in Fig. 3. This figure presents for different array structures and for all SPEC2K benchmarks how often a counter is at 0 or the maximum value [2]. Each array configuration is listed in the legend as 3-tuple: rows, columns, partitions.

The figure shows the average number of all same cycles for the structures simulated. The structures fall into two categories. For data caches (tag and data) and data TLB the number of all same cycles are below 40%. This is because these structures are heavily accessed and keeps getting filled up with blocks that break the all same behavior. Similarly for conditional predictor prediction bit, these cycles are low. For I$-tag and ITLB this average is very high due to blocks being written with mostly zeroe values. Also for IJUMPs and hysteresis bit, the average is high. Overall, the data shows that the coverage of reading from a same column is high but not sufficient for accurate and fast fault detection.

CBFD with Sweeping. One possible approach to limit CBFD's detection latency is to perform at regular intervals a sweep through an array and perform read-to-count of all array entries. Previous work [11] has shown that reinitializing core resources (flushing L1 caches and tlbs, resetting predictors state while maintaining L2) has a negligible impact on performance if it is done around 1 million or more cycles. The reason for this is that usually the number of entries in these structures is in the order of 100 to 1000 entries and they can be warmed up relatively quickly as compared to the interval length. This is particularly true for flushed L1 caches that are backed-up by an inclusive L2. Furthermore, as we explain above we do not need to perform a reinitialization but rather a read-to-count through each array without modifying the array content.

[2]The x-axis is not common between curves, i.e. a benchmark may correspond to different x-points

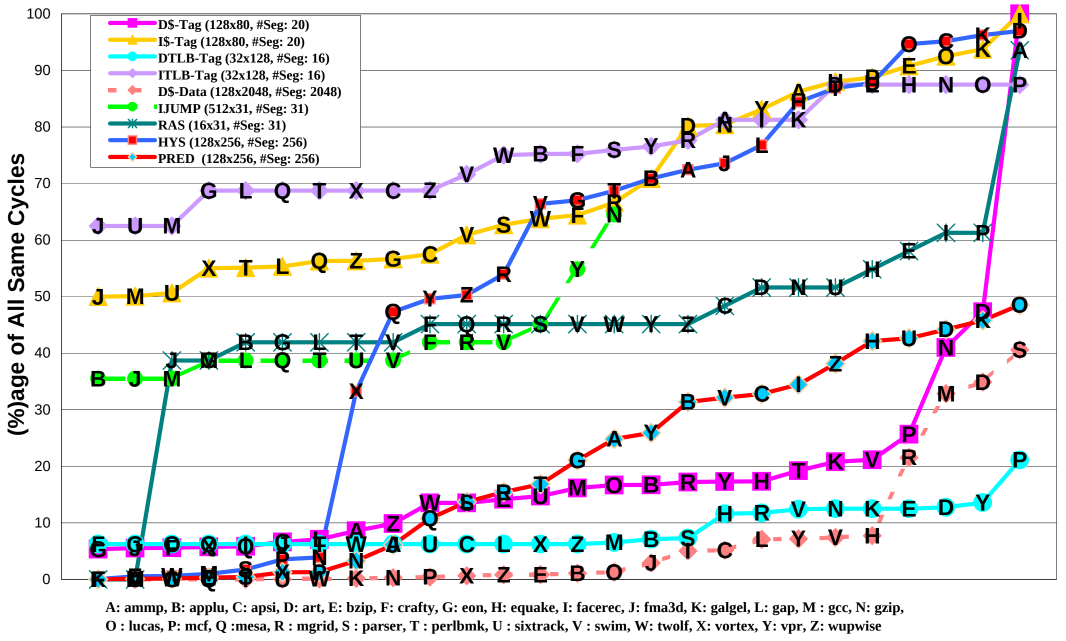

Fig. 3. Fraction of time an array partition has the same content for all cells

For a fault-free operation the counter after the sweep should become zero. If it is zero we just restore the previous counter value and resume operation, otherwise we have detected an error. The combination of CBFD and sweeping represents a low cost fault detection mechanism.

C. Overheads

A CBFD implementation incurs mainly the following overheads: the area for the counter, the energy for updating the counter, the energy for the reads-to-count, and the performance implications of the read-to-count accesses (due to potential conflicts with normal accesses).

The relative area overhead of the counter for an array with size k bits is $(1+\log_2(k))/k$, which becomes increasingly smaller with larger values of k. For instance for 128bits, 1KB, 8KB tables the counter needs to be 7+1, 13+1 and 16+1 bits respectively corresponding to 6.25%, 0.2% and 0.025% overhead. Thus area overhead of the method should be minimal as long as k is large enough. We also expect the energy overhead for updating the counter to be relatively small because the counter is small relative to the array size and is only updated on writes. We do not consider further these overheads in the paper.

Figure 4 shows the area overhead comparison of parity codes with CBFD (ECC has the same overhead as Parity for SECDED). For realistic number of partitions, the CBFD mechanism always has less hardware than ECC and parity. Partitioning the cache has no affect on ECC/parity overhead.

The remaining overheads are due to the read-to-count operations: the dynamic energy overhead of the reads-to-count and the potential performance degradation due to the delay of write operations to perform a read to count. We consider these two overheads in combination and we discuss next how to reduce them depending on array access patterns.

D. Minimizing Read-to-Count Overhead depending on Array Access Pattern

The above description of CBFD is for an array where *any* combination of reads and writes to a location is possible. However, certain processor arrays exhibit constrained access patterns that facilitate a reduction of the read-to-count overhead. For example, in several types of prediction arrays an entry is always read at the front-end before written at the back-end. Effects of wrong-path instructions can be eliminated by doing the CBFD update when instructions commit. Such access patterns, denoted using by *(RW)+*, have each write to a location preceded by a read and, therefore, the read-to-count overhead is eliminated. This is illustrated in Fig. 1.c.

The *(RW)+* access pattern happens often [12], not always, in data caches that are protected with error-correction-code when the access datum is smaller than the ECC granularity. In this case each write is preceded by read. Consequently, if it is desired to have CBFD protection for a data cache, enforcing the read-write invariance may incur small read-to-count overhead. This overhead seems to become even smaller when one considers that usually stores are less frequent than loads. For data arrays, both for instruction and data caches, a replaced block needs to be read-to-count even if it is not dirty to preserve the read-write invariance. This extra read can possibly be avoided if we can select the victim block on miss instead on fill time.

Tag and TLB arrays are written on a miss. Therefore, on a miss the tag of the victim block needs to be read-to-count. Analogous to data-arrays the read-to-count can be avoided if we can select the victim on a miss instead of on a fill.

We do not provide a full CBFD compliant data, tag and tlb array design in this work, this will be the subject of future work, we merely indicate that some of these arrays have

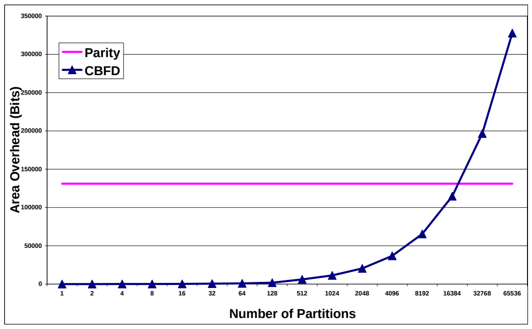

Fig. 4. Area Overhead Comparison between CBFD and Parity for Different Number of Segments (ECC has the same overhead as parity for SECDED)

properties that may make them amenable to CBFD protection.

One bit hysteresis arrays used to control update of predictions do not need a read before updating for a correct prediction. A direct write to a strong state is sufficient. For incorrect predictions the hysteresis needs to be read to decide whether to replace prediction. So incorrect predictions exhibit *(RW)+* pattern whereas for correct a read-to-count is needed.

Hysteresis arrays with two or more bits per entry need to be read before an update.Therefore, they exhibit *(RW)+* access pattern and do not need read-to-counts.

III. CBFD Coverage Comparison

Memory arrays are usually protected using information redundancy techniques, that is error detecting or error correcting codes (EDC or ECC) [8]. For n bits data words, k extra bits can be used to map the original data words to n+k bits data words. The minimum hamming distance (HD) between each pair of valid target (n+k bits) data words, determines the detection and correction capability of the used code. In particular, an EDC code can detect up to HD-1 errors, while an ECC code can correct up to (HD-1)/2 errors. A common error detection/correction code is SECDED, which stands for Single Error Correction Double Error Detection and has HD=4 (for 32bit data words needs 7 extra bits, while for 64bit 8 extra bits). Parity is the simpler protection scheme and detects one single error with the cost of 1 bit per data word (usually it is applied per byte).

We compare CBFD with parity check and ECC. Since the protection granularity of CBFD is different, we compare the three schemes in three different scenarios to get a more broad picture and show the summary in Table II.

(A) Scenario 1: Faults occur within group of bits protected by parity bit or ECC. Since CBFD is not necessarily on 64 bit basis, we consider this group of bits to be any number of bits with all faults occurring within these bits.

(i) Parity(assuming even parity): Detects all odd flips. It does not detect even flips.

(ii) ECC: Detects all one and two bit errors. In some cases more than two bit errors can be detected depending on the pattern they change the checked word into. Some three bit errors in ECC can be falsely reported as corrected errors if the resulting pattern of the word is manifested as a two bit error.

(iii) CBFD: detects more errors than parity since it is able to detect all odd flips and all even unbalanced flips. It is however not able to detect even balanced flips (We define balanced flip in which the total number of ones remain the same). This is because the counter value for such a case remains valid although there has been errors. As compared to ECC, no Hamming distance is required in CBFD to detect errors.

(B) Scenario 2: Comparing bursts of errors occurring in consecutive bits.

(i)Parity: Parity handles bursts by interleaving the bits which means that consecutive bits are checked by a different parity. Thus if a burst of error occurs it will belong to bits of different words and can be detected.

(ii) ECC: Similarly in ECC, consecutive bits are checked by different parity codes to avoid bursts of errors to lie in the same word and hence be detected by separate error checks.

(iii) CBFD: Interleaving the bits can be applied to CBFD as well. We can have a counter for one logical column that could span across several non consecutive physical columns. This will allow bursts of errors in consecutive bits to be checked by a different counter and hence detected. Interleaving counters for CBFD can be done in multiple ways with different columns or bit positions to be monitored by a separate counter.

(C) Scenario 3: Assuming multi bit errors occur in different words (protected by different check bits).

(i) Parity: Since parity is for every 8 bits, if the errors are spread in different check words, different parity codes can be used to check them and hence more errors can be detected.

(ii) ECC: In ECC as well errors across ECC granularity can be detected by different ECC codes.

(iii) CBFD: CBFD loses potential in this case if we do not have any partitioning and have one counter for the entire array. Every error will fall in the same array and will not be able to catch all multiple errors (unless they are odd or even with unbalanced flips). We can potentially solve this by partitioning the array to have more counters.

IV. Applications of CBFD

CBFD intended application is for error-detection in memory arrays but it may also be used to reduce energy. This section describes some of the CBFD applications.

CBFD is capable of detecting both soft and hard-faults in array cells as well as in the peripheral array logic (such as address decoder, column multiplexers).

Any fault that causes a cell to flip after it has been written correctly it has the potential to be detected because on the next write to the cell it will cause an incorrect counter update. Errors in peripheral logic can be detected because CBFD protects all data in an array and when data are written to the wrong location they can cause a cell flip. The per entry ECC will not detect the problem because the data and their code are correct but the address they are stored is not.

CBFD without sweeping represents a low-cost array fault detection scheme but possibly with limited fault coverage. Soft-errors are not likely to be detected but frequently occurring hard-errors will be detected.

TABLE II
ERROR DETECTION CAPABILITIES OF DIFFERENT SCHEMES

Scheme	Errors in One Word	Error Bursts	Errors in Different Words
Parity	Odd flips detected Even flips not detected	Through Column Interleaving	Detected
ECC	One and two flip detected	Through Column Interleaving	Detected
CBFD	Odd and Unbalanced Even flips detected. Balanced Even flips not detected	Through Column Interleaving	Loses Potential

CBFD with sweeping provides a low-cost high coverage fault detection method that can catch both soft and hard errors. Combining CBFD with existing protection per entry schemes, such as parity and SECDED, can improve the overall fault detection coverage due to the extra faults CBFD detects, which cannot be detected with the per-entry schemes alone.

Another application of CBFD is to combine it with a checkpointing scheme to provide a low-cost symptom based fault-tolerance.

The CBFD scheme can provide additional benefits beyond error detection. For example, counters can be used to minimize power dissipated in precharging bitlines. This can be accomplished when the counter is zero or is equal to the number of bits in the bitline. When this occurs the bitline contains all '0's or all '1's. Thus, instead of precharging a bitline in the array, the counter value can indicate the value that has to be read.

V. RELATED WORK

In [13] a two-dimensional coding scheme is proposed, which uses standard EDC/ECC per row along with EDC per column, thus allowing for detection (and correction) of clusters of faults. The method relies on read-before-writes, analogous to read-to-counts, to maintain the column parity. One of our key differences from this work is that we focus on low-cost detection where this earlier work aimed to provide both correction and detection.

It is common knowledge that most modern processors have some form of protection for architectural arrays such as caches and register files. What is less known, is that processors both in high availability systems but also embedded processors protect prediction arrays [14], [15]. This can be useful for reducing unnecessary stalls when running in lockstep with another processor.

Most EDC and ECC schemes have state overhead linear to the size of the array (k). The proposed scheme has $log_2 k$ overhead, which render it very cost effective. What is more, it can detect other types of errors, that standard parity codes cannot (like errors in peripheral logic). Last, it can be used complementary to other schemes and provide some more protection.

VI. CONCLUSIONS AND FUTURE WORK

This paper introduces a count-based fault detection scheme called *CBFD* that is based on the read-write invariance of memory arrays: every write is mostly preceded by a read. The method keeps track of the number of ones(zeros) in a memory array using only a $log_2 k$+1 counter for an array with size k bits. The paper presents the properties of CBFD as well as some applications/uses.

The proposed scheme opens up several directions of work including its more detailed evaluation/optimization for different applications as well as its combination and interaction with other error correction and detection techniques. One such line of work is to consider its usefulness for online testing. Another, interesting direction is to consider CBFD for detecting failures in DRAM memory arrays. Finally, the proposed method may have applications for power reduction, for example, it can be used to prevent precharging bitlines that all their cells store the same value.

REFERENCES

[1] S. Borkar, "Design Challenges of technology scaling," *IEEE Micro*, vol. 19, no. 4, pp. 23–29, Jul. 1999.

[2] Y. Taur, "CMOS design near to the Limit of Scaling," *IBM Journal of Research and Development*, vol. 46, no. 2/3, pp. 213–222, Mar./May 2002.

[3] S. R. Nassif, N. Mehta, and Y. Cao, "A resilience roadmap," in *DATE*, 2010, pp. 1011–1016.

[4] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *DAC '03: Proceedings of the 40th annual Design Automation Conference*. New York, NY, USA: ACM, 2003, pp. 338–342.

[5] K. Bowman, J. Tschanz, C. Wilkerson, S.-L. Lu, T. Karnik, V. De, and S. Borkar, "Circuit techniques for dynamic variation tolerance," in *DAC46*. New York, NY, USA: ACM, 2009, pp. 4–7.

[6] S. Zafar, B. Lee, J. Stathis, A. Callegari, and T. Ning, "A model for negative bias temperature instability (nbti) in oxide and high kappa," in *VLSI Technology, 2004. Digest of Technical Papers. 2004 Symposium on*, 2004, pp. 208 – 209.

[7] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off cache capacity for reliability to enable low voltage operation," in *ISCA35*, June 2008, pp. 203–214.

[8] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 26, no. 2, pp. 147–160, 1950.

[9] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. V aden, "Ibm power6 microarchitecture," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 639 –662, 2007.

[10] J. P. Kulkarni, K. Kim, and K. Roy, "A 160 mv, fully differential, robust schmitt trigger based sub-threshold sram," in *ISLPED '07: Proceedings of the 2007 international symposium on Low power electronics and design*, 2007, pp. 171–176.

[11] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec, "Performance implications of single thread migration on a chip multi-core," in *SIGARCH Computer Architecture News*, 2005, p. 2005.

[12] K. M. Lepak and M. H. Lipasti, "Silent stores for free," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 33, 2000, pp. 22–31.

[13] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. Hoe, "Multi-bit error tolerant caches using two-dimensional error coding," in *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 197–209.

[14] C. McNairy and R. Bhatia, "Montecito: a dual-core, dual-thread itanium processor," *Micro, IEEE*, vol. 25, no. 2, pp. 10 – 20, march-april 2005.

[15] "Cortex-a9 technical reference manual," infocenter.arm.com, 2010.