Error Correction and Detection for Computing Memories Using System Side Information

Clayton Schoeny, Irina Alam, Mark Gottscho, Puneet Gupta, and Lara Dolecek
Electrical and Computer Engineering, UCLA
Los Angeles, CA, 90095

***Abstract*—Error correction and detection are the core components of all modern memory systems. Current computing memory systems use simple coding schemes to simultaneously meet the resiliency and latency requirements. In this paper, we review our recent results on context-aware coding for computing memories, an approach that explicitly takes into account various intrinsic side information for improved robustness to faults. We discuss both error correction and detection, codes' theoretical properties, and provide examples of how these solutions can be implemented in practice. We explicitly describe the special case of the error localization codes. We also discuss promising future directions and connections with classical information theoretic concepts.**

***Index Terms*—Error Correction and Detection, List Decoding, Computing Memories.**

I. Introduction

Word-addressable computing memories (such as caches and main memories) have become ubiquitous and nowadays span a wide spectrum from on-chip memories embedded in Internet-of-Things (IoT) devices all the way up to off-chip main memories for high-performance server applications. Unfortunately, errors in computing memories have also increased. For example, Google has observed 70000 failures in time (FIT)/Mb in commodity on-chip memory, with 8% of modules affected per year [1], while Facebook has found that 2.5% of their servers have experienced memory errors per month [2]. With the proliferation of computing memories, and the surge in applications that use them, these trends are expected to worsen even further.

A conventional way of dealing with errors in memories is to deploy error control coding. As in all other memory and storage technologies, error-control codes (ECCs) are a classic way to build resilient computing memories by adding redundancy: A code maps each information message of length k bits (symbols) to a unique codeword of length n bits (symbols), $n > k$, that allows up to a certain number of errors to be detected and/or corrected. The theoretical development of ECCs for computing memories, including SRAM and DRAM, have thus far implicitly assumed that every message is equally likely to be stored and that every error pattern of the same weight is equally likely to occur. This approach has allowed for simplicity in the ECC design, which typically uses codes capable of correcting a small number of bit or symbol errors, e.g., $(39, 32)$ and $(72, 64)$ Single-Error-Correcting Double-Error-Detecting (SEC-DED) codes are routinely used in on-chip memories [3]–[5], and $(144, 128)_{16}$ Single-Symbol-Correcting Double-Symbol-Detecting (SSC-DSD) codes—such as ChipKill—are popular in server mainframes [6]–[9]. Here and elsewhere $(n, k)_q$ denotes a linear code that maps k input bits (symbols) into a codeword of length n bits (symbols) over a finite field of size q. If the code is binary, we omit the $q = 2$ subscript. Relatively short codes (with inevitably limited error correction) are necessary in computing memories due to stringent latency and implementation complexity constraints. However, current ECC techniques have become too weak to overcome the errors caused by the ever-increasing density of RAM in warehouse-scale computers, high-performance computer (HPC) applications [10], and low-cost on-chip SRAM memory [11]–[13].

While there has been a recent explosion of results on coding for memories [14]–[17], these works have primarily tackled non-volatile memories (NVMs), and Flash memories in particular as the main NVM technology. On the other hand, computing memories are at present predominantly volatile memories, although future computing memories may also include emerging non-volatile devices such as MRAMs. Computing memories differ from storage memories, such as Flash, in terms of function, organization, and the underlying physics, and, as a result, in terms of error types and patterns as well as acceptable coding solutions. As a result, the rich literature on coding for Flash is not readily applicable to the domain of computing memories.

As an alternative to using more expensive codes which would incur prohibitively high latency and complexity, we present a recently developed context-aware approach that relies on side information drawn from system properties, while still using relatively short and simple codes [18]–[21]. In this context-aware approach, one seeks to design more powerful ECCs for computing memories by taking into account the inherent redundancy in both the data and in the architectural organization while maintaining the simplicity of the codes. context-aware coding is a powerful machinery that may find applications in other domains including machine learning and natural language processing, see e.g., for related work [22], [23]. For the rest of this short survey paper, we will exclusively focus on the applications in computing memories. We first discuss the basic framework that uses conventional error correction/detection codes, and then we show how to expand it to error localization codes. We also provide several pointers for future investigation.

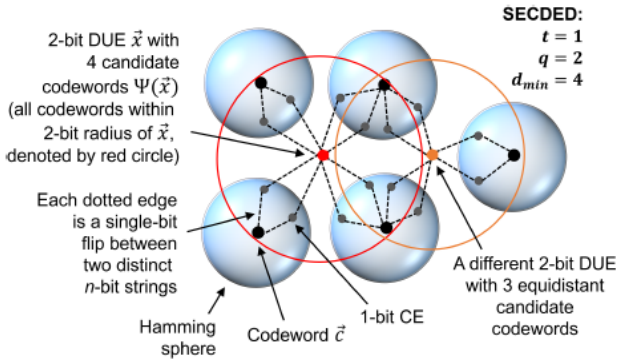

Fig. 1: Two-dimensional illustration of candidate codewords for 2-bit DUEs in the n-dimensional Hamming space of a binary SECDED code.

II. SD-ECC FRAMEWORK

We first investigate how architectural context can substantially improve the error recovery even for a fixed code, in the framework we call software-defined ECC (SD-ECC) [18]. SD-ECC uses system information and practical list decoding to improve error tolerance. As a representative exemplar of the errors we seek to correct, we focus on the detectable-but-uncorrectable errors (DUEs) in the context of (t)-symbol error-correcting, $(t+1)$-symbol error-detecting codes (for short, (t)SC$(t+1)$SD codes). DUEs are especially problematic since they typically cause the entire system to panic or rolling back to a checkpoint to avoid data corruption; both operations harm system availability and degrade the performance. For instance, even with state-of-the-art strong memory protection using a ChipKill-Correct ECC, the Blue Waters supercomputer suffers from a memory DUE rate of 15.98 FIT/GB [24]. This rate is high enough that whole-system checkpoints would likely be required every few hours, and would add a significant performance and energy overhead to HPC applications [10]. For industry-standard SECDED codes that perform better and use less energy than ChipKill, DUEs are at least an order of magnitude more frequent [24] and compound the reliability/availability problem further.

A. DUE theoretical analysis

SDECC is based on the fundamental observation that when a $(t+1)$-symbol DUE occurs in a (t)SC$(t+1)$SD code $\mathcal{C}$, there remains significant information in the received string $\vec{x}$. This information can be used to recover the original message $\vec{m}$ with reasonable certainty. It is not the case that the original message was completely lost, i.e., one need not naïvely choose from all q^k possible messages. In fact, there are exactly

$$N = \binom{n}{t+1}(q-1)^{(t+1)} \tag{1}$$

ways that the DUE could have corrupted the original codeword, which is far less than q^k. But guessing correctly out of N possibilities is still difficult. In practice, there are just a handful of possibilities: we call them candidate codewords, or CEs for short.

If the hardware ECC decoder registers a DUE, there can be several equidistant candidate codewords at the q-ary Hamming distance of exactly $(t+1)$ from the received string $\vec{x}$. We denote the set of candidates by $\Psi(\vec{x}) \subseteq \mathcal{C}$. See Fig. 1 for illustration.

The size of the candidate codeword list $|\Psi(\vec{x})|$ is independent of the original codeword; it depends only on the error vector $\vec{e}$ due to linearity of the code $\mathcal{C}$. That is,

$$|\Psi(\vec{x})| = |\Psi(\vec{c}+\vec{e})| = |\Psi(\vec{e})|. \tag{2}$$

The number of candidate codewords $|\Psi(\vec{e})|$ for any given $(t+1)$ DUE $\vec{e}$ has a linear upper bound that makes DUE recovery tractable to implement in practice.

Lemma 1. *For any error $\vec{e}$ with $wt_q(\vec{e}) = (t+1)$ in a (t)SC$(t+1)$SD linear q-ary code $\mathcal{C}$ of length n,*

$$|\Psi(\vec{e})| \leq \left\lfloor \frac{n(q-1)}{t+1} \right\rfloor.$$

Proof. The received string $\vec{x}$ is exactly q-ary distance 1 from the t-boundary of the nearest Hamming sphere(s). Thus, there are at most $n(q-1)$ single-element perturbations $\vec{p}$ such that $\vec{y} = \vec{x}+\vec{p}$ is a CE inside a Hamming sphere of a codeword. For each perturbation that results in a CE, there must be exactly t more single-element perturbations to fully arrive at a candidate codeword $\vec{c'}$. Because we cannot perturb the same elements more than once to arrive at a given $\vec{c'}$, there cannot ever be more than $\lfloor (n(q-1))/(t+1) \rfloor$ candidate codewords. □

The probability of correctly guessing the original codeword—without the use of any side information—for a specific error $\vec{e}$ is simply the reciprocal of the number of candidate codewords: $P_G(\vec{e}) = 1/|\Psi(\vec{e})|$. Let $\overline{P_G}$ be the average probability of guessing the correct codeword over all possible $(t+1)$-symbol DUEs. Also let $\sum_{\vec{e}}$ represent the summation over all possible $(t+1)$ symbol-wise error vectors $\vec{e}$. Then

$$\overline{P_G} = \frac{1}{N}\sum_{\vec{e}} \frac{1}{|\Psi(\vec{e})|}. \tag{3}$$

For a particular construction of a given code, we define $W_q(w)$ as the total number of codewords that have q-ary Hamming weight w. Then $W_q(d_{min})$ refers to the total number of minimum weight non-$\vec{0}$ codewords; its value depends on the exact constructions of the code parameters (including the generator matrix, parity check matrix, minimum distance d_{min} of the code, and the message and codeword dimensions). The average number of candidate codewords over all possible $(t+1)$-symbol DUEs is denoted as μ.

Lemma 2. *For a linear q-ary (t)SC$(t+1)$SD code $\mathcal{C}$ of length n and with given $W_q(d_{min} = 2t+2)$, the average number of candidate codewords μ over all possible $(t+1)$-symbol DUEs is*

$$\mu(n,t,q) = \frac{\binom{2t+2}{t+1} W_q(2t+2)}{\binom{n}{t+1}(q-1)^{(t+1)}} + 1.$$

Proof. In order to find the average number of candidate codewords, we must sum the number of candidate codewords for each unique $(t+1)$ q-ary error $\vec{e}_E$ where $E = i_1, i_2, \cdots, i_{(t+1)}$, and $i_1 \neq i_2 \neq \cdots \neq i_{(t+1)}$. We then divide that sum by the number of error-vectors (n choose $(t+1)$). By linearity and without loss of generality, assume $\vec{c} = \vec{0}$. We know that the only codewords $\vec{c'} \in \mathcal{C}$ that can satisfy $\Delta(\vec{c} - \vec{c'}, \vec{e}) = (t+1)$ have weight $W_q(d_{min})$. Each such $\vec{c'}$ that has $\vec{c}_{i_1} = \vec{e}_{i_1}$, $\vec{c}_{i_2} = \vec{e}_{i_2}$, etc., then has ($d_{min}$ choose $(t+1)$) distinct error-vectors $\vec{e}_E$. Thus summing over all error-vectors, each codeword $\vec{c'}$ with $wt(\vec{c'}) = d_{min}$ contributes to (d_{min} choose $(t+1)$) candidate codewords. To average, we divide $[(d_{min}$ choose $(t+1))] \times W_q(d_{min})$ by (n choose $(t+1)$). We also divide by $(q-1)^{(t+1)}$ because each non-zero element of the error vector $\vec{e}_E$ can take values from 1 to $q-1$. Finally, we add 1 to the expression since the original codeword $\vec{c}$ is a candidate codeword for every possible error-vector, and was not already counted in $W_q(d_{min})$. □

We find that μ is often easier to compute than $\overline{P_G}$ for long symbol-based codes; this is useful because $1/\mu$ is a lower bound on $\overline{P_G}$.

In the context of the current set-up, for any linear $(39, 32)$ SEC-DED code (which necessarily has minimum weight 4), we find by elementary counting arguments that there can never be more than 19 candidates for any double-bit DUE; in other codes of interest, the number is usually smaller. The most commonly used $(39, 32)$ SEC-DED code in main memory systems is the Hsiao code [5] as it minimizes the overall number of logic gates. The highest number of candidate codewords for any DUE with the Hsiao $(39, 32)$ SEC-DED code is 15. As a result, for this $(39, 32)$ SEC-DED code, simply using the basic side-information of instruction frequencies allows our SD-ECC framework to achieve 34% recovery of all possible 2-bit DUEs, as compared to 8.5% recovery by randomly picking a candidate codeword [18].

So far we have bounded the number of candidate codewords for any $(t+1)$-symbol DUE; we now show how to find these candidates. The candidate codewords $\Psi(\vec{x})$ for any $(t+1)$-symbol DUE received string $\vec{x}$ is simply the set of equidistant codewords that are exactly $(t+1)$ symbols away from $\vec{x}$. More formally, let subscripts be used to index symbols in a vector, starting from the most significant position. Then

$$\Psi(\vec{x}) = \vec{c} \cup \{\vec{c'} \in \mathcal{C} : \Delta_q(\vec{c'} - \vec{c}) = d_{min}, \vec{c'}_i = \vec{x}_i \ \forall i \text{ where } \vec{e}_i \neq 0\}. \quad (4)$$

Notice that this equation depends on the error $\vec{e}$ and original codeword $\vec{c}$, but we only know the received string $\vec{x}$.

Fortunately, there is a simple and intuitive algorithm (shown in Alg. 1) to find the list of candidate codewords $\Psi(\vec{x})$ with runtime complexity $O(nq/t)$. The essential idea is to try every possible single symbol *perturbation* $\vec{p}$ on the received string. Each *perturbed string* $\vec{y} = \vec{x} + \vec{p}$ is run through a simple software implementation of the ECC decoder, which only requires knowledge of the parity-check matrix $\mathbf{H}$ ($O(rn\log q)$ bits of storage). Any $\vec{y}$ characterized as a CE produces a candidate codeword from the decoder output.

Algorithm 1 **Compute list of candidate codewords $\Psi(\vec{x})$ for a $(t+1)$-symbol DUE $\vec{x}$ in a linear (t)SC$(t+1)$SD code with parameters $[n, k, d_{min}]_q$. For error vectors, subscripts indicate the symbol positions of errors, but not their q-ary values. For example, $\vec{e}_3$ corresponds to $[00100\ldots0]$.**

```
for i = 1 : n do
    for j = 1 : q − 1 do
        p⃗ ← j ∗ e⃗_i //(symbol i in p gets q-ary value j, all others 0)
        y⃗ ← x⃗ + p⃗
        if Decoder(y⃗) not DUE then
            c⃗' ← Decoder(y⃗) //Compute candidate codeword
            if c⃗' ∉ Ψ(x⃗) then //If candidate not already in list
                Ψ(x⃗) ← Ψ(x⃗) ∪ c⃗' //Add candidate to list
            end if
        end if
    end for
end for
```

Further pruning of the list of candidate codewords until one arrives at a single codeword can be done in a variety of ways, using a combination of system knowledge and elementary statistical properties of the data. One such example uses empirical entropy of the data, as we next discuss.

B. Experimental results

Entropy is one of the most powerful metrics to measure data similarity. Software applications typically have inherent *value locality* in their data, regardless of their hardware representation. For example, an image-processing program is likely to work on regions of pixels that exhibit similar color and brightness, while a natural language application will see certain characters and words more often than others.

Based on the above observations, we propose a simple but effective data recovery policy that chooses the candidate that minimizes the overall cacheline Shannon entropy. For example, a 512-bit cacheline is often comprised of 8 64-bit words. If one of these words yields a DUE, we plug in each candidate codeword, one at a time, and calculate the overall sample entropy.

More concretely, let $P(X)$ be the normalized relative frequency distribution of a $\ell \times b$-bit cacheline that has been carved into equal-sized Z-bit symbols, where each symbol χ_i can take 2^Z possible values. Entropy symbols are not to be confused with the codeword symbols, which can also be a different size. Then we compute the Z-bit-granularity `entropy` as follows:

$$\texttt{entropy} = -\sum_{i=1}^{\ell \times b/Z} P(\chi_i) log_2 P(\chi_i). \quad (5)$$

We observe low byte-granularity intra-cacheline entropy throughout the integer and floating-point benchmarks in the SPEC CPU2006 suite, and thus our entropy-based policy has high levels of success. Additionally, we mitigate the chance that our policy chooses the wrong candidate message by deliberately forcing a *panic* whenever the mean cacheline entropy is above a specified threshold *Panic Threshold*.

TABLE I: Percent Breakdown of SDECC *Entropy-8* Policy: (S = success, P = forced panic, M = MCE)

	panics taken			**panics not taken**			***random* baseline**		
	S	**P**	**M**	**S**	**P**	**M**	**S**	**P**	**M**
***conv.* baseline**	-	100	-						
$[39, 32, 4]_2$ **SECDED**	69.1	25.6	5.3	72.7	-	27.3	8.5	-	91.5
$[72, 64, 4]_2$ **SECDED**	71.6	23.7	4.7	75.3	-	24.7	5.0	-	95.0
$[45, 32, 6]_2$ **DECTED**	77.5	20.3	2.2	85.5	-	14.5	28.2	-	71.8
$[79, 64, 6]_2$ **DECTED**	84.0	14.5	1.5	89.0	-	11.0	20.5	-	79.5
$[36, 32, 4]_{16}$ **SSCDSD**	85.7	12.8	1.5	91.5	-	8.5	39.9	-	60.1

Results in Table I are an archetypical exemplar of our study. In this experiment, we randomly introduce $(t + 1)$-symbol DUEs into 20 SPEC CPU2006 benchmarks compiled against GNU/Linux for the open-source 64-bit RISC-V instruction set v2.0 [25]. We produce representative memory access traces, consisting of randomly-sampled 64-byte demand read cachelines. Table I contains the resulting outcome percentages, for a variety of codes, in three different scenarios: panics allowed, panics not allowed, and randomly choosing a candidate codeword (thus forming the baseline).

As more errors beyond the guaranteed error correction are being recovered from, the chance of miscorrections (MCEs) also increases. The proposed framework carefully balances the increased error tolerance over the baseline setting while controlling for the increase in MCEs. We observe that when panics are taken the MCE rate drops significantly, by a factor of up to $7.3\times$, without significantly reducing the success rate. This indicates that our *Panic Threshold* mechanism appropriately judges when we are unlikely to correctly recover the original information.

III. SD-ECC FRAMEWORK FOR ERROR LOCATING CODES

The previous section demonstrated how the SD-ECC framework can be effectively used with the existing error correcting techniques, such has SECDED. However, for certain emerging applications, e.g., microcontroller-class IoT devices, even simple error correcting method are often too costly in terms of overhead, power, and latency. Due to its simplicity, a single parity bit error-detecting code is often used; however, even in our context-aware framework, basic parity is of limited help in recovery. For example, any DUE produced by a k-bit message with parity detection will necessarily have $k + 1$ candidate codewords, since each possible bit-flip produces the correct overall parity.

We have proposed a new class of codes, *Ultra-Lightweight Error-Localizing Codes* (ULELCs) [21], that reside in between simple parity and Hamming codes, both in terms of complexity and correction capability. Generally, error-localizing codes split a codeword into fixed-sized chunks, and when an error occurs, are able to locate the chunk that contains the error. The fixed-size property is due to the creation of error-localizing codes through the tensor product operation of the parity-check matrices of an error-detecting and an error-correcting code. The primary ability of ULELC is the ability to localize any single-bit error to a specific, customizable chunk of bits in the codewords. Then, using similar recovery policies as before, we can choose the most likely candidate codeword. The customizability and non-uniformity of the chunks allow us to create the ULELC with specific data architectures in mind.

A. Theoretical Analysis

One possible way of detecting a single-bit error in a chunk is to use a parity-bit per chunk. However, the redundancy overhead of this trivial segmented parity code is equal to the number of chunks. The proposed ULELC is more efficient than that. Given r redundant parity bits, there can be $2^r - 1$ distinct non-zero columns in the parity check matrix $\mathbf{H}$. Using this fact, the ULELC can localize any single-bit error to within one of the 2^r-1 possible chunks. To create a ULELC code, we first assign to each chunk a distinct non-zero binary column vector of length r bits. Then each column of $\mathbf{H}$ is simply filled in with the corresponding chunk vector. Note that r of the chunks will also contain the associated parity-bit within the chunk itself; we call these *shared chunks*, and they are precisely the chunks whose columns in $\mathbf{H}$ have a Hamming weight of 1. Since there are r shared chunks, there must be $2^r - r - 1$ *unshared chunks*, which each consist of only data bits. Shared chunks are unavoidable because the parity bits must also be protected against faults, just like the message bits. An ULELC code has a minimum distance of two bits by construction to support detection and localization of single-bit errors. Thus, the set of candidate codewords must also be separated from each other by a Hamming distance of exactly two bits.

B. Experimental results

ULELC provides the ability to localize the error to a specific chunk of bits in the codewords. This reduces the number of possible candidate codewords for which a single bit error could have produced the received codeword. Once the error is localized, software defined heuristic recovery policies that leverage on side-information about memory contexts such as observable patterns and structure found in both instructions and data, as mentioned in the previous sections, are then used to choose the most likely codeword. Hence, we call this *Software-Defined Error-Localizing Codes* (SDELC). We have evaluated SDELC on a set of five small embedded benchmarks (`blowfish` and `sha` from the mibench suite [26] as well as `dhrystone`, `matmulti` and `whetstone`) and six larger benchmarks from the AxBench approximate computing suite [27]. For each workload, we randomly select 1000 instruction fetches and 1000 data reads from the trace and exhaustively apply all possible single-bit faults to each

of them. Separate recovery policies are used for instruction and data memories. For instruction memory, a lookup table containing the relative frequency of all instructions is pre-computed and the candidate codeword that represents the instruction with the highest frequency is chosen. For data memory, the average Hamming distance to nearby data in the same 64B chunk of memory is computed and the candidate with minimum average hamming distance is selected. SDELC correctly recovers from up to 90% (70%) of random single-bit soft faults in data (instructions) with just three parity bits per 32-bit word.

IV. Concluding Remarks

In this paper, we presented a brief overview of the new paradigm of context-aware coding for computing memories using system side information. We used simple examples and error models to illustrate the key idea; future directions abound. We now summarize extensions of the baseline framework that would be interest for future investigation:

- Theoretical analysis of the decoding list size under more refined error models and data types;
- Establishment of the structural code properties in the context of SD-ECC (e.g., distance relationships among minimum weight codewords).
- Investigation of other types of similarity metrics and the development of theoretical guarantees on the decoding performance and on the MCEs, under these metrics;
- Development of error localization codes with non-uniform segments, see e.g., [20];
- Implementation of various error models for different architectures and different data types (data vs. instruction memory; RISC-V ISA, MIPS, etc.).

Acknowledgement

Work supported by the funding support from NSF under grant CCF-BSF 1718389 and Qualcomm Innovation Fellowship.

References

[1] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," in *Proc. ACM SIGMETRICS*, vol. 37, no. 1, Seattle, Washington, Jun. 2009, pp. 193–204.

[2] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2015.

[3] P. Nikolaou, Y. Sazeides, L. Ndreu, and M. Kleanthous, "Modeling the Implications of DRAM Failures and Protection Techniques on Datacenter TCO," in *Proc. ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Dec. 2015.

[4] J. Chang, M. Huang, J. Shoemaker, J. Benoit, S.-L. Chen, W. Chen, S. Chiu, R. Ganesan, G. Leong, V. Lukka *et al.*, "The 65-nm 16-MB shared on-die L3 cache for the dual-core Intel Xeon processor 7100 series," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 4, pp. 846–852, Mar. 2007.

[5] M.-Y. Hsiao, "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, Jul. 1970.

[6] T. J. Dell, "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," *IBM Microelectronics Division*, vol. 11, pp. 1–23, Nov. 1997.

[7] X. Jian, H. Duwe, J. Sartori, V. Sridharan, and R. Kumar, "Low-Power, Low-Storage-Overhead Chipkill Correct via Multi-line Error Correction," in *Proc. IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2013.

[8] X. Jian, J. Sartori, H. Duwe, and R. Kumar, "High Performance, Energy Efficient Chipkill Correct Memory with Multidimensional Parity," *IEEE Computer Architecture Letters*, vol. 12, no. 2, pp. 39–42, Jul. 2013.

[9] S. Kaneda and E. Fujiwara, "Single Byte Error Correcting Double Byte Error Detecting Codes for Memory Systems," *IEEE Transactions on Computers*, vol. C-31, no. 7, pp. 596–602, Jul. 1982.

[10] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems," in *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2014.

[11] S. Hamdioui, G. Gaydadjiev, and A. J. van de Goor, "The State-of-art and Future Trends in Testing Embedded Memories," in *Proc. International Workshop on Memory Technology, Design and Testing (MTDT)*, Aug. 2004.

[12] S.-L. Lu, Q. Cai, and P. Stolt, "Memory Resiliency," *Intel Technology Journal*, vol. 17, no. 1, May 2013.

[13] J. Wang and B. H. Calhoun, "Minimum Supply Voltage and Yield Estimation for Large SRAMs Under Parametric Variations," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 19, no. 11, pp. 2120–2125, Nov. 2011.

[14] A. Jiang, R. Mateescu, M. Schwartz, and J. Bruck, "Rank Modulation for Flash Memories," *IEEE Transactions on Information Theory*, vol. 55, no. 6, pp. 2659–2673, Jun. 2009.

[15] M. Qin, E. Yaakobi, and P. H. Siegel, "Constrained Codes that Mitigate Inter-Cell Interference in Read/Write Cycles for Flash Memories," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 5, pp. 836–846, May 2014.

[16] E. Hemo and Y. Cassuto, "d-Imbalance WOM Codes for Reduced Inter-Cell Interference in Multi-Level NVMs," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 9, pp. 2378–2390, Sep. 2016.

[17] L. Dolecek and F. Sala, *Channel Coding Methods for Non-Volatile Memories*. Foundations and Trends in Information and Coding Theory, 2016.

[18] M. Gottscho, C. Schoeny, L. Dolecek, and P. Gupta, "Software Defined Error-Correcting Codes," in *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Toulouse, France, Jun. 2016.

[19] M. Gottscho, I. Alam, C. Schoeny, L. Dolecek, and P. Gupta, "Low-Cost Memory Fault Tolerance for IoT Devices," in *Proc. ACM/IEEE International Conference on Compilers, Architecture, and System Synthesis (CASES)*, Oct. 2017.

[20] C. Schoeny, F. Sala, M. Gottscho, I. Alam, P. Gupta, and L. Dolecek, "Context-aware resiliency: Unequal message protection for random-access memories," in *Proc. IEEE Information Theory Workshop (ITW)*, Kaohsiung, Taiwan, Nov. 2017, pp. 166–170.

[21] I. Alam, C. Schoeny, L. Dolecek, and P. Gupta, "Parity++: Lightweight error correction for last level caches," in *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Luxembourg City, Luxembourg, Jun. 2018.

[22] Y. Wang, M. Qin, K. R. Narayanan, A. Jiang, and Z. Bandic, "Joint Source-Channel Decoding of Polar Codes for Language-Based Sources," in *Proc. of IEEE Global Communications Conference (GLOBECOM)*, Dec. 2016.

[23] Y. Wang, K. R. Narayanan, and A. A. Jiang, "Exploiting Source Redundancy to Improve The Rate of Polar Codes," in *Proc. of IEEE International Symposium on Information Theory (ISIT)*, Jun. 2017.

[24] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters," in *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2014.

[25] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0," DTIC Document, Tech. Rep., 2014.

[26] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proceedings of the IEEE International Workshop on Workload Characterization (IWWC)*, 2001.

[27] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "AxBench: A Multiplatform Benchmark Suite for Approximate Computing," *IEEE Design and Test*, vol. 34, no. 2, pp. 60–68, 2017.