Karlsruhe Institute of Technology

Low-Overhead Fault-Tolerant Techniques for Emerging Memories and Computing Paradigms

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation
von

Surendra Hemaram
aus Bilara, Rajasthan, India

Tag der mündlichen Prüfung: 17.12.2025

Erster Gutachter: Prof. Dr. Mehdi B. Tahoori
Karlsruhe Institute für Technology, Germany

Zweiter Gutachter: Prof. Dr. Lorena Anghel
Grenoble INP, France

Acknowledgement

Reflecting on this joyful PhD journey, I feel truly privileged to have been supported by exceptional individuals whose contributions have been instrumental in bringing this dissertation to fruition.

First and foremost, I would like to express my heartfelt gratitude to my advisor, Prof. Dr. Mehdi Tahoori. His mentorship throughout my PhD has been nothing short of outstanding. His insightful feedback, thoughtful discussions, and support, especially during the challenging phases, have been invaluable. I also want to thank my Master's supervisor, Prof. Dr. Jai Narayan Tripathi at IIT Jodhpur, for motivating me towards the research.

I am profoundly thankful for the opportunity to collaborate with outstanding teams from across the globe. These collaborations have been a defining and enriching part of my PhD experience. I extend my sincere thanks to Prof. Dr. Francky Catthoor from imec Leuven. His guidance throughout my PhD journey has been invaluable. I am also sincerely grateful to my colleagues at imec Leuven—Siddharth Rao, Sebastien Couet, Tommaso Marinelli, and Aftab Baig—for the opportunity to collaborate on multiple research projects. Working alongside them was both intellectually enriching and personally rewarding.

My heartfelt thanks also go to my wonderful colleagues at the Chair of Dependable Nano Computing, KIT, Germany. I am grateful to our secretary, Ms. Iris Schröder-Piepka, for her exceptional skills and for ensuring that everything ran seamlessly. I would also like to thank Dr.-Ing. Dennis Gnad, Dr.-Ing. Jonas Krautter, and Dr.-Ing. Christopher Münch for their insightful discussions and technical guidance. I want to acknowledge Dr.-Ing. Mahta Mayahinia and Dr.-Ing. Soyed Tuhin Ahmed, whose technical insights and consistent support were invaluable throughout my PhD journey. I am grateful to Zhe Zhang, Sergej Meschkov, and Vincent Meyers for their kind assistance in resolving any issues related to the German language. I would also like to thank Brojogopal Sapaui, Priyanjana Pal, and Shanmukha Mangadahalli Siddaramu, with whom I shared not only professional experiences but also many cherished personal memories. Lastly, I would like to thank Dina Moussa, Atousa Jafari, Sina Bakhtavari Mamaghani, Seyedeh Maryam Ghasemi, Mahboobe Sadeghipour Rudsari, Haneen Seyam, Paula Duarte, Ali Khelejani, Gürol Saglam, Maha Shatta, Tara Gheshlaghi, Christopher Keßler, Huashuangyang Xu, Simon Pankner, and Aradhana Dube. Their support has been greatly appreciated. I am also thankful to the Dean's Office for their support and assistance.

I would like to extend my deepest gratitude to my parents, Hemaram Ji and Kamala Devi. Despite the physical distance, their unwavering love and support have been a constant source of strength throughout this journey. I would also like to express my deepest gratitude to my wife, Khushi. Her unwavering love and support have meant more to me than words can convey. I feel profoundly blessed to have such a loving and supportive family. I am also grateful to my friend Aksh and my other friends for their support and encouragement.

I would also like to extend my sincere thanks to the Government of India, the German Embassy, and the Karlsruhe Institute of Technology (KIT) for granting me the opportunity to pursue my studies in Germany. Their support played a pivotal role in shaping this academic journey. Finally, I would like to thank myself for the hard work, persistence, and resilience that carried me through this journey. Reaching this milestone would not have been possible without the dedication I consistently put forth.

In closing, I wish to express my profound admiration for those who believe in and tirelessly advocate for racial equality. Their unwavering efforts continue to inspire me and countless others around the world. I humbly and respectfully dedicate this work to them.

Hiermit erkläre ich an Eides statt, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben haben und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen – die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Karlsruhe, October 20, 2025

Surendra Hemaram

Abstract

Today, computing systems play a crucial role in everyday life, with applications ranging from compact embedded control units in automotive, healthcare, and smart devices to large-scale, high-performance computing systems. Memory is a critical component of any modern computing system, as its performance, energy efficiency, and reliability can greatly influence overall system behavior. The need for high performance and low power consumption in modern computing systems has led to aggressive technology scaling, which increasingly limits the potential of conventional Complementary Metal-Oxide Semiconductor (CMOS)-based memory technologies in advanced technology nodes. Emerging Non-Volatile Memory (NVM) have revolutionized data storage and computing performance, making them viable alternatives to CMOS memories. Several NVM technologies, including Resistive Random Access Memory (RRAM), Phase Change Random Access Memory (PCRAM), and Spin Transfer Torque Magnetic Random Access Memory (STT-MRAM), are being explored as potential replacements for conventional CMOS memories. Among various NVM technologies, STT-MRAM is the most promising candidate for memory applications, offering high density, low power consumption, high access speed, and non-volatility, making it a viable alternative for both standalone off-chip memory and System-on-Chip (SoC)-level caches in modern general-purpose computing systems, as demonstrated by several industrial implementations (e.g., TSMC, Intel, Samsung, and Everspin). However, STT-MRAM has some reliability issues fundamentally different from those of the conventional CMOS memories, such as stochastic and asymmetric switching behavior, sensitivity to process and temperature variations, short-bit failure, and oxide breakdown, leading to soft and hard errors. Mitigating these failure mechanisms through fault-tolerant/error-correction design techniques is essential to improve in-field reliability and manufacturing yield of STT-MRAM devices.

The advancements in emerging memory technologies have also led to the development of various hardware accelerator platforms designed to accelerate Neural Network (NN) computations, which demand massive storage capacity and Matrix-Vector Multiplication (MVM) operations. In particular, the Computation-in-Memory (CiM) paradigm leverages crossbar arrays of resistive NVM technologies—such as STT-MRAM, RRAM, and PCRAM—to seamlessly integrate storage with energy-efficient analog MVM directly within the memory array, often referred to as a memristive crossbar. The CiM architecture addresses the memory wall issue of the conventional Von-Neumann architecture by combining storage and computation within the same memory array. Additionally, unlike CiM, non-CiM digital Multiply and Accumulate (MAC)-based NN accelerators, which utilize memory for storage rather than computation, have also benefited from advancements in memory technologies by leveraging them as on-chip or off-chip weight storage. However, ensuring the reliability of NN computing systems is essential, as emerging memories are also prone to faults such as manufacturing defects, device non-idealities, conductance/resistance variation, temperature sensitivity, and write/read failures. These issues can lead to both soft and hard errors, which may alter stored weight values and degrade the accuracy of NN inference. Therefore, implementing fault-tolerant mechanisms is essential to maintain reliable NN operation, particularly in safety-critical applications.

In this thesis, we examine low-overhead fault-tolerant techniques that utilize Error-Correcting Code (ECC) in conjunction with architectural adaptations tailored to specific system-level requirements, aiming to enhance the reliability of emerging memory technologies used in both general-purpose computing systems and NN computing systems (NN accelerators). In the context of memory applications for general-purpose computing systems and/or standalone memory applications, we specifically target improving the run-time reliability of STT-MRAM. To this end, the thesis presents low-overhead fault-tolerant schemes to mitigate soft and hard errors resulting from intrinsic failure mechanisms in STT-MRAM technology. ① Hard errors in STT-MRAM arise from various failure mechanisms such as manufacturing defects, stuck-at faults, and oxide breakdown. To address this, we propose an efficient block error correction pointer-based strategy for hard error correction. We also incorporate experimental measurement data obtained from manufactured STT-MRAM chips, and the proposed method aligns well with our specific STT-MRAM error distribution. ② The switching characteristic of STT-MRAM is asymmetric in nature, where the likelihood of switching from 1→0 differs from that of 0→1. In such scenarios, uniform error correction coding is inefficient. To address this, we propose a memory-efficient, asymmetric, and adaptive error correction strategy based on the Hamming weight of data bits, which operates with negligible overhead in conjunction with an adaptive ECC framework. ③ Interconnects become more critical with technology scaling, primarily due to the growing impact of parasitic resistive-capacitive delays. The parasitic resistance of signal lines compromises STT-MRAM cell reliability, particularly in relation to the location of the driver. To address this, we propose a systematic, location-aware error correction strategy that protects memory sub-arrays non-uniformly.

In the context of memories for NN computing systems, this thesis targets both CiM architecture based on resistive NVMs and binary weight memories of non-CiM digital MAC-based NN hardware accelerators. The proposed fault-tolerant strategies are based on ECC schemes, co-designed with architectural adjustments and, where applicable, integrated into the NN training process. These techniques are optimized for memory efficiency and apply to both CiM architectures based on resistive NVM and binary weight memories of non-CiM digital MAC-based NN accelerators. ① Resistive NVM used in a memristive crossbar-based CiM architecture is susceptible to different failure mechanisms such as device-to-device and cycle-to-cycle variation, random-telegraph noise, and process and temperature-induced variability. These non-idealities can introduce both soft and hard errors. To address this, we propose a memory-efficient column-based online error correction strategy for memristive crossbars, which enables reliable MVM by combining the idea of a checksum and a linear block code ECC. ② The memories in digital MAC-based NN accelerators are used to store the known NN model parameters. This provides the opportunity to embed the ECC parity bits within the data bits, without increasing the number of NN parameters during the ECC encoding in case of binary weight memories, and eliminates storage requirements for parity bits. Furthermore, the approach can also be applied to CiM architecture with a binary memory element.

Zusammenfassung

Tegenwoordig spelen computersystemen een cruciale rol in het dagelijks leven, met toepassingen variërend van compacte ingebouwde besturingseenheden in auto's, gezondheidszorg en slimme apparaten tot grootschalige, krachtige computersystemen. Geheugen is een cruciaal onderdeel van elk modern computersysteem, aangezien de prestaties, energie-efficiëntie en betrouwbaarheid ervan een grote invloed kunnen hebben op het algehele gedrag van het systeem. De behoefte aan hoge prestaties en een laag stroomverbruik in moderne computersystemen heeft geleid tot agressieve technologische schaalvergroting, waardoor het potentieel van conventionele CMOS-gebaseerde geheugentechnologieën in geavanceerde technologische knooppunten steeds meer wordt beperkt. Opkomende NVM hebben een revolutie teweeggebracht in gegevensopslag en computerprestaties, waardoor ze een levensvatbaar alternatief zijn geworden voor CMOS-geheugens. Verschillende NVM-technologieën, waaronder RRAM, PCRAM en STT-MRAM, worden onderzocht als mogelijke vervanging voor conventionele CMOS-geheugens. Van de verschillende NVM-technologieën is STT-MRAM de meest veelbelovende kandidaat voor geheugentoepassingen. Deze technologie biedt een hoge dichtheid, een laag stroomverbruik, een hoge toegangssnelheid en niet-vluchtigheid, waardoor het een haalbaar alternatief is voor zowel standalone off-chip geheugen als SoC-level caches in moderne algemene computersystemen, zoals blijkt uit verschillende industriële implementaties (bijv. TSMC, Intel, Samsung en Everspin). STT-MRAM heeft echter enkele betrouwbaarheidsproblemen die fundamenteel verschillen van die van conventionele CMOS-geheugens, zoals stochastisch en asymmetrisch schakelgedrag, gevoeligheid voor proces- en temperatuurvariaties, short-bit-storingen en oxide-doorbraak, wat leidt tot soft- en hard-fouten. Het verminderen van deze storingsmechanismen door middel van fouttolerante/foutcorrectieontwerptechnieken is essentieel om de betrouwbaarheid in het veld en de productieopbrengst van STT-MRAM-apparaten te verbeteren.

De vooruitgang in opkomende geheugentechnologieën heeft ook geleid tot de ontwikkeling van verschillende hardwareversnellingsplatforms die zijn ontworpen om NN-berekeningen te versnellen, die enorme opslagcapaciteit en MVM-bewerkingen vereisen. Het CiM-paradigma maakt met name gebruik van crossbar-arrays van resistieve NVM-technologieën, zoals STT-MRAM, RRAM en PCRAM, om opslag naadloos te integreren met energiezuinige analoge MVM direct in de geheugenarray, vaak aangeduid als een memristieve crossbar. De CiM-architectuur pakt het geheugenprobleem van de conventionele Von-Neumann-architectuur aan door opslag en berekeningen binnen dezelfde geheugenarray te combineren. Bovendien hebben, in tegenstelling tot CiM, niet-CiM digitale MAC-gebaseerde NN-versnellers, die geheugen gebruiken voor opslag in plaats van berekeningen, ook geprofiteerd van de vooruitgang in geheugentechnologieën door deze te gebruiken als on-chip of off-chip gewichtsopslag. Het is echter essentieel om de betrouwbaarheid van NN-computersystemen te waarborgen, aangezien nieuwe geheugens ook gevoelig zijn voor fouten zoals fabricagefouten, niet-ideale eigenschappen van apparaten, variaties in geleiding/weerstand, temperatuurgevoeligheid en schrijf-/leesfouten. Deze problemen kunnen leiden tot zowel soft- als hard-fouten, die de opgeslagen gewichtswaarden kunnen wijzigen en de nauwkeurigheid van NN-inferentie kunnen verminderen. Daarom is het implementeren van fouttolerante mechanismen essentieel om een betrouwbare werking van NN te behouden, met name in veiligheidskritische toepassingen.

In dit proefschrift onderzoeken we fouttolerante technieken met lage overhead die gebruikmaken van ECC in combinatie met architecturale aanpassingen die zijn afgestemd op specifieke vereisten op systeemniveau, met als doel de betrouwbaarheid te verbeteren van opkomende geheugentechnologieën die worden gebruikt in zowel algemene computersystemen als NN computersystemen (NN versnellers). In de context van geheugentoepassingen voor algemene computersystemen en/of stand-alone geheugentoepassingen richten we ons specifiek op het verbeteren van de betrouwbaarheid tijdens de uitvoering van STT-MRAM. Daartoe presenteert het proefschrift fouttolerante schema's met lage overhead om zachte en harde fouten als gevolg van intrinsieke storingsmechanismen in STT-MRAM-technologie te verminderen. ① Harde fouten in STT-MRAM ontstaan door verschillende storingsmechanismen, zoals fabricagefouten, vastgelopen fouten en oxidebreuk. Om dit aan te pakken, stellen we een efficiënte, op pointers gebaseerde strategie voor harde foutcorrectie voor. We hebben ook experimentele meetgegevens van geproduceerde STT-MRAM-chips meegenomen, en de voorgestelde methode sluit goed aan bij onze specifieke STT-MRAM-foutverdeling. ② De schakelkarakteristiek van STT-MRAM is asymmetrisch van aard, waarbij de kans op schakelen van 1→0 verschilt van die van 0→1. In dergelijke scenario's is uniforme foutcorrectiecodering inefficiënt. Om dit aan te pakken, stellen we een geheugenefficiënte, asymmetrische en adaptieve foutcorrectiestrategie voor op basis van het Hamming-gewicht van databits, die met verwaarloosbare overhead werkt in combinatie met een adaptief ECC-raamwerk. ③ Interconnecties worden steeds belangrijker naarmate de technologie kleiner wordt, voornamelijk vanwege de toenemende impact van parasitaire resistieve-capacitieve vertragingen. De parasitaire weerstand van signaallijnen brengt de betrouwbaarheid van STT-MRAM-cellen in gevaar, met name in relatie tot de locatie van de driver. Om dit aan te pakken, stellen we een systematische, locatiebewuste foutcorrectiestrategie voor die geheugensubarrays op niet-uniforme wijze beschermt.

In de context van geheugens voor NN computersystemen richt dit proefschrift zich zowel op CiM architectuur op basis van resistieve NVMs als op binaire gewichtsgeheugens van niet-CiM digitale MAC-gebaseerde NN hardwareversnellers. De voorgestelde fouttolerante strategieën zijn gebaseerd op ECC-schema's, die samen met architecturale aanpassingen zijn ontworpen en, waar van toepassing, in het NN-trainingsproces zijn geïntegreerd. Deze technieken zijn geoptimaliseerd voor geheugenefficiëntie en zijn van toepassing op zowel CiM-architecturen op basis van resistieve NVM als binaire gewichtsgeheugens van niet-CiM digitale MAC-gebaseerde NN-versnellers. ① Resistieve NVM die wordt gebruikt in een op memristieve crossbars gebaseerde CiM-architectuur is gevoelig voor verschillende storingsmechanismen, zoals variaties tussen apparaten en cycli, willekeurige telegraafruis en door processen en temperatuur veroorzaakte variabiliteit. Deze onvolkomenheden kunnen zowel zachte als harde fouten veroorzaken. Om dit aan te pakken, stellen we een geheugenefficiënte, op kolommen gebaseerde online foutcorrectiestrategie voor memristieve crossbars voor, die betrouwbare MVM mogelijk maakt door het idee van een checksum en een lineaire blokcode ECC te combineren. ② De geheugens in digitale MAC-gebaseerde NN-versnellers worden gebruikt om de bekende NN-modelparameters op te slaan. Dit biedt de mogelijkheid om de ECC pariteitsbits in de databits in te bedden, zonder het aantal NN parameters tijdens de ECC codering te verhogen in het geval van binaire gewichtsgeheugens, en elimineert de opslagvereisten voor pariteitsbits. Bovendien kan de aanpak ook worden toegepast op CiM architectuur met een binair geheugenelement.

Contents

List of publications

[64] S. Hemaram, M. B. Tahoori, F. Catthoor, S. Rao, S. Couet, V. Pica, and G. S. Kar, "Soft and hard error-correction techniques in stt-mram", *IEEE Design & Test*, vol. 41, no. 5, pp. 65–82, 2024. DOI: 10.1109/MDAT.2024.3395972.

[65] S. Hemaram, M. B. Tahoori, F. Catthoor, S. Rao, S. Couet, and G. S. Kar, "Hard error correction in stt-mram", in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 752–757. DOI: 10.1109/ASP-DAC58780.2024.10473796.

[66] S. Hemaram, M. B. Tahoori, F. Catthoor, S. Rao, S. Couet, T. Marinelli, V. Pica, and G. S. Kar, "Asymmetric and adaptive error correction in stt-mram", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–14, 2025. DOI: 10.1109/TCAD.2025.3541188.

[67] S. Hemaram, M. Mayahinia, M. B. Tahoori, F. Catthoor, S. Rao, S. Couet, T. Marinelli, A. Farokhnejad, and G. S. Kar, "Intera-ecc: Interconnect-aware error correction in stt-mram", in *2025 Design, Automation & Test in Europe Conference (DATE)*, 2025, pp. 1–2. DOI: 10.23919/DATE64628.2025.10992925.

[68] S. Hemaram, T. Marinelli, M. Mayahinia, M. B. Tahoori, F. Catthoor, S. Rao, F. G. Redondo, and G. S. Kar, "Location-aware error correction for mitigating the impact of interconnects on stt-mram reliability", *IEEE Transactions on Device and Materials Reliability*, pp. 1–1, 2025. DOI: 10.1109/TDMR.2025.3627442.

[69] S. Hemaram, S. T. Ahmed, M. Mayahinia, C. Münch, and M. B. Tahoori, "A low overhead checksum technique for error correction in memristive crossbar for deep learning applications", in *2023 IEEE 41st VLSI Test Symposium (VTS)*, 2023, pp. 1–7. DOI: 10.1109/VTS56346.2023.10140077.

[70] S. Hemaram, M. Mayahinia, and M. B. Tahoori, "Adaptive block error correction for memristive crossbars", in *2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2022, pp. 1–6. DOI: 10.1109/IOLTS56730.2022.9897817.

[71] S. T. Ahmed, S. Hemaram, and M. B. Tahoori, "Nn-ecc: Embedding error correction codes in neural network weight memories using multi-task learning", in *2024 IEEE 42nd VLSI Test Symposium (VTS)*, 2024, pp. 1–7. DOI: 10.1109/VTS60656.2024.10538886.

[72] M. S. Roodsari, S. Hemaram, and M. B. Tahoori, "Non-uniform error correction for hyperdimensional computing edge accelerators", in *2025 IEEE European Test Symposium (ETS)*, 2025, pp. 1–6. DOI: 10.1109/ETS63895.2025.11049622.

[73] Z. Zhang, S. Hemaram, M. Mayahinia, C. Weis, N. Wehn, M. Tahoori, S. Nassif, G. Tshagharyan, G. Harutyunyan, and Y. Zorian, "Analysis and mitigation of radiation effects in sram-based register files", in *2025 IEEE European Test Symposium (ETS)*, 2025, pp. 1–4. DOI: 10.1109/ETS63895.2025.11049612.

[74] S. M. Siddaramu, M. Mayahinia, S. Hemaram, S. Ozev, and M. Tahoori, "Testing of passive memristive crossbars in ai hardware accelerators", in *2025 IEEE ATS/ITC-Asia (Accepted)*, 2025, pp. 1–6.

1. Introduction

1.1. VLSI Technology Scaling

The remarkable progress in Very Large Scale Integration (VLSI) technology over the past few decades has made semiconductor chips the foundation of virtually all modern electronic systems. For example, devices such as smartphones and laptops stand out as among the most ubiquitous and essential in daily life. A typical laptop incorporates numerous semiconductor components, with the SoC serving as its central and most critical element [1]. An SoC is a highly integrated VLSI circuit that combines multiple functional modules into a single chip. The advancement of VLSI technology has been predominantly driven by Moore's Law, which predicts that the number of transistors in a semiconductor chip doubles approximately every eighteen months. Technology scaling has historically improved the transistor density, computing performance, and cost-effectiveness of semiconductor devices. Figure 1.1 illustrates the evolution of microprocessor performance trends over the past several decades [2]. However, the energy efficiency gains have not kept pace with the scaling of device geometry. As transistor dimensions shrink, power density remains constant, and scaling leads to a rise in leakage power, making static power a critical design constraint. To address this, Dennard scaling advocates proportional reductions in both voltage and current alongside physical scaling. Yet, as the industry approaches the fundamental limits of both Moore's Law and Dennard scaling, the performance improvements in CPUs attributed to scaling have begun to taper off [3]–[6].

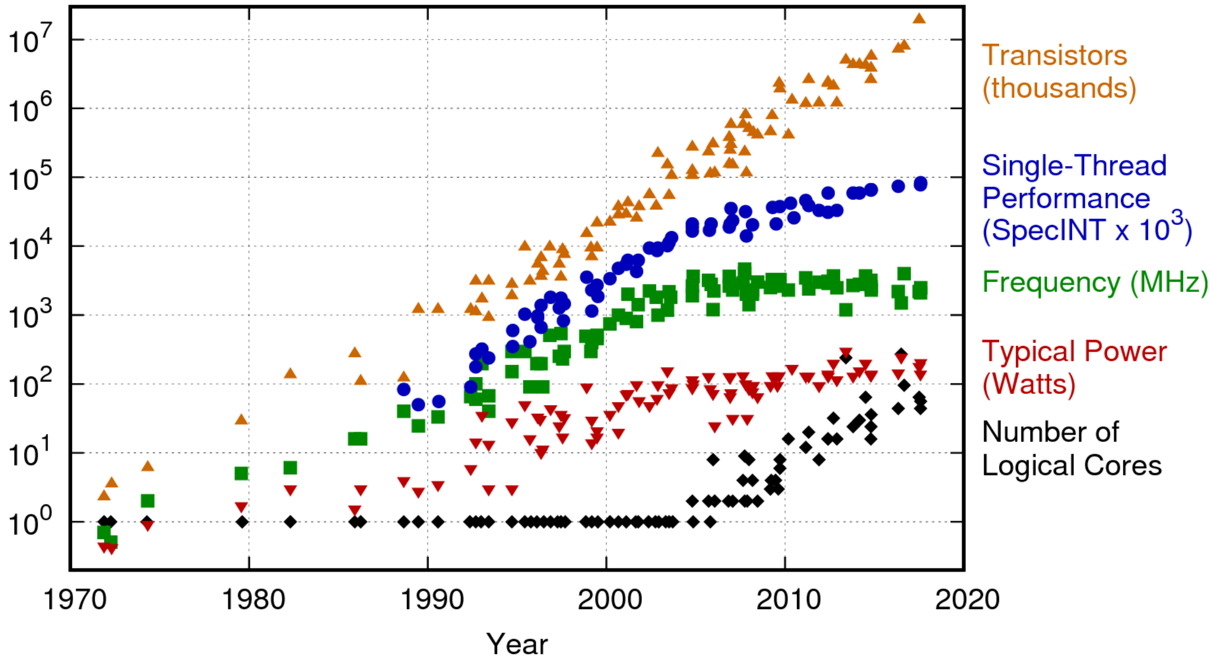


Figure 1.1.: 42 years of microprocessor trend Data (Reprinted from [2])

1.2. Power Wall and Memory Wall Issue

While continuous transistor scaling, as predicted by Moore's Law, has driven significant performance improvements in computing systems, it has also introduced critical power consumption challenges, commonly referred to as the "power wall" problem. As semiconductor technology nodes shrink, the leakage power becomes comparable to or even exceeds dynamic power consumption. This shift indicates that overall power consumption is increasingly dominated by leakage power [7], [8]. This is particularly problematic for modern integrated circuits, especially in energy-constrained environments such as mobile, edge, and IoT devices, where prolonged operation is essential [9].

In conventional Von Neumann computing architectures, the processor frequently accesses memory to fetch instructions and data, creating a performance bottleneck as memory access speed has not kept pace with the rapid improvements in processor performance, known as the "memory wall" bottleneck. This issue stems from the limited throughput between the processor and memory unit, which restricts the rate at which data can be supplied to the CPU, thereby hindering overall computational efficiency [10]–[12]. The exponential growth in data and the need to bridge the memory–processor performance gap have led to an increasing demand for high-density, low-power, and fast-access on-chip memory to overcome the memory wall and power wall bottleneck [13]–[15].

1.3. Why Emerging Memory Technologies?

Memory is a fundamental component of modern computing systems, playing a critical role in determining overall system performance. A typical memory hierarchy includes mass storage devices (such as hard drives and Flash memory), main memory (DRAM), multiple levels of cache (SRAM), and registers, as illustrated in Figure 1.2. This hierarchical structure is essential to bridge the speed gap between the processor and slower memory components by minimizing memory access latency. Conventional CMOS memory technologies have historically benefited from both Moore's Law and Dennard scaling, enabling significant improvements in memory capacity and performance over time. However, as CMOS memory technologies approach sub-nanometer nodes, their scalability is limited by intrinsic physical constraints, which adversely impact performance, energy efficiency, and area utilization [5], [16], [17]. Furthermore, the CMOS memory technologies like Static Random Access Memory (SRAM) and Dynamic Random Access Memory (DRAM) are volatile in nature and require continuous power to maintain data integrity. As a result, rising leakage power has also emerged as a critical challenge to the scalability of CMOS memory systems [18]. These challenges have collectively prompted significant research into alternative emerging NVM technologies, which offer improved scalability, energy efficiency, and performance, making them attractive candidates for next-generation computing systems.

A variety of NVM, including STT-MRAM, PCRAM, and RRAM, are under active consideration as potential replacements or alternatives to CMOS memories due to their enhanced scalability and low leakage power. Among the spectrum of NVMs, STT-MRAM has garnered exceptional attention from the research community and industry, positioning itself as a leader in the journey toward commercial viability [19]–[21]. The parameters of STT-MRAM with those of conventional CMOS-based memory technologies are shown in Table 1.1. STT-MRAM stands out with its enhanced read/write access, high endurance, scalability, and negligible leakage, making it a promising candidate compared to other NVMs, specifically in the case of SoC-level memories, which are primarily used as caches in modern computing systems [19], [22], [23]. STT-MRAM is also increasingly utilized in industrial, automotive, and embedded applications [22]–[28].

Parameters	SRAM	DRAM	NOR-FLASH	NAND-FLASH	STT-MRAM
Cell Size (F^2)	80–120	6–10	10	5	6–20
Read Latency	Low	Medium	High	High	Low
Write Latency	Low	Medium	High	High	Medium
Write Energy	Low	Low	High	High	Medium
Other Power Consumption	Leakage	Refresh	None	None	None
Supply Voltage	Low	Medium	High	High	Low
Non-Volatility	No	No	Yes	Yes	Yes
Scalability	Problem	Problem	Problem	Problem	Scalable
Endurance	High	High	Low	Low	High

Table 1.1.: Comparison of STT-MRAM memory technology with existing CMOS-based memory technology [29]

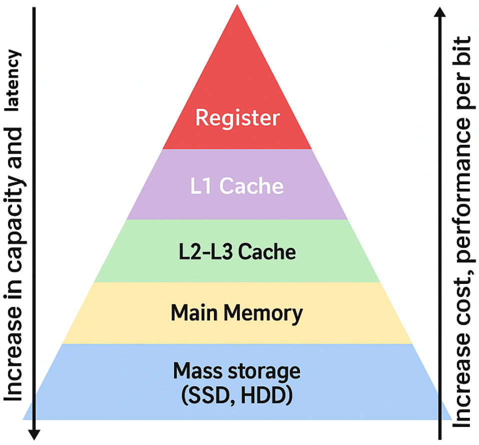

Increase in capacity and latency
Register
L1 Cache
L2-L3 Cache
Main Memory
Mass storage
(SSD, HDD)
Increase cost, performance per bit

Figure 1.2.: Typical memory hierarchy in a computing system

1.4. Emerging Memories in NN Computing

Furthermore, recent advancements in Artificial Intelligence (AI) algorithms have significantly enhanced the capabilities of NN across various applications. The NN models are computationally intensive and typically comprise millions of weight parameters, which must be stored and accessed efficiently. This requirement becomes even more challenging when deploying NNs on traditional Von Neumann architectures, where the constant data transfer between processing and memory units creates a memory wall issue [30]. Advancements of emerging memory technologies have enabled the development of specialized hardware accelerator platforms for accelerating NN computations. In particular, CiM paradigm based on a crossbar of resistive NVMs, such as STT-MRAM, RRAM, PCRAM, enables the seamless integration of storage and energy-efficient analog MVM within the memory array itself, which effectively addresses the memory wall limitation of traditional Von Neumann computing systems [31]–[36]. Additionally, unlike CiM, digital MAC-based NN accelerators, including Graphic Processing Unit (GPU), Tensor Processing Unit (TPU), Field Programmable Gate Array (FPGA), and digital Application Specific Integrated Circuit (ASIC), which use memory for storage rather than computation, have also benefited from emerging memory technologies. These accelerators also leverage high-density, low-power NVMs as on-chip and/or off-chip storage for NN weights, thereby enhancing memory efficiency and overall system performance [37], [38].

1.5. Dissertation Contributions

Although emerging technologies offer numerous advantages over traditional memory systems, they also introduce specific reliability challenges that need to be addressed for reliable implementation. Errors in memory systems can negatively impact the overall performance of the system. In the context of on-chip and off-chip memory applications in general-purpose computing systems, our focus is on STT-MRAM, a mature technology. However, STT-MRAM also presents specific reliability challenges, including stochastic and asymmetric switching characteristics, susceptibility to process variations, manufacturing defects, limited thermal stability, and location-dependent failures driven by interconnect parasitics. Together, these effects can lead to both soft errors (transient faults) and hard errors (permanent or stuck-at faults), potentially degrading the reliability of the memory system [29], [39]–[52], which must be mitigated through dedicated fault-tolerant strategies.

Beyond general-purpose computing applications, the memristive crossbar array formed by emerging resistive NVM (such as STT-MRAM, RRAM, PCRAM) is also a prominent choice for CiM realization to accelerate MVM computation, which is one of the core operations of NN computation. Among these NVM technologies, RRAM is particularly popular due to its ability to store multiple bits, making it well-suited for accommodating the large weight parameters of the NN model. Additionally, these emerging NVMs are becoming an alternative and/or complementary choice to conventional CMOS memory technologies in the case of digital-MAC-based hardware accelerators for on-chip and off-chip weight storage. However, resistive NVMs also encounter reliability challenges due to various failure mechanisms, including device-to-device variation, cycle-to-cycle variation, Random Telegraph Noise (RTN), stuck-at faults, conductance or resistance drift, temperature sensitivity, and random telegraph noise [53]–[63]. These device non-idealities and deviations can perturb the stored weights and degrade NN inference accuracy. To address these issues, it is essential to incorporate fault-tolerant mechanisms to mitigate failures and prevent performance losses.

The objective of this thesis is to develop different low-overhead fault-tolerant strategies aimed at enhancing the reliability of emerging memory technologies in both general-purpose computing and NN computing systems. The thesis objective is divided into two main parts. First, it aims to develop hardware-efficient fault tolerance strategies by combining ECC with lightweight architectural modifications to improve the reliability of STT-MRAM in general-purpose computing systems. Second, the goal is to develop fault-tolerant techniques that protect the weight memories of NN computing systems, including CiM architecture, as well as digital MAC-based hardware accelerators. The proposed methods are broadly applicable and can be adapted to other memory technologies and computing architectures with modest, technology-aware tuning. Overall, the contributions of this thesis are listed below.

Fault-tolerant techniques for reliable STT-MRAM in general-purpose computing systems

Study and analysis of error correction techniques: In this work, we provide a comprehensive study of different fault-tolerant/error-correction techniques to enhance the in-field reliability of STT-MRAM, addressing both soft (transient faults) and hard (permanent: stuck-at faults) errors, with potential applicability to other emerging NVM technologies. We have categorized existing works based on critical parameters, including error types, mitigation strategies, overhead, and architectural-level considerations. This study provides a detailed resource and analysis for researchers seeking to address similar reliability challenges in STT-MRAM or other emerging memory technologies. The work is detailed in Chapter 2 and partly based on [64].

Hard error correction to enhance stuck-at-fault tolerance: Failures, including manufacturing defects, short-bit faults, and oxide breakdown, lead to hard errors. We propose an efficient block error correction pointer strategy, which divides memory words into smaller blocks and stores the block-wise offset location of hard errors, rather than the absolute address. We have incorporated experimental measurement data from manufactured STT-MRAM chips to get the hard error distribution and design an efficient strategy. We have shown that the proposed method is hardware-efficient, with reduced decoding and memory overhead compared to existing approaches. The work is detailed in Chapter 3 and partly based on [65].

Addressing asymmetric failures: The switching characteristic of STT-MRAM is asymmetric in nature, where the likelihood of switching from 1→0 differs from that of 0→1. In such scenarios, uniform error correcting coding is inefficient, as it provides an equal probability of error correction for both switching directions. To address this, we propose an efficient, asymmetric, and adaptive error correction strategy based on the Hamming weight of data bits, which operates with negligible overhead alongside an adaptive ECC framework. We show that the proposed technique significantly enhances reliability, as measured by a last-level cache block error rate. The work is detailed in Chapter 4 and partly based on [66].

Mitigating the impacts of interconnect parasitics on bit cell reliability: Interconnects become increasingly critical as technology scales, primarily due to the growing impact of parasitic resistive-capacitive delays. The parasitic resistance of signal lines worsens cell writability and readability, especially in cells farther from the write driver. This work proposes a systematic location-aware non-uniform error correction strategy, where rows closer to the driver receive standard error correction, while rows farther from the driver are assigned more robust error correction. This strategy reduces ECC parity requirements by selectively applying stronger error correction to specific rows within the memory array, rather than uniformly across the entire array. The work is detailed in Chapter 5 and partly based on [67], [68].

1.5.1. Fault-tolerant techniques for weight memories in NN computing system

Error correction in a memristive crossbar-based CiM architecture: In this work, we propose a column-based error correction in memristive crossbars, which enables reliable MVM by combining the idea of a checksum and a linear block code. Firstly, we propose a scaled checksum-based strategy for performing single-column error correction. The proposed approach alleviates the problem of storing the checksum value into multiple columns of the crossbar, which arises from the limited number of stable levels in resistive NVMs. This is achieved through checksum scaling and checksum-aware NN training. The work uses Algorithm-based Fault Tolerance (ABFT) and a Hamming code-based checksum to perform error correction with a scaled checksum. Secondly, we demonstrate a ECC-based checksum for block error correction in a memristive crossbar, which protects a block of the memristive crossbar at a time, enabling multi-column error correction. The work is detailed in Chapter 6 and partly based on [69], [70].

Protecting binary weight memories of NN accelerator: Unlike memories in general-purpose computing systems, which can store arbitrary values, the memories in NN accelerators are used to store the known model parameters. This provides the opportunity to embed the ECC parity bits alongside the data bits without extending the memory size. We propose NN-ECC, a multi-task learning objective that integrates binary linear block ECC into the NN parameters during training. The proposed strategy does not increase the number of NN parameters during the ECC encoding process and eliminates all storage requirements for parity bits. The proposed approach is versatile and can accommodate various ECC schemes with different error correction capabilities. We have demonstrated zero memory overhead up to a 4-error correction capability, while incurring negligible accuracy drop due to ECC encoding. The proposed ECC encoding can also be extended to CiM architecture with a binary memristive device. The work is detailed in Chapter 7 and partly based on [71].

1.6. Dissertation Outline

In this chapter, we have presented the motivation and contributions of this thesis. The remaining sections of the thesis are organized into the following chapters:

- Chapter 2 introduces the relevant background and state-of-the-art to understand the scope of this thesis.
- Chapter 3 demonstrates a hard error correction strategy in STT-MRAM aimed at improving stuck-at-fault tolerance.
- Chapter 4 discusses adaptive and asymmetric error correction in STT-MRAM to mitigate asymmetric failures.
- Chapter 5 presents a systematic location-aware non-uniform error correction in STT-MRAM to address reliability issues caused by interconnect parasitics.
- Chapter 6 introduces a checksum-based error correction in memristive crossbar-based CiM architectures used for NN acceleration.
- Chapter 7 presents a zero-overhead ECC scheme tailored for binary weight memories of NN hardware accelerators.
- Chapter 8 concludes the thesis and outlines potential directions for future research.

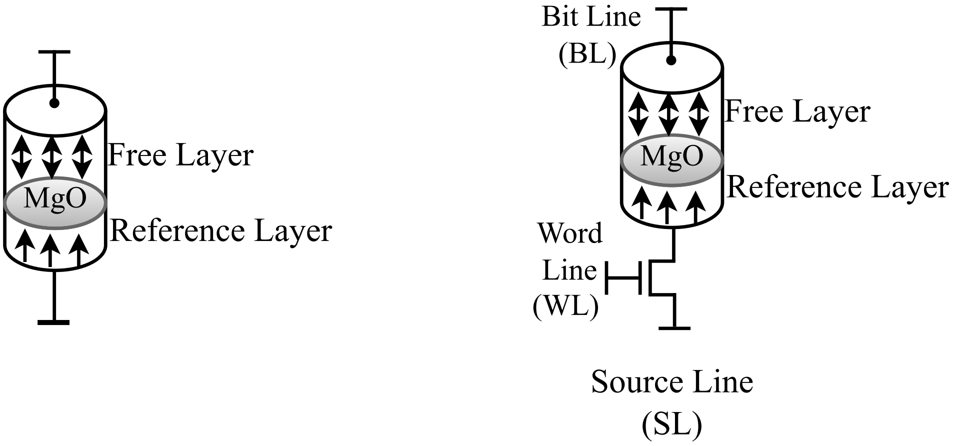
2. Background

This chapter begins with an overview of STT-MRAM and other emerging resistive NVM technologies, and also discusses their reliability challenges. It then introduces the fundamentals of NN computation and NN hardware accelerators, highlighting how memory faults impact NN performance. Next, we examine fault tolerance classification across different abstraction levels and standard fault-tolerant techniques to protect memories and computing systems. Finally, the chapter presents a comprehensive analysis of state-of-the-art techniques related to fault tolerance for STT-MRAM and weight memories in neural network computing systems, outlining key bottlenecks and research opportunities.

2.1. STT-MRAM

2.1.1. Basics of STT-MRAM

STT-MRAM employs a Magnetic Tunnel Junction (MTJ) as a basic storage element to store data. MTJ consists of two ferromagnetic layers separated by a thin oxide layer (e.g., MgO), as shown in Fig 2.1(a) [29], [39], [75], [76]. The reference layer has a fixed magnetic orientation, while the free layer can freely rotate its magnetic orientation. The resistance of the MTJ depends on the magnetic orientation of the free layer with respect to the reference layer. The parallel magnetic orientation (P) of both layers results in a Low Resistance State (LRS) (R_P). Conversely, the anti-parallel orientation (AP) leads to aHigh Resistance State (HRS) (R_{AP}). The magnetic orientations of the free layer can be switched by passing a spin-polarized current in the proper direction, thereby storing binary data as either LRS or HRS.



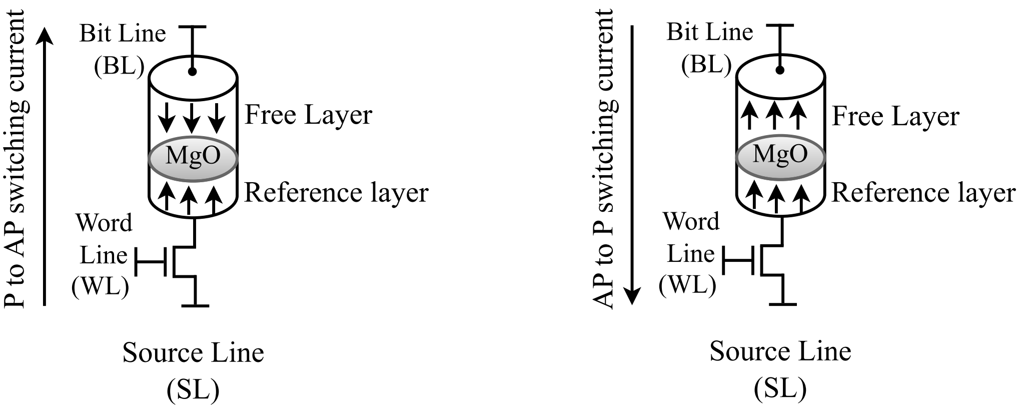
(a) MTJ

(b) Bit-cell

Figure 2.1.: STT-MRAM bit cell

Fig. 2.1(b) illustrates a 1T–1MTJ STT-MRAM bit cell, where the access transistor is connected in series with the MTJ, forming a three-terminal configuration. In this setup, the Bit Line (BL) and Source Line (SL) constitute the current path, while the Word Line (WL) governs the transistor gate to enable bit-cell selection. Fig. 2.2 depicts how the bit cell stores binary information: the high resistance state (HRS), corresponding to the anti-parallel ('AP') magnetization, represents binary '1', whereas the low resistance state (LRS), corresponding to the parallel ('P') magnetization, represents binary '0'.

To retrieve or read the stored value from the MTJ, a sense amplifier senses a low unidirectional current. The sensed current is compared with the reference current to get the information about the read data. If the sensed current exceeds the reference current, the MTJ is in the low resistance state (LRS, R_P); otherwise, it is in the high resistance state (HRS, R_{AP}). The accuracy of the sensing process relies on the relative resistance between 'P'-state and 'AP'-state, defined by a factor Tunneling Magnetoresistance (TMR), as described in Equation (2.1.1). Fig. 2.3 illustrates the resistance distribution of R_P and R_{AP} states of experimental measurement data for the 1 Mbit array. This experimental measurement setup considers the MTJ size of 60 *nm*, a pitch size of 200 *nm*, and a total of $1024 \times 1024 = 65,536$ cells. The resistance measurement is performed using a standard 4-point Kelvin measurement technique.



(a) 'AP' State: MTJ Storing '1' (HRS)

(b) 'P' State: MTJ Storing '0' (LRS)

Figure 2.2.: STT-MRAM bit-cell storing binary data

$$TMR = \frac{R_{AP} - R_P}{R_P} \times 100 \tag{2.1}$$

STT-MRAM is a type of NVM designed to retain stored data even when the device is in standby mode for a specific period, known as Retention Time (RT). The RT of STT-MRAM is primarily determined by the thermal stability factor (Δ), which quantifies the energy barrier (E_B) that the free layer magnetization has to overcome to switch its magnetic orientation. This factor is formally defined in Equation (2.1.1), where μ_0 is the vacuum permeability, M_s denotes saturation magnetization, K_B represents the Boltzmann constant, H_k indicates the magnetic anistropy field, V is volume of free layer, and T depicts temperature. The statistical RT model is widely used and is expressed in Equation (2.1.1), where τ_0 denotes the time constant, which is the inverse of attempt frequency (typically $1ns$), and P_{RT} is the switching probability of MTJ state after RT. A higher thermal stability factor (Δ) leads to a longer data RT. Typically, a data storage application requires longer retention (in years) compared to a cache memory application (*ms*-scale).

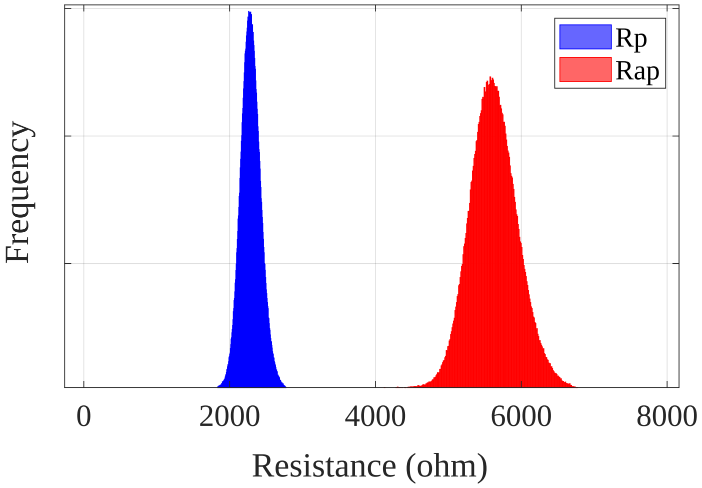


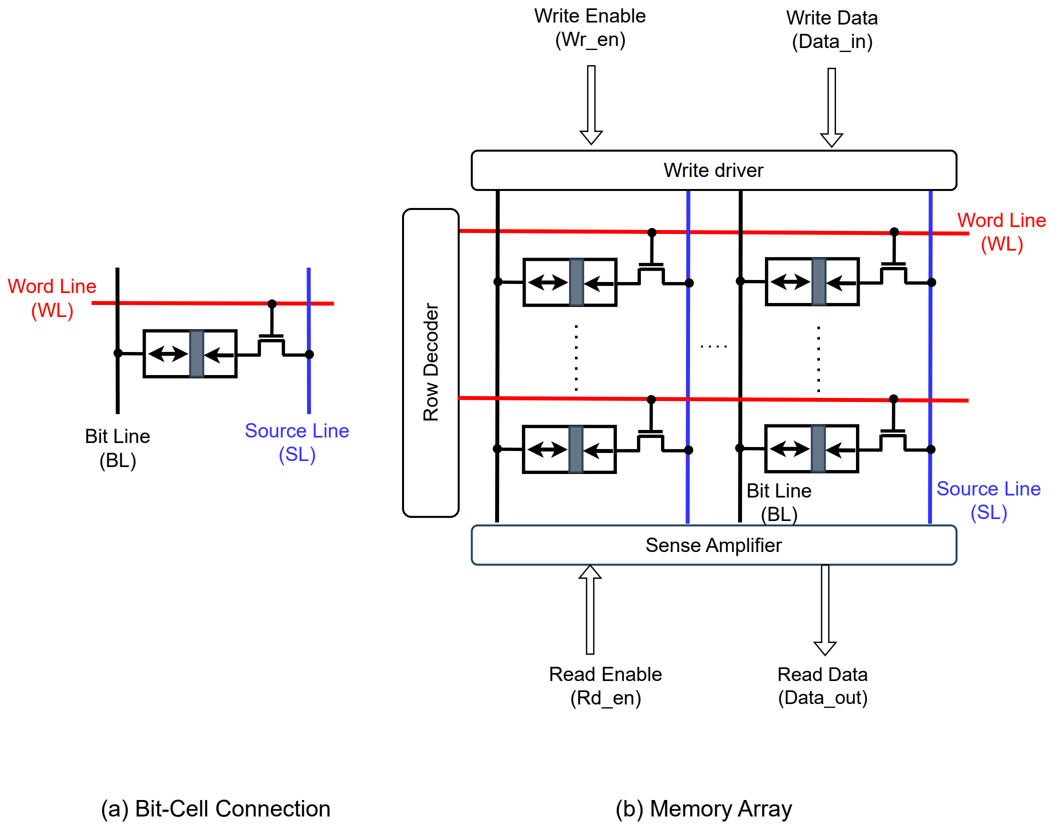
Figure 2.3.: Resistance distribution of R_P and R_{AP}. Array size: $1Mb$, MTJ size: $60nm$, Pitch size: $200nm$

$$\Delta = \frac{\mu_0 V H_k M_s}{2K_B T} \tag{2.2}$$

$$RT = \frac{\tau_0 \exp(\Delta)}{1 - P_{RT}} \tag{2.3}$$

Fig. 2.4 depicts a typical organization of a STT-MRAM memory array. It comprises the 1T-1MTJ (one access transistor and one magnetic tunnel junction) bit-cell and peripheral circuitry. Each bit cell in an array is connected to the CMOS peripheral components via three signal lines: WL, BL, and SL. The WL is used to control the access of the bit-cell, which is connected to the gate of the access transistor. For a given memory address, a specific WL is activated to access a particular row in a memory array. The remaining two terminals of the bit cell are connected to the BL and SL. The voltage signals on BL and SL are responsible for managing the write and read operations of the internal MTJ device. These operations are governed by the magnitude and polarity of the voltage applied across the MTJ via BL and SL.

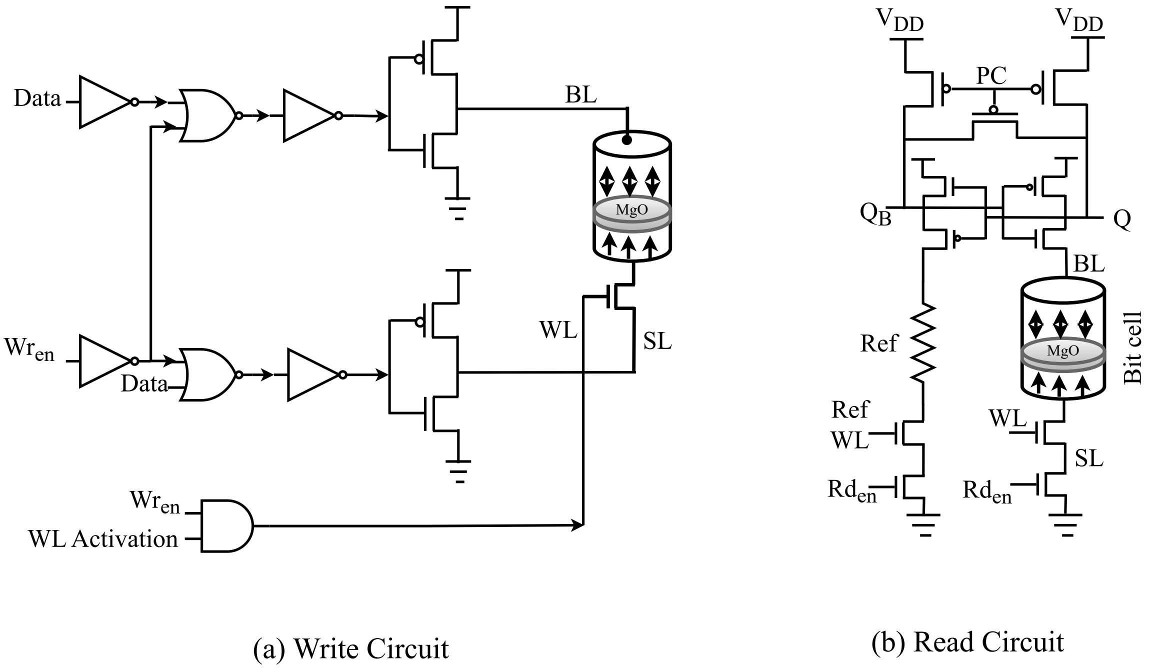
Typical write and read circuits for STT-MRAM are shown in Fig. 2.5 [29], [77], [78]. The write circuit consists of control logic paired with a write driver. When the write enable signal (Wr_{en}) is asserted high, the control logic allows the input data to pass through to the input port. During a write operation, the write driver generates the appropriate voltage polarity and amplitude across the BL and SL lines, producing a spin-polarized current that alters the state of the MTJ. For a read, a small bias is applied over the same path, and the resulting cell current is compared to a reference by a sense amplifier to determine the stored value. Fig. 2.5(b) shows a pre-charge sense amplifier to read the stored data. The read process is initiated when the read enable signal (Rd_{en}) is asserted high. The sense amplifier compares the current flowing through the target bit cell with that of a reference cell. The resistance of the reference bit cell is set to $Ref = \frac{R_{AP}+R_P}{2}$, representing the midpoint between the anti-parallel and parallel resistance states. If the cell current I_{cell} exceeds the reference current I_{ref}, the sense amplifier outputs logic '0' on the Q node; conversely, if $I_{cell}<I_{ref}$, it outputs a logic '1'. The read margin of STT-MRAM is constrained by limited TMR; robust sensing is essential. There also exist other sense-amplifier architectures that incorporate reference trimming and disturb-free reference cells to improve read operation [27], [28].



(a) Bit-Cell Connection

(b) Memory Array

Figure 2.4.: Typical STT-MRAM array structure



(a) Write Circuit

(b) Read Circuit

Figure 2.5.: Typical write and read circuits for STT-MRAM

2.2. Reliability challenges associated with STT-MRAM

2.2.1. Faults and Errors

A fault is an unwanted physical condition within a system that can lead to an error. An error is a deviation in the logical state of a data bit from the expected value. A failure occurs when such errors impact a system's functionality [79]. Faults are generally classified into two categories: transient and permanent. **Transient** faults are temporary disruptions caused by environmental factors like radiation, electromagnetic pulses, temperature variation, and voltage fluctuation. They lead to **soft** errors—temporary data corruption that persists until overwritten [80]–[82]. Advancements in CMOS technology, including higher transistor density and faster speeds, make systems more susceptible to soft errors [83]. It is difficult to detect the presence of soft errors during functional testing as they occur during runtime. ECCs are widely employed to recover corrupted data bits during runtime [84].

Permanent faults are typically caused by irreversible physical damage, such as manufacturing defects, extreme temperature fluctuations, radiation exposure, and process variations. At the storage or processing level, the permanent faults translate into **hard** errors, which persist over time [80], [82]. As device dimensions continue to shrink, hard error rates are likely to increase as effects such as electromigration and dielectric breakdown become more common. Hard errors are modeled by stuck-at faults (stuck-at-1 or stuck-at-0) and generally require bit-cell replacement. The location of most hard errors can be detected during the manufacturing testing process, and redundancy repair (i.e., extra rows or columns) is the most commonly used strategy in memory systems to address these errors [85], [86].

2.2.2. Soft errors in STT-MRAM

The STT-MRAM is also known to be susceptible to various sources of soft errors. Soft errors in STT-MRAM predominantly arise from the characteristics of the free layer, including parameters such as the thermal stability factor, as well as the stochastic and asymmetric nature of the bit-cell switching [29], [39]–[49].

2.2.2.1. Read disturb

In MTJ, the read and write currents share the same path within the bit-cell. This introduces the possibility of unintended switching of the bit-cell content when the read current is applied, resulting in a read disturb, which is quantified as read disturb probability (P_{RD}) and given by Equation (2.4). The variable τ represents an intrinsic attempt time constant equal to 1 ns. The Δ value represents the thermal stability factor, which defines the retention property of the bit-cell during standby mode. The Δ value is a function of the volume of the free layer, saturation magnetization, effective anisotropy field, and temperature. I_c is the critical current, corresponding to the minimum current required to change the state of the bit-cell, and I_r is the read current for t_r read duration. Additionally, the retention failure probability (P_{RF}) for a specific period (t) can be calculated using the P_{RD} with $I_r = 0$. Environmental factors during runtime, such as thermal fluctuations and stray magnetic fields, may also lead to resistance drift, meaning a change in the resistance values of R_P and R_{AP}. This resistance drift may result in both read and retention failures.

$$P_{RD} = 1 - exp[\frac{-t_r}{\tau.exp[\Delta(1 - \frac{I_r}{I_c})]}] \tag{2.4}$$

2.2.2.2. Write error

The switching behavior of STT-MRAM differs significantly from conventional CMOS-based memory technologies, as STT-MRAM relies on a magnetic switching mechanism that is inherently stochastic and asymmetric in nature, and some desired data may not be stored correctly on MTJ. This issue arises primarily due to two reasons: Firstly, the MTJ exhibits inherent asymmetric switching behavior, which is a function of the spin-transfer efficiency factors of the MTJ and the reduced current density resulting from the voltage degradation of the access transistor. The asymmetric switching is characterized by a higher likelihood of errors during transitions from P('0')→AP('1') than from AP('1')→P('0'). Secondly, the switching of MTJ magnetic orientation is stochastic due to random thermal fluctuations of magnetization. This implies that the switching delay of MTJ magnetization is not deterministic. If the write current is terminated before the MTJ completes its switching, resulting in a write error, it is given by Equation (2.5). Where t_w is the write latency, and C is a technology-dependent parameter, I is the ratio of the write current (I_w) to the critical current (I_c). At the device level, the critical current is determined by technology parameters, while at the circuit level, it is defined by the switching time.

$$WER_{bit}(t_w) = 1 - exp[\frac{-\pi^2.(I-1).\Delta}{4(I.exp[C(I-1)t_w]-1)}] \tag{2.5}$$

2.2.3. Hard errors in STT-MRAM

Hard errors in STT-MRAM primarily arise from the characteristics of the thin oxide barrier sandwiched between two ferromagnetic layers, such as its thickness (t_{ox}) and TMR ratio. The hard errors are referred to as permanent faults, which cause the STT-MRAM cell to be stuck-at-0 or stuck-at-1, irrespective of the applied switching current [29], [40], [48]–[52].

Achieving high switching speed in STT-MRAM to meet the demands of modern computing systems requires a high current density, as defined by Equation (2.6). This can be accomplished in two ways: first, by increasing the bias voltage (V), and second, by reducing the resistance-area product (RA). Additionally, improving switching speed often necessitates thinning the oxide barrier. However, these modifications may lead to the breakdown of the oxide barrier, resulting in hard errors.

The preference for a thin oxide barrier also introduces significant variation. The MTJ resistance exhibits an exponential relationship with the oxide barrier thickness. The variation in oxide barrier thickness may exceed the adequate TMR ratio, which can disrupt sensing operations and lead to hard errors. Additionally, applying bipolar stress (AC) may reduce the lifetime of the STT-MRAM cell and increase the risk of breakdown in the oxide barrier.

$$J = \frac{V}{RA} \tag{2.6}$$

Hard error in STT-MRAM also arises from a rough oxide barrier, known as pinhole defects. These occur when CoFeB particles fill the concave of the rough oxide barrier during the deposition process, forming a leakage path between two ferromagnetic layers and potentially leading to oxide breakdown, as shown in Fig. 2.6. The presence of pinholes shunts the current path, which degrades the TMR and can ultimately lead to oxide breakdown. Furthermore, significant variations in the manufacturing process may also lead to permanent faults in STT-MRAM. This is because critical parameters of the MTJ, including magnetic anisotropy, oxide barrier, TMR, and saturation magnetization, can be adversely impacted.

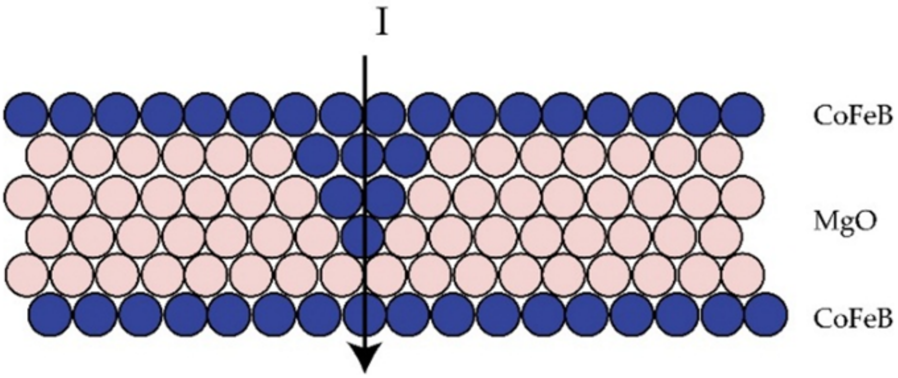
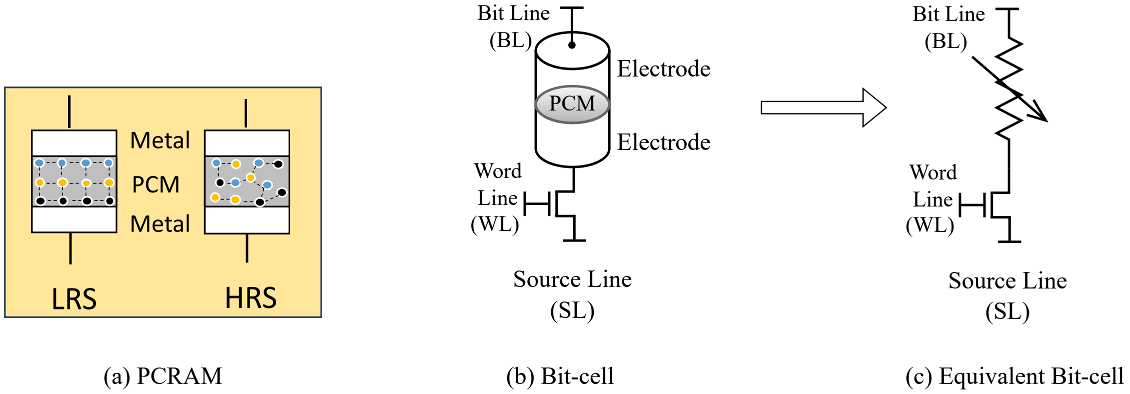


Figure 2.6.: Demonstration of pinhole generation in STT-MRAM (Reprinted from [40])

2.3. PCRAM

PCRAM is also a non-volatile storage technology often grouped among resistive NVMs that stores data as HRS/LRS states. A PCRAM storage element consists of a volume of Phase-Change Material (PCM), sandwiched between two electrodes, as shown in Fig.2.7. Data are encoded in the significant resistance contrast between the PCM's crystalline (LRS, high-conductivity) and amorphous (HRS, low-conductivity) phases. This phase change is accompanied by pronounced shifts in electrical and optical properties of PCM: the amorphous phase exhibits high resistivity and low optical reflectivity, while the crystalline phase shows low resistivity and high reflectivity. To switch between these phases, an adequate electrical current pulse is applied, also known as the RESET and SET operations. The applied switching electrical pulses produce Joule heating, which drives phase transitions. A SET operation (HRS to LRS switching) can be initiated by applying a longer, lower-amplitude pulse that raises the active region below the melting point but above the crystallization range. Similarly, for LRS to HRS, i.e., the RESET operation, a short, high-amplitude pulse is used to heat the material above its melting temperature and then rapidly quench it, thereby freezing an amorphous phase. The transition from amorphous to crystalline phase (i.e., a set process) determines PCRAM performance (speed). PCRAM performance depends critically on the properties of PCM. A memory read operation involves measuring the device's electrical resistance to determine its state: high resistance indicates the amorphous phase (logical '0'), while low resistance indicates the crystalline phase (logical '1'). Furthermore, PCRAM can also offer the benefit of supporting multi-bit storage per cell, commonly known as multi-level memory. [87]–[91].



(a) PCRAM (b) Bit-cell (c) Equivalent Bit-cell

Figure 2.7.: PCRAM technology

PCRAM offers a promising memory solution; however, it also faces reliability challenges. The reliability issue may be more pronounced in the case of a multi-level cell, as adjacent levels have a narrow resistance window [92]–[96]. Table 2.1 categorized different reliability issues associated with PCRAM and their impact on bit cell [92]. Most reliability concerns in PCRAM trace back to the metastability of the amorphous state. This can evolve over time through crystallization and structural relaxation. Other reliability challenges include cycling endurance, programming disturbance, read disturbance, and retention failure. Repeated SET/RESET cycling leads to gradual drifts in the SET and RESET states, ultimately resulting in stuck-at faults. Program disturb occurs due to thermal interference in the array caused by unintentional heat spreading induced by programmed cells. Read disturb can result in unwanted switching and resistance degradation in the reset state. Data retention is also another limitation, where the RESET state's retention can degrade at elevated temperatures.

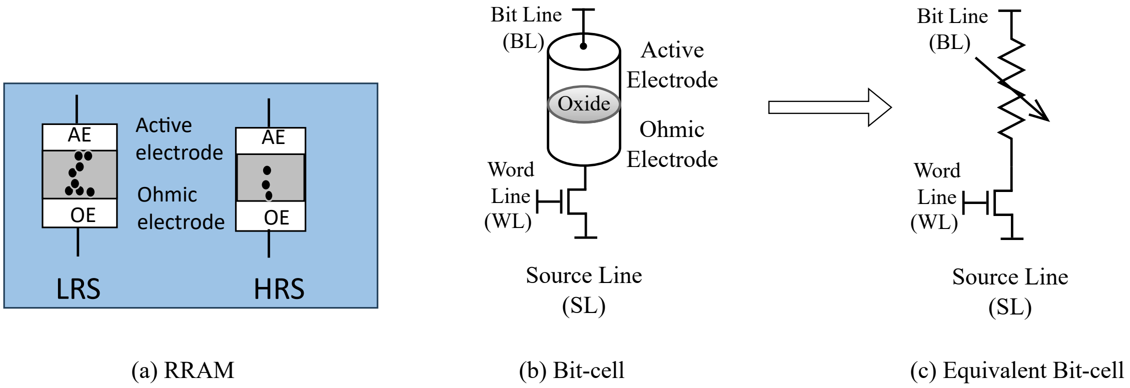
Table 2.1.: Reliability issues for PCRAM cell [92]

Reliability Issue	**Impact on cell**
Crystallization	Resistance decrease
Structural relaxation	Resistance increase
Cycling endurance	Stuck set/reset
Program disturb	Resistance decrease/increase
Read disturb	Switching and resistance decrease

2.4. RRAM

RRAM is also one of the leading candidates among emerging NVM technologies, showing potential as an alternative to traditional memory and computing applications. Its operation is commonly attributed to the formation and partial rupture of nanoscale conductive filaments within an insulating oxide, enabling reversible switching between LRS and HRS. Fig. 2.8 shows the basic structure of RRAM technology, which represents the metal-insulator-metal stack, where the oxide layer is sandwiched between an active electrode and an ohmic electrode. In RRAM, under an applied bias, oxygen vacancies drift and local oxidation/reduction reactions (redox reaction) alter the oxide stoichiometry, which creates, reshapes, or ruptures a conductive filament between the electrodes. A SET pulse forms the conductive filament and places the cell in the LRS; a RESET pulse ruptures the filament and returns the cell to the HRS. These two resistance states are used to encode logic values. RRAM can operate in two polarity-dependent switching modes: unipolar and bipolar. In unipolar switching, both SET and RESET are executed with the same voltage polarity, differing only in amplitude and/or pulse duration. In bipolar switching, voltage pulses of opposite polarity are required: one polarity drives SET, and the opposite drives RESET [97]–[101].

Furthermore, RRAM is well suited for CiM architectures that accelerate NN workloads. Its ability to realize multi-level conductance/resistance states allows model weights to be stored *in situ* and mapped directly to crossbar conductances; the same array then performs the required MVM, reducing data movement and improving energy–latency efficiency [102], [103]. However, RRAM devices also exhibit certain reliability challenges—device non-idealities, device-to-device and cycle-to-cycle variability, resistance drift, and RTN—which can lead to soft errors (transient faults) and hard errors (permanent faults). Device-to-device variability involves stochastic differences between individual cells, cycle-to-cycle variability pertains to changes from one switching cycle to another, and RTN noise is related to read instability observed in RRAM, which represents the variation of current amplitude during reading, which typically exhibits stochastic changes [97], [98], [104]–[107].



(a) RRAM

(b) Bit-cell

(c) Equivalent Bit-cell

Figure 2.8.: RRAM technology

2.5. NNs and hardware accelerators

2.5.1. Basics of NN computation

The concept of NN is inspired by the biological NNs in the human brain, enabling computers to solve problems in ways that mimic human thought processes. An NN is a computing graph in which nodes represent neurons, and connections are weights, as shown in Fig. 2.9. Mathematically, NN can be defined as a function $\mathcal{F}_\theta$ with learnable parameters θ (weights) that map a given input vector X to an output vector $\hat{Y}$. During NN training, given a dataset for a task $\mathcal{D}_{train} \subset (X_{train}, Y)$, the parameters θ are learned with a learning algorithm by minimizing the loss $\mathcal{L}$ [108]. Delving deeper, the advent of Deep Neural Network (DNN) signifies an extension of NN architectures to encompass more than just a few layers to perform more complex tasks. In the domain of NN computation, there exist four principal topologies: spiking neural networks (SNNs), which aim to closely mimic the actual biological processes of the brain; fully connected networks, where each neuron is connected to every neuron in the adjacent layer; Convolutional Neural Network (CNN), renowned for their prowess in processing data with a grid-like topology, such as images; and recurrent neural networks (RNNs), which excel in handling in time-series or sequential data due to their inherent memory of previous inputs [108]–[111].

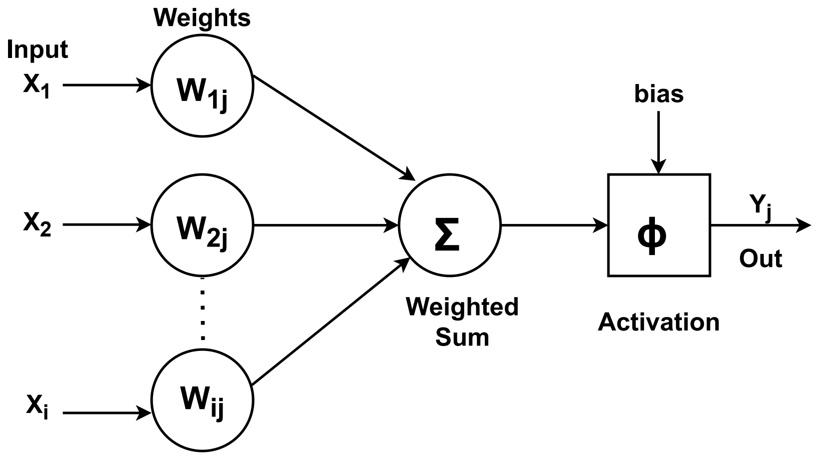


Figure 2.9.: Basic structure of NN computation

2.5.2. NN hardware accelerators

Deep learning algorithms have become increasingly important in various fields, ranging from mobile devices to data center operations. These algorithms involve numerous MVM operations on large datasets. However, a major challenge in efficiently deploying NN algorithms on traditional computing architectures is the bottleneck caused by data transfer between the processor and memory. This bottleneck, also commonly referred to as the memory wall problem, can significantly impact both the performance and energy efficiency of computing systems [10]–[12].

To address the memory wall bottleneck inherent in traditional von Neumann computing architectures and enhance the performance of NN computations, significant research efforts have focused on deploying NNs on various specialized silicon hardware platforms. Hardware platforms such as FPGA, Digital ASIC, GPU, and TPU have been utilized to enhance NN performance in the domain of digital hardware accelerators [112]–[125]. These hardware accelerators have been engineered to optimize the execution of deep learning algorithms, improving computational efficiency. Alongside digital approaches, the evolution of analog and mixed-signal (AMS)-based NN accelerators is also promising. These solutions have the potential to increase computational efficiency, which makes them particularly suitable for edge computing scenarios [126]–[132]. The AMS-based NN accelerator domain has seen significant advancements, particularly with the introduction of memristive crossbar-based CiM architectures, where MVM is executed within the memory [128]–[132].

A GPU, also known as a graphics processor, display core, or graphics chip, is a specialized microprocessor designed for graphics workloads in computers, workstations, game consoles, and mobile phones. Thanks to its massively parallel architecture and high memory bandwidth, the GPU has become a popular accelerator for deep learning, where developers leverage its data-parallel execution model to accelerate training and inference. A hardware accelerator based on GPU [117], [119], [120] leverages parallelism enabled by numerous cores within its architecture, as shown in Fig. 2.10, resulting in substantial computational speedup. GPU harnesses extensive data-level parallelism in applications using the Single Instruction Multiple Thread (SIMT) execution model. The decision to integrate GPU largely depends on the expected benefits associated with increased memory bandwidth, larger dataset sizes, and the optimization of long-running tasks.

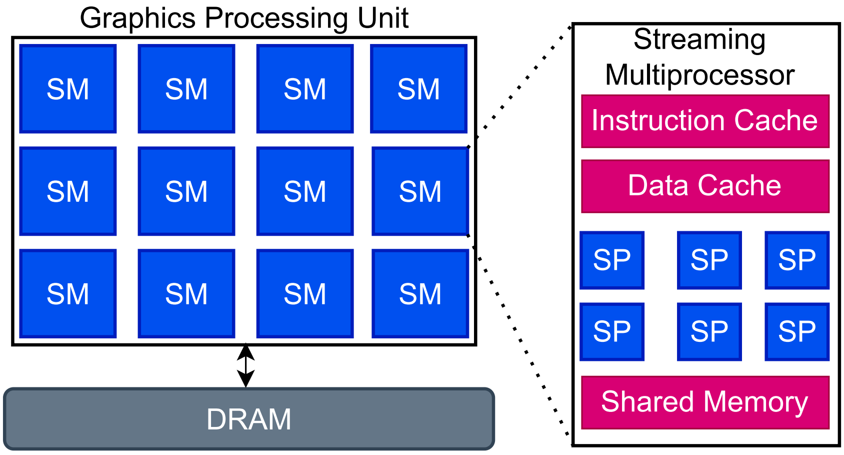


Figure 2.10.: Overview of GPU architecture. SP: Scalar processor with private and on-chip shared memory

TPUs [117], [120], [121] are specifically designed as matrix processors tailored to efficiently handle NN tasks. They are equipped with thousands of MAC units, or Processing Element (PE)s, interconnected to form an extensive physical matrix, as shown in Fig. 2.11. Each PE can have a single or multiple cores and executes computations in a Single Instruction Multiple Data (SIMD) fashion. Data are exchanged between off-chip memory and PEs through an on-chip controller. The controller loads activations and parameters into the on-chip staging buffers and also fetches the low-level instructions executed by the PEs. Each PE can consist of one or more cores, with each core featuring multiple compute lanes to enable SIMD-style parallel execution. The compute lanes in these cores feature MAC units, enabling efficient computation between activations and model parameters, which is a core operation of NN computation.

FPGA-based accelerators [117], [123] can serve as a viable alternative to ASIC-based accelerators, offering superior performance at an affordable cost with reconfigurability, low power dissipation, fast iteration, and low non-recurring engineering costs. They support parallelism and speed up computations by mapping them to parallel hardware. FPGA-based accelerators can provide several orders of magnitude speed-up compared to baseline CPUs. FPGAs enable designers to implement only the necessary logic in hardware, based on the target application. The architecture of FPGA-based DNN accelerators mainly consists of a host computer and an FPGA part to implement NN algorithms.

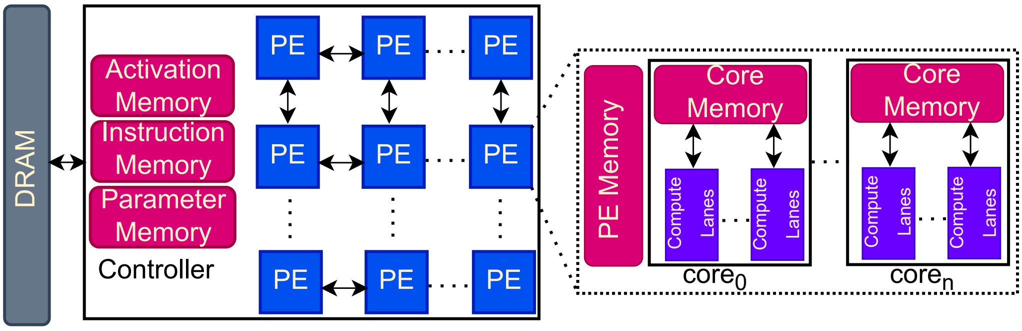


Figure 2.11.: Overview of Edge TPU architecture [121]

In the case of an AMS-based accelerator, the memristive crossbar-based CiM architecture is gaining significant attention due to its efficiency in performing matrix-vector multiplication (MVM) [129]–[132]. The CiM architecture can also be realized using CMOS memories, such as SRAM, with modest modifications to the bit cell structure and periphery. This might be an attractive option at an advanced technology node. However, the SRAM-based CiM architecture would suffer from leakage power, which can be significant and thus undesirable for edge inference that spends substantial time in standby [103]. Consequently, resistive NVM technologies, such as RRAM, PCRAM, and STT-MRAM, are at the forefront of developments for the realization of CiM architecture, also known as memristive crossbars [133]–[141]. In particular, RRAM is widely used for CiM realization to accelerate NN computation, and it can also enable multi-bit storage per bit cell [103], [133]–[137]. Fig. 2.12 shows the basic structure of the memristive crossbar array used in CiM architecture. The memristive crossbar, formed by an array of bit cells, is utilized as an accelerator module for MAC operations in an analog manner. The bit cell is used to store the weight value in the form of device conductance/resistance. The BL represents the summation of the multiplication of the input vector (V) and the conductance matrix (G), i.e., $I_N = \sum_{i=1}^{M} V_i G_{iN}$. A Digital-to-Analog (DAC) converter and Analog-to-Digital (ADC) are required to maintain the interaction between the analog and digital data flows. Moreover, the writing circuitry, address decoder, and controller are the other periphery circuits needed for the functionality of the MAC accelerator.

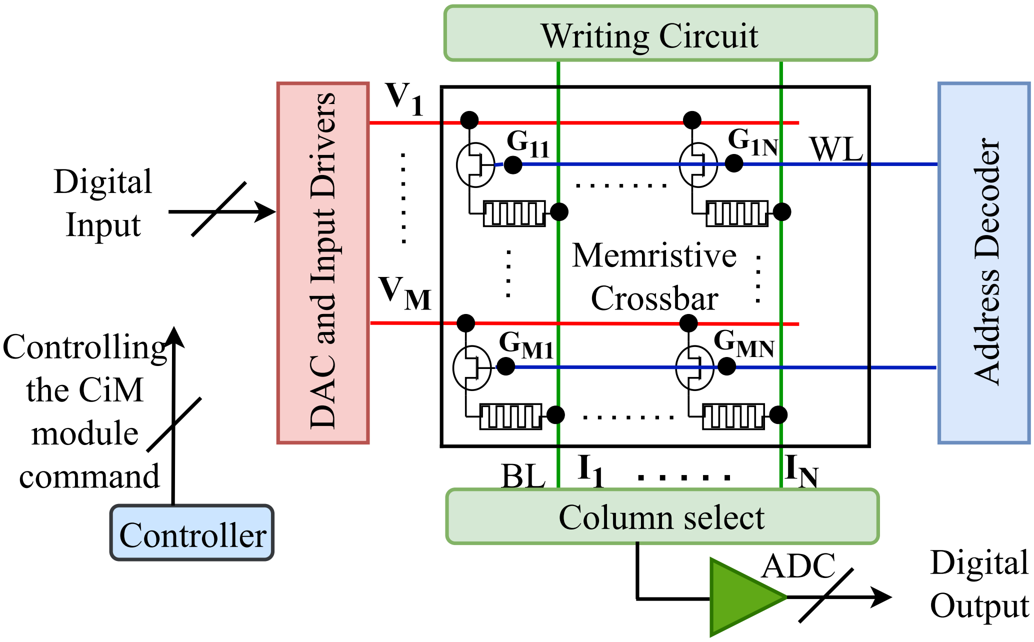


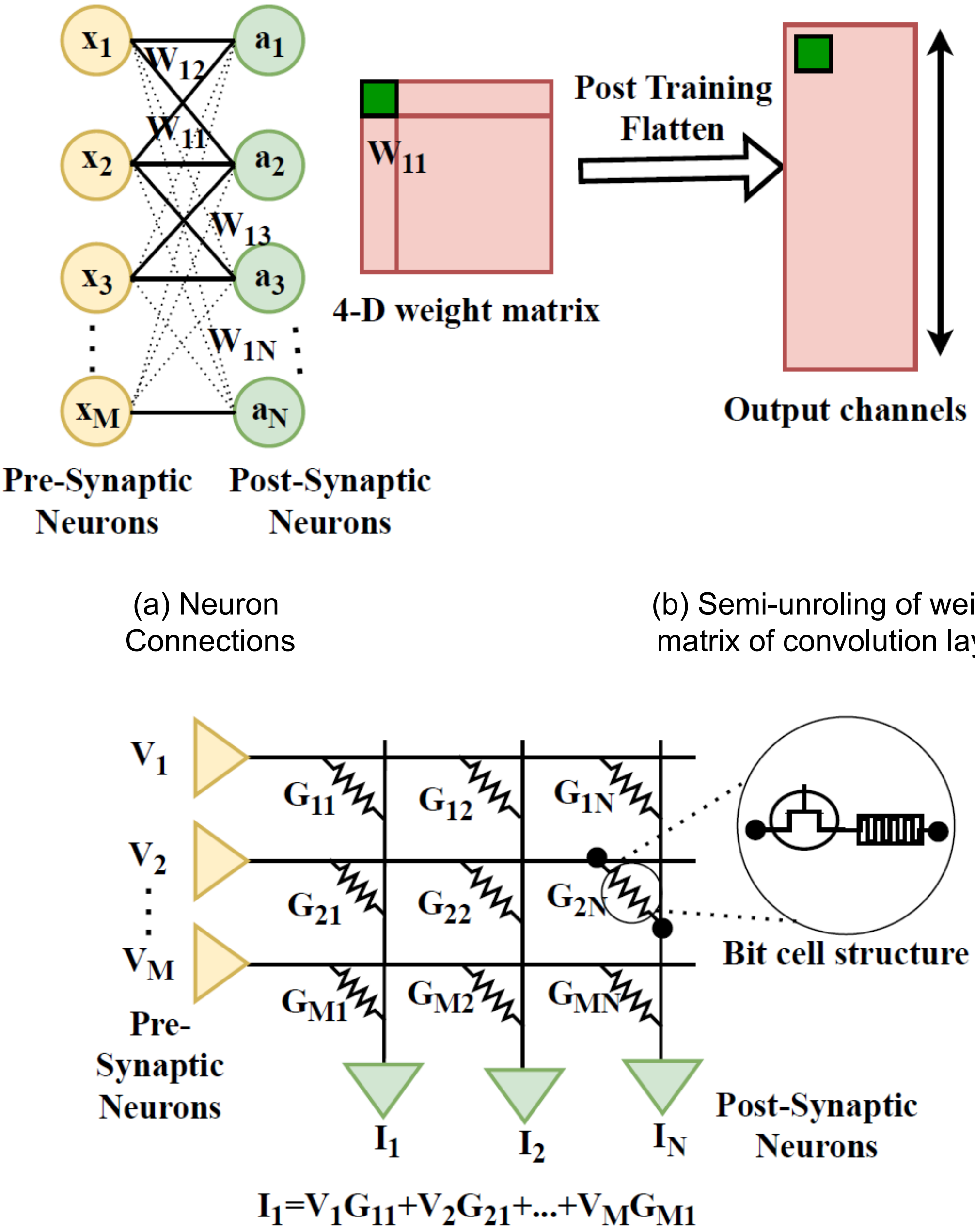
Figure 2.12.: Memristive crossbar-based CiM Architecture. The resistive NVMs are also referred to as memristive devices

In a memristive crossbar architecture, the number of bits in the output produced by each BL during MVM operation depends on the several factors: the number of rows in the crossbar (M), the input bits (V_b), and the resolution of the memory device (b). This configuration can be used to decide the resolution of the ADC. The relationship can be expressed by Equation (2.7), where R represents the output resolution. The expression shown in the Equation (2.7) is valid when the values of V_b and b are greater than 1; otherwise, the Equation (2.8) needs to be used to get the output resolution [142]. Here, when not all the rows are activated at a time, the value of M should be equal to the number of activated rows at a time.

$$R = \log(M) + V_b + b \tag{2.7}$$

$$R = \log(M) + V_b + b - 1 \tag{2.8}$$

The NN inference can be made using memristive crossbar-based CiM, in which synaptic weight can be stored into these crossbar arrays, which efficiently perform the required expensive MVM in the memory array, as shown in Fig. 2.13 (c). A one-time write operation is required to map the trained weights of the NN to the crossbar array. Fig. 2.13 (a) represents the single layer of a NN. Each connection of the two-layer represents the synaptic weight value, which is represented by the conductance values (G) of memristive cells in a memristive crossbar structure. The conductance (resistance) values (G) of memristive cells represent the synaptic weights (W) of NN. The weights of fully connected layers are 2-D shaped so that they can be mapped directly to the crossbar array. In the case of convolution layers, a 4-D weight matrix is flattened to a 2-D weight matrix by semi-unrolling. This flattened weight matrix is mapped to the crossbar as shown in Fig. 2.13 (b). The peripheral circuit performs batch normalization and non-linear activation to get the activation for the next layer [143].



(a) Neuron Connections

(b) Semi-unroling of weight matrix of convolution layer

(c) Memristive crossbar array for MVM

Figure 2.13.: Synaptic weight mapping to the memristive crossbar array

2.5.3. Impacts of faults (or errors) on NN performance

NNs contain millions of weight parameters that must be stored in the memory system available on the deployed hardware accelerator. These memory solutions span from CMOS memories to a variety of cutting-edge NVMs [144]–[156]. However, these memory devices are susceptible to different faults that can lead to both soft and hard errors. Such errors can alter the stored weights, resulting in computational inaccuracies and potentially causing functional task failures [157]–[161]. This interconnection is conceptually illustrated in Fig. 2.14, which showcases a NN assigned with computational roles and deployed on a hardware platform.

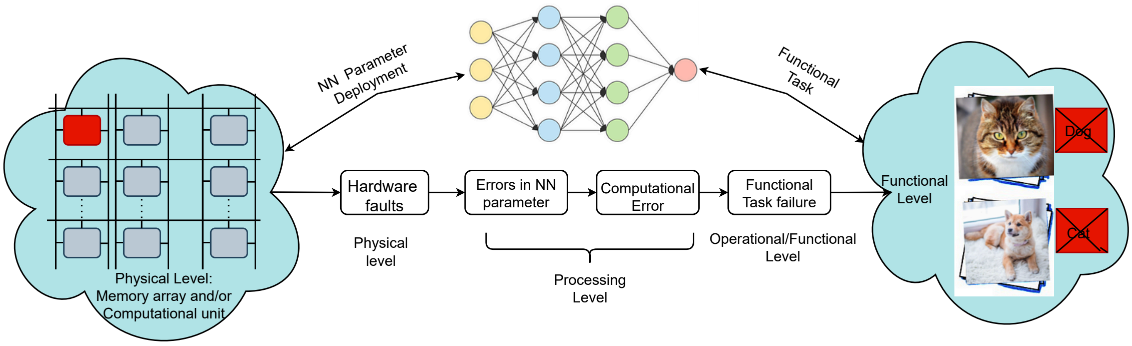


Figure 2.14.: Overview of the relationship between fault and its impact on NN functional task

In the domain of NN hardware accelerators, numerous faults often have minimal impact on performance. They are either masked before they can affect the desired output or result in acceptable minor changes that do not compromise the overall NN performance. This robustness can be attributed to several key factors, like the inherent sparsity of NN, over-provisioning, the distributed nature of computing architecture, and the specific nature and sequence of mathematical computations that are executed. However, some faults are significant and detrimental to system performance, which can be crucial in safety-critical applications. Fig. 2.15 represents the overview of fault criticality classification [157]. It is essential to evaluate how faults propagate through NN computation and affect the performance of NN hardware accelerators. This facilitates early reliability analysis and assists in developing practical and economical strategies for fault detection, fault tolerance, and fault repair [159], [160].

For assessing fault tolerance, it is preferred to use a fault injection technique to understand how a system behaves under different error conditions. This is particularly useful for identifying critical components that may need protection against specific faults in their physical implementation. The fault injection in an NN hardware accelerator can be performed at different levels of abstraction, such as software model, RTL level, micro-architecture level, transistor level, gate level, etc. [162]–[167]. By introducing faults into an NN model on a probabilistic basis, researchers can evaluate the extent of the failure and its impact on the neural computations being performed. Recent studies predominantly focus on bit-flips and/or stuck-at-faults occurring in memory cells and registers that store NN parameters, such as synapse weights and neuron activations [159]–[161], [168]–[170]. Bit flips can be introduced at a specific bit error rate, which represents the probability of faults occurring. This rate can also be represented as a fault rate, which is the percentage of faulty bits. Similarly, stuck-at faults can also be modeled as a permanent bit-flip fault model, rather than temporarily flipping the bits. The comprehensive study on the different fault modeling and fault injection experiments can be found in the survey work [157], [158].

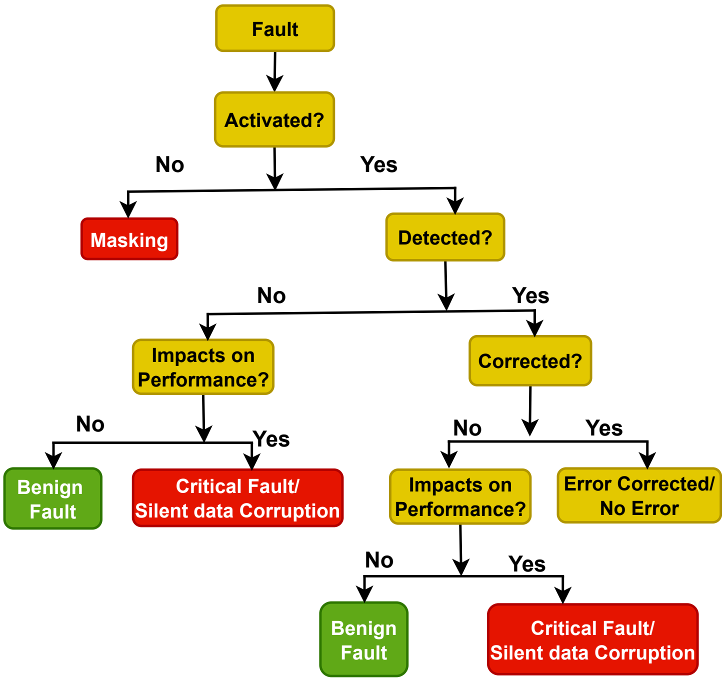


Figure 2.15.: Overview of fault criticality classification [157]

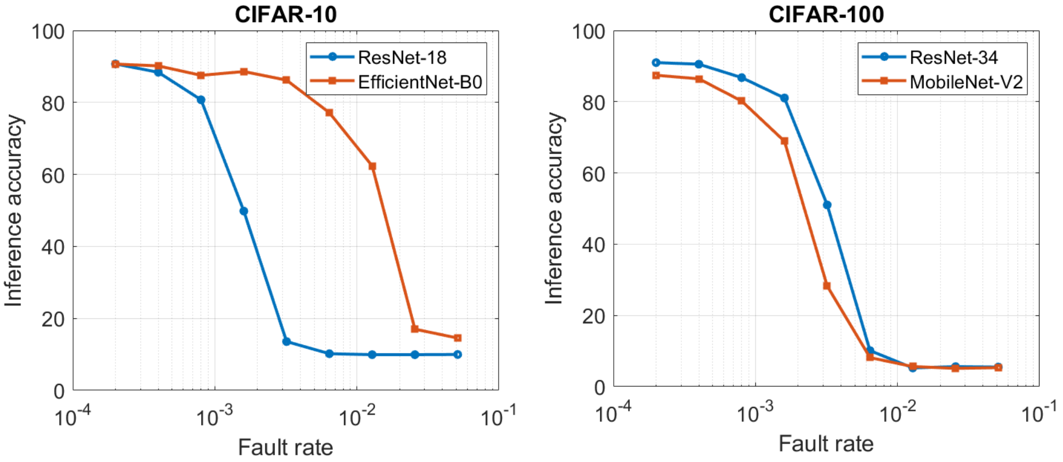


Figure 2.16.: Impact on NN baseline accuracy as a function of different fault rates using the bit-flip fault model

Figure 2.16 shows the impacts of faults on NN baseline accuracy (mean of 100 Monte Carlo simulations) for different NN models under different fault rate configurations for CIFAR-10 and CIFAR-100 datasets. The weight parameter is an 8-bit quantized weight case. In the context of fault injection, bit-flip faults are introduced into the weights of NN models by randomly sampling from a uniform distribution. Here, the fault rate indicates the percentage of faulty bits injected to assess the reliability of the NN. It can be observed that as the fault rate increases, there is a significant drop in accuracy. We have performed 100 Monte Carlo simulations and reported the mean accuracy for different NN models. Additionally, it is evident from Figure 2.16 that different NN models exhibit varying degrees of inherent fault tolerance, which is influenced by their distinct architectural designs.

2.6. Overview of fault-tolerance

To enhance the reliability (continuity of correct service) of systems, fault-tolerant techniques are commonly employed. A fault-tolerant strategy integrates reliability assurance by incorporating testing, diagnosis, and redundancy into a system's design and operation. A widely accepted definition of a fault-tolerant computing system is one that can continue to perform its functions correctly, without needing external intervention, even in the presence of specific faults. In other words, fault tolerance is aimed at failure avoidance and is carried out via error detection and system recovery. Table 2.6 outlines the techniques involved in fault tolerance. In general, redundancy techniques are extensively used to safeguard against operational faults and can be implemented at various levels, including hardware, software, temporal, and informational [79], [171]. Hardware redundancy involves adding extra components to enhance fault tolerance and can be divided into two types: static and dynamic redundancy. Static redundancy masks (or corrects) the errors. Dynamic redundancy handles faults after they manifest, and it is implemented via a two-step process: first, detection of the fault, followed by recovery. Hardware redundancy methods such as N-modular redundancy, redundancy repair, and ECC are primarily used to protect memory and computer systems at the hardware level. Software redundancy includes additional programs, routines, or instructions that would not be necessary in a fault-free system. Time redundancy refers to the practice of re-executing or validating operations at various levels of detail, including micro-operations, single instructions, code segments, or entire programs, to identify and correct errors [171]. In this thesis, we focus explicitly on hardware-level redundancy, particularly the ECC-based strategy [84], which can detect and correct errors at runtime by utilizing additional redundancy, as discussed next.

Fault Tolerant Strategies	
Error Detection	Detects the presence of an error
Concurrent Detection	During normal service delivery
Preemptive Detection	While normal service is suspended
Error Recovery	Transforms a system state with errors/faults into a state without detected errors and without faults that can be activated again
Error Handling	Eliminates errors from the system state
Rollback	Brings the system back to a saved state before the error occurred
Rollforward	State without detected error is new state
Compensation	Uses redundancy to mask or correct errors
Fault Handling	Prevents faults from being activated again
Diagnosis	Identifies and records the cause of errors
Isolation	Performs physical or logical exclusion of the faulty components from further participation
Reconfiguration	Switches to spare component or reassigns tasks among non-failed components
Reinitialization	Checks, updates and records the new configuration

Table 2.2.: Different abstractions of fault-tolerant techniques (adopted from [79])

2.7. Error-Correcting Codes (ECCs)

2.7.1. Basics of ECCs

Low error rates are essential for achieving high manufacturing yield and density in memory systems. ECCs are recognized as an effective fault-tolerant technique against different faults (or errors) in memory systems. ECC employs finite field arithmetic to facilitate error detection and correction in information bits. Linear block codes are a class of ECCs that operate on blocks of data. The linear block code can be defined as a (n, k) code, where k is the length of data bits (u), n is the length of the codeword (c), and $p = n - k$ is the number of redundant parity bits [84], [172], [173]. The basic architecture of an ECC is illustrated in Fig. 2.17(a). In the ECC architecture, the encoding process is performed by converting k-bit data (u) into an n-bit codeword (c) by multiplying the k-bit data (u) by a generator matrix (G) using modulo-2 arithmetic, as shown in Equation (2.9). The resulting n-bit codeword is stored in a memory system. In this way, ECC uses redundant parity bits in each word line of memory systems.

$$[c]_{1\times n} = [u]_{1\times k} \times [G]_{k\times n} \tag{2.9}$$

For ECC decoding process, the received n-bit vector (r) is processed using a parity-check matrix (H), as shown in Equation (2.10). This matrix operation generates a syndrome vector, which is used to detect, locate, and correct errors. In typical memory systems, the process of decoding is carried out individually for each word line as it is accessed sequentially. The strength of an ECC is quantified by its minimum Hamming distance (d_{min}), which directly influences its error-correcting capability. Hamming distance indicates the number of different bits between any two valid codewords in a set of 2^k valid codewords. A linear block code with minimum Hamming distance d_{min} guarantees correcting all the error patterns of $t = \lfloor\frac{(d_{min}-1)}{2}\rfloor$ or fewer errors [172].

$$[s]_{1\times n-k} = [r]_{1\times n} \times [H]^T_{n-k\times n} \tag{2.10}$$

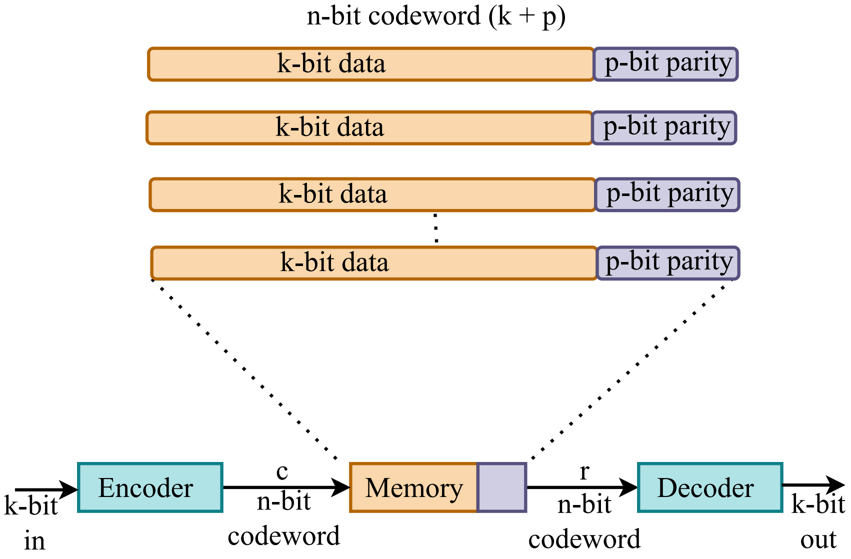


Figure 2.17.: Basic block diagram of ECC structure

2.7.2. SEC/SEC-DED ECCs

Hamming code is the first class of linear block codes devised for Single Error Correction (SEC). For any positive integer $p \geq 3$, there exists a (n, k) Hamming code with a data block size of $k = 2^p - p - 1$, code length, $n = 2^p - 1$ and the number of parity bits, $p = n - k$, where k is the data bits [84], [172], [173]. The Hamming code can also be used to detect double errors by extending one extra parity bit, which is referred to as Single Error Correction-Double Error Detection (SEC-DED) code. For example, the (71,64) Hamming code can be used as a SEC-DED code by extending a single parity bit, which results in a new code dimension of (72,64). SEC-DED is widely used to increase computer memory reliability in most real-world applications. There also exist other SEC-DED codes, which are a class of optimal minimum odd-weight-column [174].

2.7.3. Multi-bit ECCs

Bode-Chaudhuri-Hocquenghem (BCH) codes are renowned for their proficiency in multi-bit error correction scenarios, especially those involving multi-bit errors [172], [175]. For any positive integers $m \geq 3$ and $t < 2^{m-1}$, there exists a binary BCH code with codeword length $n = 2^m - 1$, number of parity $p \leq mt$ and $d_{min} \geq 2t + 1$. BCH codes are expensive in terms of memory overhead compared to SEC-DED. The required number of parity bits (p-bits) for t error correcting BCH code can be roughly estimated by $p = t \cdot \lceil \log_2 (k + 1) \rceil$.

BCH codes can also be extended by adding 1-bit parity for enhanced 1-bit error detection capabilities. The storage overhead for the SEC, SEC-DED, and t error correcting $t + 1$ error detecting BCH code for different memory blocks is represented in Fig. 2.18. Additionally, the decoding procedure of the BCH code is computationally expensive in terms of hardware overhead, specifically in terms of decoding area and latency overhead, as it requires multiple cycles to perform the decoding process to obtain the corrected data bits [172], [175], [176]. There are also parallel error-correcting designs of multi-bit BCH codes, which utilize parallel combinational logic to accelerate the decoding process [177].

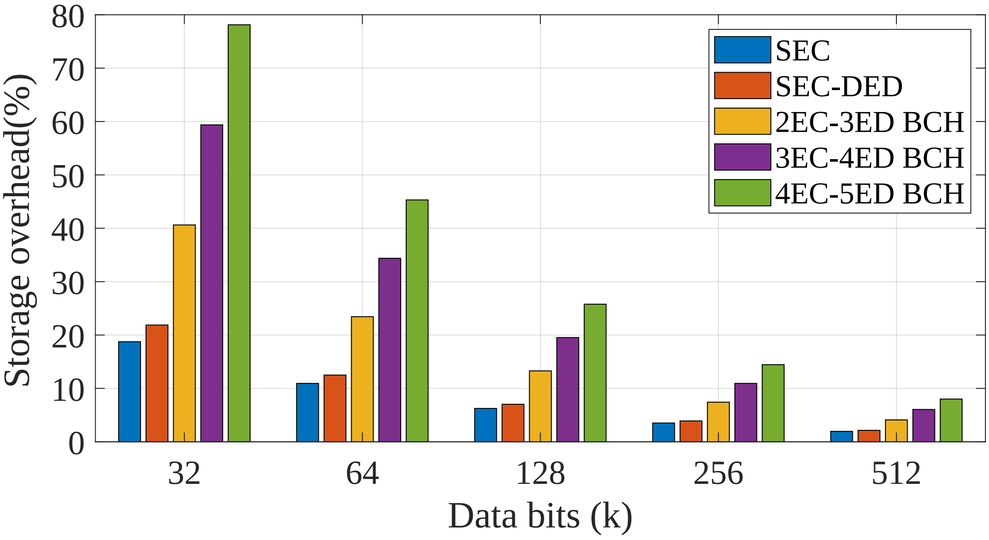


Figure 2.18.: Storage overhead for single and multi-bit ECCs

2.7.4. Block failure probability

An ECC strategy for correcting t errors in k-bit data can be represented as (n, k, t), where n is the codeword size, also known as word (or block) size after the ECC encoding process. If the failure probability at the bit level, known as the bit error rate, is denoted as P_b, then the failure probability of a memory block, also known as the word error rate or Block Error Rate (BER), P_{BER}, can be expressed as follows:

$$P_{BER} = 1 - \sum_{i=0}^{t} \binom{n}{i} P_b^i (1 - P_b^{n-i}) \tag{2.11}$$

2.8. Algorithm-based Fault Tolerance (ABFT)

ABFT was introduced to perform fault-tolerant matrix computation [178]. ABFT encodes matrices using checksums, representing the sums of each row or column as shown in Fig. 2.19. This technique involves performing computations on both the original and encoded data, resulting in an output matrix that contains the computed results and checksums. By comparing the row and column sums with the corresponding checksums in the output matrix, faults can be located at the intersection of the row and column where the checksums are incorrect. Additionally, when processing matrices with large integer elements, the same data length (k) can be used to store elements of the summation vector if residue arithmetic is applied, with the modulus (2^k) applied to the summation vector. Further, the work in [179] extended the concept of ABFT with the weighted checksum technique to perform error detection and correction in matrix computation. The technique combines non-weighted and weighted checksums to extract the signature for error correction, as shown in Fig. 2.20. In the case of the weighted checksum, a linear or exponential weight factor is applied, correlated with either the row or the column number. For example, the checksums computed during encoding are denoted as A_{W_c} and A_{NW_c} for weighted and non-weighted checksums, respectively. The error correction signature is computed as the ratio of the weighted deviation to the non-weighted deviation, expressed as $A_{faulty} = S_2/S_1$, where $S_1 = A_{W_c} - \sum_{i=1}^{N} a_i$ and $S_2 = A_{NW_c} - \sum_{j=1}^{N} W_{f_i} a_i$. The correction can be done by subtracting the S_1 from the faulty output whose column/row address is indicated by A_{faulty}. The weight factor W_{f_i} can be defined as linear: $W_{f_i} = 1, 2, 3, ...N$ or exponential: $W_{f_i} = 2^1, 2^2, 2^3, ...2^N$.

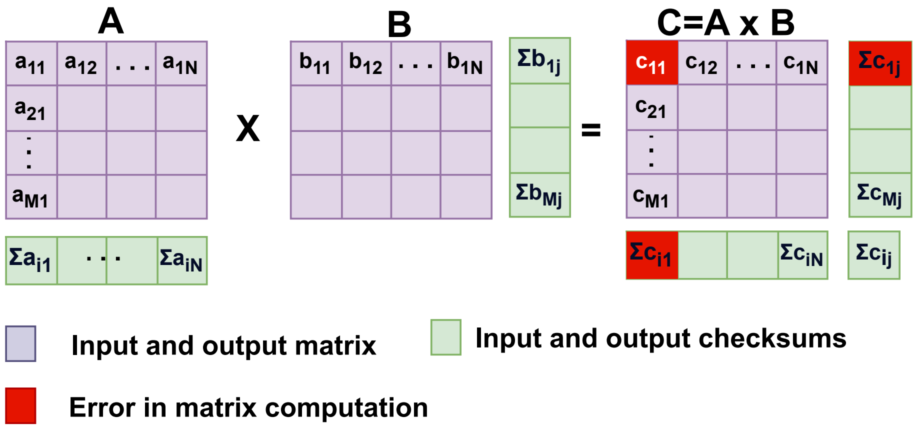


Figure 2.19.: ABFT based on row/column checksum

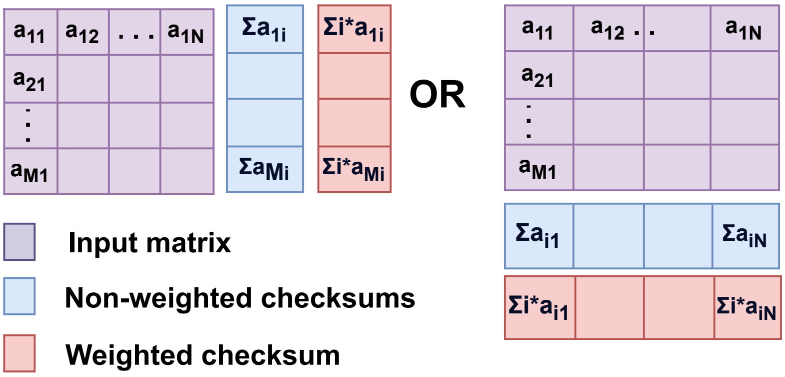


Figure 2.20.: ABFT based on weighted and non-weighted checksum

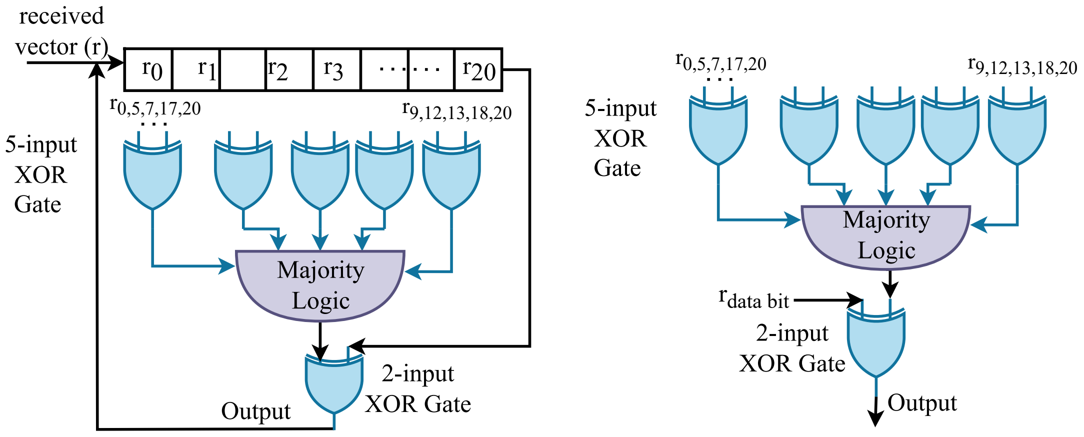
2.9. State-of-the-art fault-tolerant techniques for STT-MRAM

This section provides a comprehensive study of state-of-the-art fault-tolerant techniques for STT-MRAM in general-purpose computing systems. We briefly review the existing strategies, focusing on prominent redundancy-based fault-tolerant techniques, as well as cross-layer schemes that combine architectural and device-level approaches. These failure-specific techniques are the keystone for enhancing reliability in the face of both soft and hard errors. We also highlight opportunities and bottlenecks associated with the existing strategies. We present a more detailed overview of existing strategies; however, later in each chapter, we also give context of existing related work with respect to our proposed strategies.

2.9.1. Soft error correction (Transient fault mitigation)

ECCs are widely used to mitigate run-time errors in memory and computing systems. As the failure mechanism in STT-MRAM differs significantly from that in conventional CMOS-based memories, using ECCs along with architectural enhancements has become a common practice to ensure the robustness and dependability of STT-MRAM. Several fault-tolerant techniques have been introduced in the literature to mitigate soft errors in STT-MRAM. These techniques include general error-mitigation-based strategies, read disturbance mitigation, and write error mitigation [180]–[202].

Wang et al. [180] proposed the use of One-Step Majority-Logic Decodable (OS-MLD) codes, such as Orthogonal Latin Square (OLS) codes, Euclidean geometry codes, and difference-set codes. These codes are a class of multi-bit ECCs, relying on orthogonal parity-check sums on the error digit. The correctness of the data bit under decoding is determined by the majority voting logic results of all the parity-check sums, as depicted in Fig. 2.21(a). However, the serial iterative decoding of OS-MLD code leads to high decoding latency. They proposed a parallel decoder implementation of OS-MLD codes to address this issue, as shown in Fig. 2.21(b). This enables parallel decoding by independently replicating the majority logic and XOR gates for each data bit, thereby reducing decoding time while sacrificing decoding hardware area overhead. Although the OS-MLD codes have simple decoding logic due to their parallel, low-latency operation, they encounter certain limitations. Both Euclidean geometry and difference-set codes have restricted useful code parameters, including codeword sizes and error correction capabilities. While the OLS code necessitates considerably high storage overhead for parity bits, this increases significantly as the number of bits to be corrected increases, as shown in Fig. 2.28.



(a) Serial decoding logic for DS(21,11) code

(b) Parallel decoding logic for DS(21,11) code

Figure 2.21.: Decoding logic for OS-MLD with DS(21,11) code. DS code refers to the Difference Set code [180]

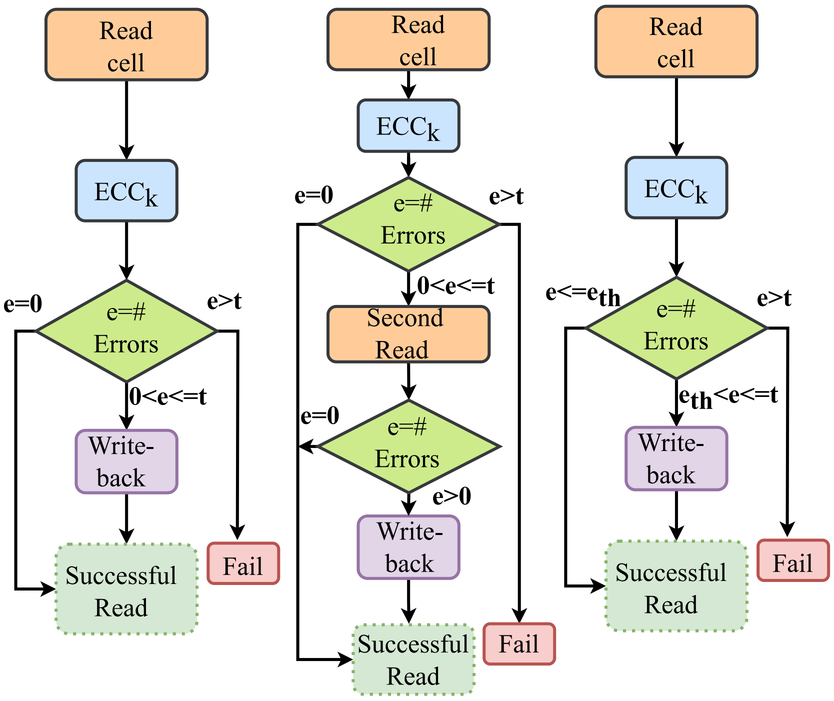
In one of the studies, Li et al. [181] explored the use of Low-Density Parity Check (LDPC) codes in single-level cell and multi-level cell STT-MRAM designs to enhance memory reliability against soft errors. This method employed a holistic channel model to assess the asymmetric effects in STT-MRAM and developed a dedicated asymmetric LDPC soft decoding tailored explicitly for the asymmetric nature of STT-MRAM. The benefits of asymmetric LDPC soft decoding are more pronounced in multi-level STT-MRAM cells than in single-level STT-MRAM cells (i.e., binary storage cell). However, this also results in increased storage overhead for multi-level STT-MRAM cells. Similarly, Mei et al. [182] presented a method to optimize polar codes for the STT-MRAM channel, where they used an approach to symmetrize the STT-MRAM channel, then constructed the polar codes and optimized their performance for the STT-MRAM channel. While these codes exhibit a significant reduction in the block error rate, it is essential to note that the decoding complexity of these codes poses a substantial hardware challenge.

The ECC uses redundant parity bits that might affect the lifetime of the STT-MRAM cache. Farbeh et al. [183] introduced Floating-ECC, which exploits the difference in write activity between ECC bits and data bits. Floating-ECC dynamically circulates the physical position of ECC bits within a cache line using swap and restore logic, ensuring that no specific bits are subjected to excessive wear. It operates in two phases: the normal phase and the repositioning phase. During the normal phase, the Floating-ECC controller increments a counter with each write operation or cache miss and triggers the next repositioning phase when the threshold is exceeded. The repositioning phase periodically changes the physical location of the ECC word within the cache lines, ensuring an even distribution of ECC activity across all bits. This method improved cache lifetime, but at the cost of one additional read and write access. Additionally, regular cache operations are suspended for a specific period during the repositioning phase.

There is an expectation of an increase in failure rate with the scaling of the storage element of STT-MRAM, which will demand strong multi-bit ECC and periodic scrubbing to ensure reliability. To address this issue, Guo et al. [184] proposed a scheme that aims to reduce the scrubbing frequency by utilizing strong BCH codes applied to long codewords that span multiple cache blocks. The idea is to minimize the need for fetching multiple blocks from memory and conducting strong ECC checks during every read by predicting and proactively scrubbing memory regions that are likely to be accessed in the near future. It employs a hierarchical error protection mechanism, wherein strong ECC is used for infrequent scrubbing, while simple ECC is employed for most ordinary memory accesses, because recently scrubbed memory blocks tend to accumulate relatively fewer errors.

The hybrid multi-level cache architecture is an emerging solution meeting the growing demand for high-performance storage systems, with STT-MRAM as a promising candidate for the first-level cache. To address this context, Hadizadeh et al. [185] introduced an STT-MRAM aware dynamic ECC allocation-based method to enhance the reliability of a hybrid cache architecture, where STT-MRAM is employed as the first-level cache and solid-state drive as the second-level cache. This approach uses (72,64) SEC-DED code to protect all pages uniformly throughout the first-level cache. In contrast, for dirty pages, a dynamically applied strong multi-bit ECC module (Double Error Correction-Tripple Error Detection (DEC-TED)) is utilized to provide strong protection. This is achieved by allocating idle or under-utilized pages from the first-level cache to meet the required ECC redundancy for the dirty pages. This method avoids the added overhead of strong multi-bit ECC by using less-utilized cache pages for extra redundancy. However, it increases cache controller memory overhead for first-level and second-level cache.

Seyedzadeh et al. [186] developed different policies to achieve read disturbance mitigation, as shown in Fig. 2.22. The first policy, called write after error, utilizes ECC to detect errors and performs a write-back operation to clear persistent errors. The second policy, known as write after the persistent error, filters out false reads by reading the data a second time when an error is detected. This involves a trade-off between write and read energy consumption. The third policy, write after error threshold, selectively leaves cells with incorrect data behind, up to a certain threshold error (e_{th}), when the number of errors is below the ECC capability. The reliability of these approaches varies depending on the dominant type of error. When false reads are more prevalent, the write after error and write after the persistent error policies are found to be more reliable than write after error threshold and ECC_t, where t represents the number of bits that ECC can correct. Conversely, when the rate of read disturbance increases, the write-after-error policy becomes more reliable due to its consistent write-back to eliminate read disturbance, but at the cost of higher energy overhead and additional memory bandwidth consumption.



(a) Write after any error detection

(b) Second read after any error detection

(c) Leave e_{th} errors behind, e_{th}<t

Figure 2.22.: Leveraging ECC to mitigate read and write faults [186]

Mittal et al. [187] propose a method to reduce the read disturbance issue in Last Level Cache (LLC) STT-MRAM by data compression and selective duplication. This can work with any compression technique, and they illustrated using the base-delta-immediate compression algorithm. Firstly, the method avoids writing any bits for all-zero data because it can be reconstructed from the compression encoding bits. Secondly, it maintains two copies of the data within the block when the data has a compressed width of at most 32B. During the next read operation, the copy experiences read disturb errors that are not corrected since the second copy remains unaffected. Lastly, for any data with a compressed width exceeding 32B, including uncompressed data items, it writes a single copy of the data, restored after each read operation. This technique reduced the read disturbance at the cost of base-delta-immediate compression and decompression logic overhead.

In a typical *w*-way set-associative cache, each read operation from a line in a set triggers *w*-1 additional reads from the other lines in the same set, leading to the accumulation of read disturbance errors. Asadi et al. [188] proposed a Read Error Accumulation Preventer (REAP) cache architecture, as shown in Fig. 2.23. This approach involves a slight modification to the cache read path, ensuring that ECC checks are conducted not only for the requested block but also for all other blocks within the cache set that are read concurrently. In this scheme, the read path is modified by adding the ECC decoder unit for each line accessed in parallel, enabling the simultaneous checking of all data blocks within the cache. As a result, ECC checking is guaranteed for every cache line after each read operation. This method increased the Mean Time To Failure (MTTF) of the cache, but at the cost of a slight increase in cache area and energy consumption due to the large number of ECC checks per read access.

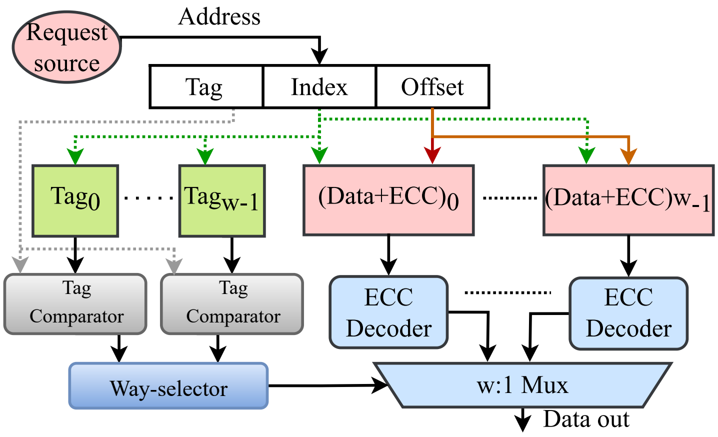


Figure 2.23.: Cache Architecture of REAP-cache scheme [188]

Wu et al. [189] exploited the thermal gradient from the read reliability perspective of the STT-MRAM LLC, where the ECC protection strength for a bank changes with the on-chip thermal distribution. The overview of their approach is illustrated in Fig. 2.24. Each bank is categorized as either a hot or a cool bank. Hot banks have temperatures higher than the thermal threshold, requiring stronger ECC protection, such as DEC-TED, compared to cool banks, which need only SEC-DED. The CPU sends a cache read request with a virtual address that comprises a bank address, set index, tag address, and data offset. The bank address is used to locate the data bank and access the bank coding table. If the bank uses the SEC-DED code, the corresponding table bit is set to 0; otherwise, it is set to 1. In case of a cache hit and when the bank uses the DEC-TED ECC code, the data and corresponding ECC code are read into a data buffer. The DEC-TED ECC decoder generates and sends the correct data to the CPU. The thermally aware adaptive ECC reduces the average ECC encoding and decoding latency, thereby improving performance and decreasing energy consumption.

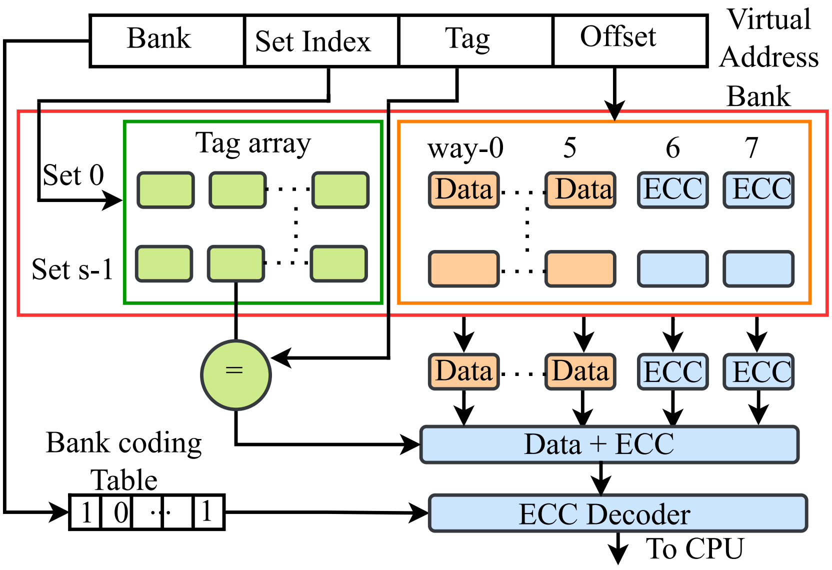


Figure 2.24.: Working of thermal aware adaptive ECC [189]

An excessive number of read accesses to the tag array may significantly increase the read disturbance rate. To address this issue, Cheshmikhani et al. [190] proposed a scheme called Read Disturbance Rate Reduction in STT-MRAM Caches by Selective Tag Comparison (3RSeT) to mitigate errors by eliminating a substantial portion of unnecessary tag reads. The concept behind 3RSeT is to minimize the number of reads from tag cells during each access by eliminating unnecessary reads. Aboutalebi et al. [191] presented the device and architecture-level techniques to mitigate and tolerate read disturbance. At the device level, they recommended enlarging the STT-MRAM cell volume (enlarging the MTJ area and free layer thickness) and increasing the anisotropy energy. These changes increase the critical switching current (I_c) and the value of the thermal stability factor (Δ), effectively decreasing the rate of read disturbances. Additionally, at the architectural level, they proposed a restore-aware memory controller design that employs a restore-aware policy selection to tolerate read disturbance. While the architectural-level technique can improve performance and energy efficiency, it is essential to consider the tradeoff at the device level, as altering device parameters may significantly degrade other important design metrics at the array level, such as area overhead.

In STT-MRAM, switching is asymmetric in nature, i.e., the write failure rate at 0→1 (P to AP) switching can be several orders of magnitude higher than 1→0 (AP to P) switching. The number of 0→1 (P to AP) switching in the write operation is proportional to Hamming weight, i.e., the number of 1s in the incoming block that should be written to the target cache line. Wen et al. [192] proposed a dynamic differential encoding strategy to reduce the Hamming weight of incoming cache data blocks to reduce the asymmetric write errors. The block diagram of this method is shown in Fig. 2.25, where the data blocks with a Hamming weight higher than the threshold are XORed to produce new data with reduced Hamming weights. This approach effectively reduces the block error rate due to asymmetric switching in STT-MRAM by Hamming weight reduction. However, it incurs higher storage overhead than SEC-DED as it requires an extra 12-bit flag per 512-bit word line to store the encoding status. In addition, this method may not be optimal when dealing with applications where the cached data blocks within most word lines have very low value locality or data correlation. This can pose a significant challenge in reducing the Hamming weight.

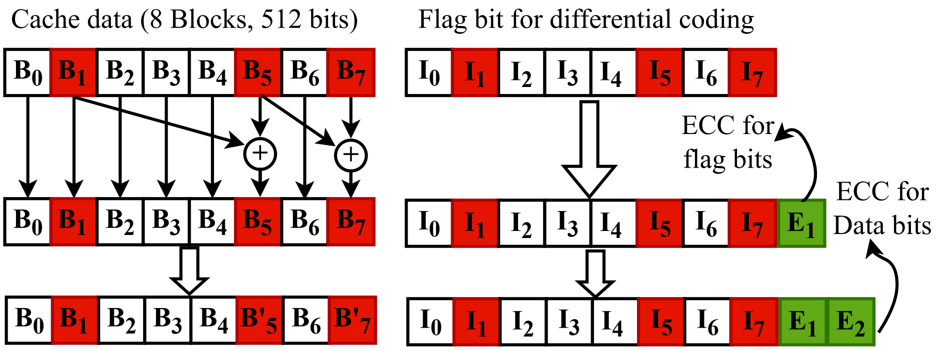

Cache data (8 Blocks, 512 bits)
Flag bit for differential coding
B_0 B_1 B_2 B_3 B_4 B_5 B_6 B_7
I_0 I_1 I_2 I_3 I_4 I_5 I_6 I_7
ECC for flag bits
ECC for Data bits
B'_5 B'_7
E_1 E_2

Figure 2.25.: Dynamic differential coding scheme [192]

Azad et al. [193] also proposed a technique called Asymmetry-Aware Protection (A^2PT) to protect the STT-MRAM LLC against asymmetric write error. The cache block is divided into groups or subsets, each with a different level of ECC protection. The incoming data is mapped to the respective group depending on their Hamming weight. The data blocks with high Hamming weights require strong ECC protection using the 7EC-8ED BCH code, while those with very low Hamming weights require weak ECC protection. Their method used four BCH codes (2EC-3ED,3EC-4ED,4EC-5ED, and 7EC-8ED) for an error protection mechanism (each corresponds to one group). They also proposed a write error rate-aware replacement policy to map incoming data to the respective cache group efficiently. The method aimed to reduce storage overhead by employing robust multi-bit ECC for large data blocks. However, the robust multi-bit BCH code requires a very complex decoding structure. Also, the method needs a lot of architectural modifications in cache policies, which can further affect the system's performance.

Wang et al. [195] proposed an adaptive ECC method to mitigate write failures. The method divides the cache block into regions, each protected by a different ECC. The error rate of data is dynamically estimated, and the data is allocated to a specific partition that offers the necessary error-correcting capability. All the cache blocks belonging to the same set are divided into groups; the simplest case is two groups having different ECC protection (one with (72,64) SEC-DED, and the other with (523,512) SEC-DED). The work in [194] also proposed a similar two-level approach using (72,64) SEC-DED for cache lines above a specific Hamming weight and (523,512) SEC-DED for the rest. However, relying solely on a (523,512) SEC-DED for entire cache lines does not achieve the desired reliability, and runtime reliability may be affected. It may fall short of modern memory ECC requirements, which typically include at least (72,64) SEC-DED per 64-bit, and in some cases, ECC_2 as well. Additionally, the method lacks protection for flag bits, which is crucial, as it may adversely affect the data bits. Additionally, these approaches necessitate architectural modifications, which can further impact the performance.

In the work presented in [196], data compression techniques were employed to reduce the number of bits that need to be written to the STT-MRAM array. However, the process of compressing and decompressing data demands complex circuit designs, which can lead to an increase in performance overhead. For instance, the data compression algorithm such as Bit-Plane Compression [203] used in [196] incurs five cycles for compression/decompression. Similarly, the work in [197] also uses the base-delta-immediate compression/decompression logic [204] to store the additional ECC parity bits in the case of the compressible cache line. Though the work uses OLS code, which has low decoding complexity, the OLS code suffers from high memory overhead, as it requires a large number of parity bits when error correction capability increases [180].

Sayed et al. [198] employed the idea of the Lazy-ECC technique, which performs speculative computation based on unchecked data, allowing the exclusion of decoding latency from read access. Fig. 2.26 illustrates the Lazy-ECC for the instruction cache, where both the ECC decoder and the processor pipeline stages receive unchecked data. While speculative computation occurs based on unchecked data, the decoder unit simultaneously checks for errors. If the read data is correct, execution continues uninterrupted. However, if an error is detected, an error signal is activated, and the erroneous instruction is reverted from the pipeline before progressing to the next stage. The pipeline then inserts no-operation instructions until the error is corrected. This method achieved a performance improvement by excluding the high decoding latency associated with strong multi-bit ECC from the critical read path. Nonetheless, it comes with the drawback of high storage overhead due to the use of a strong multi-bit BCH code. Sayed et al. [199] also demonstrated the use case of systematic unidirectional error-detecting codes, which are suitable for error detection only and not for error correction. The data can only be recovered in case of a write-through policy, as a copy of the data exists in the lower-level cache memory. Therefore, unidirectional error detection alone is insufficient to meet the overall reliability requirements.

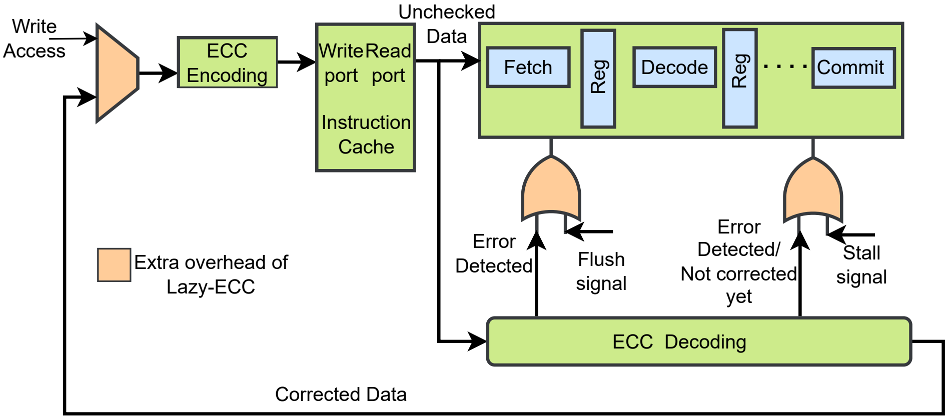


Figure 2.26.: Block diagram of Lazy-ECC (instruction cache) [198]

Bishnoi et al. [200] presented a static and dynamic circuit-level strategy to improve the write performance. The static method addresses MTJ asymmetry by boosting write current during slow-write operations. Additionally, a pMOS transistor is introduced in the bit-cell parallel to the nMOS access transistor to avoid the voltage drop across the access transistor. The dynamic technique mitigates the stochastic switching behavior by dynamically increasing the write current by activating parallel transistors of the write circuitry in a step-wise manner to drive more current. The method achieves equal switching latencies for both transitions, improving performance and reducing energy consumption. However, all these improvements come at the cost of high area overhead, up to a 20% area overhead for a 512 kB memory.

In [201], a NAND-SPIN-based LLC is proposed to improve write bandwidth, with two write policies: fixed width invert (FWI) and adaptive buffer entry (ABE). FWI sets all bits to 1 with the spin-orbit-torque effect, and checks if the 0 count exceeds half the buffer width, then inverted data bits are written using the STT effect, resulting in 6.25% memory overhead. The ABE policy checks if a number of 0s exceeds 256 in the 512-bit cache line to invert the data bits, and a single flag bit is used to indicate the inverting status. Further, the ABE policy also tracks 0s in each buffer entry to optimize the write process, requiring an additional overhead of 2.98%. Speculatively, work in [201] might face write reliability issues, showing a relatively high bit error rate of 14e-3 while writing a few selective cells of a NAND-SPIN group. Additionally, it involves counting 0s at two stages, in addition to the two-step write, which may contribute to performance overhead.

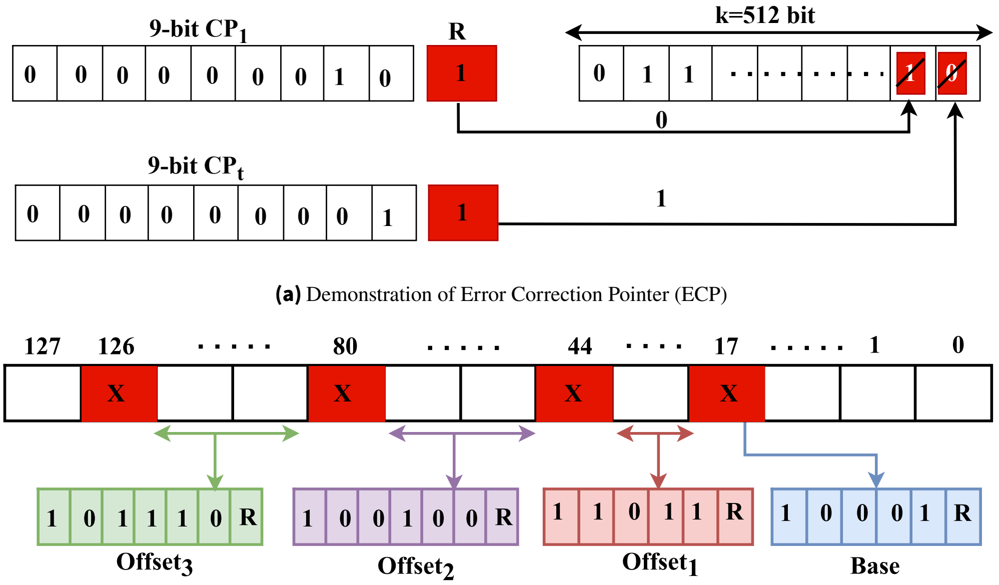
The work in [202] introduced ROCKY, a cache replacement policy designed to enhance the reliability of the STT-MRAM cache. ROCKY takes care of two things: keeping the number of write operations imposed by the L1 to L2 (hybrid SRAM-STT-MRAM) cache low and reducing the overall hamming weight ratio of data blocks in the STT-MRAM cache. However, the work requires a modified controller and a replacement policy algorithm not only for the STT-MRAM cache but also for the other-level cache. Furthermore, ROCKY employs a hybrid SRAM-STT-MRAM L2 cache, where SRAM stores data blocks with high Hamming weight, which can possess higher leakage power and reduced non-volatile storage compared to the L2 cache made up of fully STT-MRAM technology.

2.9.2. Hard error correction (Permanent fault mitigation)

The Redundancy Repair (RR) is the most commonly used technique in nearly all memory systems for addressing hard faults during post-manufacturing testing, utilizing redundant or spare rows/columns. The RR approach, however, tends to require excessive redundancy, as it demands an entire row/column to repair a single fault. This becomes particularly challenging in the case of NVMs when hard faults are dispersed throughout the array rather than clustered [205]. The primary concern, however, lies in addressing hard faults that occur when the memory is already in the field, necessitating dedicated hard error correction logic. The RR approach is very general and similar for all the memory systems. Therefore, our research primarily focuses on a dedicated, efficient hard error correction strategy tailored to when the memory is in the field and can handle diverse hard fault scenarios. Conventional ECCs, designed primarily for soft errors, are also not well suited to combat hard errors. However, integrating ECC with architectural-level modification, additional hardware logic, or combining ECC with RR and/or error correction pointers can effectively mitigate both hard and soft errors. Several studies have been reported that explicitly target hard error correction and hybrid hard and soft error correction in different NVMs [205]–[210].

Schechter et al. [206] proposed a concept of Error Correction Pointer (ECP) for hard error correction in resistive memories. ECP works by directly storing the address of memory cells that are found to have permanently failed during the verification of a memory write. Similar to conventional ECC, ECP also operates at the word level within each memory block. In the ECP_t scheme, t Correction Pointer (CP) are utilized to indicate the addresses of failed cells. Each pointer is paired with a replacement memory cell (R), as shown in 2.27(a). When performing a write operation, if the stuck-at value in a memory cell does not match the value that should be written in that cell, the corresponding replacement bit is set to 1; otherwise, it is set to 0. The value of the replacement cell is XORed with corrupted data bits to produce the corrected data value. ECP incurred significant storage and decoding overhead with an increase in error correction capabilities, as shown in Fig. 2.28 (b).

Swami et al. [207] proposed a Error Correction String (ECS) to efficiently utilize the correction pointers, which stored the hard error location for each memory word line. In the case of ECS, the offsets (distances) between successive failed cells are encoded instead of the absolute addresses of the failed cells. Fig. 2.27(b) demonstrates the simple example of the ECS method for k=128-bits memory block that adopts the base-offset approach to store the address of faulty cells. Here, the base is the address of the first faulty cell, and the offsets are the distances between successive failed memory cells. ECS permits each word line in the memory array to have offsets of different lengths, but all the offsets within a particular word line are of equal length. ECS uses redundant bits to store the offset value, replacement cell, and offset identifier bits, all together constituting a total of 66 bits for a 512-bit word line. ECS achieved lower storage overhead by efficiently utilizing correction pointers compared to ECP, with an increased burden of decoding the offsets to recover the absolute locations of the failed cells. ECS becomes inefficient when one of the offset sizes becomes large, which limits the optimal utilization of correction pointers.



(a) Demonstration of Error Correction Pointer (ECP)

(b) Demonstration of Error Correction String (ECS)

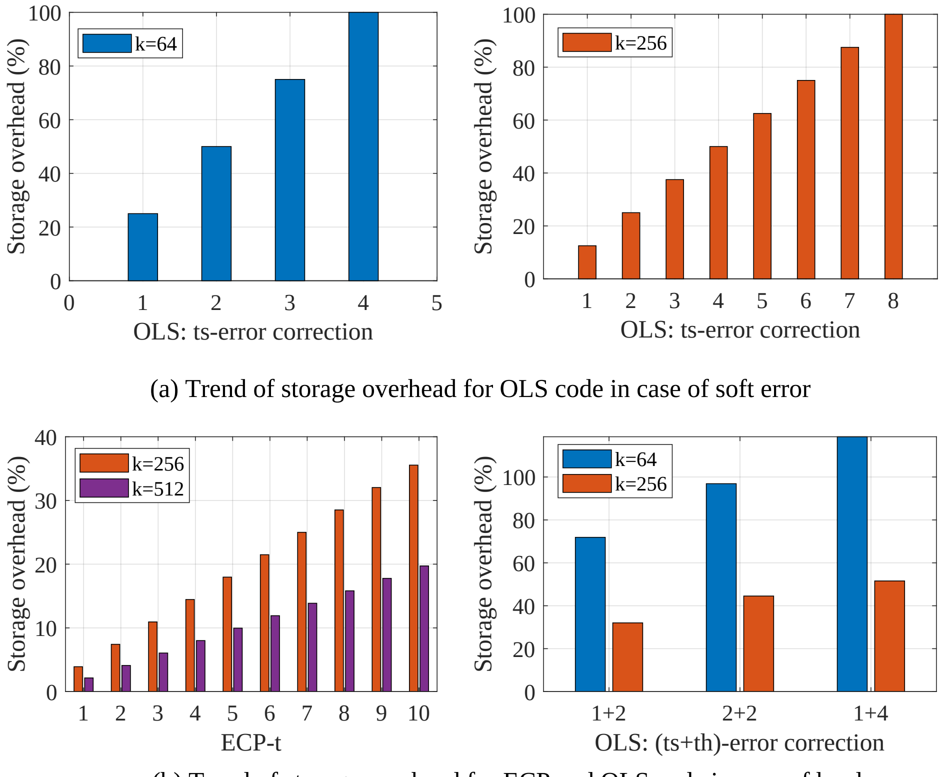
Figure 2.27.: Error Correction Pointer (ECP) and Error Correction String (ECS) for hard error correction

The stuck-at-fault mitigation scheme developed by Seong et al. [208] exploits the fact that a permanently failed cell can still be readable, allowing for the continued use of those memory cells to store data. Their method introduced a dynamic data block partitioning technique that guarantees that each partition contains at most one faulty bit. This characteristic enables the recovery of each partition by utilizing single-bit error correction techniques such as SEC-DED. After that, data block inversion is used to read the value from faulty cells in a partitioned group, where data can be stored in an inverted form if a failed memory cell has a value opposite to the data being written. This method improves lifetime; however, it is restricted to having only one fault in each group.

Qureshi et al. [211] proposed a method called Pay-As-You-Go (PAYG) for hard error correction in PCRAM. The key idea is that PAYG splits the correction mechanism into two groups: first, a local error correction that can correct up to a single error per memory word line, and second, a global error correction that contains multiple correction pointers to provide strong error correction for the word lines that have multiple errors that exceed the capability of the local error correction mechanism. The method requires additional memory access when accessing the correction pointer entries from the global error correction pool, which may introduce a performance penalty.

Yoon et al. [209] introduced a method called FREE-p (Fine-grained remapping with ECC and Embedded Pointers) to tolerate both hard and soft errors in resistive NVMs. This approach focuses on identifying and remapping only the small memory block of a physical page that has wear-out failures, leaving the rest of the page functional. FREE-p can tolerate up to six errors by employing a 6EC-7ED BCH code, comprising two soft errors and four wear-out or permanent failures, for a 512-bit word line. In this method, the responsibility of identifying a remap region for the failed block lies within the scope of the operating system. The FREE-p technique is capable of addressing both soft and hard errors. However, it comes with a significant decoding overhead due to its utilization of the 6EC-7ED BCH code. Additionally, the operating system needs to support the FREE-p scheme to allocate pages for mapping failed blocks and monitor free space in all allocated pages.

Kang et al. [205] proposed a synergistic framework, named sECC, that integrates both the ECC and fault masking to simultaneously address the permanent and transient faults in STT-MRAM. This approach masks permanent faults using a linear block ECC to mask the stuck-at faults and correct the transient faults simultaneously with the same codeword. Additionally, they proposed an enhanced integration of redundancy repair and sECC (iRRsECC) to optimize system performance. In this, all single isolated faults are masked, transient faults are corrected using sECC, and other permanent faults are repaired through redundant rows or columns. The hardware overhead primarily arises from storing the generator and parity-check matrices, redundant rows/columns, and the ECC codec structure. This method exhibits lower hardware efficiency compared to conventional redundancy repair techniques when the probability of permanent fault distribution is small. However, it shows improved performance in cases with a higher probability of permanent faults.



(a) Trend of storage overhead for OLS code in case of soft error

(b) Trend of storage overhead for ECP and OLS code in case of hard error

Figure 2.28.: Trend of storage overhead for ECP and OLS code. t_s: no. of soft error correction, t_h: no. of hard error correction

Das et al. [210] introduced a technique for hard and soft error correction in STT-MRAM LLC using a low decoding complexity OLS code. Unlike traditional OLS codes that only correct soft errors using majority voter logic, this approach employs a threshold voting scheme that can handle both soft and hard errors. This method leverages the use of a threshold voting scheme to mask the inputs to the voter that have hard errors in one or more of their constituent data bits. During the encoding process, it stores the address of the hard error locations using the write-verify read procedure along with the encoded codeword for soft error correction. In the decoding process, hard error locations are masked before being passed to the threshold voting logic, which effectively outvotes both soft and hard errors. The OLS code employed in this approach requires reduced decoding overhead compared to the multi-bit BCH code, thanks to its simple, parallel XOR-based hardware implementation. However, the technique exhibits substantial storage overhead, which escalates considerably even for minor enhancements in error correction capability, as demonstrated in Fig. 2.28 (b).

2.9.3. Summary of state-of-the-art fault-tolerant techniques for STT-MRAM

Although existing literature extensively covers soft error mitigation techniques for STT-MRAM, particularly those related to ECCs, architectural-level design, circuit and device level, and the combination of ECCs and architectural-level approaches, further improvements are needed, particularly in terms of decoding overhead, and storage overhead, to make STT-MRAM a viable option for SoC-level memory, which is also referred to as cache in modern computing systems. Regarding hard error correction (permanent or stuck-at fault) correction, most existing strategies focus on other NVMs, such as PCRAM, and have their own limitation as well. While techniques developed for other NVMs may be helpful for STT-MRAM, hard error correction for STT-MRAM has not been extensively explored, and the techniques developed for other NVMs may not be tailored to specific fault types in STT-MRAM. The existing hard error correction strategies, designed explicitly for STT-MRAM, are also expensive in terms of storage and decoding overhead, indicating a need for further research to improve these aspects and open up an option for new efficient run-time hard error correction.

2.10. State-of-the-art fault-tolerant techniques for weight memories of NN computing systems

In conventional memories in general-purpose computing systems, redundancy-based strategies have been employed to ensure fault-free operation up to a certain fault rate. However, this comes at a considerable storage overhead. Several works in the literature target fault-tolerant strategies for NNs deployed over different hardware accelerators [157], [158], [212]–[238]. Many of these strategies utilize redundancy-based approaches, such as ECCs, ABFT, and a checksum-based strategy as an alternative solution that guarantees the mitigation of faults by restoring the erroneous weights stored in the memory of the NN hardware accelerators, thereby enhancing reliability [221]–[238]. We categorize these strategies into two domains: one for digital MAC-based accelerators, and the other for resistive NVM-based CiM architecture for NN acceleration.

2.10.1. Digital MAC-based NN accelerators

In the case of digital MAC-based hardware accelerators, memory units are used to store the NN parameters (weight, bias, etc.), and processing elements carry out MAC operations to achieve the desired MVM functionality by accessing data from the memory units associated with the NN accelerators. In such a scenario, the ECC framework might be slightly modified compared to the ECC implementation used in conventional memory systems. The work in [221], [222] proposed an error detection mechanism based on even/odd parity. The Least Significant Bit (LSB) of a q-bit quantized weight is used as the parity. An error is detected when an odd number of bit errors occur for a weight, and this erroneous weight is replaced with a zero value. The work in [221] uses a 16-bit quantization representation for each weight, where 15 bits are used to represent the binary expansion of weights, and the LSB bit is used as a parity bit such that the modulo-2 sum of 16 bits is zero, as shown in Figure 2.29. Similarly, the work in [222] uses 8-bit quantized weights, where parity is applied over the group of 8 or 32 weights. Since these approaches use weight zeroing once an error is detected, it might be crucial in the case of safety-critical applications, as this corrective action may result in information loss since the erroneous weights are replaced with zeros, which may reduce NN performance. These approaches focus only on error detection and lack explicit error correction.

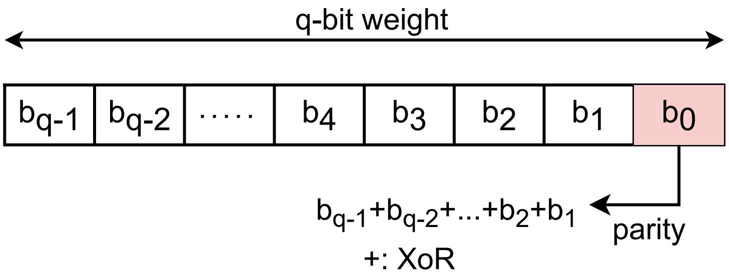


Figure 2.29.: Odd parity for error detection

Safety is the most critical aspect of an autonomous driving system. Accurate object detection and classification inference are crucial for accurately mapping the vehicle's surroundings. The work in [223] analyzes the vulnerability of NNs to random hardware faults in GPUs, specifically within the context of autonomous systems where meeting safety requirements of auto safety standards, such as ISO 26262, is crucial. The study demonstrates vulnerability to random hardware faults, such as silent data corruption and permanent faults, which can cause functional failures, including poor object detection and degradation in classification accuracy. The work also evaluates the effectiveness of chip-level safety mechanisms in GPU architectures, utilizing ECC to detect and correct random hardware faults. The target hardware platform used in their study is the NVIDIA VOLTA GPU architecture, where SEC-DED safeguards the HBM2 memory subsystem. Similarly, various on-chip SRAMs are also protected using SEC-DED.

Undervolting below the nominal voltage is an effective solution for enhancing energy efficiency in FPGAs. However, reducing the voltage without scaling the frequency can lead to timing-related faults. The work in [224] experimentally demonstrated that voltage underscaling would lead to an exponential increase in fault rate for on-chip memories, or block RAMs in FPGA. Consequently, running a FPGA-based NN accelerator at low voltage can significantly reduce its classification accuracy for a given functional task. The work demonstrated that ECC can effectively mitigate undervolting faults in NNs deployed over FPGAs. To achieve power savings for the accelerator without compromising NN accuracy, they leverage the built-in ECC of FPGA BRAMs. Consequently, the NN classification error rate reduces with the use of ECC during under-voltage operation. Using built-in ECC offers the advantage of not requiring any software or hardware modifications. However, this approach needs further evaluation regarding ECC requirements for more complex NN architectures and larger datasets.

The study in [225] proposed a selective protection scheme that chooses only a subset of weights under ECC protection to minimize the redundancy overhead associated with ECC parity bits. For instance, k_{total} is the total number of bits used to represent the NN weights, and k_{pro} denotes the number of bits protected with ECCs. A (n, k) code is used to enable error correction, and then the memory overhead for ECC redundancy can be given by Equation (2.12). A deep reinforcement learning-based algorithm is used to identify the critical components that require protection. A trade-off between ECC redundancy and DNN performance is assessed using ResNet-18 and VGG16 with the CIFAR-10 and MNIST datasets. The finite-length ($n = 8191, k = 6787$) BCH code is used for error correction, which is capable of correcting up to 115 errors. However, using such a robust BCH code for a large word length (k) is not computationally efficient from a hardware and performance perspective, as it would cause significant decoding area and latency overhead.

$$Redudancy = \frac{k_{pro} \times (n - k)}{k_{total} \times k} \tag{2.12}$$

The approach in [226] also employs selective protection to reduce the storage overhead associated with error correction logic. The work utilized explainable AI (XAI) to identify the most critical weights that require protection to prevent performance loss, and selectively applied TMR to the most sensitive locations in the feature maps. TMR can correct single bit-flips, assuming no more than one-bit error occurs during a computation cycle. The method exhibits superior inference accuracy in cases of higher redundancy. However, it drops when redundancy is reduced. However, applying TMR can be slightly computationally expensive, though it is applied selectively. For instance, TMR for 10% of values of feature maps might lead to about 30% computational complexity. Furthermore, they have also tested the approach with BCH code, similar to the work done in [225]. For the same redundancy case using the BCH code, the XAI-based protection performs better than the deep reinforcement learning-based algorithm utilized in [225]. However, using such a robust BCH code for a large word length (k) is not computationally efficient from a hardware and performance perspective, as it would cause significant decoding area and latency overhead.

The works in [227], [228] proposed a methodology to achieve zero memory overhead ECC for fault tolerance in NNs. The method introduced in [227] proposed an in-place zero-space ECC for single error correction in NN weights. The primary motivation behind their approach is that the weights of well-trained CNNs are mostly small values. The absolute values of the weights are less than 64 in most cases, so an adequate number of weight bits would be less than 8, and the remaining bit could be used for ECC parity storage. They used SEC-DED code with code dimension ($n = 64, k = 57$), applied over a group of eight 8-bit weights (64 bits). Out of 64 bits, 57 bits represent the weight value, and 7 bits store parity bits, as shown in Fig. 2.30. This way, there can not be more than one large weight in every 8 consecutive weights. Also, extra space would be needed to store the location of large weights, as it is required during the ECC decoding stage to identify the location of ECC parity bits. A modified training is required to ensure weight distribution in a specific range to eliminate the need to store large weight locations. This allows a second higher-order bit (b_6) of each weight to store the parity bits (p) of the SEC-DED, as shown in Figure 2.30. The output of the ECC logic is used to recover the original weights, and for each small weight, the sign bit (S) is just copied to parity bits to get the original weights. However, this method lacks multi-bit error correction capability, as it cannot store extra parity bits. Additionally, the approach requires a large number of modified training iterations to achieve the desired weight distribution, resulting in significant computational overhead.

The value-aware parity insertion ECC for fault-tolerant NN is proposed by [228], with reduced parity storage overhead compared to the conventional ECC approach. The method leverages the fact that most weights fall within a specific range, making it feasible to use a few weight bits for parity. Specifically, in the case of 8-bit sign-magnitude representation, when the absolute value of the weight is less than 0.5, then the b_6b_5 is always guaranteed to be 0, as shown in Table 2.3. This creates an opportunity to use weights with absolute values less than 0.5 to store the parity bits in place of b_6 and b_5. The original weight value can be recovered by zeroing the parity bits at b_6b_5. In contrast, for weights larger than or equal to 0.5, the b_1b_0 is used to store the parity. However, the original weight value may not be recovered fully, as lower-order bits are not always 0. The method uses a Double Error Correction (DEC) code for 64-bit data. During the encoding process, 14-bit parity is generated using (n=64, k=50) code and stored in the two free bits of each of the seven weights among the eight weights, as shown in Figure 2.31. The decoding process is conducted before the start of inference to perform error correction. Once the error correction is performed, the parity bits within the weights are no longer utilized, and the original weights are restored by masking the parity bits with 0s. The method necessitates additional storage (0.1%) for identifying the location of parity bits during the decoding process. Additionally, the encoding method slightly impacts the baseline accuracy, as the technique sacrifices a few weight bits to store the ECC parity bits, resulting in an accuracy loss of up to 4%. This limitation may further constrain their approach from achieving zero memory overhead for ECCs requiring more parity bits.

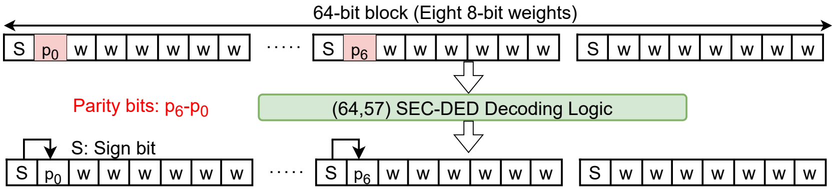


Figure 2.30.: Hardware logic for in-place zero-space ECC protection [227]

Table 2.3.: Sign-magnitude binary representation of 8-bit weight value.

Weight Value	Sign-magnitude representation
0.0625 (-0.0625)	00000100 (10000100)
0.125 (-0.125)	00001000 (10001000)
0.25 (-0.25)	00010000 (10010000)
0.5 (-0.5)	00100000 (10100000)
0.5625 (-0.5625)	00100100 (10100100)

Few studies have focused on using ABFT, which are used to enable reliable matrix operation in NN computation [229]–[231]. A sanity checksum mechanism is proposed in [229] to safeguard both fully connected and convolutional layers in DNNs. Spatial checksums are implemented using an additional neuron, while a temporal checksum is facilitated through an additional input. In [230], the authors proposed ABFT-based checksum techniques aimed at improving the reliability of CNN computations. The work has designed different ABFT schemes for CNN implementation to detect and correct errors at runtime. The first approach implements a full checksum, deriving checksums from both input and weight data. The second is related to the row checksum of the output. The third scheme is related to the column checksum of the output. The last scheme is related to checksum-of-checksum, which consists of neither input nor weight but only their checksum. ABFT-based checksum is also used in [231] to improve the reliability of matrix multiplication kernels for neural networks on GPUs. The strategy enhances traditional ABFT approaches by optimizing the use of additional rows and/or columns. Specifically, it reduces the number of matrix rows and columns required for multiplication, replacing them with checksum rows and columns. The checksums based on ABFT are widely used in matrix operations; however, decoding them can be slightly more complex than the conventional ECC operation, as it involves division, along with addition and multiplication.

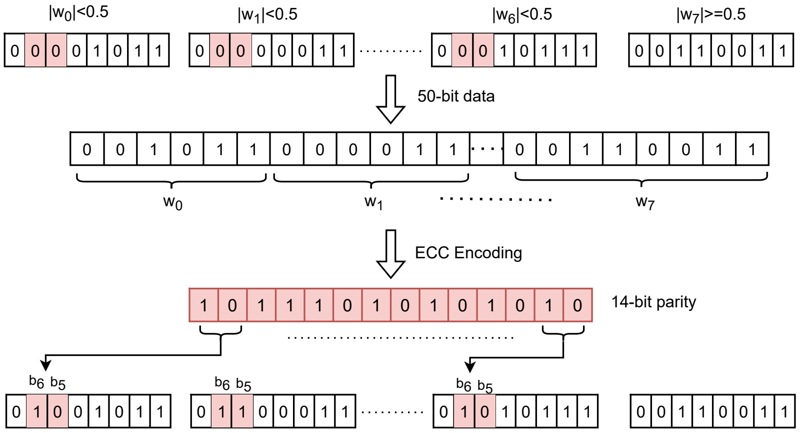


Figure 2.31.: Encoding logic for parity-aware insertion ECC [228].

2.10.2. Resistive NVM-based CiM architecture (memristive crossbar) for NN acceleration

Applying ECC for CiM-based accelerators slightly differs from traditional memory or digital accelerators, where data decoding is typically performed on each memory word line. In CiM architecture, however, multiple word lines within the crossbar are activated simultaneously to perform MAC operations in order to enable parallel MVM operations. This parallelism necessitates a ECC approach that accommodates the arithmetic nature of CiM operations, making arithmetic decoding essential for accurate error correction. As illustrated in Fig. 2.32, unencoded operands are added, and the result is encoded using the SEC (7,4) Hamming code ($f(x+y)$). On the right-hand side, however, the encoded operands are added ($f(x)+f(y)$), yielding a result that does not match the left-hand side because Hamming distance is not equal to zero (i.e., $f(x+y)! = f(x)+f(y)$). This underscores the need for arithmetic distance-based decoding logic, which can conserve the additive property essential for CiM architectures. Figure 2.33 illustrates the basic structure of a binary ECC scheme for a CiM macro. Like conventional ECC methods, encoding is applied in a similar manner, especially when bit-cells store binary values. However, the decoding process deviates from traditional ECC; instead of conventional decoding, it requires arithmetic, i.e., algebraic decoding logic, as the output represents the cumulative result of the MAC operation. Moreover, when utilizing multi-bit memristive cells for weight storage, the binary ECC encoding logic often transitions to an algebraic checksum operation rather than a simple XOR function.

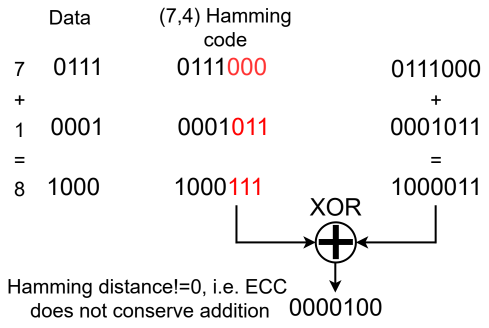
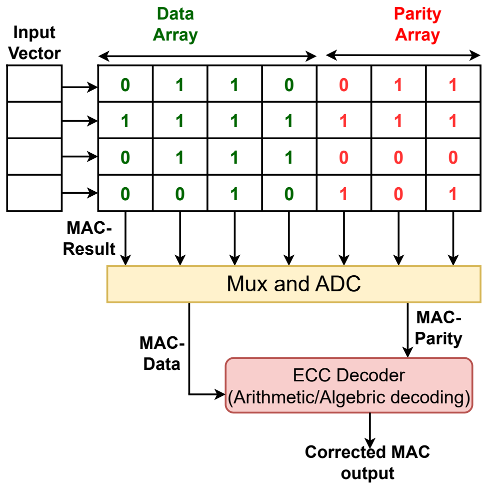


Figure 2.32.: Example demonstrating the SEC Hamming code does not conserve the additive property



0	1	1	0	0	1	1
1	1	1	1	1	1	1
0	1	1	1	0	0	0
0	0	1	0	1	0	1

Figure 2.33.: Demonstration of CiM macro with ECC

The work in [232] utilized LDPC code to correct non-ideality-induced errors during MVM operation in a memristive crossbar. Fig. 2.34 shows the overview of the error correction logic based on the LDPC coding. The pre-trained, quantized NN weights are encoded through the generator matrix (G), and these encoded weights are mapped to the memristive crossbar. During the inference stage, the binary input is applied to the crossbar, and the resultant current is converted to voltage and applied as an input to the ADC to give a digital output. The binary output is then decoded through the parity check matrix (H). The result along each column of the crossbar is a multi-level value, so the multi-level decoding through belief propagation [239] over a finite field $GF(q)$ with $q > 2$ can be performed to give the correct results. The generator matrix with $q = 2^b$, where b is the number of bits per memristive cell, is used for the encoding. The work also tried to encode only a few MSB bits of weight to reduce the storage overhead. This encoding method works well when there is low variation in the read current of the crossbar and when the device has low precision, such as $b = 1$ or 2 bits. However, when variation is significant, accuracy drops significantly, especially for 2-bit and 3-bit memristive devices. Also, the complexity of LDPC decoding is nearly proportional to the q^2, so a higher b value would impose significant computational overhead.

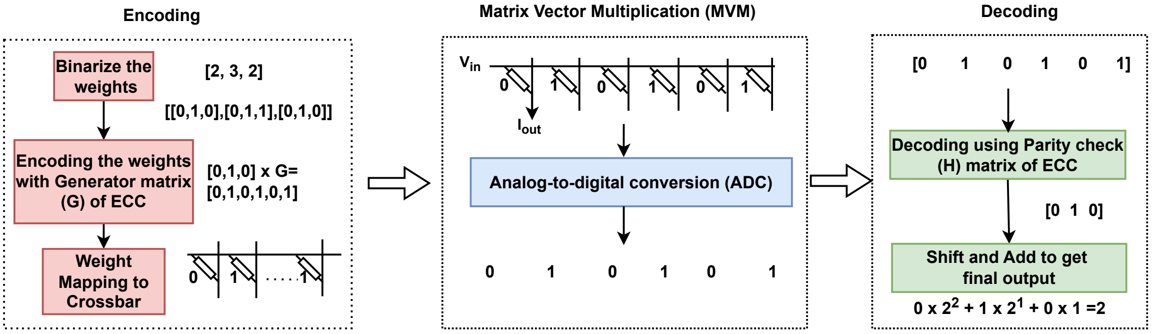
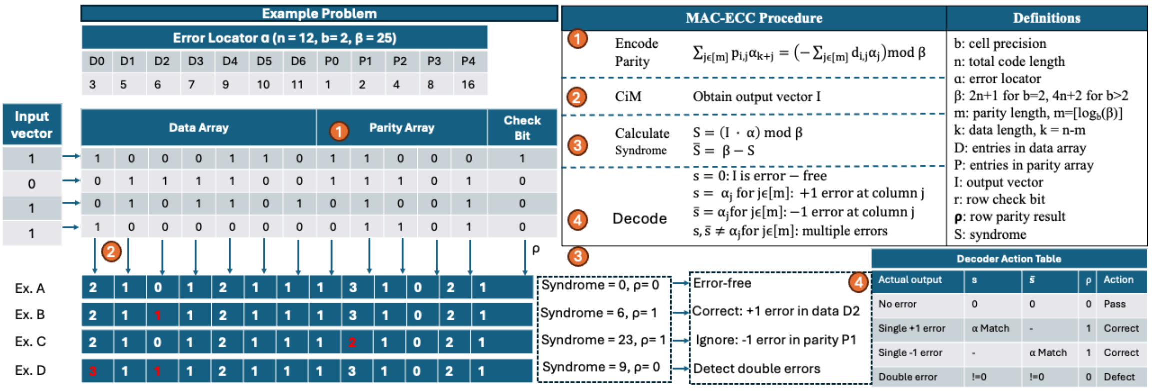

Encoding
Binarize the weights
[2, 3, 2]
[[0,1,0],[0,1,1],[0,1,0]]
Encoding the weights with Generator matrix (G) of ECC
[0,1,0] x G= [0,1,0,1,0,1]
Weight Mapping to Crossbar
0 1 1
Matrix Vector Multiplication (MVM)
V_in
0 1 0 1 0 1
I_out
Analog-to-digital conversion (ADC)
0 1 0 1 0 1
Decoding
[0 1 0 1 0 1]
Decoding using Parity check (H) matrix of ECC
[0 1 0]
Shift and Add to get final output
0 x 2^2 + 1 x 2^1 + 0 x 1 =2

Figure 2.34.: Fault-tolerant CiM architecture with LDPC ECC [232]

MAC-ECC, proposed in [233], leverages ECC techniques from [240] to perform error correction in CiM architectures based on binary RRAM cells. The MAC-ECC corrects errors based on arithmetic distance rather than Hamming distance. MAC-ECC is denoted as (n, k, r), where n is the codeword length, k is the length of data bits (or columns), and r denotes the presence or absence of the check bit column. The parity length can be calculated as $p = n - k$. The MAC-ECC encoding and decoding process is illustrated in Figure 2.35. In step 1, encoding is performed similarly to conventional ECC encoding. In step 2, the MAC operation is carried out. In step 3, the error locator and output vector are used to compute the syndrome (S) vector. In step 4, the syndrome and its complement are utilized for error correction. A ρ value of 0 indicates no errors or an even number of errors, while a ρ value of 1 indicates an odd number of errors. Different approaches are used for designing a suitable MAC-ECC scheme tailored to a specific DNN workload, target accuracy, and required error rate.

Similarly, the work in [234] also proposed SEC-DED ECC-based logic for binary RRAM-based CiM architecture targeting NN acceleration. The basic idea of the encoding and decoding logic is shown in Figure 2.36. Since it involves binary RRAM cells, the encoding process resembles that of a conventional linear binary block code, where the XOR operation is applied among the data bits to generate the parity bits. However, to enable double error detection, the method requires two additional bits: one for error detection (S_0) and the other for sign detection (S_1). The sign computation is obtained using a LUT, which stores entries for different possible combinations of data output and parity output during MVM operation. The method employs ($n = 39 + 1, k = 32$) SEC-DED ECC. From a hardware point of view, the (n=39+1, k=32) ECC hardware logic accounts for 3.5% of total area, and ECC parity bits (including associated read circuits) account for 13.3% of total area, respectively.


Example Problem

Error Locator α (n = 12, b= 2, β = 25)

D0	D1	D2	D3	D4	D5	D6	P0	P1	P2	P3	P4
3	5	6	7	9	10	11	1	2	4	8	16

Input vector	Data Array							1 Parity Array					Check Bit
1	1	0	0	0	1	1	0	1	1	0	0	0	1
0	0	1	1	1	1	0	0	1	1	1	0	1	0
1	0	1	0	1	1	0	1	0	1	0	0	1	0
1	1	0	0	0	0	0	0	0	1	1	0	1	0

													ρ	3	4
Ex. A	2	1	0	1	2	1	1	1	3	1	0	2	1	Syndrome = 0, ρ= 0	Error-free
Ex. B	2	1	1	1	2	1	1	1	3	1	0	2	1	Syndrome = 6, ρ= 1	Correct: +1 error in data D2
Ex. C	2	1	0	1	2	1	1	1	2	1	0	2	1	Syndrome = 23, ρ= 1	Ignore: -1 error in parity P1
Ex. D	3	1	1	1	2	1	1	1	3	1	0	2	1	Syndrome = 9, ρ= 0	Detect double errors

MAC-ECC Procedure		
1	Encode Parity	$\Sigma_{j\epsilon[m]} p_{i,j}\alpha_{k+j} = (-\Sigma_{j\epsilon[m]} d_{i,j}\alpha_j) \bmod \beta$
2	CiM	Obtain output vector I
3	Calculate Syndrome	$S = (I \cdot \alpha) \bmod \beta$ $\bar{S} = \beta - S$
4	Decode	$s = 0$: I is error − free $s = \alpha_j$ for $j\epsilon[m]$: +1 error at column j $\bar{s} = \alpha_j$ for $j\epsilon[m]$: −1 error at column j $s, \bar{s} \neq \alpha_j$ for $j\epsilon[m]$: multiple errors

Definitions

b: cell precision
n: total code length
α: error locator
β: 2n+1 for b=2, 4n+2 for b>2
m: parity length, m=[log_b(β)]
k: data length, k = n-m
D: entries in data array
P: entries in parity array
I: output vector
r: row check bit
ρ: row parity result
S: syndrome

Decoder Action Table

Actual output	s	$\bar{s}$	ρ	Action
No error	0	0	0	Pass
Single +1 error	α Match	-	1	Correct
Single -1 error	-	α Match	1	Correct
Double error	!=0	!=0	0	Defect

Figure 2.35.: Illustration of MAC-ECC encoding and decoding logic [233]

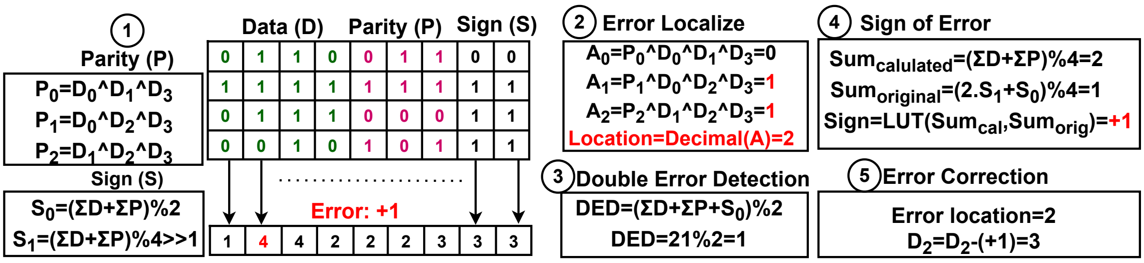


Figure 2.36.: Illustration of MAC-ECC encoding and decoding logic [234]

The work in [235] proposed an arithmetic code-based error correction scheme for analog memristive crossbar-based NN accelerators. During encoding, the weight matrix is multiplied by A before being mapped to the crossbar. The detection and correction capability of AN code highly depends on the value of A; the larger the A, the higher the correction capability will be, but at the same time, the code complexity will also increase. The work also extended the concept of AN code to data-aware ABN code, where B is a small prime number multiplied by a factor A. During decoding, errors are corrected using A, followed by error detection on the corrected result using the B value. The error correction mechanism consists of three elements: two units for computing the residuals of A and B, and a correction table used to map each residual to a syndrome, as shown in Figure 2.37. The memristive devices considered in their work have a multi-bit (2 bits per cell) storage capability. The area overhead incurred by error correction logic is about 3.4%, and the overhead incurred by the additional parity bit is around 7%. The work in [241] also proposed AN-based error correction coding for RRAM-based CiM architecture.

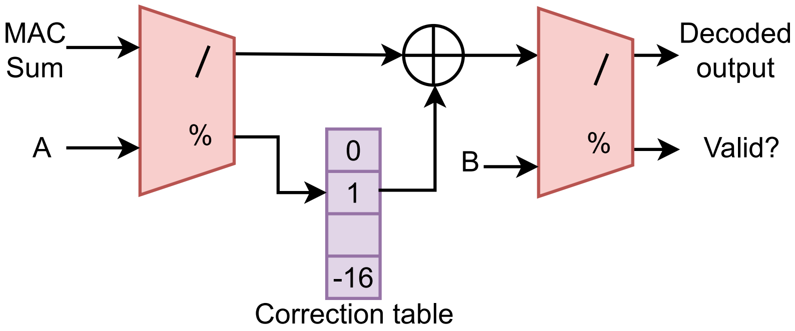


Figure 2.37.: CiM with AN code [235]

In [236], analog error-correcting codes are introduced to enable fault-tolerant MVM in memristive crossbars for NN applications. The analog ECC is implemented by expanding the CiM area with an additional encoder matrix, which generates the parity output simultaneously with the actual data output. The decoder matrix is used to check for the presence of errors. If all inaccuracies are within a defined tolerance ($\delta = 0.5$), then the decoder output will be close to zero, indicating an error-free output. The design of the decoder matrix guarantees that the pattern for any substantial deviation is unique, so that the location of the error can be determined by looking up this unique pattern. The error can be corrected accordingly based on the output magnitude that exceeds the threshold, as specified in the work [240], which demonstrates the information theory behind the analog error-correcting codes. The effectiveness of the analog ECC is validated by deviating the device conductance from its target value, programming it to a very high conductance state, and then analyzing the accuracy before and after applying analog ECC. However, the work does not include an evaluation of CNN. Also, the evaluation of the hardware overhead associated with the encoder and decoder matrix is not specified.

The work in [237] proposed extended-ABFT (X-ABFT) inspired by ABFT-based checksum logic to improve the reliability of RRAM-based crossbar for NN acceleration. The method extends the classical ABFT along the time dimension by sampling the outputs from multiple clock cycles. Similar to ABFT, the encoding is done by using extra checksum columns to store the non-weighted and weighted checksums, respectively, as shown in Equation 2.13. The non-weighted ($G_{C_{i1}}$) and weighted ($G_{C_{i2}}$) checksum for the i^{th} ($i = 1, 2...M$) row can be expressed by Equation 2.13. Where W_{f_j} represents the weight factor, and G_{ij} is the quantized conductance value (quantized weight value of the given weight matrix). The location of the faulty column can be identified by extra checksum columns used in the crossbar. To enable the purpose of extra checksum rows, the method uses extra input voltages in an additional test cycle, as shown in Figure 2.38, which is used to identify the address of a faulty row. The X-ABFT has two kinds of overhead, one related to test time, and the other is hardware overhead for redundant checksum columns. The time overhead is a function of the number of computation cycles between two test rounds and the number of test inputs (or test block size). The hardware redundancy is associated with checksum columns, as RRAM cells have limited states, so storing checksums involves a few additional columns per crossbar. For example, 1% faults while considering a 4×16 sub-array under test, the method can cover 81% of faults; however, it incurs 5% of time overhead and 33% hardware redundancy. Furthermore, the ABFT error detection and correction logic will also incur additional area overhead and decoding overhead to extract the signature for error detection and correction.

$$G_{C_{i1}} = \sum_{j=1}^{N} G_{ij} \quad \text{and} \quad G_{C_{i2}} = \sum_{j=1}^{N} W_{f_j} G_{ij} \tag{2.13}$$

$$\begin{bmatrix} 1 & 3 & 0 & (1+2+0) & (1\times1+2\times3+3\times0) \\ 1 & 3 & 0 & (1+2+0) & (1\times1+2\times3+3\times0) \\ 1 & 3 & 0 & (1+2+0) & (1\times1+2\times3+3\times0) \\ 1 & 3 & 0 & (1+2+0) & (1\times1+2\times3+3\times0) \end{bmatrix}$$

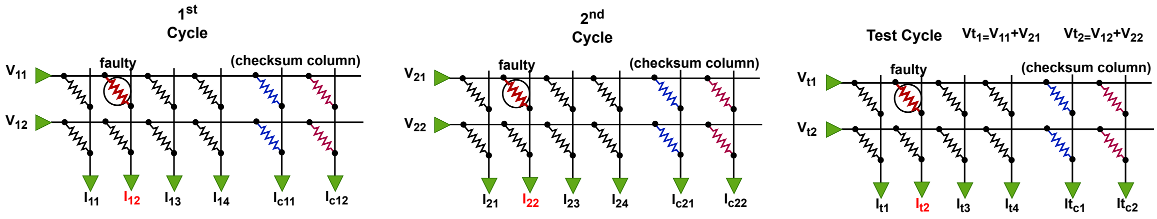
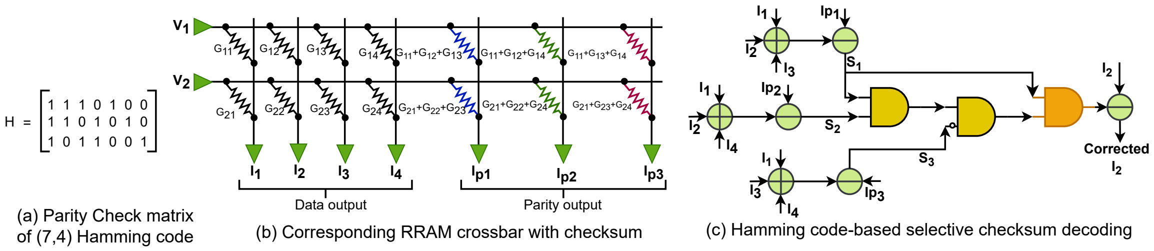


Figure 2.38.: Extension of ABFT along the time dimension [237]. V: Input, I: Output, c: Checksum, I_t: Test output



(a) Parity Check matrix of (7,4) Hamming code
(b) Corresponding RRAM crossbar with checksum
(c) Hamming code-based selective checksum decoding

Figure 2.39.: Hamming code-based selective checksum [238]

The work in [238] proposed a Hamming code-based checksum and majority voting-based checksum for error correction in the RRAM-based memristive crossbar. The method encodes the weight matrix using a binary parity check matrix, similar to conventional linear block codes, but computes the checksum by adding data instead of XOR operations, as shown in Figure 2.39(b). The number of parity checksum computations needed for k data symbols and p parity checksums is given by $k + p \leq 2^p - 1$. The decoding is performed on the final MVM output. The method can correct any number of errors in a single column at a time, also referred to as the t-column error correction strategy. The decoding can also be done algebraically instead of an XOR operation, where the magnitude of error can be added/subtracted to the received data symbol, as shown in Figure 2.39(c). The method also used a majority voting-based checksum strategy, where decoding is based on majority voting. The encoding and decoding logic would be slightly different, as the parity check matrix differs from that of the Hamming code. Also, the decoding is based on a majority voting scheme. In the case of a majority voting scheme, the number of parity checksum computations needed for the k data symbol and p parity checksum is given by $k \leq \frac{p(p-1)}{2}$. In the case of a checksum-based strategy, due to the limited number of stable states in memristive devices, checksum-based techniques require more than one memristive cell to store such a large checksum value, resulting in the need for several checksum columns in a memristive crossbar array, which leads to significant memory overhead.

2.10.3. Summary of state-of-the-art fault-tolerant techniques for weight memories of NN computing systems

The existing works cover various redundancy-based fault mitigation techniques designed to enhance the reliability of weight memories in NN computing systems. While existing literature covers a wide range of strategies, further improvements are necessary, particularly in terms of storage overhead. NN contains millions of parameters, which can result in significant redundancy for error correction strategies. Additionally, there is a lack of low-overhead multi-bit error correction methods, resulting in considerable storage overhead due to the use of parity bits. Moreover, in the case of multi-bit resistive NVM-based CiM architecture, checksum-based techniques also introduce significant memory overhead for storing the checksum values. This aspect requires further enhancement to enable a more efficient approach.

2.11. Summary

This chapter provides an overview of STT-MRAM and other emerging resistive NVM technologies, highlighting their associated reliability challenges. It then introduces foundational concepts in NN computation and explores various hardware accelerators, also highlighting the impact of memory faults on NN performance. Further, the chapter examines different levels of fault tolerance abstraction, with a focus on redundancy-based strategies such as ECCs. Finally, the chapter presents a comprehensive analysis of state-of-the-art fault-tolerant techniques in the context of this thesis, which covers fault-tolerant techniques for STT-MRAM and weight memories in NN computing systems.

3. Hard Error Correction Strategy in STT-MRAM: Enhancing Stuck-at-Fault Tolerance

Hard errors in STT-MRAM are primarily caused by oxide barrier breakdown, which can lead to cells being in a stuck-at state. This can severely impair the manufacturing yield and its large-scale industrial adoption. This chapter introduces an efficient hard error correction strategy for STT-MRAM to enhance the stuck-at-fault tolerance. In this chapter, we develop a block-based correction-pointer scheme that employs block-wise base-offset address encoding to effectively repair hard faults, supported by hard fault distribution data obtained from chip measurements. This design focuses on minimizing storage costs and ensuring simple decoding logic. The proposed strategy is designed for STT-MRAM, but this approach can also be adapted for other types of NVMs.

3.1. Introduction

Although STT-MRAM has many promising features, it has some reliability issues, including soft and hard errors. Hard errors (permanent faults: stuck-at-1 or stuck-at-0) in STT-MRAM are primarily caused by oxide barrier breakdown and variations in oxide layer thickness [29], [40], [48]–[52]. STT-MRAM achieves high switching speeds through a high current density. However, this demands thinning of the oxide barrier sandwich between two ferromagnetic layers, which can indeed lead to oxide barrier breakdown. Furthermore, the variability of oxide barriers can create resistance fluctuations that result in hard errors. Extreme process variations during manufacturing can also contribute to these hard faults. Furthermore, rough oxide barriers can result in pinhole defects, which may further grow over time, causing the memory cell to become stuck in LRS. A detailed discussion of the various sources of hard errors in STT-MRAM is discussed in Section 2.2.3. While some of these faults can be repaired using spare bits during post-manufacturing testing, there are limitations. If the faulty bits are dispersed randomly across the array, there may not be enough redundant rows and columns available to repair these faults. Furthermore, these errors can arise during memory operation. It is crucial to address hard errors during runtime to ensure reliable operation.

Several studies on hard errors, specifically in PCMs, have been reported in the literature, as discussed in Chapter 2. Despite the progress made by state-of-the-art approaches in mitigating hard errors in resistive NVMs, the challenges related to the underutilization of error-correcting resources and decoding overhead remain active areas of research. Although the techniques developed for PCMs may be applicable to STT-MRAM, the hard-error correction for STT-MRAM has not been extensively explored, and the techniques developed for other NVMs may not be tailored to the specific fault types in STT-MRAM. In this chapter, we developed an efficient block error correction pointer-based strategy to store the hard fault location and correct them during the run-time decoding process. We discuss various variants of the proposed strategy, outlining their encoding and decoding logic. Additionally, we use experimental measurement data obtained from a manufactured STT-MRAM chip to support our design strategy.

The hard fault distribution obtained from our experimental measurement data demonstrates that the faults are scattered throughout the large memory word line, with no specific sub-block having clustered faults. This demonstrates that using spare/redundant rows or columns is not efficient to replace faulty bits. This finding motivates us to propose a sub-block-specific technique to address hard errors, which is simple yet highly effective. In this paper, we propose a storage-efficient and low-decoding-overhead Block Error Correction Pointer (BECP) to correct hard failures in STT-MRAM. It adopts a block-wise base-offset approach, in which a large word is divided into smaller sub-blocks. Each sub-block has its dedicated base value and correction pointer to store the offset address of the hard error. The offset for each sub-block is determined by subtracting the hard error location from the base value. Therefore, we only need to store an offset instead of the actual address of the hard error. This enables efficient utilization of correction pointers, resulting in a reduction of memory overhead and decoding overhead while providing a high number of error correction capabilities. We utilize experimental measurement data obtained from manufactured STT-MRAM arrays at various die locations to get the fault distribution, enabling the efficient design of the proposed approach. Such experimental validations are not considered in the related work.

3.2. STT-MRAM array and analysis of measurement data

3.2.1. STT-MRAM array description

An experimental approach is adopted to analyze the hard fault behavior and distribution of fabricated STT-MRAM chips and develop a practical and technology-relevant correction scheme. The measurement involves a considerable number of memory elements, exceeding one million, which is statistically suitable to provide a solid practical perspective for the fault behavior. We describe the details of the $1Mbits$ STT-MRAM memory array, which is used as our experimental vehicle. Fig. 3.1 shows the block diagram of the $1Mbit$ (1024×1024) macro, which consists of sixteen blocks in columns (A-P) and eight blocks in rows (0-8). The $(0, 0)$ cell is physically located at the bottom right, and the $(1023, 1023)$ cell is at the top left of the array structure. Fig. 3.2 shows the general structure with the array dimension. The sixteen blocks along the column constitute the 1024 cells, each block having a size of 64 bits. The eight blocks in a row constitute 1024 cells, with each block having a size of 128 bits. Overall, this array consists of sixteen *major block columns* (MBC), each having size equal to 1024×64. The block in blue color indicates the local word line (LWL) driver and global word line (GWL) repeater. The block in red color indicates the local bit line (LBL) periphery. The GWL is divided into four segments (four repeaters). Transmission gates connect one LBL to the proper GBL and supply voltages depending on the read and write operations.

Fig. 3.3 shows the physical dimensions of the $1Mb$ memory array. A $60nm$ device is designed in a $200nm$ pitch and measuring devices in every $4um$ (i.e., in every 20 devices). The structure consists of three parts: memory elements, measuring devices, and bottom electrodes. The size of memory elements and the bottom electrode is the same for all sixteen *major block columns* (MBCs). In this study, we utilize several datasets of the fabricated memory array obtained from various wafers and distinct die positions. Fig. 3.4 illustrates the distribution of resistance in the parallel state (R_P) and anti-parallel state (R_{AP}) of measured data from one of the wafers.

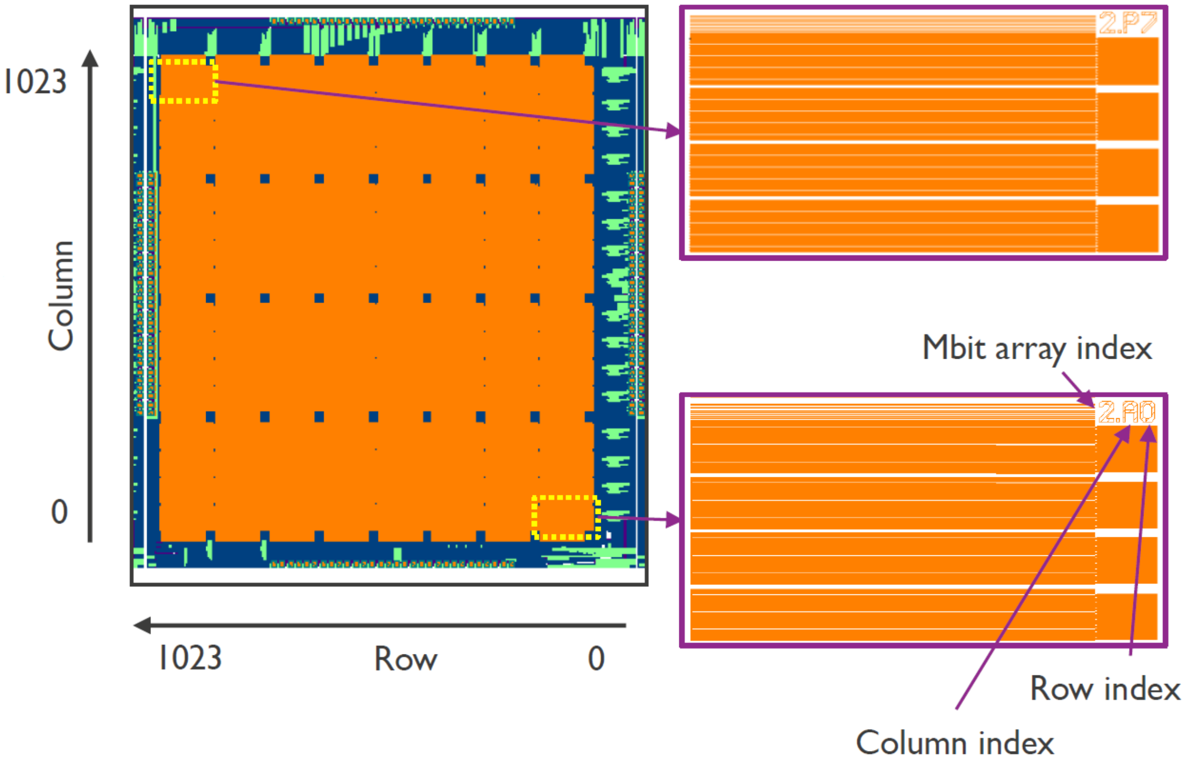


Figure 3.1.: Physical structure of 1 Mbit array macro

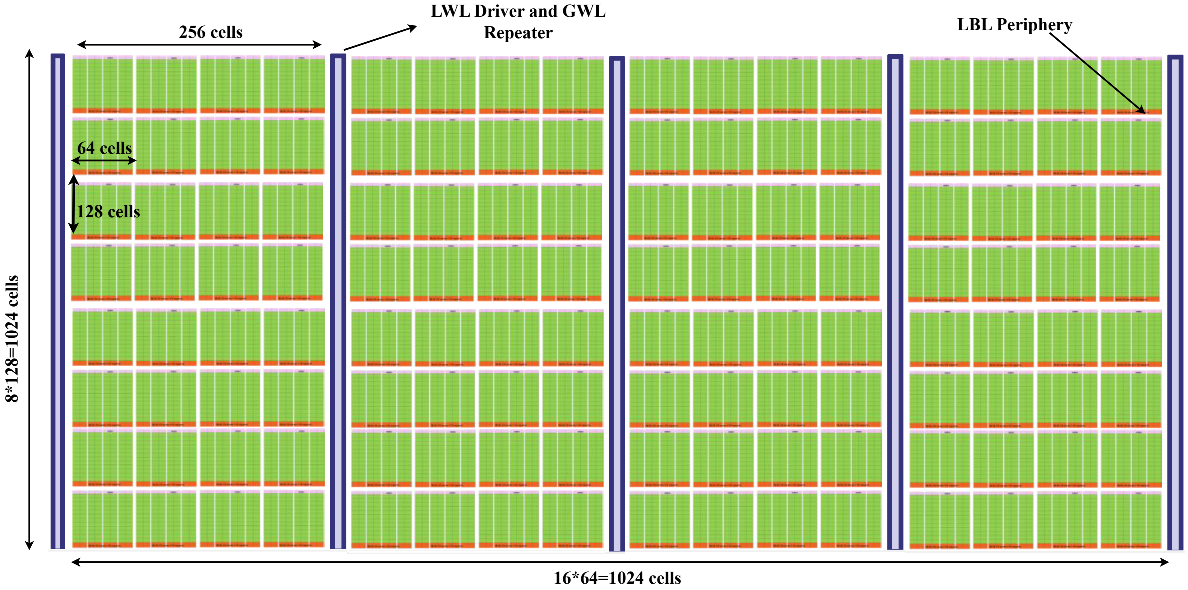


Figure 3.2.: Array Dimension of 1 Mbit array macro

Symbol	Legend
▨	Memory element
🟥	Measuring device
🟩	Bottom electrode

Item	Design	
	Device(nm)	Pitch (nm)
60p200	60	200

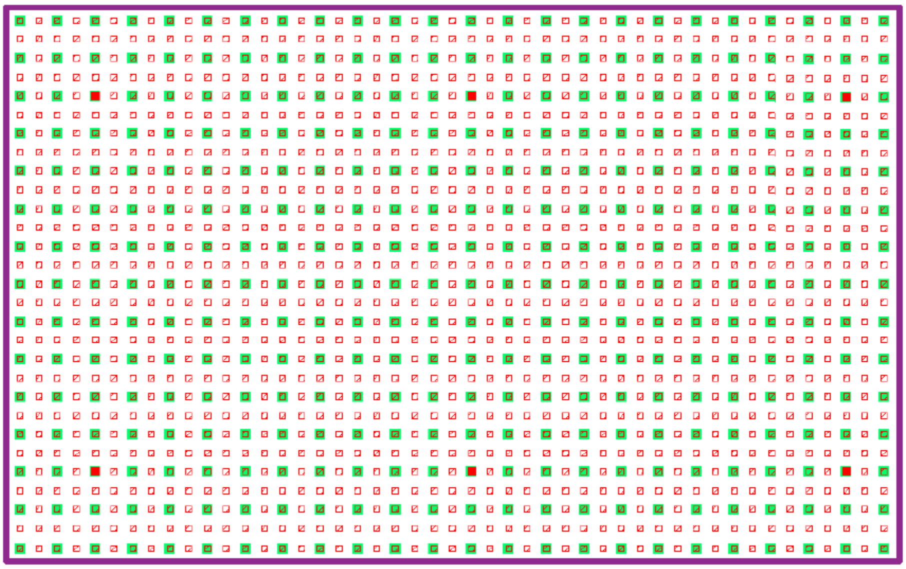


Figure 3.3.: Physical dimension of 1*Mbit* array macro

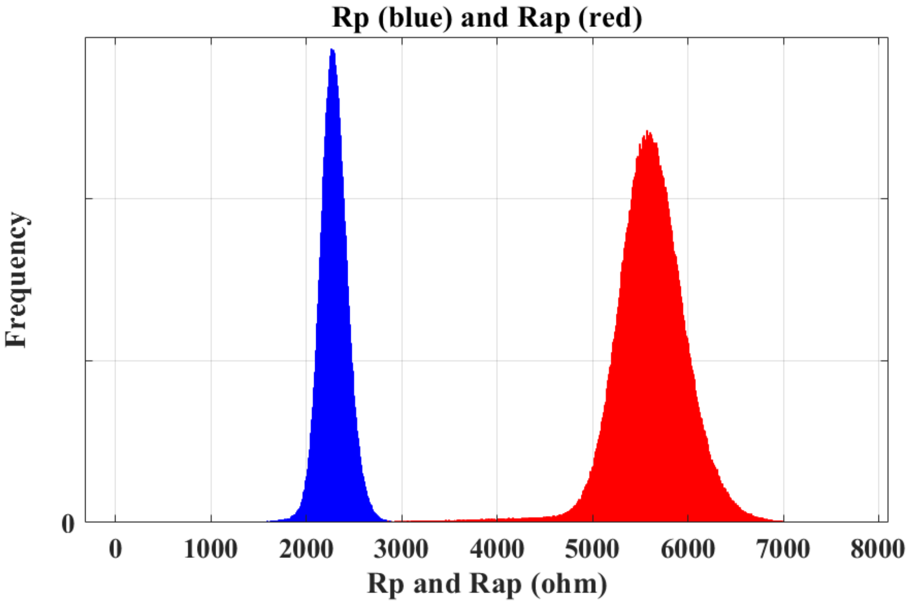


Figure 3.4.: Resistance distribution of parallel (R_P) and anti-parallel state (R_{AP}). The values of parallel and anti-parallel state resistance are determined through chip measurements.

3.2.2. Analysis of fault distribution with measured chips

To design an efficient hard error correction strategy that minimizes both memory usage and decoding overhead, we analyze the fabricated STT-MRAM arrays to examine the fault distribution. This analysis serves as the basis for our proposal of an efficient hard error correction method for the STT-MRAM array. The details of this analysis are discussed below.

3.2.2.1. Case 1: Number of faults per sub-block

In this analysis, we analyze the fault distribution across different-sized blocks, referred to as sub-blocks. Figure 3.5 presents the analysis of hard fault distribution for various sub-block sizes, using data collected from two separate wafers at three different die locations. The hard fault information is obtained through the "write-verify read" procedure (write verification) [206], [209], [210]. When analyzing larger sub-block sizes, such as 256 bits, we observe many instances where more than one fault occurs within the sub-blocks, as illustrated in Figure 3.5. Even with a sub-block size of 128 bits, up to three faults are found in some sub-blocks, although cases with four faults are infrequent, occurring in only up to 0.048% of the instances. Reducing the sub-block size to 64 bits reveals that only a few blocks exhibit more than one fault, as depicted in Fig. 3.5. The percentage of sub-blocks with two faults is up to 0.701%, significantly lower than that of 256 and 128-bit sub-block sizes. The occurrence of three faults in a sub-block is almost negligible (up to 0.043%). By considering even smaller sub-block sizes, such as 32 bits, the percentage of sub-blocks with more than one fault becomes even smaller compared to sub-block size =64 bits. i.e., up to 0.2% for two fault cases, and nearly zero for three fault cases, i.e., up to 0.009% only.

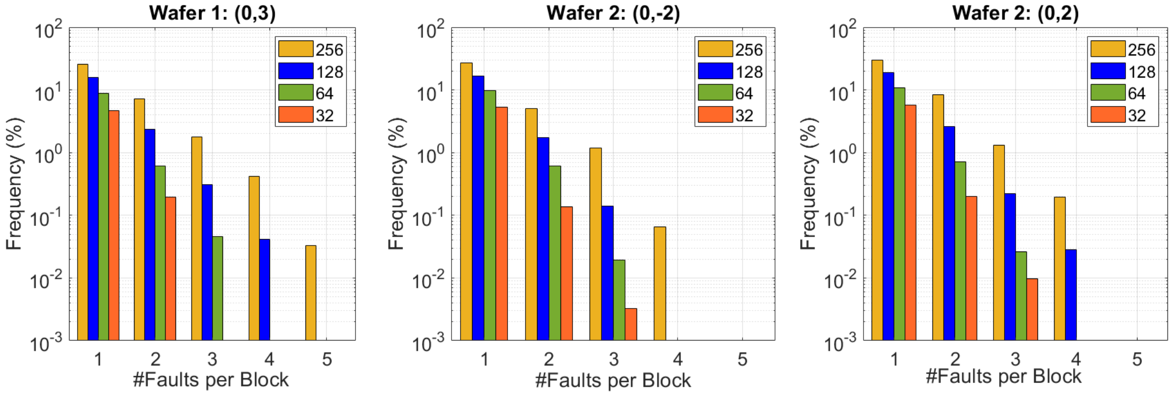


Figure 3.5.: Fault distribution with different sub-block sizes

3.2.2.2. Case 2: Multiple sub-blocks having more than one fault in a word line

We have also examined the scenario in which multiple blocks within a word line contain more than one fault. Consider a 512-bit word (two sub-arrays of size 1024 × 512 per 1Mbits) and a maximum sub-block of 64 bits. In wafer 1, the percentage of word lines in which two sub-blocks have more than one fault (up to 2 faults only) is up to 0.048%, i.e., one word line only among both sub-arrays. In the wafer 2 (0,2) dataset, it is up to 0.097%, i.e., one-word line only per sub-arrays, and it is zero in the wafer 2 (0,-2). This case was not observed for a sub-block size of 32 bits. While these occurrences are negligible, we still consider further enhancing the error correction capability.

3.2.2.3. Analysis of choosing a sub-block size

Based on the analysis above, using a sub-block size of 64 bits or smaller decreases the likelihood of encountering multiple faults. A 512-bit block can be divided into either eight sub-blocks for a 64-bit size or sixteen sub-blocks for a 32-bit size. The number of CPs equals the number of sub-blocks correcting a single hard error per sub-block. Therefore, choosing a smaller sub-block size (such as 32 bits) will increase the error correction but will also lead to high storage overhead. However, Choosing a smaller sub-block size may provide storage efficiency for word lengths less than 512 bits. A sub-block size of at least 64 bits is recommended to strike a sensible balance for a 512-bit memory block.

3.3. BECP: Block Error Correction Pointer

The above analysis serves as a basis for introducing the proposed BECP scheme, which involves breaking down large word lengths into smaller sub-blocks and applying block-wise base-offset encoding of the absolute address of the hard faults. Each sub-block is assigned a dedicated base value and a correction pointer to encode the absolute address of failed memory cells in terms of their offset value. By encoding the faulty cell location in terms of offset for each sub-block, the BECP scheme not only decreases memory overhead but also significantly reduces decoding complexity. In situations where sub-blocks have multiple faults or when there are multiple sub-blocks with more than one fault in a memory word, a single correction pointer per sub-block is not sufficient to correct the faults. We also propose a modification to BECP, called Multi-Block Error Correction Pointer (MBECP), to address these practical scenarios presented in Section 3.2.2.

3.3.1. Block-wise base-offset encoding of the hard faults

Block-wise base-offset encoding is significantly different from the encoding used in ECS, where the address of the first fault within a word is treated as the base, and the offsets are the distances between successive failed memory cells [207]. However, in the block-wise base-offset approach, a fixed base value is assigned per sub-block and is not dependent on the address of the first faulty cell in a word. Figure 3.6 illustrates the block-wise base-offset encoding of the absolute address of a failed cell. In this method, a large k-bit word is divided into multiple sub-blocks, each with a size of k_{sub}-bits. The number of sub-blocks (N_{sub}) is determined by the ratio of the original word length (k) to the size of each sub-block (k_{sub}). Each sub-block has a specific base value, and to find the offset of the faulty cell, the absolute address of the faulty cell is subtracted from this base value. The maximum address of each sub-block serves as the base value to determine the offset location of hard faults.

Consider a 16-bit data block divided into four blocks, as shown in Fig. 3.6 (b). To store the absolute address of the faulty cell, we need a 4-bit ($log_2(16)$) Correction Pointer (CP) for each failed cell, but with block-wise offset, we need only a 2-bit CP for each failed cell. Here, the important assumption is that there is one fault per sub-block, as we are using one CP for each sub-block. The BECP scheme requires N_{sub} CPs to store the offset of faulty locations and N_{sub} replacement bits to replace the faulty bits, which are adequate to correct N_{sub} hard faults. The number of bits needed to store each correction pointer is given by ($\log_2 k_{sub} = \log_2 k - \log_2 N_{sub}$), where k is the word size, and $k_{sub} = \frac{k}{N_{sub}}$ is the sub-block size. Therefore, the fractional-storage overhead (S_{BECP}) required by BECP for a k-bit word is given by the following equation:

$$S_{BECP} = \frac{N_{sub} + N_{sub} \times (log_2 k - log_2 N_{sub})}{k} \tag{3.1}$$

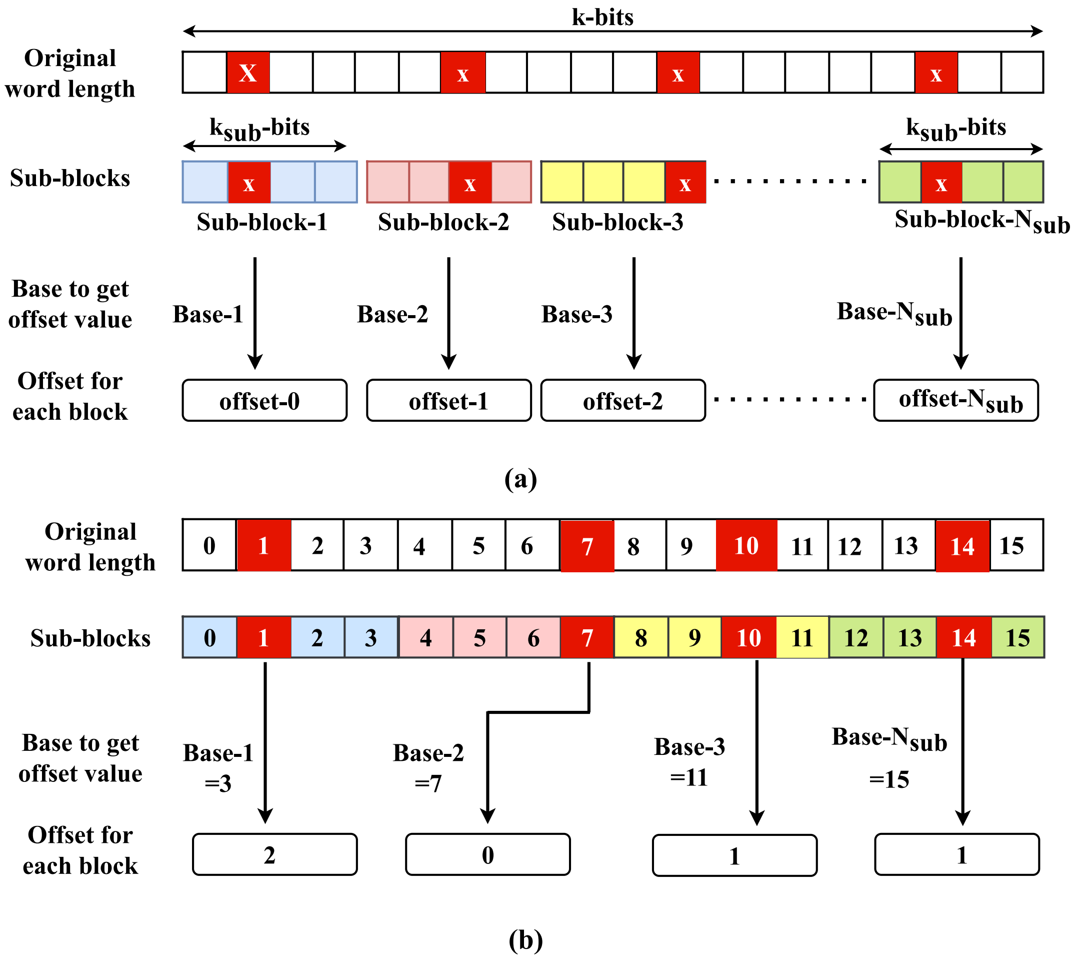


Figure 3.6.: Block-wise base-Offset encoding of BECP

3.3.2. Decoding hardware implementation for BECP

Fig. 3.7 shows the decoding logic of the proposed BECP. Each sub-block implements decoding logic separately by replicating the decoding structure of ECP_1 [206], but with the use of a much smaller decoder (6 to 64-way) instead of a large decoder (9 to 512-way). A replacement bit (R) is used to invert the cell stuck at the wrong value. If the cell is stuck at the wrong value, the replacement bit (R) is programmed to logic high; otherwise, logic zero. Depending on the value of the correction pointer, one of the output lines of the row decoder would be logic high, and the replacement cell performs an error correction by inverting the faulty cell value. The interesting point in this logic is that it does not require any additional combinational logic to restore the absolute address of the faulty cell, unlike the ECS method [207], which results in a further reduction in decoding hardware and latency. The offset bits themselves are sufficient to indicate the actual address of the failed memory cell by reversing the connection of decoder lines. Thanks to the assumption of a single error per sub-block, which can reduce both storage and hardware complexity overheads.

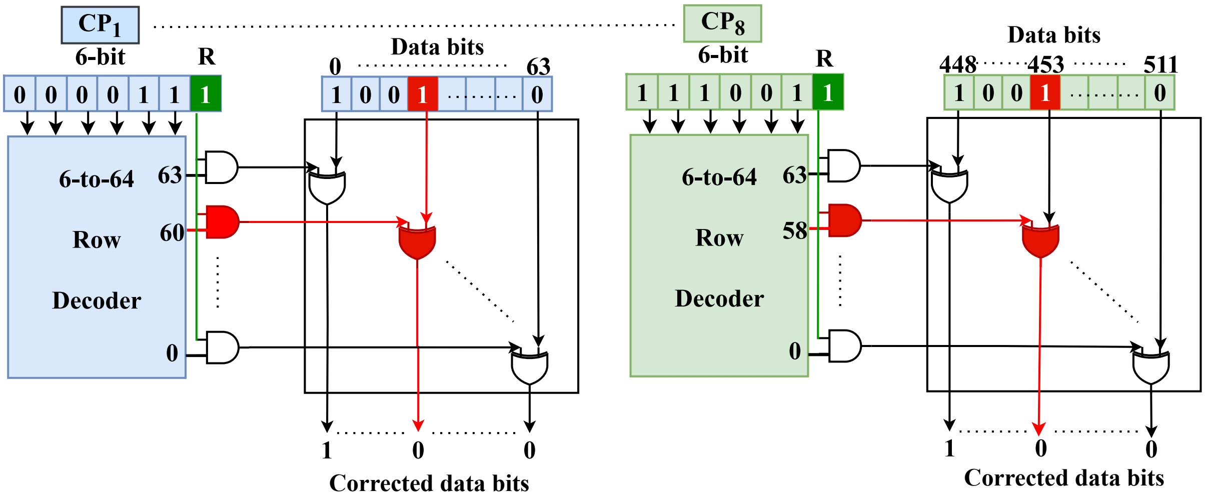


Figure 3.7.: Decoding logic for BECP (k = 512, and k_{sub} = 64-bits)

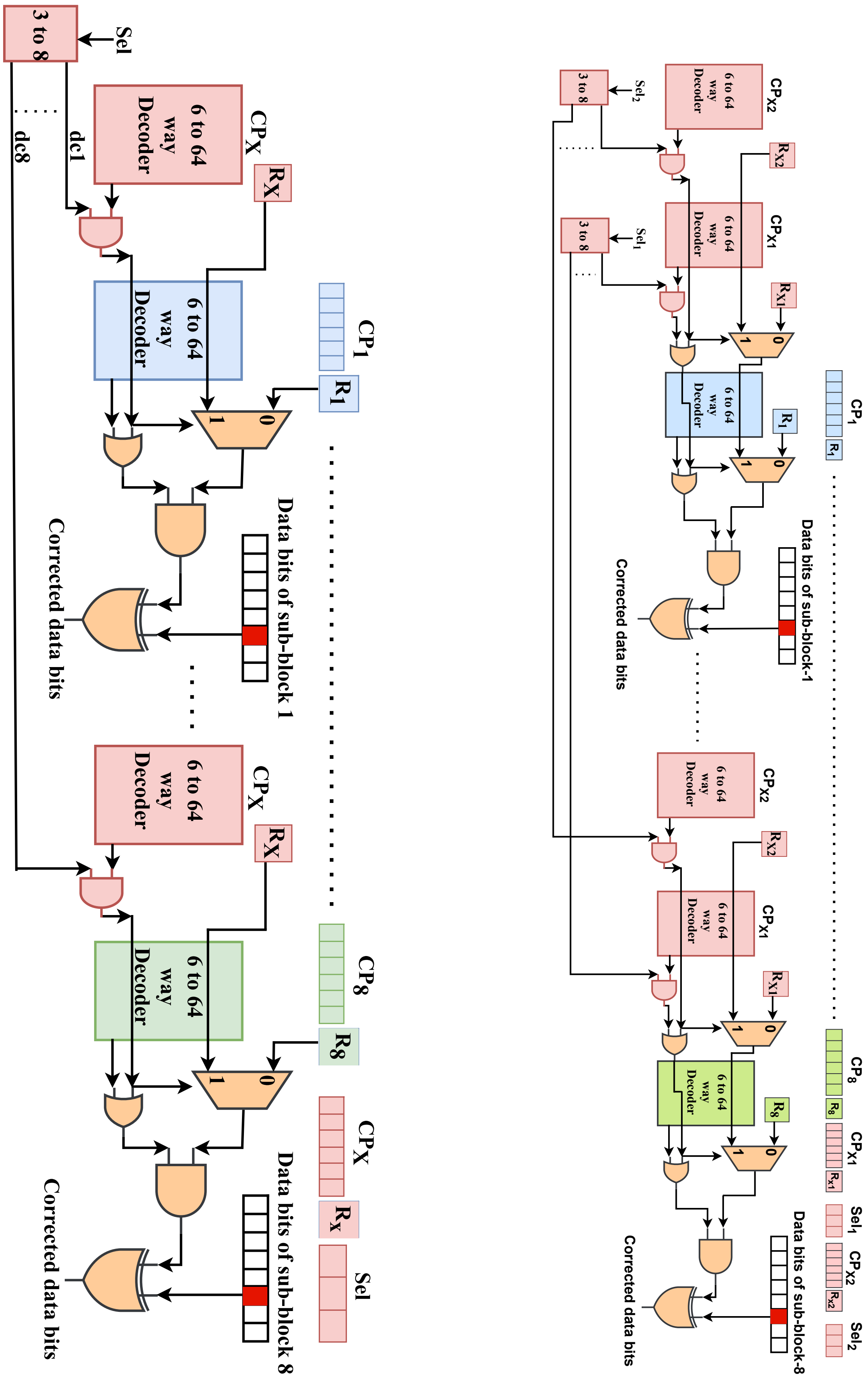
3.3.3. Multi Block Error Correction Pointer 1: $MBECP_1$

Based on the fault distribution analysis, it has also been observed that a few sub-blocks potentially experience up to two faults. To address this, we propose $MBECP_1$ as a modification to BECP, as shown in Fig. 3.8(a). It includes one CP per sub-block to correct the single fault, and an additional CP (CP_X) is used per word line for the second fault. A selection bit ($Sel = log_2 N_{sub}$) is used to indicate the specific sub-block in which the double fault occurred. A DEC decoding strategy is developed for each sub-block, and a 3-to-8-way decoder is required to activate the DEC strategy for any sub-block. If a double fault occurs in any of the sub-blocks, the output of the decoder and replacement cell (R_X) corresponding to CP_X can be passed to the right by enabling the AND gate of the corresponding sub-block. Expression (3.2) defines the fractional-storage overhead (S_{MBECP_1}) for a word line containing k-bit data.

$$S_{MBECP_1} = \frac{1 + log_2 k + N_{sub} + N_{sub}(log_2 k - log_2 N_{sub})}{k} \tag{3.2}$$

3.3.4. Multi Block Error Correction Pointer 2: $MBECP_2$

According to the fault distribution analysis discussed in Section 3.2, we have also observed that the occurrence of more than two faults in a sub-block or multiple sub-blocks with the multi-fault (up to two faults) scenario in a word line is extremely rare. However, we propose a further modified design, $MBECP_2$, illustrated in Figure 3.8(b), to enhance the fault correction ability in cases where measurements are available on STT-MRAM subarrays, which would require even more than two errors per sub-block. This design utilizes two additional CPs and applies the three error correction decoding logic to each sub-block. Similar to the previously modified approach, 3 to 8-way decoders are used to generate a control signal that depends on the select signal corresponding to each additional CP (Sel_1 and Sel_2). The control signals trigger the precedence logic, which is responsible for detecting the activation of a higher correction entry. Control signals trigger precedence logic, activating the higher correction entry. If such an entry is activated, the replacement cell of the corresponding CP can be passed to the right. The fractional storage overhead (S_{MBECP_2}) for k-bit word is given by Expression (3.3). For a specific scenario where k = 512 and the block size is 64, the number of sub-blocks (N_{sub}) is equal to 8, and the number of select bits required for each select signal to address these sub-blocks is three ($Sel = log_2 N_{sub}$).



(a) Decoding Hardware logic of *MBECP*$_1$

(b) Decoding Hardware logic of *MBECP*$_2$

Figure 3.8.: Decoding hardware logic for *MBECP*$_1$ and *MBECP*$_2$ (sub-block size = 64 bits).

$$S_{MBECP_2} = \frac{2(1 + log_2k) + N_{sub}(1 + (log_2k - log_2N_{sub}))}{k} \tag{3.3}$$

3.3.5. Handling errors in Correction Pointers metadata

The likelihood of errors in CP metadata is lower than that of data bits because metadata receives minimal write traffic compared to memory data bits, as they are updated only during the write-verify read process to get the hard fault location. Despite the negligible possibility of failure in CPs, our proposed modified approach can also handle failures in CPs. We utilize additional CP for each row to cover multi-fault situations, enabling the replacement of a faulty CP in any sub-block. The additional CP (CP_X) can be used to replace the faulty CP of any sub-block. In cases where two CPs point to the same cell, the correction entry with a higher index takes precedence over the one with a lower index. Furthermore, our experimental analysis depicts that the faults are scattered rather than clustered, suggesting that traditional redundancy repair using spare rows/columns [242] will not be efficient [243]. In practical scenarios, the occurrence of a large number of faults in any word would be rare, and such chips with high defectivity rates are usually discarded. This means our method can effectively replace the costly redundant rows/columns scheme.

3.4. Results and discussions

3.4.1. Simulation setup

This section discusses the simulation setup and evaluation of the proposed methodology. Based on the fault distribution analysis using our experimental measurement datasets, we consider the maximum sub-block size to be 64 bits. Additionally, we use a memory word of $k = 512$ bits, which is consistent with the typical memory block size. To provide insight into even smaller word lengths, we also analyze such scenarios by considering word length k=256 bits. The error correction schemes are implemented using Verilog HDL and synthesized using Synopsys Design Compiler with the GF 22nm technology. The proposed method is also evaluated against the existing state-of-the-art methods using the same technology node. However, it is essential to note that the proposed methodology can be extended to other word lengths and is not restricted to this specific length. The comparison is based on storage overhead, the number of tolerable error capabilities, and decoding overhead, such as area and latency. We have evaluated three configurations of the proposed strategy, including the BECP, $MBECP_1$, and $MBECP_2$.

3.4.2. Performance evaluation against ECP and ECS (k=512)

The comparison of the proposed technique with correction pointer-based methods (ECP and ECS)is presented in TABLE 3.1. This comparison assesses storage overhead for correction pointers, the number of correctable errors, and decoding overhead, encompassing both area overhead and decoding latency. Various error correction capabilities (ECP_n) have been considered for the ECP method. The ECP_t scheme can correct up to t errors, and as the value of t increases, the storage overhead also increases drastically compared to the proposed approach. For instance, the storage overhead of ECP_8 is greater than that of the proposed modified MBECP methods, $MBECP_1$ and $MBECP_2$, by 22.72% and 16.57%, respectively. The storage overhead for ECP increases substantially with an increase in the error correction capabilities, as shown in Fig. 3.9.

To perform conditional error correction up to nine errors for a 512-bit word, the ECS requires an additional 66 bits, resulting in a storage overhead of 12.89%. However, the $MBECP_1$ requires the same storage as ECS for the same number of correction capabilities. The $MBECP_2$ shows better correction capability than ECS but with slightly higher storage overhead, as shown in the TABLE. 3.1. In ECS, all the offsets within a specific word have the same length. A lower offset value increases the likelihood of effective error correction in ECS. However, if one of the offsets in a word line becomes large, the error correction capability can decrease. ECS permits each word line in the memory array to have offsets of different lengths, but all the offsets within a specific word line have the same length, limiting the efficient utilization of correction pointers. Practically, having a lower offset may not always be possible, which limits the correction capability.

Table 3.1.: Comparison of proposed method with correction-pointer schemes ECP and ECS (k = 512, k_{sub} = 64)

Performance metric	ECP [206]					ECS [207]	Proposed			
	ECP_6	ECP_7	ECP_8	ECP_9	ECP_{10}		$BECP$	$MBECP_1$		$MBECP_2$
Number of tolerable errors	6	7	8	9	10	9	8	9		10
Storage overhead (%)	11.91	13.88	15.82	17.77	19.73	12.89	10.93	12.89		14.84
Decoding area (μm^2)	4197	5246	5834	6883	7472	7376	712	1781	1839	2450
Decoding latency (ps)	912.71	958.55	958.55	1004.39	1004.39	1234.25	134.26	326.49	286.48	305.40

Table 3.2.: Comparison of proposed method with ECP (k = 256, k_{sub} = 64)

Performance metric	ECP [206]				Proposed			
	ECP_3	ECP_4	ECP_5	ECP_6	$BECP$	$MBECP_1$		$MBECP_2$
Number of tolerable errors	3	4	5	6	4	5		6
Storage overhead (%)	10.93	14.45	17.96	21.48	10.93	14.45		17.96
Decoding area (μm^2)	1002	1302	1831	2131	356	889	916	1218
Decoding latency (ps)	481.79	503.48	521.17	521.17	134.26	297.04	237.91	275.95

Table 3.3.: Comparison of proposed method with ECP (k = 256, k_{sub} = 32)

Performance metric	ECP [206]					Proposed			
	ECP_6	ECP_7	ECP_8	ECP_9	ECP_{10}	$BECP$	$MBECP_1$		$MBECP_2$
Number of tolerable errors	6	7	8	9	10	8	9		10
Storage overhead (%)	21.48	25.00	28.51	32.03	35.54	18.75	22.26		25.78
Decoding area (μm^2)	2131	2660	2961	3491	3790	362	921	993	1261
Decoding latency (ps)	521.17	563.78	563.78	606.40	606.40	86.64	218.87	177.40	214.45

The proposed method achieves significantly lower decoding overhead compared to existing correction pointer-based techniques, as demonstrated in Table 3.1. This is due to the fact that the proposed method requires a simpler decoder (i.e., 6 to 64-way for sub-block size = 64) to decode the offset location of the faulty cell. In contrast, existing schemes require a 9-to-512-way decoder per correction pointer entry. Additionally, the proposed methodology utilizes a reduced number of chains of OR gates and Mux logics, which further decreases the decoding overhead. For instance, even ECP_6 exhibits approximately 2.4× and 1.7× higher decoding area overhead compared to proposed $MBECP_1$ and $MBECP_2$ techniques, respectively. The decoding latency is also much lower in the case of the proposed strategy.

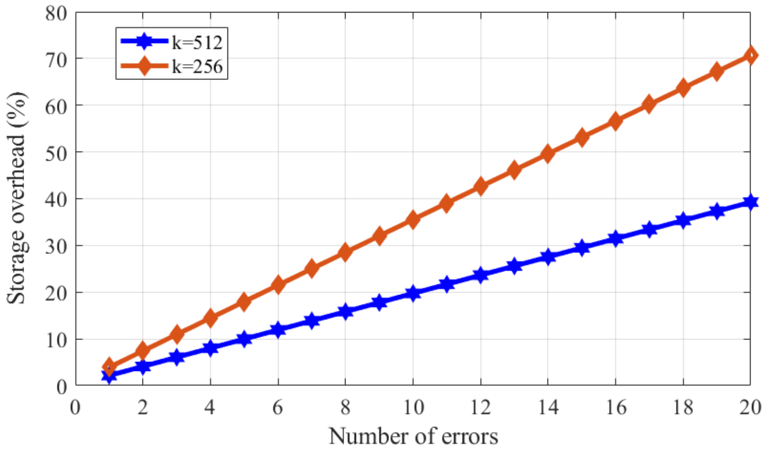


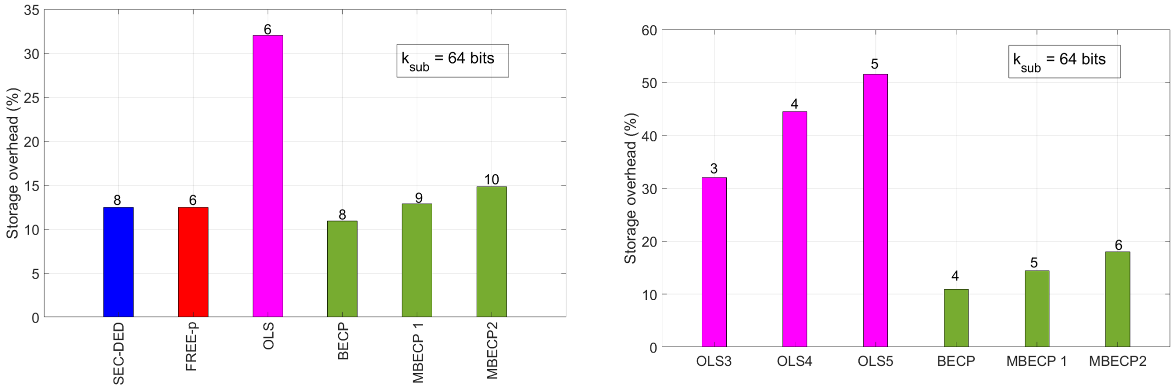
Figure 3.9.: Trend of storage-overhead for ECP [206]

In contrast, the ECP has a higher decoding latency, as shown in TABLE. 3.1. In general, it follows the order of $O(logd) + O(logn) + O(1)$. The factor $O(logd)$ corresponds to d-way decoder latency, $O(logt)$ is attributed to the chain of OR logic and Mux logic, and $O(1)$ is for the final AND and XOR gate to invert the faulty bit. In ECP, there are certain scenarios where the increase in latency is negligible. This is attributed to the optimization carried out during the mapping process by the synthesizer because the logic structures of ECP_{t-1} and ECP_t are nearly identical, with the exception of one additional level of Mux and OR logic in ECP_t. As illustrated in the TABLE. 3.1, ECS has an even higher decoding overhead than ECP because the ECS requires the use of ECP_9 decoding logic and additional logic to recalculate the original address of the faulty cell. The decoding area overhead of ECS is almost 4.2× and 3× higher than the proposed $MBECP_1$ and $MBECP_2$, respectively.

3.4.3. Performance evaluation against ECCs for k=512 bits

Fig. 3.10 (a) presents the comparative analysis of the proposed method with ECC techniques for k=512 bits word length. It illustrates the storage overhead related to storing hard fault locations and the number of tolerable or correctable errors for each respective scheme. The FREE-p scheme [209] is capable of correcting up to 6-bit errors, including two soft errors and four hard errors. Compared to the FREE-p scheme [209], the proposed technique offers better error correction capability while incurring slightly higher memory overhead. The FREE-p scheme utilizes the 6EC-7ED BCH code, which has a highly complex decoding structure resulting in a very high decoding area and latency compared to the proposed method [176]. Additionally, the FREE-p technique relies on the operating system to allocate pages for mapping failed blocks. The (72,64) SEC-DED ECC can be applied to each sub-block of size 64 and can correct a single error per sub-block. The (72,64) SEC-DED ECC incurs higher storage overhead compared to the proposed BECP.

The OLS code [210] incurs a high storage overhead compared to the proposed method. The OLS code can correct three errors per 256-bit word, which includes one soft error and two hard errors. For a 512-bit word length, two OLS codes are necessary because OLS codes are typically applied to word lengths that are powers of 2, such as 16, 64, 256, and so on. As per data reported in [210], the OLS technique would result in significantly greater decoding area overhead compared to the proposed method (≈ 6.5× $MBECP_1$ and ≈ 4.5× $MBECP_2$), while the decoding latency would be roughly twice compared to the proposed $MBECP_1$ and $MBECP_2$.



(a) k = 512 bits

(b) k = 256 bits

Figure 3.10.: Comparison of the proposed technique with ECCs. Numbers above bars indicate tolerable errors

3.4.4. Performance evaluation against ECP and ECCs for k=256 bits

In TABLE.3.2 and Table 3.3, the proposed method is compared with the existing ECP for a word k=256 bits with the sub-block size of k_{sub}=64 and k_{sub}=32 bits, respectively. The proposed method was not compared with ECS for k=256-bit words, as the offset analysis in the case of ECS is done explicitly for a word length of k=512 bits only. The proposed method requires smaller storage and decoding overheads compared to the ECP. The comparative analysis between the proposed method and the ECC, such as OLS code with three different configurations: $OLS3$ (1 soft error + 2 hard errors), $OLS4$ (2 soft errors + 2 hard errors), and $OLS5$ (1 soft error + 4 hard errors) for a word length of k=256 bits, is presented in Figure 3.10 (b). Although the OLS code corrects both soft and hard errors, it requires significantly higher storage overhead, even for lower error correction capabilities. As an illustration, the storage overhead for $OLS4$ is 3.08× that of $MBECP_1$ and 2.47× that of $MBECP_2$, respectively. The decoding overhead for OLS is also higher for k=256 bits, following a similar trend to the case of k=512 bits.

3.5. Summary

In this chapter, a hard error correction strategy for STT-MRAM is presented. The proposed design is developed based on the analysis of hard fault behavior in STT-MRAM chips that have been experimentally fabricated. The proposed technique utilizes a block-wise correction pointer and the block-wise base-offset encoding of the address of faulty cells for hard error correction. The performance of the proposed strategy is also compared with the state-of-the-art techniques. It is worth noting that the proposed technique is efficient in terms of storage and decoding complexity while achieving considerable error correction capability. The low decoding complexity of the proposed method also makes it a viable option in the case of STT-MRAM as a last-level cache. Notably, the use of experimental measurement results obtained from memory arrays manufactured on different wafers and die locations supports the validity and effectiveness of our proposed method. A similar approach can also be applied to other NVMs as well.

4. Adaptive and Asymmetric Error Correction in STT-MRAM: Addressing Asymmetric Failures

STT-MRAM has emerged as a promising alternative to conventional CMOS memory technologies for on-chip cache replacement. Due to its superior access speeds, high endurance, and scalability, it is being extensively considered a promising candidate for last-level cache replacement. This technology has reached considerable industrial maturity, with several foundries now offering this emerging technology. Despite its advantages, STT-MRAM faces reliability challenges, primarily due to its asymmetric switching error characteristics, where the likelihood of a bit transitioning from 1→0 differs from that of 0→1. Conventional ECCs do not account for such asymmetry between these bit-flip types and fall short of providing balanced error correction. This chapter presents an effective asymmetric and adaptive error correction method for STT-MRAM that leverages the Hamming weight of data bits while operating with minimal overhead alongside an adaptive ECC framework. We propose two-level strategies: the first focuses solely on reducing Hamming weights within the standard uniform ECC framework, and the second emphasizes Hamming weight reduction in conjunction with an adaptive non-uniform ECC framework. The efficacy of the proposed strategy is tested by reliability enhancement, measured by a cache word/block error rate, across the last-level cache data for various SPEC CPU2017 benchmarks [244]. This enhancement in reliability is achieved without introducing excessive memory and hardware overhead, and without compromising system performance, presenting a compelling case for improving the operational reliability of STT-MRAM.

4.1. Introduction

The switching behavior of STT-MRAM differs significantly from conventional CMOS-based memory technologies, as STT-MRAM relies on a magnetic switching mechanism that is inherently stochastic and asymmetric in nature. The asymmetric switching is characterized by a higher likelihood of errors during transitions from P('0')→AP('1') than from AP('1')→P('0') [182], [192], [193], [245], [246]. The asymmetric switching error rate ratio of P('0')→AP('1') switching to AP('1')→P('0') switching can be up to in the order of 10^3, for a typical write pulse width of $10ns$ [192], [193], [245], [246]. Though switching parameters are typically configured for worst-case P('0')→AP('1') switching scenario, however, due to the stochastic nature of magnetization, the P('0')→AP('1') switching duration is higher and shows more significant variability than AP('1')→P('0') switching, leading to a higher failure rate. Furthermore, using a longer write pulse width and a larger write current magnitude during the write operation proved inefficient from an energy consumption perspective [245], [247]. Also, asymmetric switching becomes even more prominent for higher pulse widths [192], [193], [245]. Moreover, using high pulse widths and current magnitudes diminishes write endurance and increases the risk of junction barrier breakdown, potentially causing permanent bit cell failure [48]–[50]. Such asymmetric errors challenge the effectiveness of conventional symmetric ECCs, typically designed to handle uniform error probabilities across all bits in the memory array.

In the existing literature, research has shown some attempts to enable asymmetric error correction in STT-MRAM [181], [182], [186], [192]–[197], [201], [202]. Some of these efforts have concentrated on reducing the asymmetric error rates by tailoring strategies to the data content dependent, such as lowering the Hamming weight or integrating more robust ECC and architectural modifications according to the Hamming weight of data bits [192]–[196], [201], [202]. However, existing approaches are hindered by memory overhead associated with the Hamming weight reduction approach, high decoding complexity due to the deployment of a robust multi-bit ECC configuration, and necessary architectural modifications at both the cache and system levels, which can potentially adversely impact system performance. There is still significant work needed to improve asymmetric error correction in STT-MRAM without imposing excessive hardware and performance overhead. These challenges need to be addressed to maintain a balance between reliability and efficiency in STT-MRAM.

4.2. Asymmetric errors in STT-MRAM

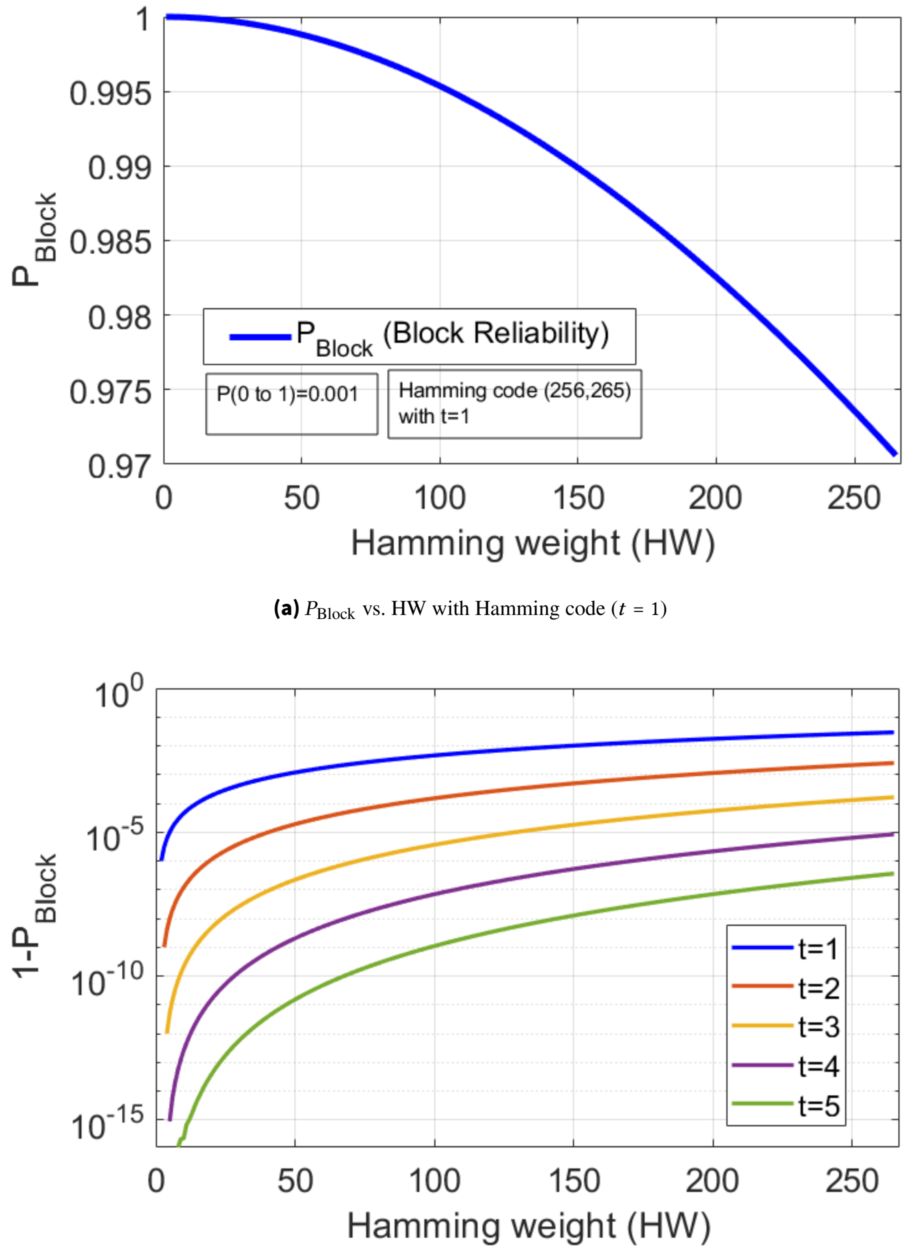
The writing data in STT-RAM cells is inherently stochastic due to the impact of thermal fluctuations [41], [42], [48], [243]. These thermal fluctuations during the magnetization of MTJs result in stochastic switching behaviors. If the write current is terminated before the MTJ completes its switching, resulting in a write error. The write switching probability in STT-MRAM is governed by Equation (2.5) [42]. Specifically, in STT-RAM cells, transitioning the MTJ from a low-resistance state to a high-resistance state (0→1) requires a larger current or longer write pulse due to lower spin-transfer efficiency. Furthermore, the variability in the switching time during the 0→1 transition is more significant, contributing to an increased likelihood of errors during this transition. As a result, the switching error rate from $0 \rightarrow 1$ is higher compared to the switching direction from $1 \rightarrow 0$ [182], [193], [245], [246].

Similarly, this asymmetry can also be reflected in the read operation. In STT-MRAM, the read and write currents share the same path, which introduces the possibility of unintended switching of the bit cell data when the read current is applied during the sensing process, resulting in a read disturbance [42]. While the retrieved data may initially appear accurate, the stored value could become corrupted, leading to potential multiple bit flips in future reads from the same location. The read-disturb phenomenon occurs solely in one direction. However, this uni-directionality of read disturb depends on whether the read current path is aligned with the write-0 or write-1 direction [182], [201]. The unintended asymmetric bit-flip can occur during reading, for example, i.e., 1→0 bit-flip when reading a ‘1’ [196].

4.3. Constraints of uniform ECCs

Studies have demonstrated that the number of 0→1 bit-flipping during write operation is proportional to the Hamming weight, i.e., the number of ‘1’s in the incoming data blocks being written to the target word/cache line [192], [193], [245]. Therefore, the Hamming weight serves as an upper limit for the number of 0-to-1 switching. Fig. 4.1 (a) demonstrates how memory block reliability (P_{Block}) correlates with Hamming weight, for a 256-bit block size with Hamming code having error correction capability $t = 1$. The dominant bit error rate of 0→1 switching ($P_{b_{0\rightarrow 1}}$) is considered to represent a fully asymmetric case. The block level reliability (P_{Block}), which indicates the successful flipping of all bits, reduces as the Hamming weight of the word increases, as shown in Fig. 4.1 (a). For high Hamming weight, achieving acceptable reliability may necessitate the use of a multi-bit ECC, as indicated in Fig. 4.1 (b), which relates word/block error rate (P_{BER}=1-P_{Block}) to Hamming weight for various ECC strengths (t).

ECCs are typically designed with fixed error correction capability and offer symmetric error correction capability to both bit-flipping directions (i.e., 1-to-0 and 0-to-1 errors are equally likely and equally correctable). As a result, conventional ECCs may struggle to efficiently address asymmetric error rates. Furthermore, employing strong multi-bit ECC to cover the worst-case highest Hamming weight scenario (i.e., the worst-case highest number of 0-to-1 switching), as such cases are rare and highly unlikely to occur, is also inefficient due to increased ECC memory overhead and decoding costs. This highlights the necessity for asymmetric and adaptive error correction in STT-MRAM to improve the overall reliability.



(a) P_{Block} vs. HW with Hamming code ($t = 1$)

(b) P_{BER} $(1 - P_{\text{Block}})$ vs. HW with various ECC strengths (t).

Figure 4.1.: Word/Block-level reliability vs. Hamming weight (HW) for an asymmetric error rate $P_{b_{0\to 1}} = 0.001$

4.4. Asymmetric and adaptive error correction

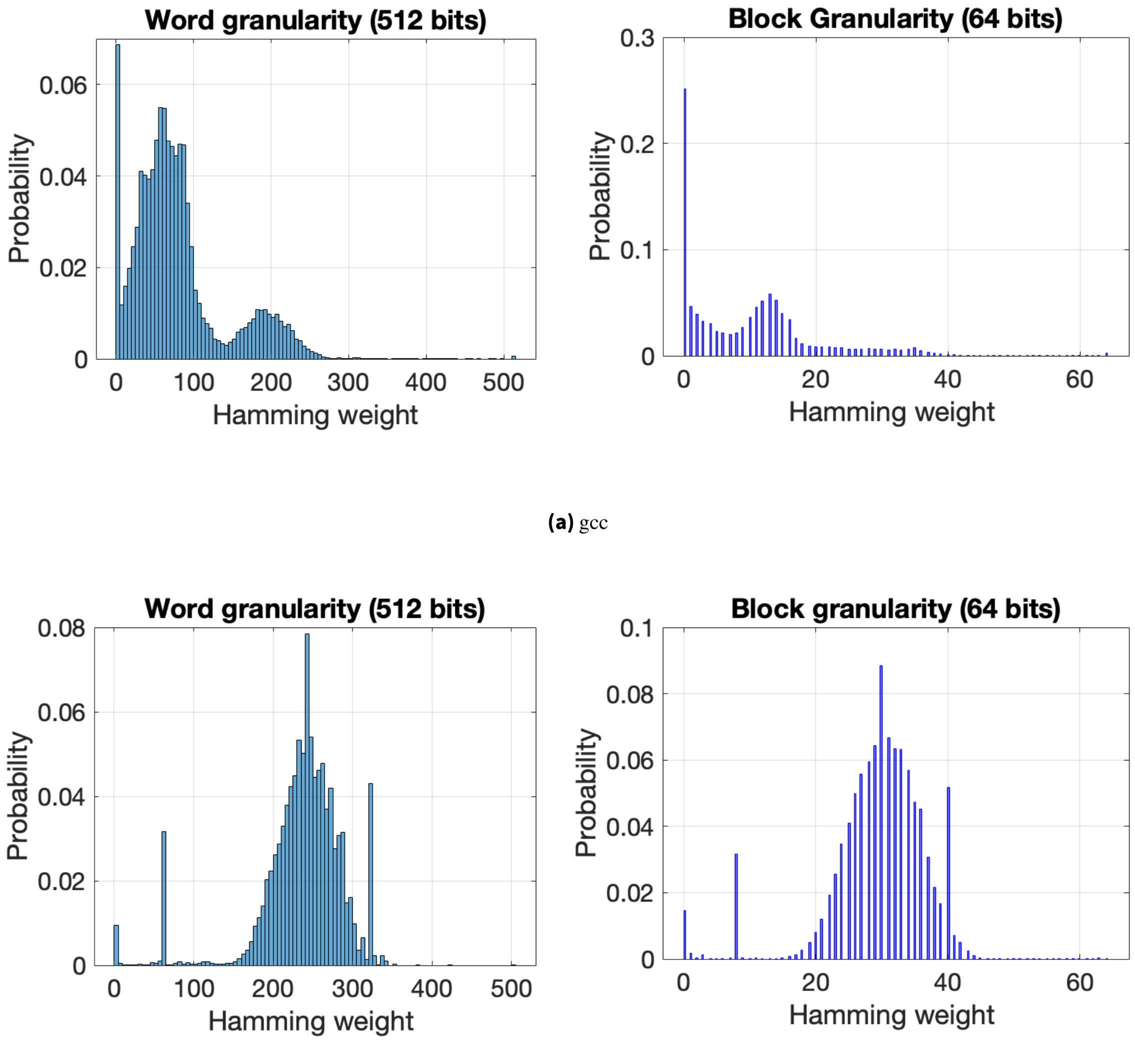
This section explores the proposed strategy for asymmetric and adaptive error correction. First, we introduce the concept of applying XOR-based differential encoding selectively guided by the Hamming weight of the incoming word/cache line being written to the memory, aiming to reduce the Hamming weight, which eventually minimizes the asymmetric 0-to-1 switching. Following this, we employ a standard SEC-DED ECC on encoded data bits. The application of ECC is essential after the first level of asymmetric switching reduction because even when memory cells store either 0 or 1, a runtime error can still occur due to external factors such as thermal variation, interference of magnetic field, oxide barrier breakdown, etc. To mitigate these issues and enhance the overall reliability, ECC protection is always essential, safeguarding the memory from runtime errors and improving long-term stability. Additionally, we introduce the concept of an adaptive ECC strategy to enhance the STT-MRAM reliability when data blocks have low Hamming weight reduction.

4.4.1. Hamming weight reduction

Fig. 4.2 shows the Hamming weight distribution of last-level cache data for two representative SPEC CPU2017 benchmarks [244], based on write cache traces, as our focus is on the Hamming weights of incoming data bits being written to memory. Here, two scenarios are presented: the first involves a 512-bit cache line, and the second involves a block granularity of 64 bits. The block granularity is important because ECC is applied at the block level within a word line. As shown in Fig. 4.2, most of the Hamming weight distribution does not lie in the extreme range. Therefore, using uniform, robust multi-bit ECC to cover the extreme range is overkill from the perspectives of memory storage and decoding overhead. In the case of cache data, it is also observed that data correlation (value locality) exists within the word line, i.e., data is correlated at the block level granularity [195]. An XOR-based differential encoding can be a common approach to de-correlate the data blocks [192]. However, real-world data may disrupt this correlation, particularly in the case of low Hamming weight blocks between highly correlated blocks.

To address this issue, we adopt selective XOR-based differential encoding based on the Hamming weight of the entire word line (also referred to as cache line) to reduce the Hamming weight of incoming data bits before they are written to the memory array, as shown in Fig. 4.3. The underlying concept is that when a word line has a high Hamming weight, there is a greater likelihood of correlation among the data blocks within that word line. In such cases, performing the XOR operation between data blocks effectively reduces the Hamming weight, as XORing two blocks with a higher number of 1s will result in a higher number of 0s. If the Hamming weight of the word line is greater than the defined threshold (HW_{th1}) value, then we only perform the XOR-based differential encoding between the data blocks in that word line. Otherwise, the data blocks are transmitted without encoding, as demonstrated in Fig. 4.3.

The first data block is selected as the base, and XOR operations are performed between consecutive blocks to obtain encoded data (B'). This process is mathematically expressed as: B'_1=B_1, and B'_i=$B_i \oplus B_{i-1}, i \geq 2$. This way, the proposed strategy bypasses the differential encoding of word lines with lower Hamming weight to avoid the adverse effect. As we consider the Hamming weight of the entire word line, we need only a single flag bit to store the encoding status. The Hamming weight distribution might differ slightly for different workloads, so the Hamming weight threshold (HW_{th1}) depends on the workloads. Hence, to determine the adequate HW_{th1}, we simulated the Hamming weight distribution of cache data of various workloads from the SPEC CPU2017 benchmark suite with different thresholds. We consider the size of the word line to be 512 bits; so we varied the threshold up to this range. The threshold that significantly reduced the Hamming weight in most benchmark datasets is selected.



(a) gcc

(b) roms

Figure 4.2.: Normalized distribution of Hamming weight of cache data for SPEC CPU2017 benchmarks (gcc and roms)

4.4.2. SEC-DED ECC and reconstruction of original data bits

After the initial level of Hamming weight reduction is achieved using selective differential encoding, the SEC-DED ECC is employed to further enhance overall reliability. We apply ECC at the block level granularity, utilizing the standard (72,64) SEC-DED Hamming code for 64-bit block granularity. During the ECC encoding process, the encoded data generated through the selective differential encoding process is fed to the ECC encoder, producing a 72-bit codeword consisting of 64 data bits and 8 parity ECC bits. An additional flag bit is used to indicate the data encoding status, indicating whether the differential encoding is applied or not for a word/cache line. During the decoding phase, ECC decoding is conducted first to detect and correct errors. The overall process is summarized in Fig. 4.4. The ECC decoder takes encoded data bits and parity bits as input, computes the syndrome vector to detect the occurrence of an error, and performs the error correction. Following the ECC decoding, the status of the flag bit is identified; if it is 1, the differential encoding is reapplied to retrieve the original data. Conversely, this reversal step is skipped if the value of the flag bit is 0.

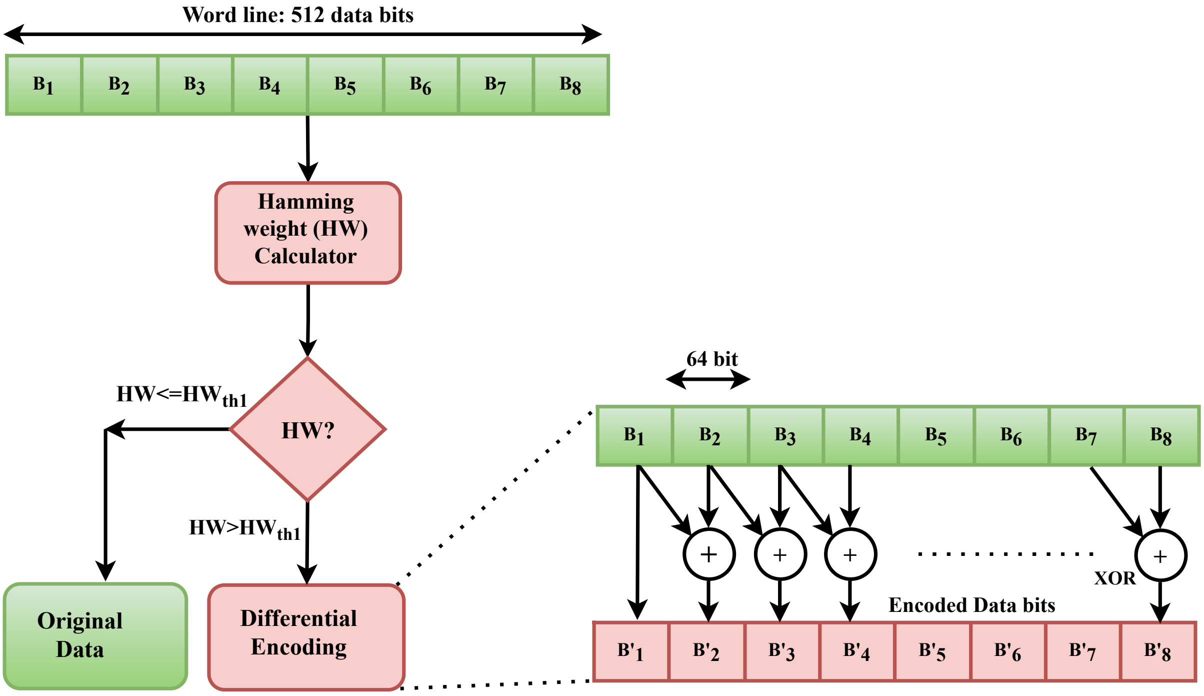


Figure 4.3.: Overview of the proposed Hamming weight reduction strategy. HW: Hamming weight, and HW_{th1}: Hamming weight threshold for a word line. We consider a data block size of 64 bits, which is aligned with the standard data block size for ECC implementation and most suitable from the Hamming weight reduction point of view, as per our simulation

Addressing potential errors in the flag bit is also a critical concern, as it can cause unintended differential encoding to the data bits, which needs to be taken care of. To prevent the need for extra memory overhead for protecting the flag bit, utilizing a shortened code can be a helpful strategy, i.e., deriving the shortened code dimension from the larger code dimension [248], [249]. Consider an ECC code dimension (n, k), with the generator matrix $([G]_{k\times n})$, and data block $([B]_{1\times k})$, where n represents the codeword size and k represents the data bits. The ECC encoded codeword can be obtained using Equation (4.1), where the input data block (B) is modulo-2 multiplied by the generator matrix (G) of the ECC. The number of parity bits required by ECC is equal to $r = n - k$.

$$[C]_{1\times n} = [B]_{1\times k} \times [G]_{k\times n} \tag{4.1}$$

In order to protect a flag bit, it is necessary to incorporate it into the ECC encoding process, thereby altering the data size to $[B]_{1\times k+1}$. This modification necessitates a revision in the original code dimensions to (n_{new}, k_{new}), where $k_{new} = k + 1$, with a corresponding generator matrix $([G_{new}]_{k_{new}\times n_{new}})$. To create the generator matrix for this adjusted code dimension, the first step is to compute a generator matrix $([G]_{k\times n})$ for a code dimension that exceeds the newly required parameters. This is done by determining a (n, k) Hamming code, where $r = n - k = 1 + log_2(k_{new})$. Following this, the process entails the removal of $k - k_{new}$ rows and columns from $[G]_{k\times n}$ to get the new generator matrix $([G_{new}]_{k_{new}\times n_{new}})$ for (n_{new}, k_{new}) SEC Hamming code. The (n_{new}, k_{new}) code can be made to behave as SEC-DED by appending an additional parity bit.

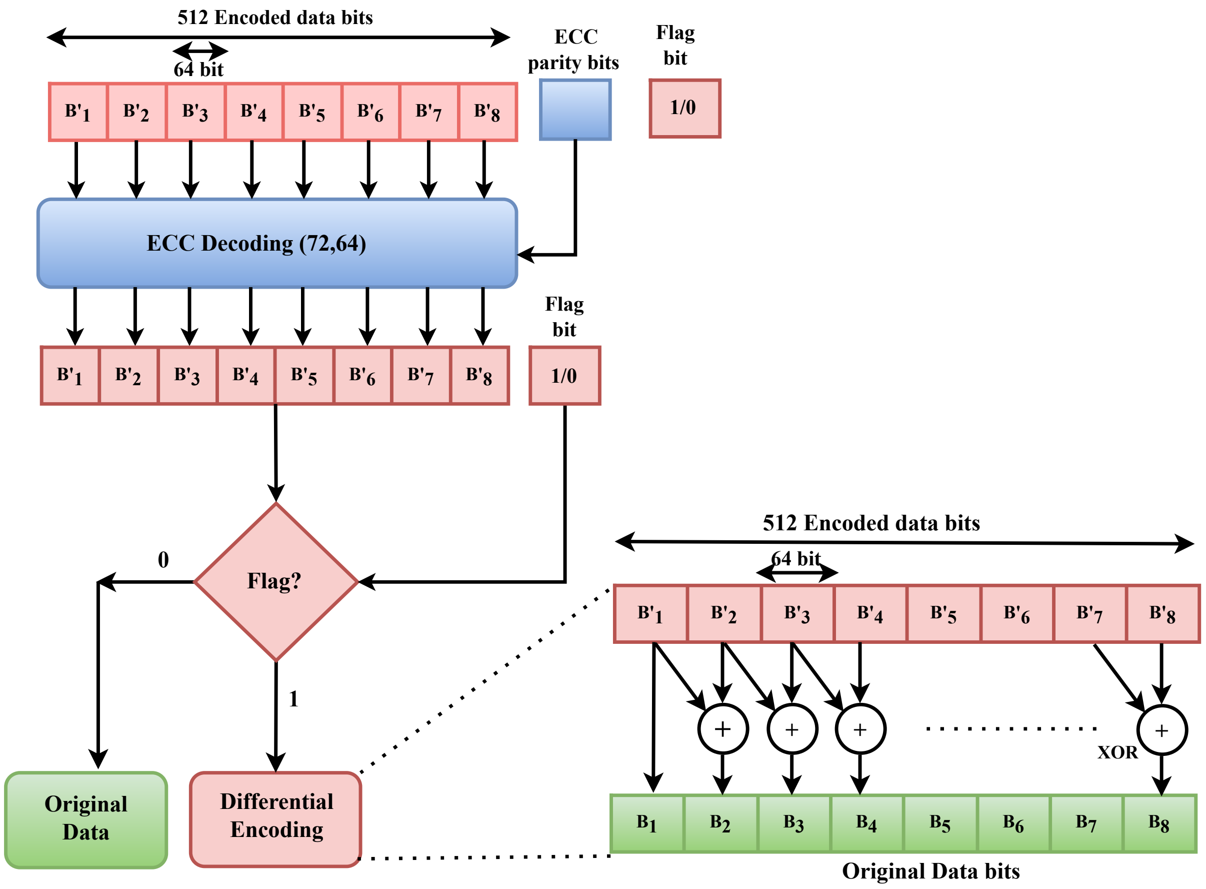


Figure 4.4.: Decoding process during the memory read operation

We use the (72,64) Hamming code for the 64-bit block. The adjusted block size becomes $k_{new} = 64+1 = 65$ upon incorporating a single flag bit. For such a scenario, a ($n = 127, k = 120$) SEC Hamming code with generator matrix ($[G]_{120\times127}$) can be used to derive a code dimension suitable for k_{new}=65 bits. This reduction involves eliminating 55 rows and columns from the $[G]_{120\times127}$, resulting in a new generator matrix $[G_{new}]_{72\times65}$. This process effectively tailors the code (n_{new}=72, $k_{new} = 65$) having single error correction capability. An additional parity bit is appended to obtain the SEC-DED code, which results in a ($n_{new} = 73$, $k_{new} = 65$) code, necessitating the same memory overhead as the $(72, 64)$ code and similar decoding complexity.

In the scenario where a word line comprises eight 64-bit blocks, totaling 512 bits, the strategy to protect a single flag bit involves employing the new code dimension of $(73, 65)$ for just one of these blocks. The remaining seven blocks can be safeguarded using the standard $(72, 64)$ SEC-DED Hamming code. For practicality and ease of implementation, it is advantageous to apply the $(73, 65)$ code dimension to the last block of the word line. The hardware overhead would be similar to the case of $(72, 64)$ SEC-DED, as it also needs eight parity bits and a similar decoding structure. The overview of the modified decoding structure is shown in Fig. 4.5.

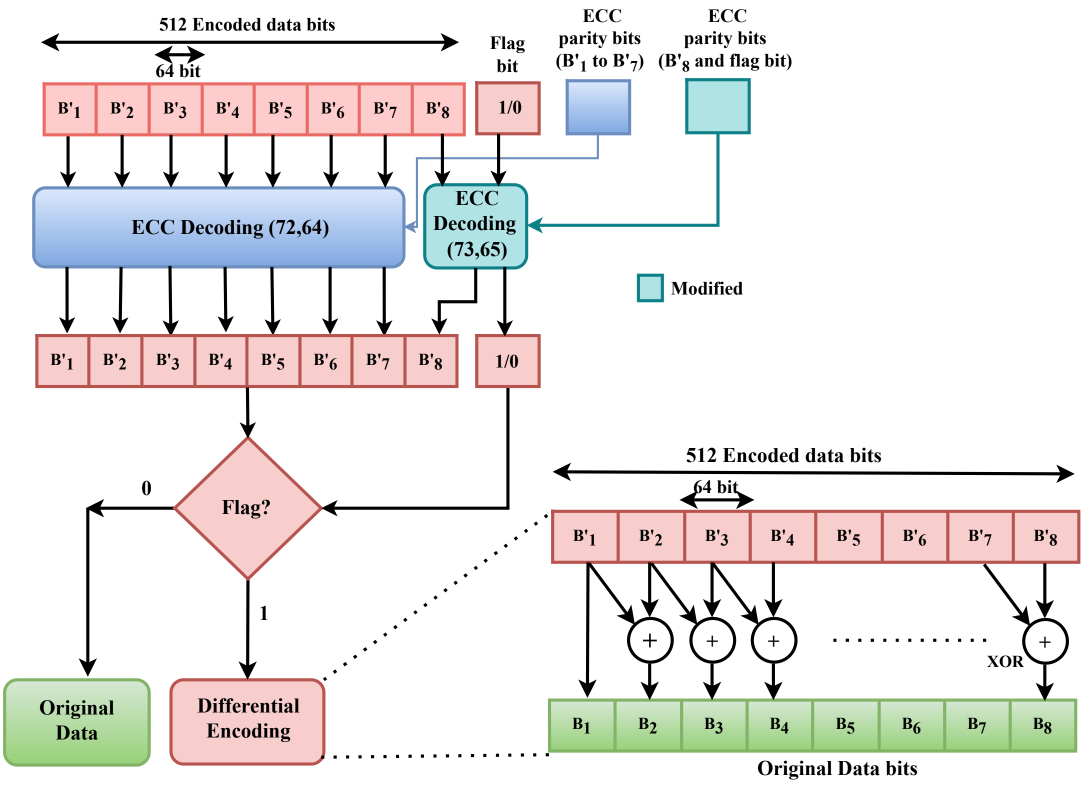


Figure 4.5.: Illustration of modified data decoding process

4.4.3. Adaptive error correction at the block level

In some applications, the data blocks within most word lines may possess a very low-value locality. This presents a significant challenge in reducing the Hamming weight. Fig. 4.6 illustrates the simulated Hamming weight distribution of encoded data for a few example benchmarks. The Hamming weight distribution of the original and encoded data shows substantial overlap, with a minimal shift towards the lower range, indicating a negligible reduction. It has also been observed in existing robust techniques for reducing the Hamming weight of data bits, such as the high memory overhead dynamic differential encoding [192], which uses multiple flag bits (12-bits) to store the encoding status, does not work effectively in such hard scenarios, as shown in Fig. 4.6. Therefore, using a high memory overhead solely for Hamming weight reduction could not be efficient in improving reliability.

To address such scenarios, we introduce an efficient adaptive ECC technique. This technique selectively applies a robust multi-bit ECC only to those blocks within a word line with high Hamming weight, even after an initial Hamming weight reduction, instead of using strong ECC for all the blocks. This approach is intended to reduce the memory and decoding overhead associated with the robust multi-bit ECC while enhancing memory reliability. Fig. 4.7 presents an overview of the adaptive ECC strategy, specifically devised to align with the varying Hamming weight observed at the block level in a word line. After initially reducing the Hamming weight at the word level, blocks that exhibit a Hamming weight exceeding a predefined block level threshold Hamming weight (HW_{th2}) receive protection via robust multi-bit ECC_2, a DEC-TED BCH code. Meanwhile, an SEC-DED Hamming code (ECC_1) protects blocks with lower Hamming weight. In the adaptive ECC, each block is accompanied by a flag bit to indicate the status of the ECC. The flag bits across all eight blocks of the word line are also protected through the application of shortened block codes.

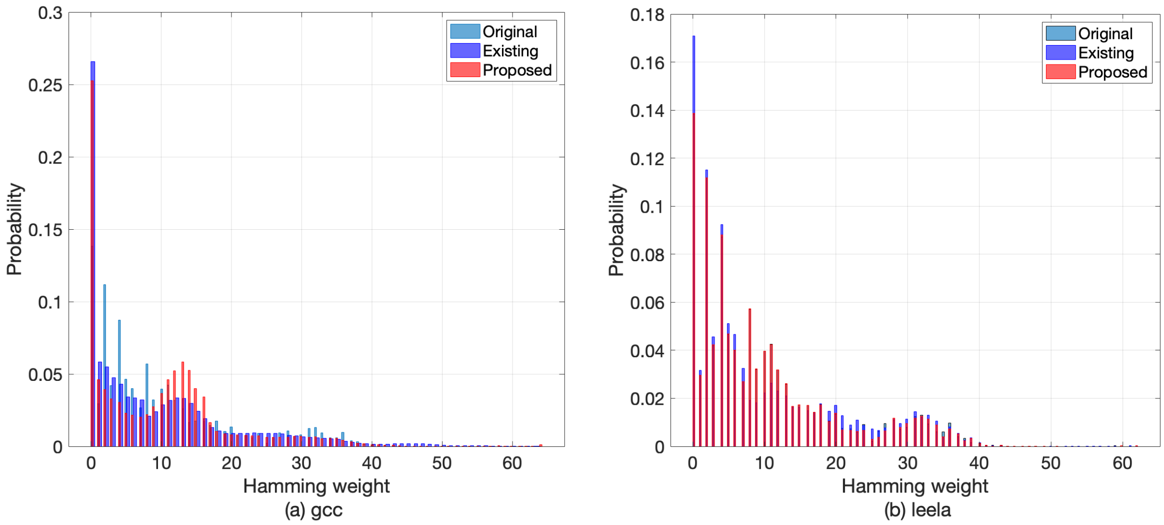


Figure 4.6.: An example case illustrating minimal reduction in Hamming weight

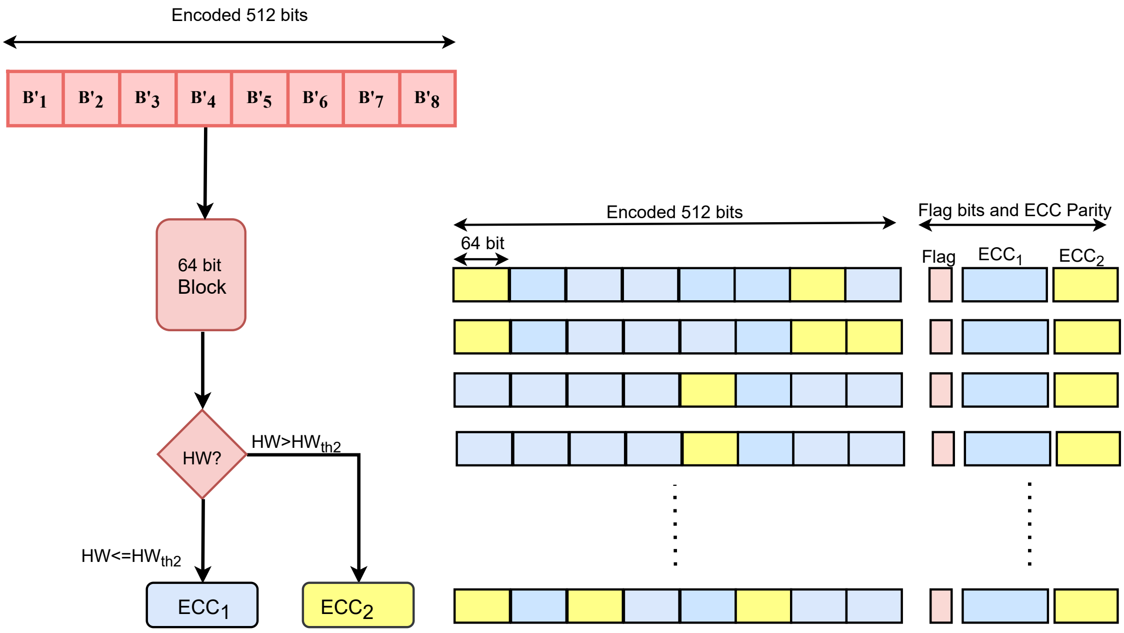


Figure 4.7.: Overview of the proposed Adaptive ECC strategy. HW: Hamming weight, HW_{th2}: block-level threshold

Determining the number of blocks in a word line that needs protection by ECC_2 is crucial. This can be achieved by analyzing the distribution of blocks whose Hamming weight exceeds the block-level Hamming weight threshold HW_{th2}. Fig. 4.8 represents the variation in the distribution of a number of blocks requiring ECC_2 with different HW_{th2}, following the initial Hamming weight reduction at the word level. The analysis reveals that the frequency of a number of blocks requiring ECC_2 is not uniform. The number of blocks requiring ECC_2 differs for different HW_{th2}. A lower HW_{th2} results in more blocks requiring ECC_2, while a higher HW_{th2} leads to fewer blocks with ECC_2. Additionally, a higher frequency is noted for up to two or three blocks, which progressively decreases as the number of blocks increases. This trend is even more evident with a higher HW_{th2}, as fewer blocks require ECC_2.

In most cases, protecting up to two or three blocks with strong ECC is sufficient to improve overall reliability. Nevertheless, there is a trade-off between the number of blocks protected by ECC_2, the memory overhead associated with ECC parity bits, and the overall improvement in reliability. The detailed study of the trade-off between memory overhead and reliability is discussed later in this chapter. This visualization helps in understanding the impact of threshold settings on the need for ECC_2 and facilitates decision-making for balancing reliability with ECC overhead. We present one example benchmark, and the distribution may differ slightly for other benchmarks, as it depends on the data correlation. However, a similar conclusion can also be drawn for other benchmarks.

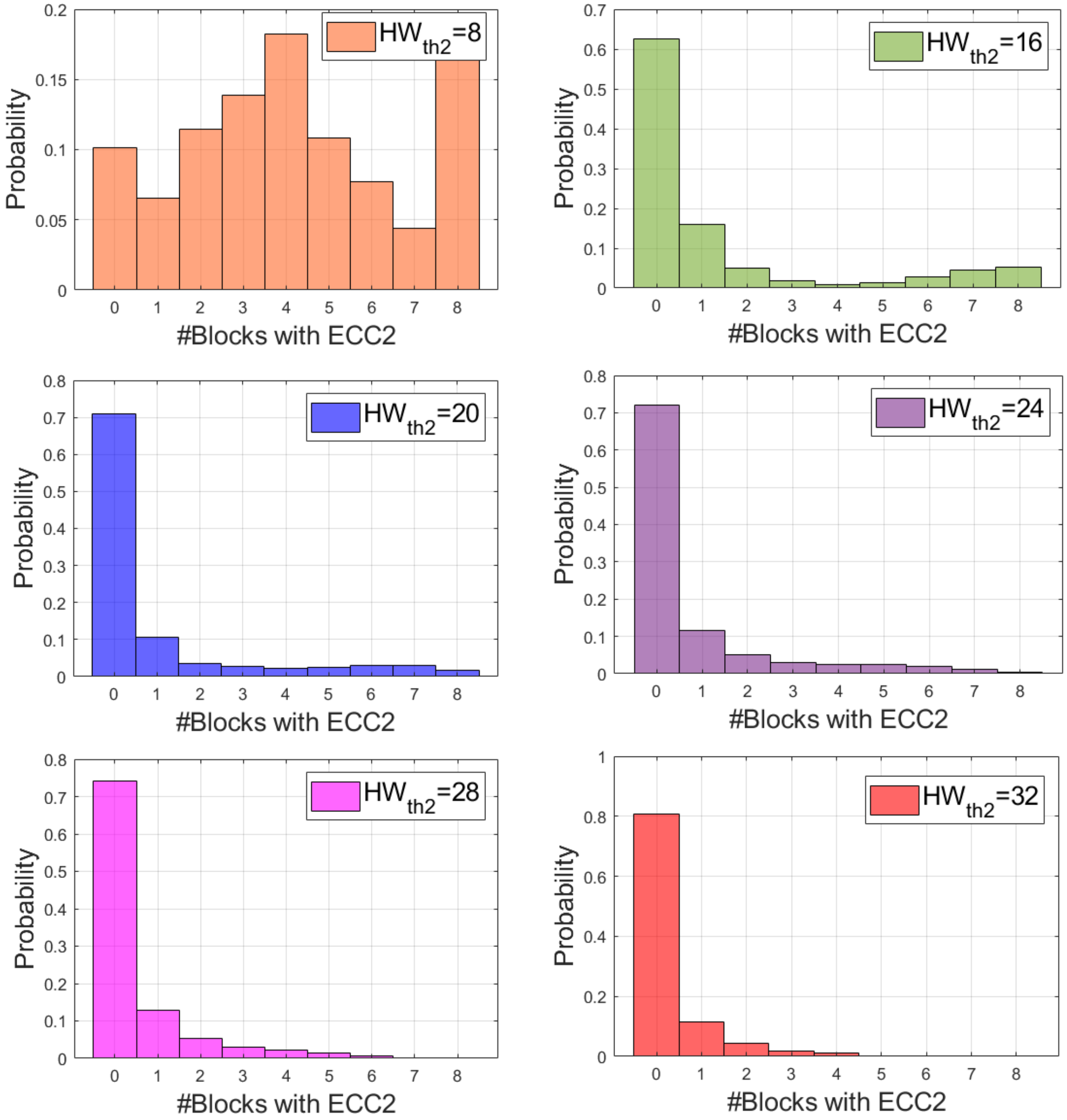


Figure 4.8.: Illustration of a number of blocks requiring protection by ECC_2 for different HW_{th2} for example benchmark:‘gcc’

4.5. Results and discussions

4.5.1. Simulation setup

To test our proposed strategy, we analyze the reliability of cache data of a wide range of applications from the SPEC CPU2017 benchmark suite [244]. The benchmarks have been chosen to take into account multiple factors: first, not selecting applications that behave too similarly, following indications from previous work [250]. Secondly, mixing benchmarks that stress the memory system as well as more compute-intensive ones [251]. Assuming STT-MRAM serves as the last-level L2 cache and the gem5 simulator [252] is used to produce the L2 cache traces, and the most representative workload segment is extracted through the Simpoint method [253].

The basic details about the evaluation methodology are illustrated in Table 4.1. We consider two different CPU configurations: in-order and out-of-order (OoO). Firstly, all the analyses are conducted for in-order CPUs, and the analysis related to out-of-order CPUs is presented in Section 4.5.6. We obtain the encoded cache data block for each benchmark using the proposed encoding scheme designed to minimize the Hamming weight. Subsequently, we calculate the probability of word failure (BER: block error rate) under the application of the Hamming code (ECC_1) for each cache line in the case of both encoded and original data blocks. We present the average BER for both cases in each benchmark suite. Finally, we evaluated our adaptive ECC approach by analyzing its impact on BER. Regarding the bit error rate, we consider the dominant 0 to 1 switching failure probability ($P_{b_{0\to1}}$) as a bit error rate to compute the BER [246]. Considering that currently manufactured STT-MRAM chips may exhibit varying bit error rates [254]. The bit error rate (P_b) range from 10^{-4}-10^{-5} is considered for the evaluation. However, the relative improvement in BER remains consistent across different bit error rates.

Table 4.1.: Simulation Details for Reliability Analysis

Benchmarks	602.gcc, 649.fotonik3d, 605.mcf, 654.roms, 638.imagick, 625.x264, 628.pop2, 641.leela, 623.xalancbmk, 607.cactusBSSN
CPU Model	ARM HPI (high performance in-order) ARM Out-of-Order (OoO) CPU Clock frequency: 2 GHz
L1I/SRAM	16 kB, Associativity:2, Read/Write cycles: 2/2
L1D/SRAM	16 kB, Associativity:4, Read/Write cycles: 2/2
L2/STT-MRAM	256 kB/1MB, 4/16 Bank, Assoc.16, Read/Write cycles: 12/20
Word/Cache line	512 -bits (64 Byte)
Data Block Size	64 bits (total eight blocks)

For a memory block of Hamming weight (HW), the block reliability (P_{Block}) under an asymmetric bit error rate (P_b) protected by t-bit error correction coding strategy can be given by Equation (4.2) [255]. The block reliability (P_{Block}) demonstrates the successful flipping of all the bits in a memory block. The block failure probability or BER (P_{BER}) can be computed as given by Equation (4.3).

$$P_{Block}(HW,t) = \sum_{i=0}^{t} \binom{HW}{i} P_{b_{0\to 1}}^{i} (1 - P_{b_{0\to 1}})^{HW-i} \tag{4.2}$$

$$P_{BER}(HW,t) = 1 - P_{Block}(HW,t) \tag{4.3}$$

$$P_{BER}(HW,t) = 1 - \prod_{i=1}^{N_B} P_{Block_i}(HW,t) \tag{4.4}$$

In general, ECC is applied at the block level within a large word line, where the word line is divided into multiple blocks and each block is protected by a dedicated ECC. In situations where a word/cache line is divided into multiple blocks (N_B), such as in our example with eight 64-bit blocks constituting a 512-bit word line, and where each block is safeguarded by its own dedicated ECC, the cache BER can be determined using the following approach as shown in Equation (4.4).

Moreover, we also present an analysis of the hardware overhead entailed by the proposed strategy. Our assessment centers on memory overhead, along with the overheads associated with encoding and decoding processes. The realization of this approach is executed using Verilog HDL and synthesized via the Synopsys Design Compiler, leveraging the Global Foundries 22nm technology library for this purpose.

4.5.2. Evaluation of Hamming weight reduction

We simulated the Hamming weight distribution of the cache data for different benchmarks using the proposed strategy. The Hamming weight threshold for the word line was established at $HW_{th1} = 185$. Fig. 4.9 and Fig. 4.10 illustrate the Hamming weight distribution at the word and block level granularity, respectively. The Hamming weight distribution shifts towards the region of lower Hamming weight, demonstrating the effectiveness of the proposed encoding strategy, which utilizes only a single flag bit. The benchmarks 'x264', 'pop2', 'roms', and 'imagick' demonstrate average Hamming weight reductions of 15%, 21%, 42%, and 64%, respectively. The more considerable reductions observed in 'roms' and 'imagick' are attributed to the higher correlation among the data blocks.

It has also been observed that setting a lower threshold in cases where benchmarks have a very high correlation among the data blocks, such as 'imagick', can result in a significant decrease in Hamming weight. Our simulation depicts that in the case of 'imagick' when the threshold value was lowered below HW_{th1}=120, the average Hamming weight was reduced by 70%. This reduction remained almost consistent up to HW_{th1}=85. Further threshold reduction led to an impressive decrease of nearly 87% in the Hamming weight of data bits.

We further assessed the performance of the proposed strategy against the existing robust dynamic differential encoding [192]. Fig. 4.11 represents the percentage reduction in Hamming weight across various benchmarks, demonstrating that our strategy is capable of reducing Hamming weight by as much as 64%, nearly matching the performance of the existing robust encoding method [192]. This demonstrates the efficiency of our proposed strategy, which achieves a notable reduction in Hamming weight using only a single flag bit (only 0.19% memory overhead), as opposed to the existing strategy that requires a 12-bit (2.35% memory overhead) for the flag bits.

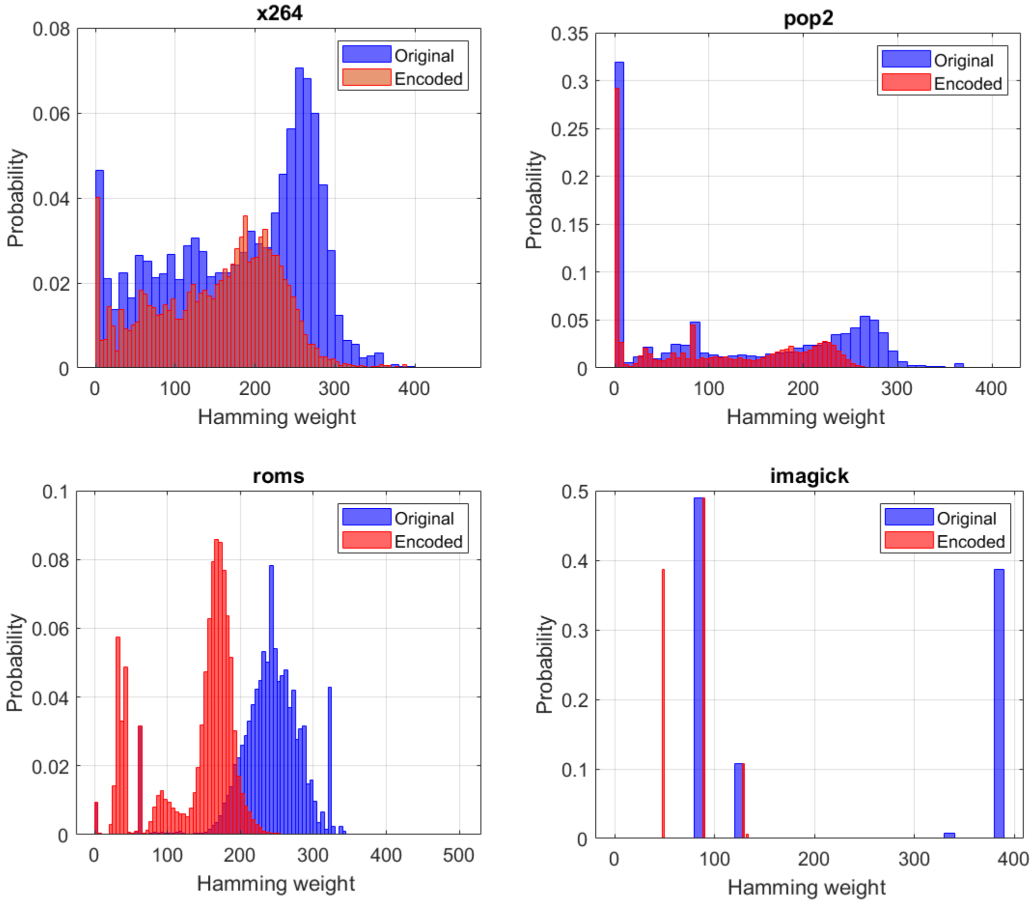


Figure 4.9.: Simulated Hamming weight distribution at the word level (512-bit) granularity

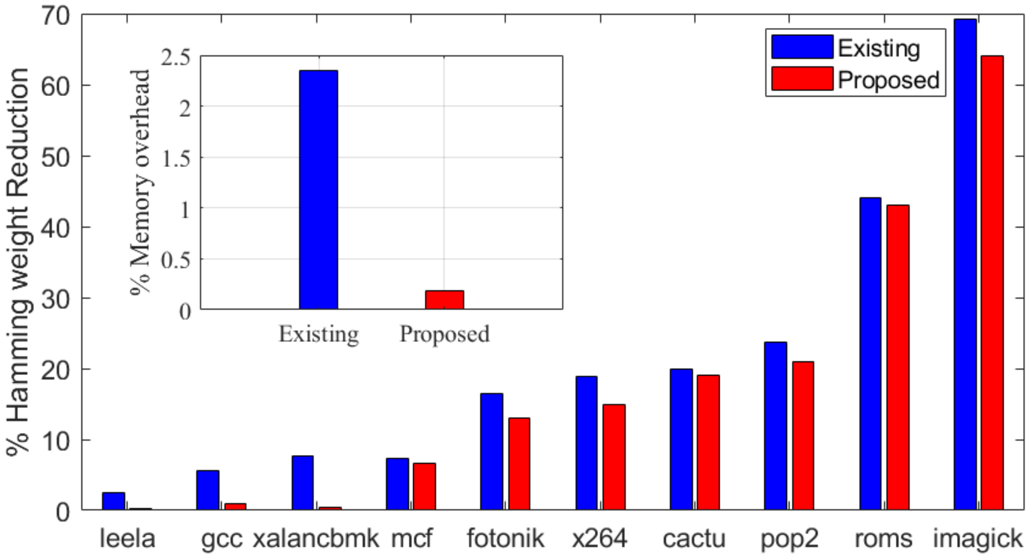


Figure 4.11.: Comparative analysis of Hamming weight reduction with existing strategy

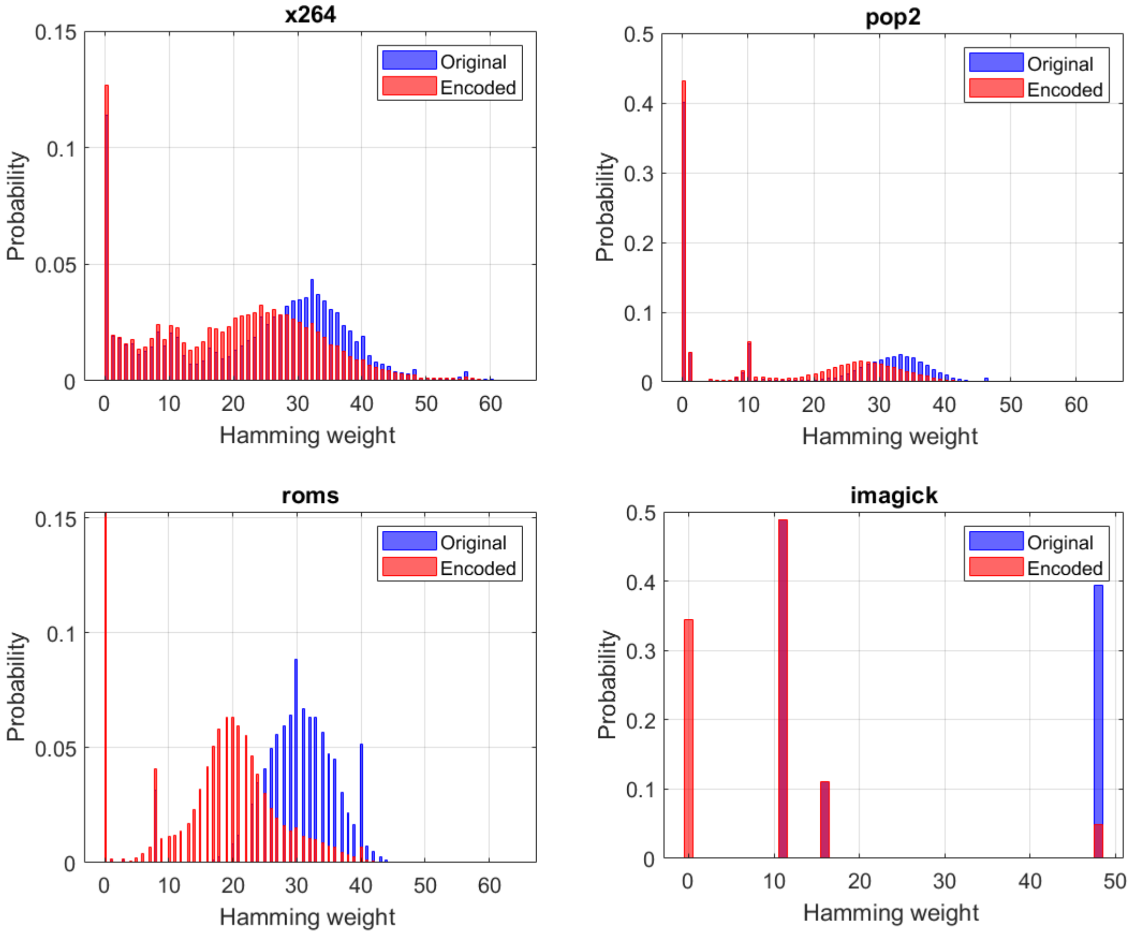
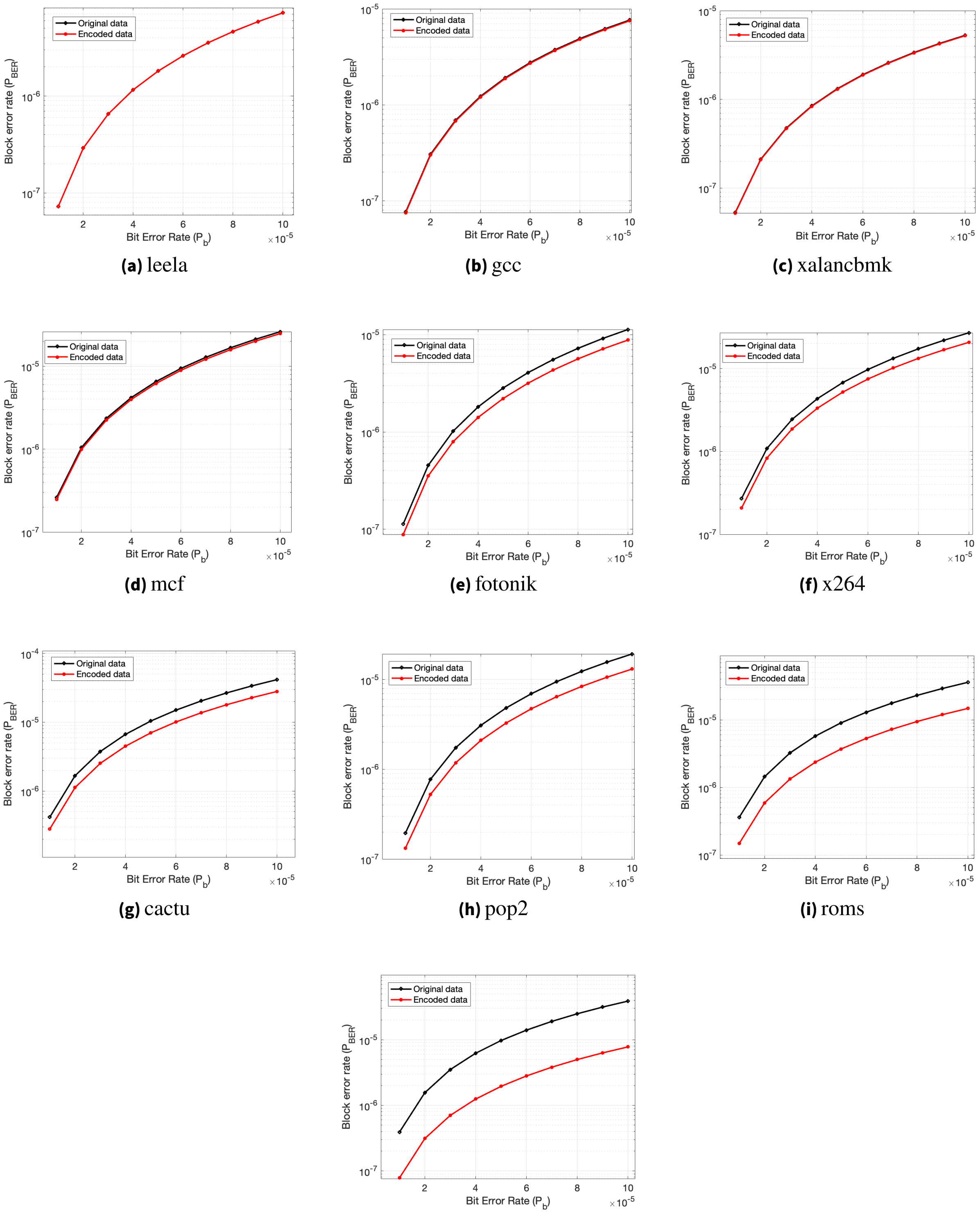


Figure 4.10.: Simulated Hamming weight distribution at the block level (64-bit) granularity

4.5.3. Reliability evaluation with SEC-DED Hamming code

The BER is a crucial metric for assessing memory reliability. To conduct a detailed and comprehensive assessment, we systematically calculate the BER for each word line of cache data across various applications for different asymmetric switching error rate scenarios. In this analysis, encoded data using the proposed strategy with the SEC-DED Hamming code is compared against the baseline original data with the SEC-DED Hamming code per 64-bit data in a word line.

Figure 4.12 illustrates the simulated BER across various applications. The proposed strategy significantly reduces the BER in the case of encoded data sets. The observed reduction in average block error rates was significant, up to 22% for ‘fotonik’, 24% for ‘x264’, 32% for ‘pop2’, 33% for ‘cactu’, 59% for ‘roms’, and 80% for ‘imagick’, thereby indicating notable enhancements in memory system reliability. Conversely, for some benchmarks like ‘leela,’ ‘gcc,’ and ‘xalancbmk,’ the improvement in error rates was relatively modest. This outcome was anticipated, given the negligible impact on Hamming weight reduction observed in these particular benchmarks.



(a) leela **(b)** gcc **(c)** xalancbmk

(d) mcf **(e)** fotonik **(f)** x264

(g) cactu **(h)** pop2 **(i)** roms

(j) imagick

Figure 4.12.: Simulated average block error rate using the proposed strategy with SEC-DED Hamming code for different $P_{b_{0\to1}}$

4.5.4. Hardware overhead with SEC-DED Hamming code

The proposed method employs the standard SEC-DED Hamming code per 64-bit data blocks, incurring a memory overhead of 12.5%. The SEC-DED encoder and decoder incur an area overhead of $134\mu m^2$ and $274\mu m^2$ for each block, with a latency of $145ps$ and $200ps$, respectively. The data encoding logic, which includes Hamming weight calculation and differential encoding, demands an area overhead of $1750\mu m^2$ and incurs a latency of $425ps$, as shown in Table 4.2(a). Remarkably, the encoding process imposes a minimal memory overhead (0.19%), leveraging merely a single flag bit. During critical read operations, the data decoding adds only $27ps$ to the inherent latency of SEC-DED decoding. The additional hardware overhead required for the data encoding and decoding logic to reduce the Hamming weight is minimal, $< 0.3\%$, normalized to the L2 cache area, estimated based on NVSim [256].

Table 4.2.: Hardware overhead of data encoding/decoding logic on top of ECC

(a) Hardware overhead with SEC-DED Hamming code

Hardware Logic	Area (μm^2)	Latency (ps)
Data Encoding	1750	425
Data Decoding	448	27

(b) Hardware overhead with Adaptive ECC

Hardware Logic	Area (μm^2)	Latency (ps)
Data Encoding	2938	685
Data Decoding	572	132

We also evaluate the impact of computational overhead incurred by the proposed strategy with SEC-DED on system performance. In this, we measure the system performance in terms of Instructions Per Cycle (IPC). Figure 4.13 shows the normalized IPC using the proposed scheme, with respect to the baseline ECC_1. The average IPC reduction across different benchmarks is just under 0.1%. This is because the proposed method adds only one extra cycle during writing, while the critical read cycle remains unchanged, as low-latency data decoding can be performed within the same cycle as ECC decoding. In this way, the critical decoding path remains unaffected, preserving system performance close to the baseline scenario, i.e., just ECC without extra logic.

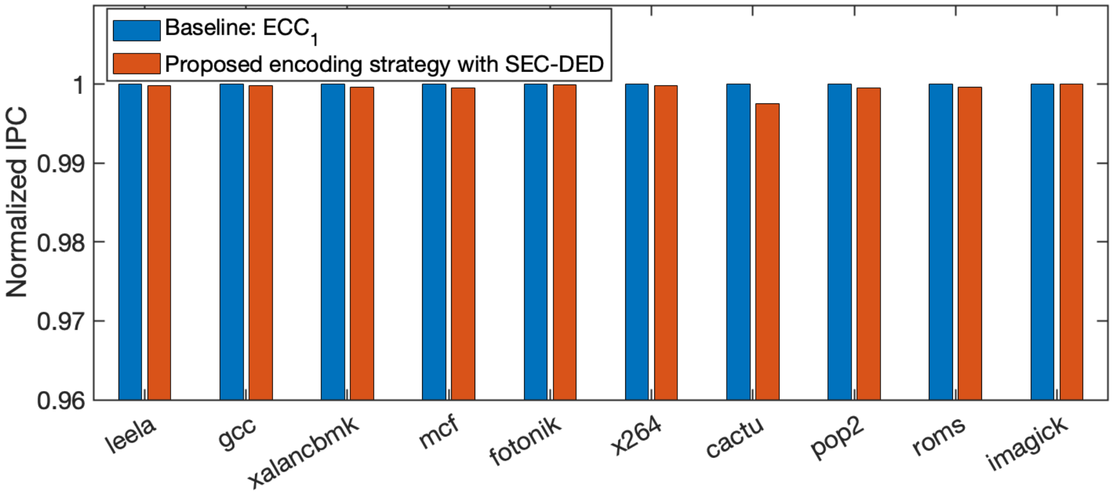
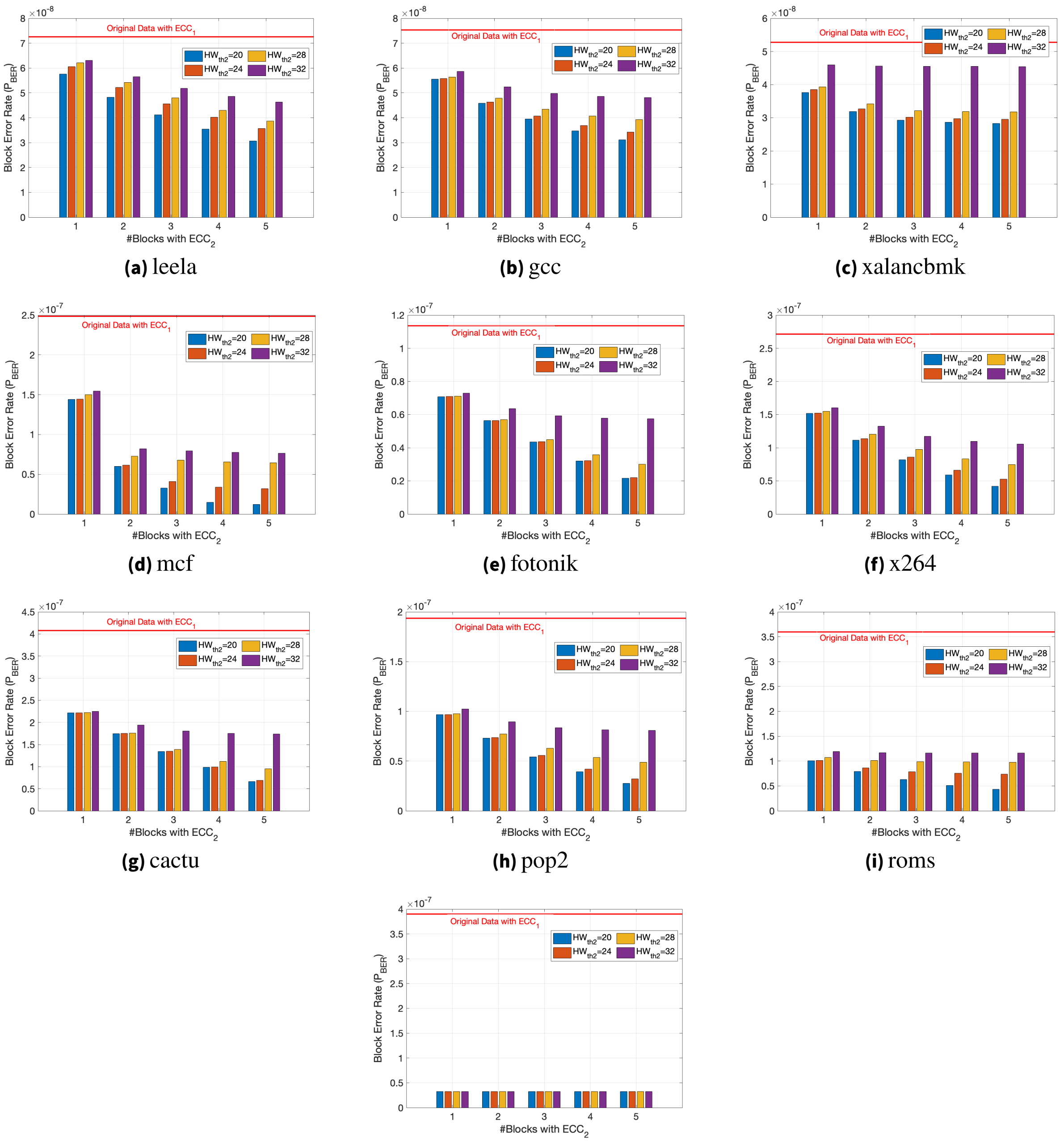


Figure 4.13.: Normalized IPC under the proposed strategy with SEC-DED ECC

4.5.5. Evaluation of the adaptive ECC technique

We investigate the reduction in BER with an adaptive ECC framework across various SPEC CPU2017 application benchmarks. Fig. 4.14 shows average BER under varying configurations where different numbers of blocks in a word line are protected by DEC-TED BCH code (ECC_2), considering different block-level threshold (HW_{th2}). As the number of blocks using ECC_2 increases, there is a noticeable reduction in the BER compared to the baseline original data with ECC_1. For instance, in the case of 'gcc' with HW_{th2}=24, the reduction in BER is 26%, 38%, 46%, 51%, and 55% with the number of blocks using ECC_2 being 1, 2, 3, 4, and 5, respectively. For benchmarks such as 'imagick', the BER reduction remains relatively consistent. This is because the encoding itself significantly reduces the Hamming weight, resulting in a minimal ECC_2 configuration. Moreover, the BER reduction slightly differs for different HW_{th2}. Because a high threshold results in a lesser number of word lines with blocks utilizing ECC_2, as shown in Fig. 4.15, which yields a smaller BER reduction. However, for benchmarks such as 'mcf' and 'imagick', the frequency of word lines with block utilizing ECC_2 follows nearly a similar trend even for higher HW_{th2}, as illustrated in Fig. 4.15. This consistency is due to the similar trend in the number of blocks utilizing ECC_2 per word line across various HW_{th2} values. Although a lower HW_{th2} may lead to a greater reduction in block error rate, it leads to higher memory overhead since it increases the number of blocks requiring ECC_2 in most of the word lines. It is adequate to use a block-level threshold of around HW_{th2}=24, or a maximum of up to 32, to achieve a satisfactory reduction in block error rate while keeping the memory overhead at a reasonable level. Fig. 4.16 illustrates the trade-off between reliability improvement (reduction in BER) and memory overhead with HW_{th2}=24. For example, in the case of mcf, the percentage reduction in BER is approximately 45%, 77%, 84%, 87%, corresponding to memory overheads of 15.62%, 17%, 18.35%, and 19.72%, respectively. Although it may appear intuitive that the reduction in BER increases with more blocks using ECC_2, it comes with a higher memory overhead. However, ECC parity bit requirement can be reduced by up to 33.3%, 27.5%, 21.67%, and 15.83% for 1, 2, 3, and 4 blocks, respectively, with adaptive ECC_2 configuration compared to applying uniform ECC configuration (uniform ECC_2 requires 23.4% of memory overhead for ECC parity bits).

In the case of an adaptive ECC strategy, the ECC_2 decoding logic is based on the work described in [177]. The ECC_2 encoding and decoding logic for a block results in an area overhead of $160\mu m^2$ and 2570 μm^2, and a latency of $300ps$ and $693ps$, respectively. The critical ECC decoding latency would correspond to ECC_2 logic. However, the average ECC decoding latency would be lower, as about 70% of word lines will not have blocks requiring ECC_2 decoding logic in most benchmarks, as shown in Fig. 4.15. The estimated hardware overhead for ECC_2 logic is based on prior work in [177], [257]. During data encoding, the selective differential encoding results in overhead similar to that detailed in Table. 4.2. The block-level Hamming weight calculation and additional logic are needed to determine which block needs to be triggered with ECC_2, altogether resulting in an area of $1188\mu m^2$ and a latency of $260ps$. The adaptive ECC supports up to N_B blocks with ECC_2. If the number of blocks requiring ECC_2 goes beyond N_B, the system is programmed to process only N_B blocks with ECC_2. We considered N_B=3; however, even if we increase N_B, the change in hardware overhead is negligible. During the data decoding, the additional hardware logic is required to identify the blocks that need ECC_2 decoding logic, resulting in an area overhead of $124\mu m^2$ and a latency of $105ps$. Finally, selective differential encoding is reapplied based on the flag bit to recover the original data, which results in an area of $448\mu m^2$ and a latency of $27ps$, similar to that detailed in Table 4.2(b). Overall, the additional hardware overhead needed for the data encoding and decoding logic in adaptive configuration is minimal, at under 0.5%, normalized to the L2 cache area, estimated based on NVSim [256]. Furthermore, the power-delay product caused by the data encoding and decoding logic in the adaptive ECC strategy is minimal, accounting for under 1%, normalized to the L2 cache access energy, estimated based on NVSim [256].



(a) leela **(b)** gcc **(c)** xalancbmk

(d) mcf **(e)** fotonik **(f)** x264

(g) cactu **(h)** pop2 **(i)** roms

(j) imagick

Figure 4.14.: Simulated average block error rate using adaptive ECC configuration for different benchmarks with $P_{b_{0\rightarrow 1}} = 10^{-5}$

We also assessed how computational overhead affects system performance, which is measured in terms of IPC. The data encoding logic accounted for one extra cycle during writing, in addition to ECC encoding. During the critical read phase, the data decoding logic incurs no additional cycles beyond those required for ECC decoding because its latency is quite low. As a result, it can be completed within the same cycle as ECC decoding. However, the average decoding latency is expected to be slightly lower, as not all word lines require ECC_2. Therefore, we consider both the scenario for adaptive ECC configuration: one with adaptive ECC_2 application and another worst case scenario in which ECC_2 latency applies to all word lines, even though not all word lines require ECC_2.

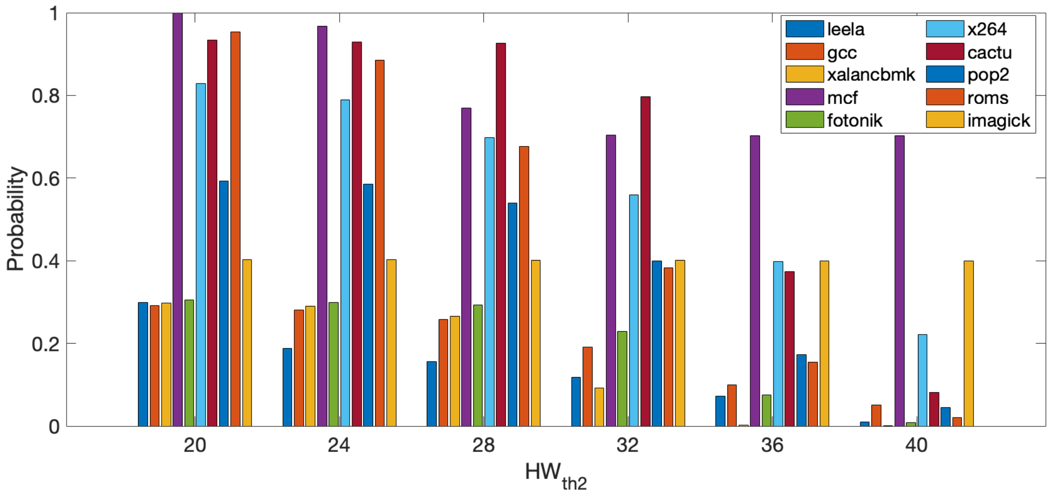


Figure 4.15.: Normalized distribution of Word line having Number of blocks utilizing ECC_2 for different HW_{th2}

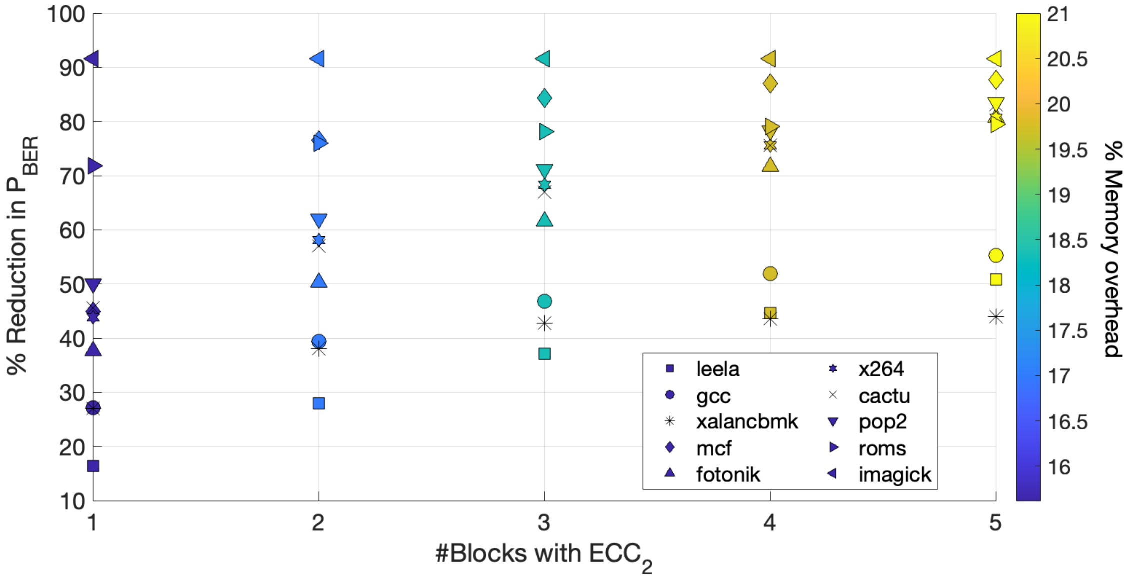
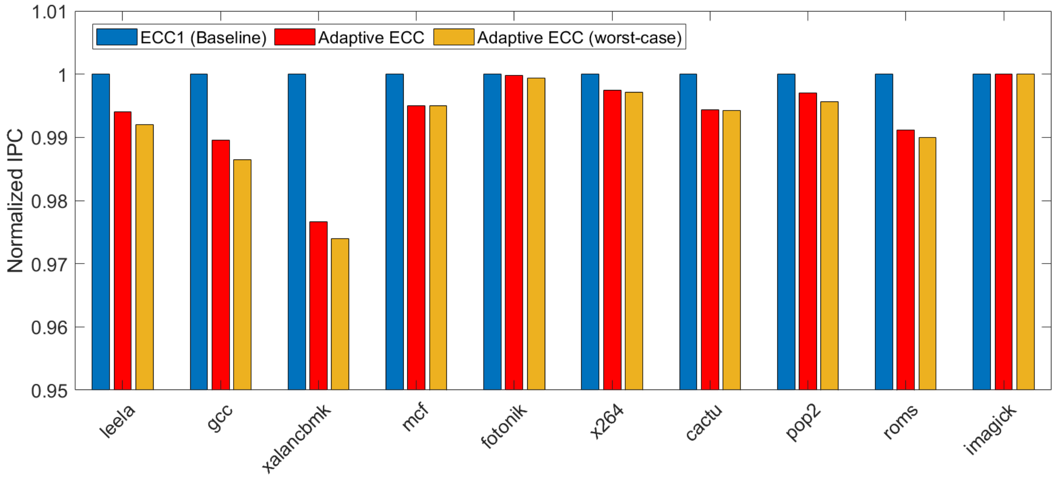
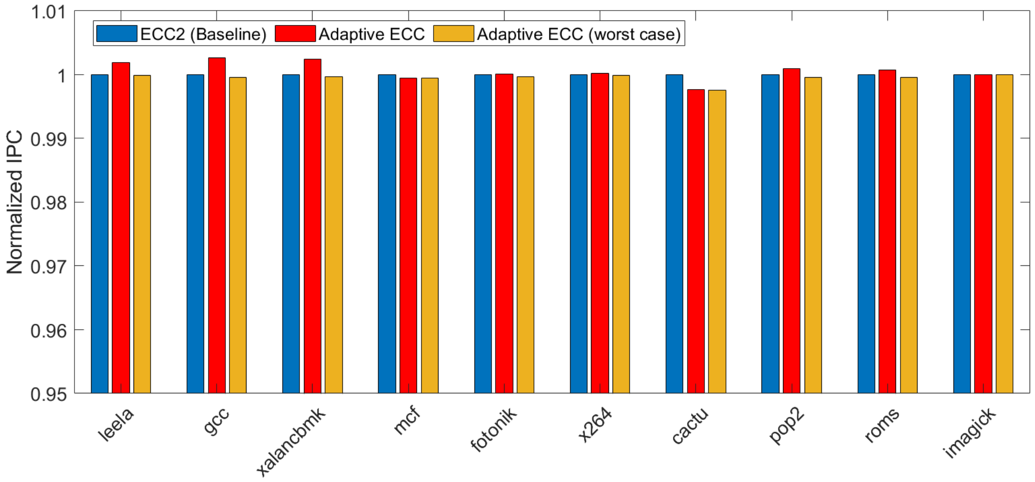


Figure 4.16.: Trade-off between reliability improvement and memory overhead for a threshold HW_{th2} = 24

Figure 4.17 presents the normalized IPC for various benchmarks using the proposed adaptive ECC scheme relative to baseline ECC_1 and ECC_2. Compared to the ECC_1 configuration, the average IPC reduction across different benchmarks is within 1%, whereas compared to ECC_2, the average IPC reduction across different benchmarks is also minimal, at less than 0.1%, resulting in negligible performance overhead. This is the worst-case scenario, considering similar decoding latency for ECC; however, there would be no overhead compared to uniform ECC_2, as shown in Figure 4.17(b), instead it would perform better, as not all cache lines require ECC_2 decoding in the case of an adaptive ECC configuration, as shown in Figure 4.15.



(a) Normalized IPC under the adaptive ECC configuration compared to ECC_1

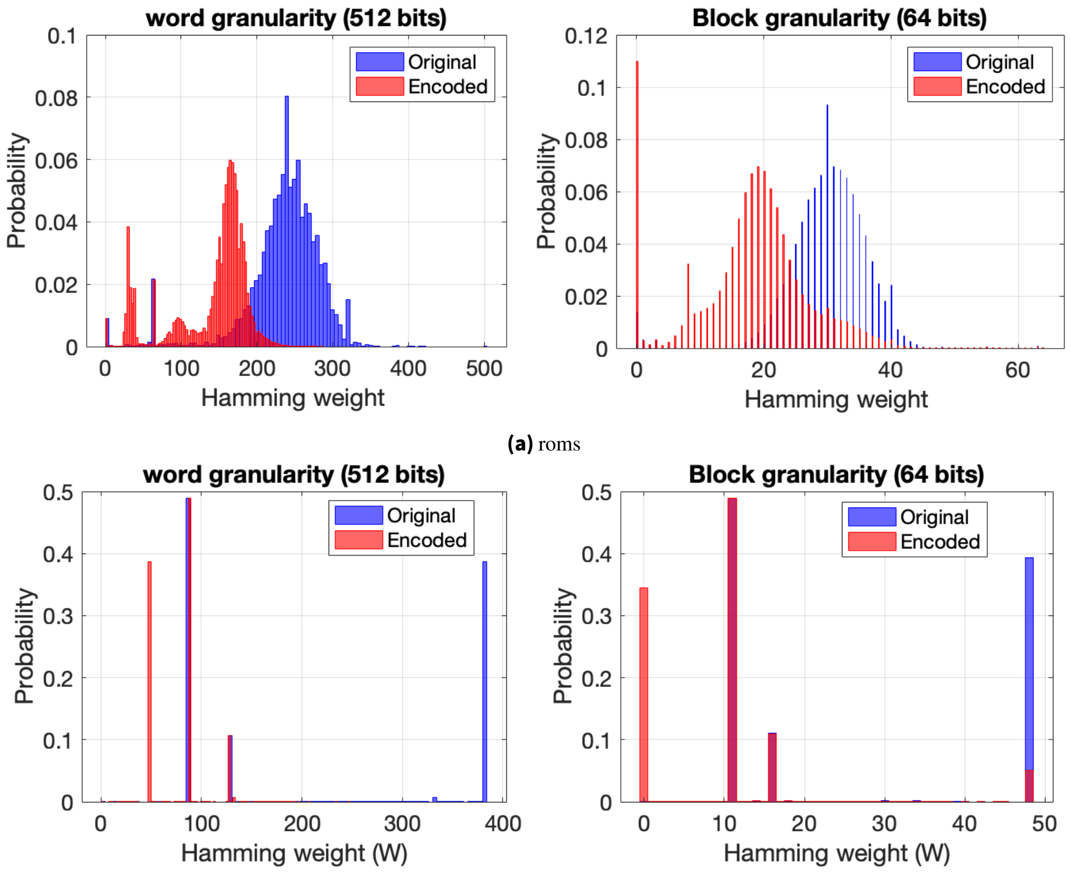


(b) Normalized IPC under the adaptive ECC configuration compared to ECC_2

Figure 4.17.: Comparison of normalized IPC and another metric under adaptive ECC configuration.

4.5.6. Evaluation with out-of-order CPU configuration

We expanded our evaluation to the OoO CPU with an L2 cache size of 1MB organized in 16 Banks while keeping the other configurations as outlined in Table 4.1. Figure 4.18 shows the Hamming weight distribution for a few example SPEC CPU 2017 benchmarks. The Hamming weight distribution notably shifts towards the left after the application of the proposed encoding strategy, highlighting its effectiveness. Furthermore, Figure 4.19 displays the percentage of Hamming weight reduction across some example benchmarks, which exhibits an almost similar trend to the findings discussed in Section 4.5.2.



(a) roms

(b) imagick

Figure 4.18.: Normalized distribution of Hamming weight of cache data (OoO CPU)

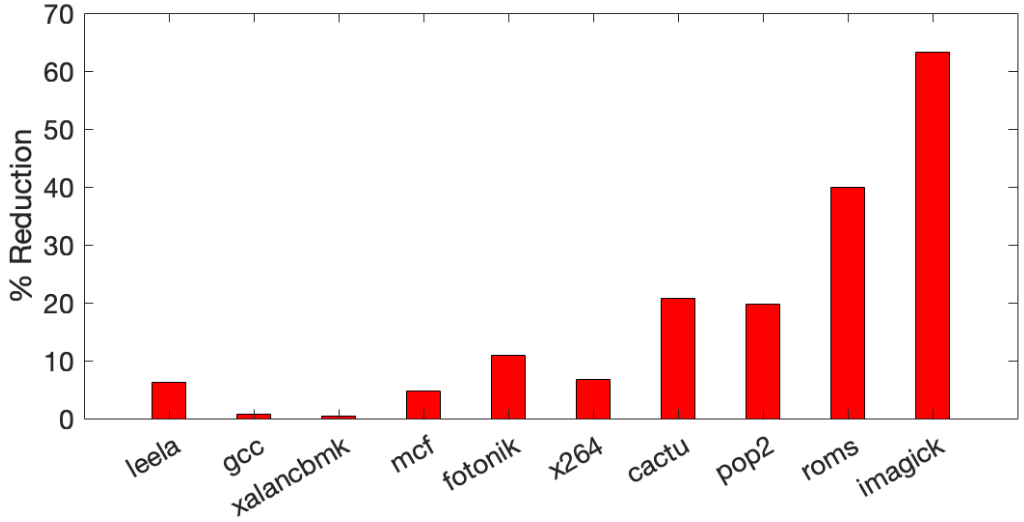


Figure 4.19.: % Hamming weight reduction (OoO CPU)

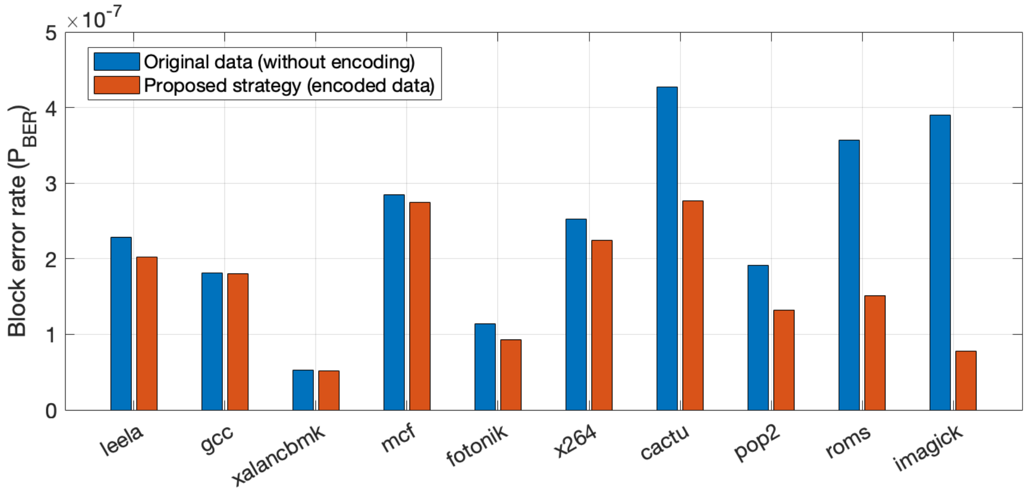


Figure 4.20.: Simulated average block error rate using the proposed strategy with SEC-DED for $P_{b_{0 \to 1}}$=10^{-05} (OoO CPU)

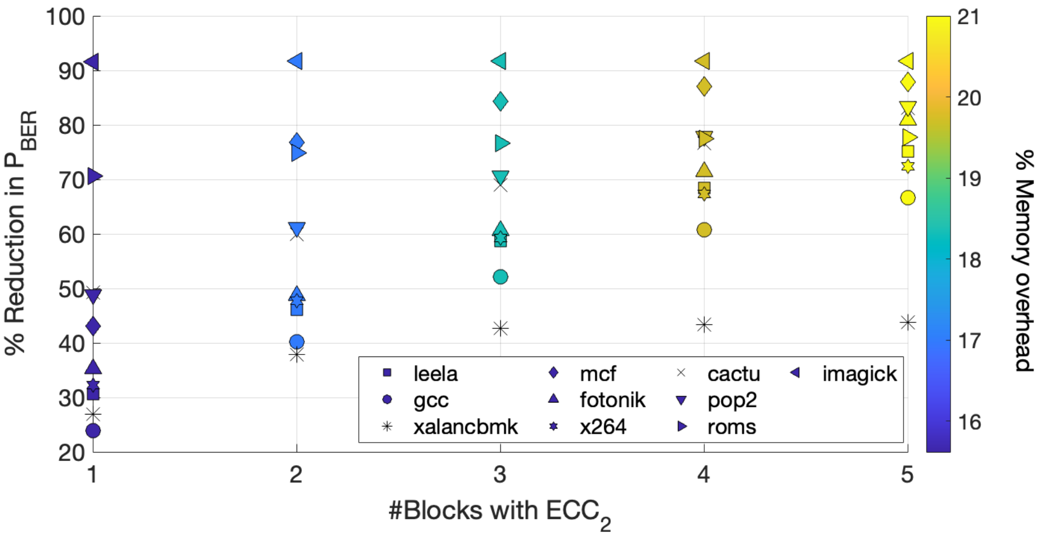
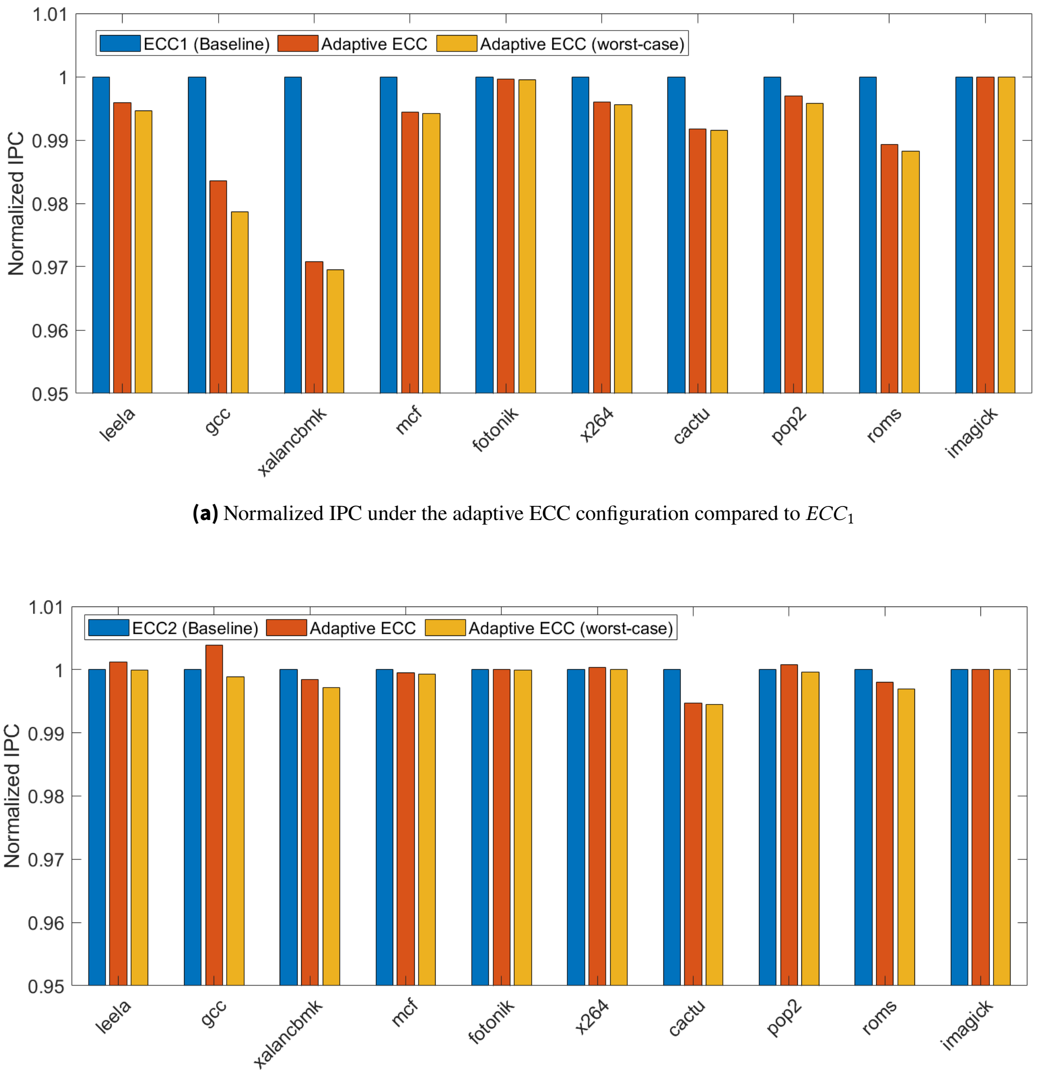


Figure 4.21.: Trade-off between reliability and memory overhead for HW_{th2}=24 in case of proposed adaptive ECC (OoO CPU)

Figure 4.20 demonstrates the simulated average BER using the proposed encoding strategy with SEC-DED. The observed reduction in average BER was significant, up to 11% for ‘x264’, 35% for ‘cactu’, 31% for ‘pop2’, 58% for ‘roms’, and 80% for ‘imagick’, respectively. Conversely, for some benchmarks, the improvement was relatively modest due to minimal impact on Hamming weight reduction, as we discussed previously. This is further addressed by the proposed adaptive strategy, as shown in Figure 4.21. For example, in the case of mcf, the percentage reduction in BER is approximately 43%, 77%, 84%, 87%, corresponding to memory overheads of 15.62%, 17%, 18.35%, and 19.72%, respectively. Similar reductions in BER are also evident in other benchmarks. Moreover, we also evaluate the impact on system performance in the case of OoO CPU configuration. Figure 4.22 illustrates the normalized IPC using the proposed adaptive ECC compared to the baseline ECC_1 and ECC_2. The average IPC reduction across all the benchmarks is under 1% compared to ECC_1. The average IPC reduction is less than 0.15% compared to the baseline ECC_2, signifying negligible impacts on system performance. This is the worst-case scenario, considering similar decoding latency for ECC; however, there would be no overhead compared to uniform ECC_2, as shown in Figure 4.22(b), as not all cache lines require ECC_2 decoding.



(a) Normalized IPC under the adaptive ECC configuration compared to ECC_1

(b) Normalized IPC under the adaptive ECC configuration compared to ECC_2

Figure 4.22.: Normalized IPC under the proposed adaptive ECC strategy (OoO CPU configuration)

4.6. Summary

This chapter presents an efficient method for facilitating asymmetric and adaptive error correction in STT-MRAM. The proposed approach leverages the Hamming weight reduction strategy on top of standard SEC-DED to mitigate the predominant asymmetric switching failure rate while imposing minimal additional memory overhead for flag bits, i.e., just a single bit per word line. Furthermore, we have developed an adaptive ECC configuration that avoids the need for strong multi-bit ECC uniformly, thereby improving reliability when Hamming weight reduction is ineffective. The adaptive ECC approach significantly improves reliability without imposing substantial hardware overhead and impacting system performance, as demonstrated across different SPEC CPU2017 benchmark suites. This presents a strong argument for boosting the operational reliability of STT-MRAM. The proposed method is highly adaptable and can also be applied to other types of memories.

5. Location-Aware Non-Uniform Error Correction in STT-MRAM: Mitigating Interconnect-Induced Reliability Challenges

The reliability challenges of STT-MRAM become more pronounced as technology scales down, primarily due to the increasing impact of interconnect parasitic resistive-capacitive (RC) effects. In this chapter, we propose an efficient location-aware non-uniform error correction strategy for mitigating the impact of interconnect parasitics on the reliability of STT-MRAM bit cells. By applying a non-uniform error correction mechanism across different memory zones, our approach increases correction strength with distance from the driver. The proposed approach eliminates the need for uniformly strong error correction across the entire memory zone, thereby reducing ECC parity bit memory overhead while enhancing reliability in the vulnerable memory zone. The efficacy of the proposed strategy is tested by reliability enhancement, measured by a cache word/block error rate, across the last-level cache data for various SPEC CPU2017 benchmarks [244]. Additionally, we also conduct a system-level evaluation to assess the impact of our proposed strategy on overall system performance, measured by IPC.

5.1. Introduction

The reliability challenges of STT-MRAM macro as a resistive memory are further aggravated as interconnects become more critical with technology scaling, primarily due to the growing impact of resistive-capacitive parasitic effects [258]–[260]. The parasitic resistance of signal lines (word, bit, and source line) worsens cell reliability [261]. This is problematic for cells farther from the driver, where the voltage drop caused by parasitic resistance is more pronounced. Therefore, the failure probability is more likely to gradually increase as the distance of word lines (rows) from the write driver increases. This highlights the need for a non-uniform error correction mechanism tailored to the varying susceptibility of memory rows based on their proximity to the write driver.

In the literature, several studies based on ECC have been reported to improve the reliability of STT-MRAM by addressing different failures. Existing reliability improvement techniques have attempted to enhance various aspects of reliability metrics. However, prior works mainly consider the failure behavior at either the device level or in a workload-aware manner. These strategies lack consideration of the impacts of interconnect parasitics on bit-cell reliability at the memory array level. Additionally, some foundry works have also explored device and circuit-level strategies [28], [261], [262]. The work in [28] attempted to reduce parasitic resistance by employing a two-column common-source line. Similarly, the work in [262] employs an eight-column common source line and a hybrid resistance reference. These strategies aimed to optimize STT-MRAM performance at the circuit and design levels; however, they do not explicitly target the error correction in the most error-prone regions. The recent work in [261] employed design-process-test co-optimization to improve STT-MRAM reliability, demonstrating that most test failures occur farther from the driver, highlighting the need for location-aware error correction.

In this chapter, we propose an efficient, location-aware error-correction strategy to mitigate random failures and interconnect-induced reliability degradation in STT-MRAM, enhancing overall reliability. The proposed strategy offers robust protection against location-dependent failures, where the failure probability increases with distance from the driver due to voltage drops resulting from interconnect resistance. The proposed strategy divides the memory subarray into different zones based on the proximity of rows to the driver. Each zone is protected by ECC of varying strength, with stronger error correction applied to rows farther from the driver, which are more vulnerable to errors. The proposed approach eliminates the need for uniformly strong ECC across the entire memory by strategically applying ECC of varying error correction levels, thereby optimizing both reliability and ECC parity overhead. We evaluate the reliability metric by simulating the memory word/cache block error rates (i.e., the probability of word or cache block failure) under the presence of the proposed strategy. Additionally, we thoroughly examine the trade-offs between ECC parity overhead and varying error correction mechanisms across different memory zones. Furthermore, we perform detailed system-level performance analysis with STT-MRAM as the last-level (L2) cache and simulate realistic workloads from the SPEC CPU2017 benchmark suite [244]. The system-level simulation results indicate that our strategy has a negligible impact on system performance, as measured by IPC.

5.2. Motivation and Problem Definition

In STT-MRAM, errors stem from stochastic switching, process variations, and manufacturing defects, which can affect cell reliability, including weakening writability and readability. These reliability concerns become more dominant depending on the row's location within the subarray due to interconnect parasitics. Fig. 5.1 shows the STT-MRAM subarray with peripheral blocks. As the distance from the write driver increases, parasitic resistance leads to significant voltage drops in the signal lines, making cells more prone to failures. An STT-MRAM subarray with interconnect parasitics is simulated using the parameters shown in Table 5.1. We consider a perpendicular magnetic anisotropy MTJ [263]. The memory simulator NVSim [256] is used to compute RC parasitics, following the configuration outlined in Table 5.1(a). The switching failure of the worst-case (last) cell in a few rows is simulated using $5K$ Monte Carlo (MC) SPICE runs at different switching voltages. Conducting a large MC run, such as $100K$, to obtain the typical failure rate is highly time-consuming. To address this, the observed failures are correlated with the corresponding cell current obtained with a single-run SPICE simulation, which is expected to vary across rows due to parasitic resistance. An exponential model is then applied to extrapolate the failure probability based on the current vs. failure trend, as shown in Fig. 5.2. The switching failures in the row gradually increase with distance from the write driver and become more severe in the farthest group of rows (last ~64 rows), as shown in Fig. 5.3. Failure is normalized to the farthest row from the driver. Although the magnitude of failure may vary with different simulation setup parameters, the trend remains the same: a higher failure probability as we move away from the driver.

Additionally, we demonstrate the impact of interconnect parasitics on switching failure probability using experimentally measured datasets. The measurement setup considers the MTJ size of $60nm$, a pitch size of $200nm$, and a switching pulse duration of $10ns$. Fig. 5.4 shows the switching failure probability for four different chips ($192kb$ are being measured in each chip). The subarray is divided into four blocks, each containing an equal number of rows. The $Block_1$ is close to the write driver, while the $Block_4$ is farthest. It is noticeable that the failure gradually increases as we move away from the driver. This trend is even acknowledged in recent studies by GlobalFoundries [261], which indicate that most failures occur in cells farther from the driver in a $40Mb$ fabricated STT-MRAM chip. These failures were attributed to voltage drop due to increased interconnect resistance.

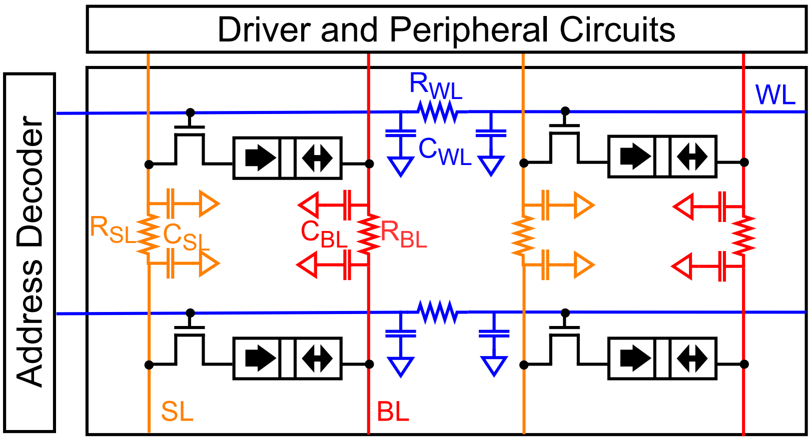


Figure 5.1.: A Subarray structure of STT-MRAM memory macro. WL: Word line, SL: Select line, BL: Bit line

Table 5.1.: Technology parameters for simulating switching failures. RA: Resistance-Area product, TMR: Tunneling-Magnetoresistance, t_{ox}: oxide layer thickness, t_{sl}: Free-layer thickness, r: MTJ radius. For interconnect parasitic calculation using NVSim, we consider bit cell area of 0.046 μm^2 [264] and cell aspect ratio of 2.

(a) Technology specification and MTJ parameters.

Node	VDD	Pulse	RA	TMR	t_{ox}	t_{sl}	r
$22nm$	0.9	$10ns$	7.5	2	$0.8nm$	$1.2nm$	$20nm$

(b) Interconnect specification (R:Ω, C: aF).

Line	WL		BL		SL	
Metal Width	$44nm$		$90nm$		$90nm$	
Parasitic	R_{WL}	C_{WL}	R_{BL}	C_{BL}	R_{SL}	C_{SL}
	1.5Ω	$33aF$	0.7Ω	$60aF$	0.7Ω	$65aF$

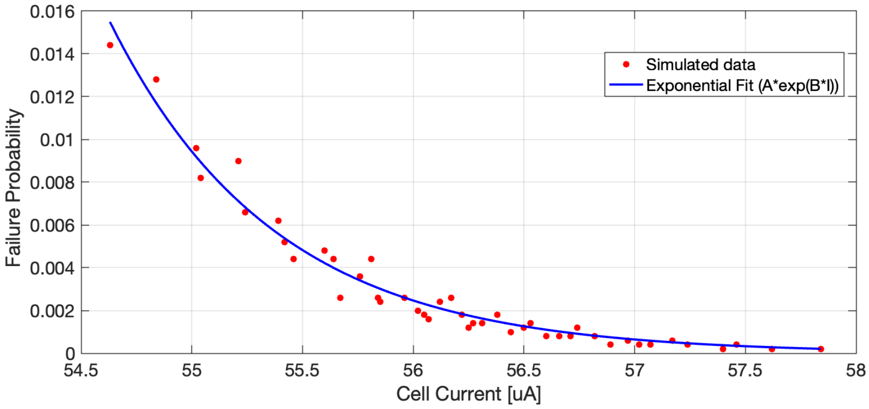


Figure 5.2.: Switching failure trend vs bit cell current

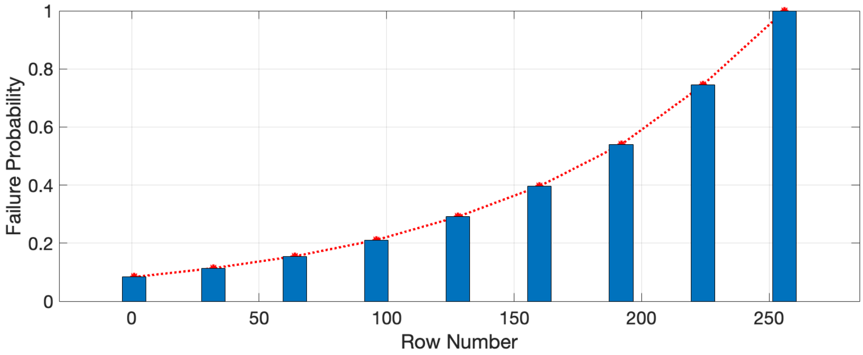


Figure 5.3.: Trend of switching failure with respect to row distance from the driver

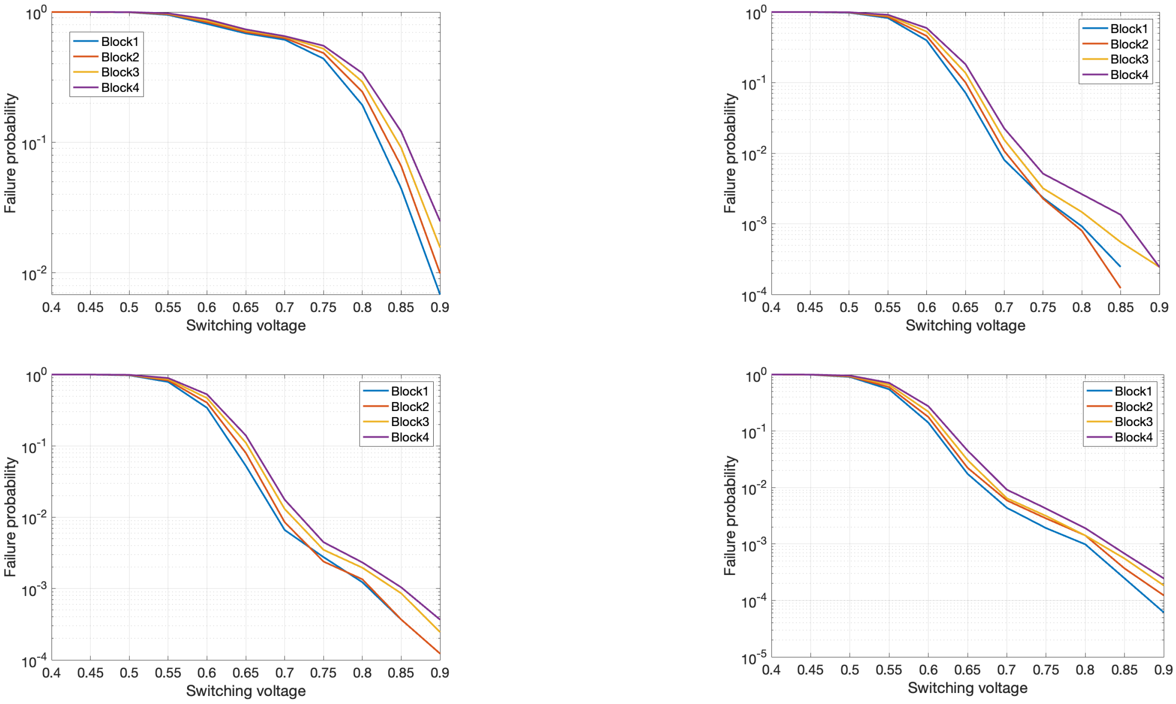


Figure 5.4.: Switching failure probability for four different chips

Additionally, the read reliability of STT-MRAM can also be influenced by interconnect parasitics. These interconnect parasitics can affect the read window during memory access and, similar to their impact during switching, they can degrade the applied read voltage. Existing studies from foundries indicate that read failures increase due to the effects of interconnect parasitics [261], [262]. This trend mirrors that of switching, where a greater distance from the driver results in a higher likelihood of failure [261]. The increased failure rate in cells located farther from the driver increases the probability of cache block failure, as depicted by the cache block error rate (BER: P_{Block}), detailed in Section. Conventional ECC offers the same level of error correction to all the rows in a subarray. However, using uniform ECC with a higher error correction strength for all rows to cover the worst-case row scenario is too costly in terms of memory overhead and decoding overhead, as not all rows require enhanced error correction. This underscores the need for non-uniform ECC that not only corrects random bit failures across the array but is also tailored to mitigate the extra reliability degradation induced by interconnect parasitics in rows farthest from the driver.

5.3. Memory Organization

The hierarchical breakdown of a typical STT-MRAM organization, as shown in Fig. 5.5(a), has a bank as the top-level unit (a fully functional memory unit that operates independently). Each bank has multiple mat units that can be operated simultaneously to serve a required cache access request, and within each mat, there are multiple memory subarrays [256]. In the experimental and simulation setup section of this chapter, an example 1 *MB* STT-MRAM last-level cache is integrated into an ARM CPU. The hierarchical breakdown of the considered STT-MRAM, shown in Fig. 5.5(a), consists of 16 banks, with each bank containing four mat units, and each mat unit containing two subarrays. Accessing a 64-byte cache block requires activating a group of mats and a single subarray in each activated mat, with each subarray providing 32 bytes. A unique 14-bit internal address is used to identify each cache block at its physical location in the cache memory, as shown in Fig. 5.5(b). This address is derived from a subset of the 40-bit physical address, which may vary across architectures, along with additional way bits obtained from the tag comparison logic, as the cache follows a 16-way set associativity. Column bits are not needed since the entire row is accessed. However, if only a subset is accessed, column bits are used, adjusting the number of banks, mats, and subarrays to align with the internal address bits.

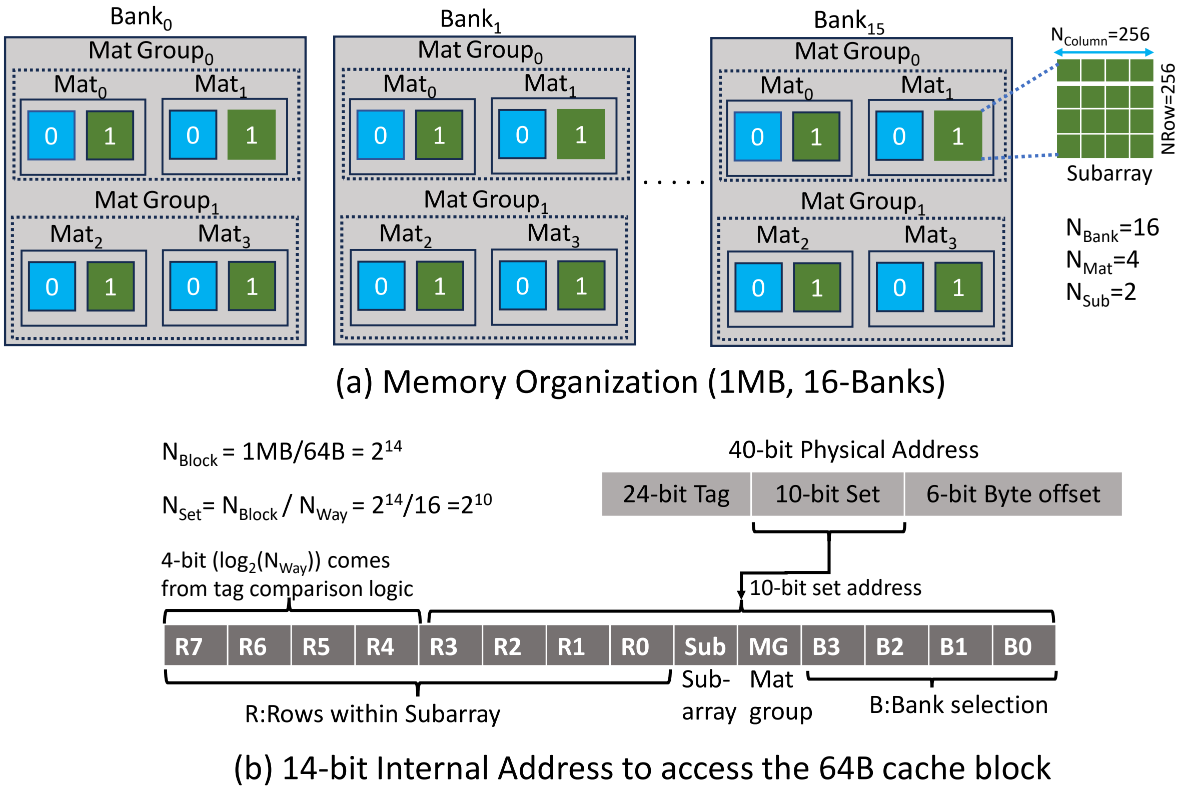


Figure 5.5.: Example 1*MB* STT-MRAM organization

5.4. Location-Aware Non-Uniform Error Correction Strategy

The proposed strategy provides non-uniform error correction by dividing the memory subarray into multiple zones (M-zones), with each zone protected by a different ECC strength. We partition the N_{Row} rows into M disjoint zones $Z_0, Z_1, \ldots, Z_{M-1}$, with $\sum_{i=0}^{M-1} |Z_i| = N_{Row}$ and order them from the cold zone Z_0 (group of rows nearest the driver) to the hot zone Z_{M-1} (group of rows farthest from the driver). Our goal is to address both random and location-dependent failures by using simpler ECC in the cold zone Z_0 and stronger ECC in the hot zone Z_{M-1} to mitigate the increased failure probability caused by interconnect parasitic resistance. The hot zone comprises roughly 0.25% of the array, i.e., the last ~64 farthest rows from the driver, which are most prone to failure. However, there is a trade-off between the number of rows in the hot zone and memory overhead for ECC parity bits. The intermediate zones ("warm zones") $Z_1, \ldots, Z_{m-1}$ can share ECC of intermediate strength, enabling a fine-grained trade-off between reliability and ECC parity-bit overhead. We propose two variants of the proposed strategy to improve the STT-MRAM reliability:

- **SEC/SEC-DED with distinct data block sizes**: Reducing the ECC data block size within the word line can boost the error correction capabilities. The number of error corrections in the hot zone is enhanced by using a smaller data block size for the SEC/SEC-DED code compared to the other zone.
- **Hybrid ECC (SEC/SEC-DED and DEC/DEC-TED)**: The second variant implements a double error correction (DEC) or double error correction-triple error detection (DEC-TED) multi-bit BCH code in the hot zone, while the SEC/SEC-DED code protects the other zone. Applying stronger error correction only to specific rows improves memory reliability without incurring unnecessary memory overhead.

A detailed discussion of the proposed location-aware non-uniform error correction strategy for STT-MRAM is presented in the following subsections.

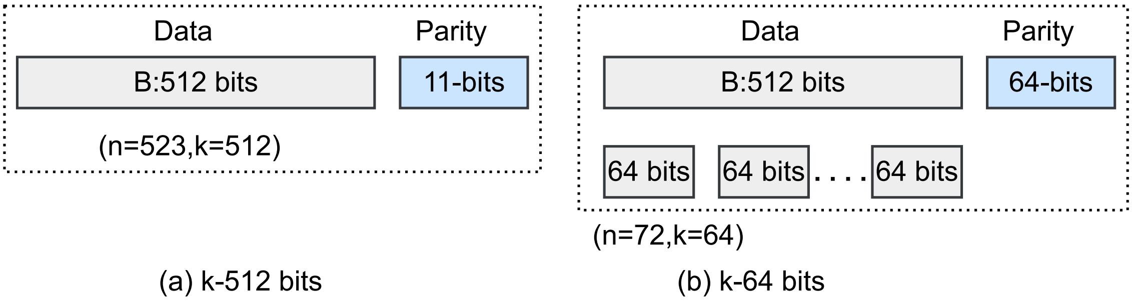
5.4.1. SEC/SEC-DED (ECC_1) with distinct data block size

In general, the memory is protected by SEC/SEC-DED code or double error correcting BCH code for a typical data block size of 64 or 128 bits [25], [265]–[267]. Instead of applying an ECC to the entire cache block (B-bits), it is common practice to divide the large cache line into multiple smaller blocks (k-bits) and protect each one individually, as shown in Fig. 5.6. Smaller ECC data block sizes improve the number of error corrections but also increase memory overhead due to increased ECC parity bits, as shown in Table 5.2. Where t represents the number of error corrections for a 512-bit word size with a single error correction per k-bit block. Memory overhead per block is given by Equation (5.1), and the total memory overhead is obtained by multiplying this by the number of blocks ($\frac{512}{k}$) in 512-bit word lines. For instance, applying SEC/SEC-DED to a 512-bit block incurs 10/11-bit parity, whereas applying SEC/SEC-DED to each 64-bit block results in 56/64-bit parity (7 parity bits per block: 7×8=56 bits for SEC and 8 parity bits per block: 8×8 = 64 bits for SEC-DED).

$$MemoryOverhead = \frac{\text{Parity}}{k} \times 100 \tag{5.1}$$

Table 5.2.: ECC parity bits for various block sizes (k-bits) in case of SEC/SEC-DED

k	SEC			SEC-DED		
	t	Parity bits	Memory Overhead	t	Parity bits	Memory Overhead
512	1	10	1.95%	1	11	2.14%
256	2	9	3.51%	2	10	3.90%
128	4	8	6.25%	4	9	7.03%
64	8	7	10.93%	8	8	12.50%
32	16	6	18.75%	16	7	21.87%



(a) k-512 bits

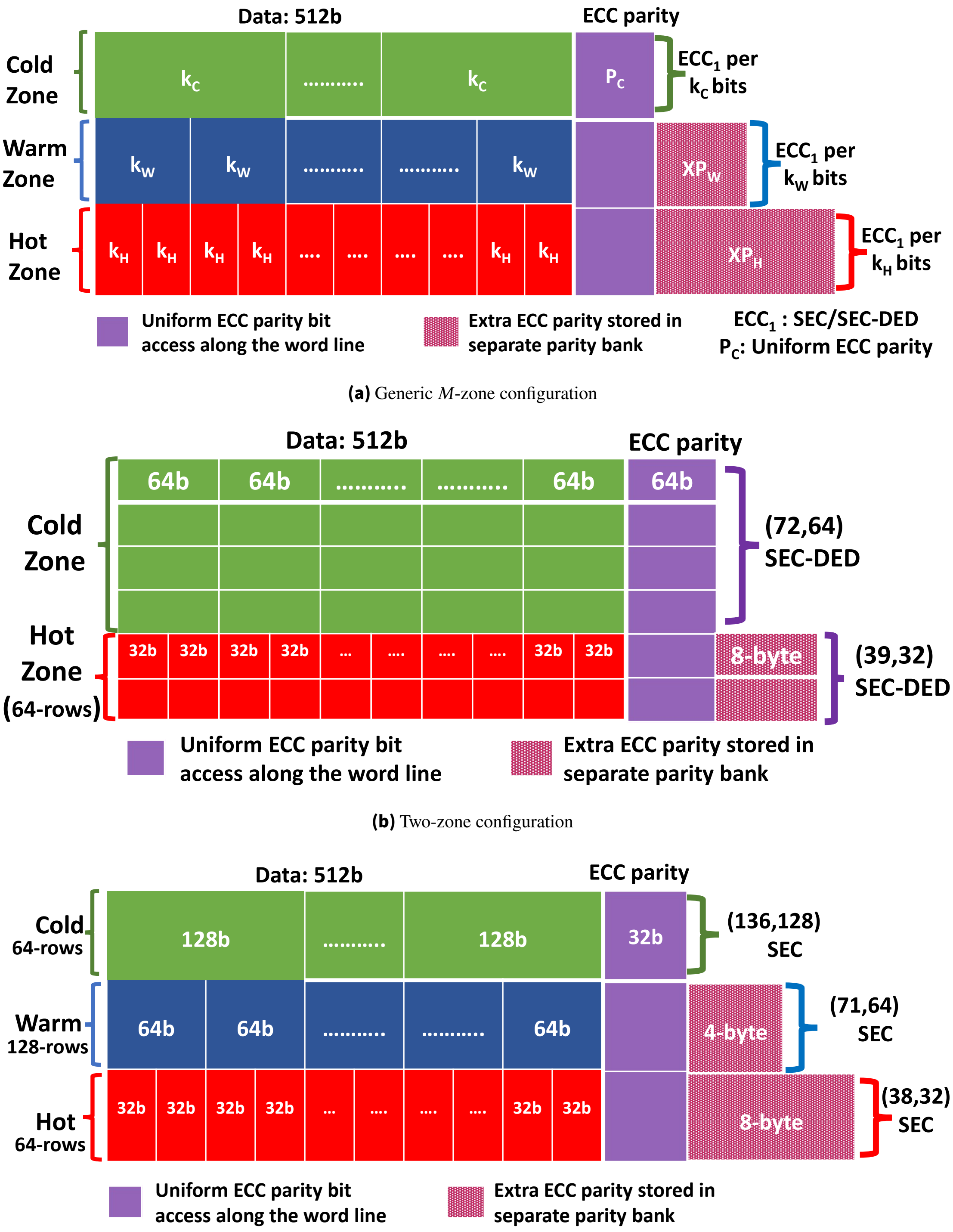
(b) k-64 bits

Figure 5.6.: Demonstration of a SEC-DED Hamming code for 512 bits

In this strategy, we apply the SEC/SEC-DED Hamming code (ECC_1) with varying data block sizes across different zones in a memory subarray. Fig. 5.7 (a) illustrates the overview of the proposed strategy using ECC_1 with distinct data block sizes, where the 512-bit cache block mapped to the different zones is protected by ECC_1 with distinct data block sizes (k). The hot zone uses the smallest block size to enhance error protection. Each zone requires a different number of ECC parity bits; smaller data block sizes result in more parity bits. Since different zones require different parity bits, the memory array must be a regular structure. Therefore, we use uniform parity overhead, corresponding to the cold zone parity (P_C), for all cache blocks mapped to any zone on the subarray to maintain the symmetric array structure. To avoid uniform additional storage overhead for cache blocks mapped to other than the cold zone, we introduce a separate parity bank (a group of subarrays) dedicated to storing the additional parity bits (XP_W and XP_H) associated with warm and hot zones.

Adding extra parity rows within the subarray is inefficient, as it not only complicates the parity mapping process but also incurs a performance penalty due to the additional read/write operations. To enhance hardware efficiency and avoid these extra operations, the additional parity (or extra rows) can be efficiently mapped to a separate bank, enabling parallel access. This can be achieved by utilizing a multi-bank memory architecture. Given that the cache block is 512 bits, the system must be able to access the 512-bit data and the associated parity bits for ECC encoding/decoding in the hot zone. A key consideration is that accessing the parity stored in a separate bank requires an explicit address mapping strategy, which should not introduce substantial hardware overhead.

In this work, we demonstrate two configurations for the proposed location-aware ECC: a two-zone configuration (cold and hot), as shown in Fig. 5.7 (b), and another with an extended three-zone configuration (cold, warm, and hot), as shown in Fig. 5.7 (c). However, the proposed strategy is generic and can be applied to more zones. The details are discussed next.



(a) Generic *M*-zone configuration

(b) Two-zone configuration

(c) Three-zone configuration

Figure 5.7.: The proposed location-aware error correction strategy employs ECC_1 with distinct data block sizes (k). The cache/word line size is 512 bits. The cold zone consists of rows near the driver, and the hot zone includes the rows farthest from the driver. b stands for bits, XP_W denotes extra parity bits/bytes associated with the warm zone, and XP_H denotes extra parity bits/bytes related to the hot zone

5.4.1.1. Two Zone Configuration

Fig. 5.7 (b) shows the structure of a two-zone configuration, where the memory subarray is divided into two zones: cold and hot zones. The cold zone is protected by an SEC-DED with a 64-bit block size, while the hot zone is protected with SEC-DED using a 32-bit block size. The cold zone is protected by a (72,64) SEC-DED code, requiring a 64-bit ($P_C = 8 \times 8$) parity for a 512-bit cache block. The hot zone, on the other hand, is protected by a (39,32) SEC-DED code and requires a 112-bit ($P_H = 16 \times 7$) parity. We use a uniform 8-byte (64-bit) parity overhead for all cache blocks mapped to any zone on the subarray. In the cold zone, the parity bit can be easily accessed due to the uniform 8-byte parity storage, which is sufficient to store the parity associated with the (72,64) code. For the hot zone, each block requires 7 bits of parity, which can be treated as a 1-byte parity chunk. In this case, the parity for the first eight blocks can be accessed from the uniform parity storage, while the remaining 8 bytes are mapped to a separate parity bank, with the parity subarray operating at an 8-byte word level access.

For the hot zone, an additional 8-byte parity corresponding to the remaining eight ECC data blocks in a 512-bit cache or word line must be mapped to a separate parity bank. Storing extra parity in a parity bank requires address calculation corresponding to the data address, which may lead to additional hardware overhead. To make parity mapping hardware efficient, we ensure that the number of rows in each zone is a power of 2. The total number of extra rows required for memory depends on the number of rows in the hot zone (N_{Row_H}), and the extra parity bit associated with the cache block mapped to the hot zone (XP_H). This, in turn, determines the total number of parity subarrays within the parity bank. The number of subarrays required for the parity bank associated with the hot zones is given by Equation (5.2). Alternatively, it can also be given by N_{Sub_P}= $\left\lceil (\frac{N_{Row_H}}{N_{Row}} \times N_{Block} \times \frac{8 \times XP_H}{N_{Column}} \times \frac{1}{N_{Row}}) \right\rceil$. All parameters are treated as powers of 2, and the units are bytes for both parity and cache block size. The parity bank with four subarrays (256×256) is sufficient when considering the last 64 rows under the hot zone, for the organization shown in Fig. 5.5.

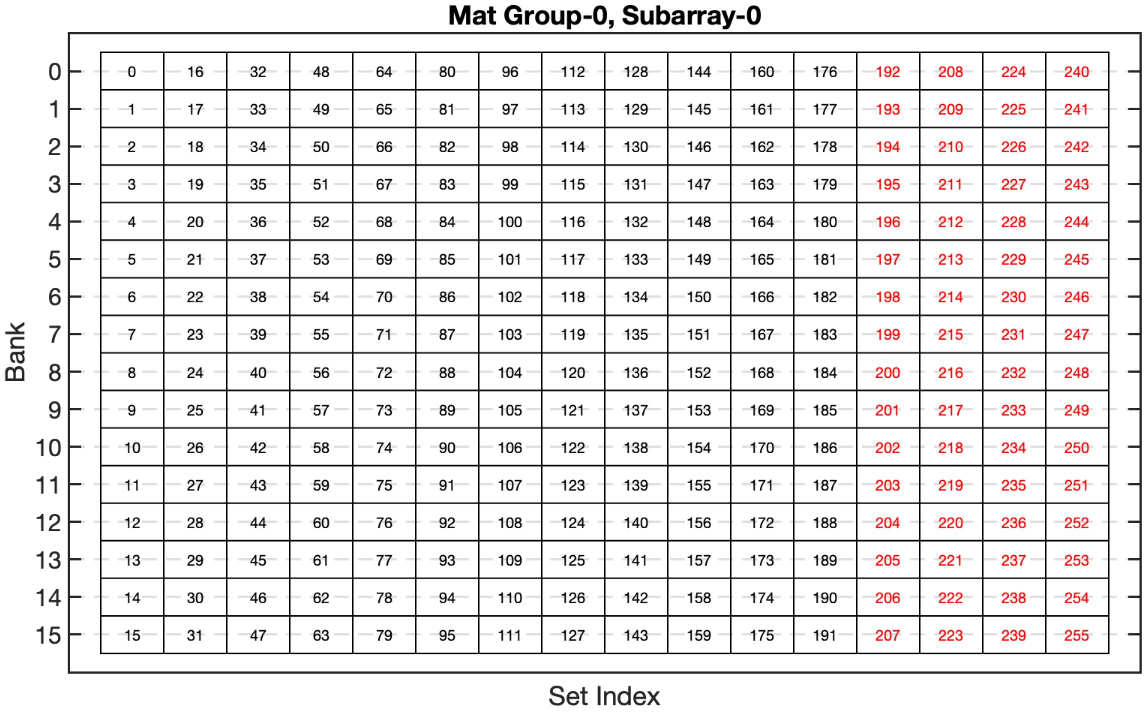
$$N_{\mathrm{Sub}_P} = \left\lceil \left(\frac{N_{\mathrm{Row}_H}}{N_{\mathrm{Row}}} \times \frac{XP_H}{B} \right) \times (N_{\mathrm{Bank}} \times N_{\mathrm{Mat}} \times N_{\mathrm{Sub}}) \right\rceil \tag{5.2}$$

The details of the address mapping procedure for the parity bank are outlined as follows:

- **Cache block distribution:** The cache configuration is 16-way set-associative, with each way containing a 64-byte block. The physical memory location of all the sets is distributed across 16 banks. Fig. 5.8 shows an example of how sets are mapped to Mat Group-0, Subarray-0 of Bank 0. The 14-bit address (B_3, B_2, B_1, B_0, MG, Sub, R_7, R_6, R_5, R_4, R_3, R_2, R_1, R_0) is sufficient to specify the row address of each 64-byte blocks mapped to a subarray.
- **Memory Zone identification:** The Most Significant Bit (MSB) of the row address (R_7 and R_6) in the 14-bit internal address are used to identify whether a cache block resides in the cold or hot zone. If the block falls within the hot zone, the (39,32) SEC-DED code is applied; otherwise, the (72,64) SEC-DED code.
- **Parity Bank Organization:** A parity bank with four 256×256 subarrays is sufficient to store the extra parity bits associated with the hot zone, as shown in Fig. 5.9 (a) while considering $N_{Row_H} = 64$ number of rows under the hot zone. Since each cache block mapped to a hot zone requires an additional 8 bytes of parity, the parity subarrays within the parity bank operate at an 8-byte word level to enable 8-byte parity access.

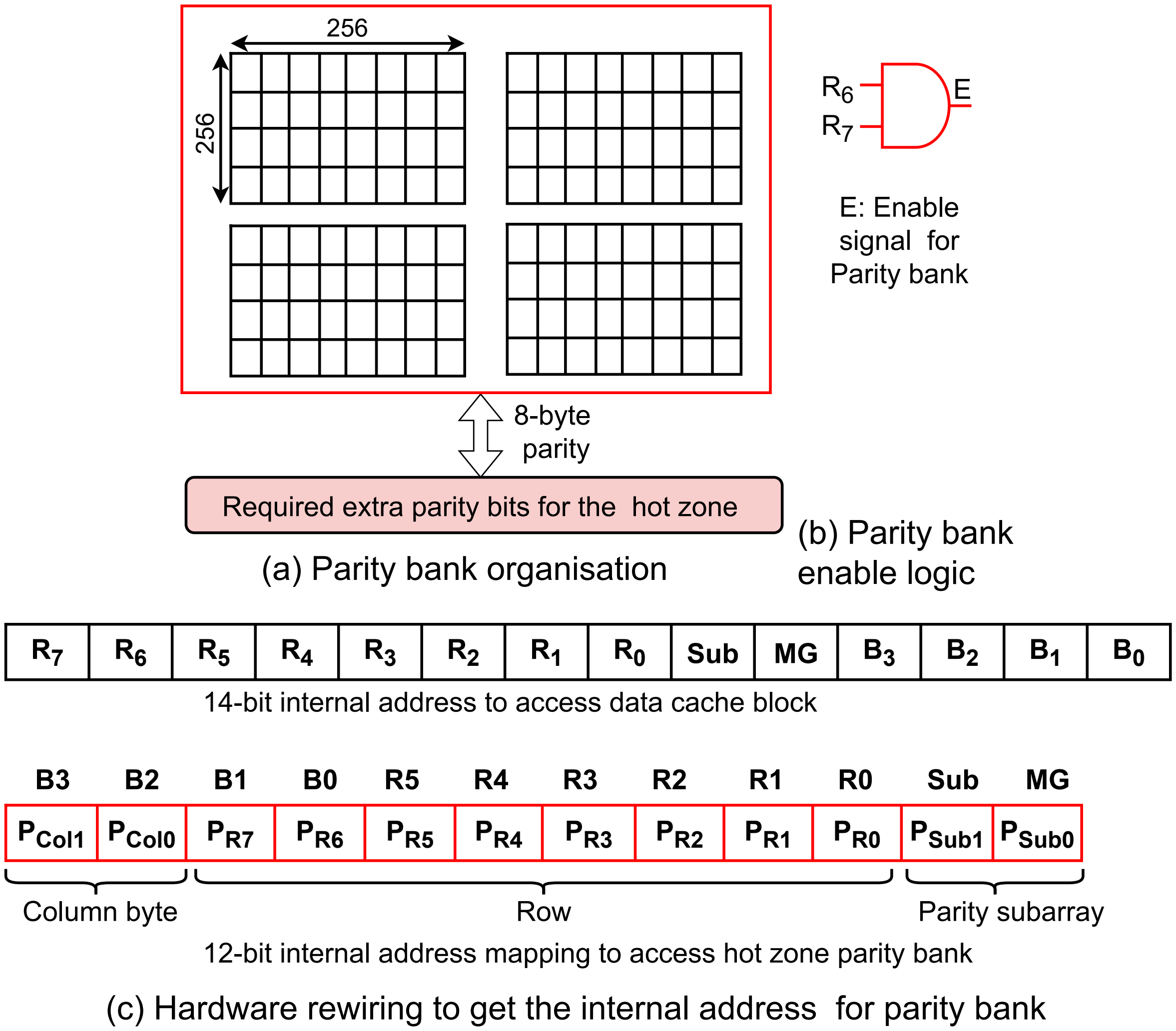
- **Address bits for Parity Bank:** To access the required extra parity bits, the parity bank addressing scheme requires a 12-bit address, which includes 2-bits (P_{Sub}) for the selection of a parity subarray, 8-bits (P_R) for the row address within a subarray, and 2-bits (P_{Col}) to select the column group containing the 8-byte parity.
- **Address mapping for parity bank:** The enable logic for the parity bank is derived by performing an AND operation on the MSB bits of the row address of the cache block, as shown in Fig. 5.9(b). The address corresponding to the 64-byte cache block can be mapped to the 12-bit parity bank address by rewiring the required data address bits, as shown in Fig. 5.9(c). The data address (B_3, B_2, B_1, B_0, R_5, R_4, R_3, R_2, R_1, R_0, Sub, MG) is mapped to the parity bank address (P_{Col1}, P_{Col0}, P_{R7}, P_{R6}, P_{R5}, P_{R4}, P_{R3}, P_{R2}, P_{R1}, P_{R0}, P_{Sub1}, P_{Sub0}), enabling access to the 8-byte parity. Fig. 5.10 illustrates an example of parity address mapping for the hot zone corresponding to Mat Group-0, Subarray-0 of Bank-0. In this example, the parity is mapped to column byte-0 of the first 64 rows in Parity Subarray-0. Similar mappings can be applied to other bank, mat, and subarray configurations.

The parity bank requires no separate ECC protection since it only stores the ECC parity, which the ECC already safeguards against errors in both the data and parity bits. The proposed strategy is adaptable to any memory organization. For ECC encoding and decoding logic in the hot zone, the (72,64) SEC-DED encoder/decoder can be reused for the first eight 32-bit blocks by zero padding to align the data bit length of the (72,64) code, while the remaining eight blocks require dedicated (39,32) SEC-DED encoding and decoding logic. However, due to the inherently low latency of the (72,64) ECC logic, it is also practical to reuse the same (72,64) encoder/decoder sequentially across all sixteen 32-bit blocks.



0	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
1	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
10	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
11	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
12	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
13	13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
14	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
15	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

Figure 5.8.: Cache Set distribution across different banks in the case of mat group-0 and subarray-0, where the red regions indicate hot zones—cache blocks mapped to the last 64 rows of the memory subarray. A similar configuration applies to other mats and subarrays


256
256
8-byte parity
Required extra parity bits for the hot zone
R6
R7
E
E: Enable signal for Parity bank
R7 R6 R5 R4 R3 R2 R1 R0 Sub MG B3 B2 B1 B0
14-bit internal address to access data cache block
B3 B2 B1 B0 R5 R4 R3 R2 R1 R0 Sub MG
PCol1 PCol0 PR7 PR6 PR5 PR4 PR3 PR2 PR1 PR0 PSub1 PSub0
Column byte
Row
Parity subarray
12-bit internal address mapping to access hot zone parity bank

(a) Parity bank organisation

(b) Parity bank enable logic

(c) Hardware rewiring to get the internal address for parity bank

Figure 5.9.: Address mapping logic for extra parity bits associated with the hot zone to a designated parity bank in case of a two-zone configuration. The wiring logic described is applicable to any memory configuration. However, the specific wiring details will depend on the parameters of the memory configuration

5.4.1.2. Three Zone Configuration

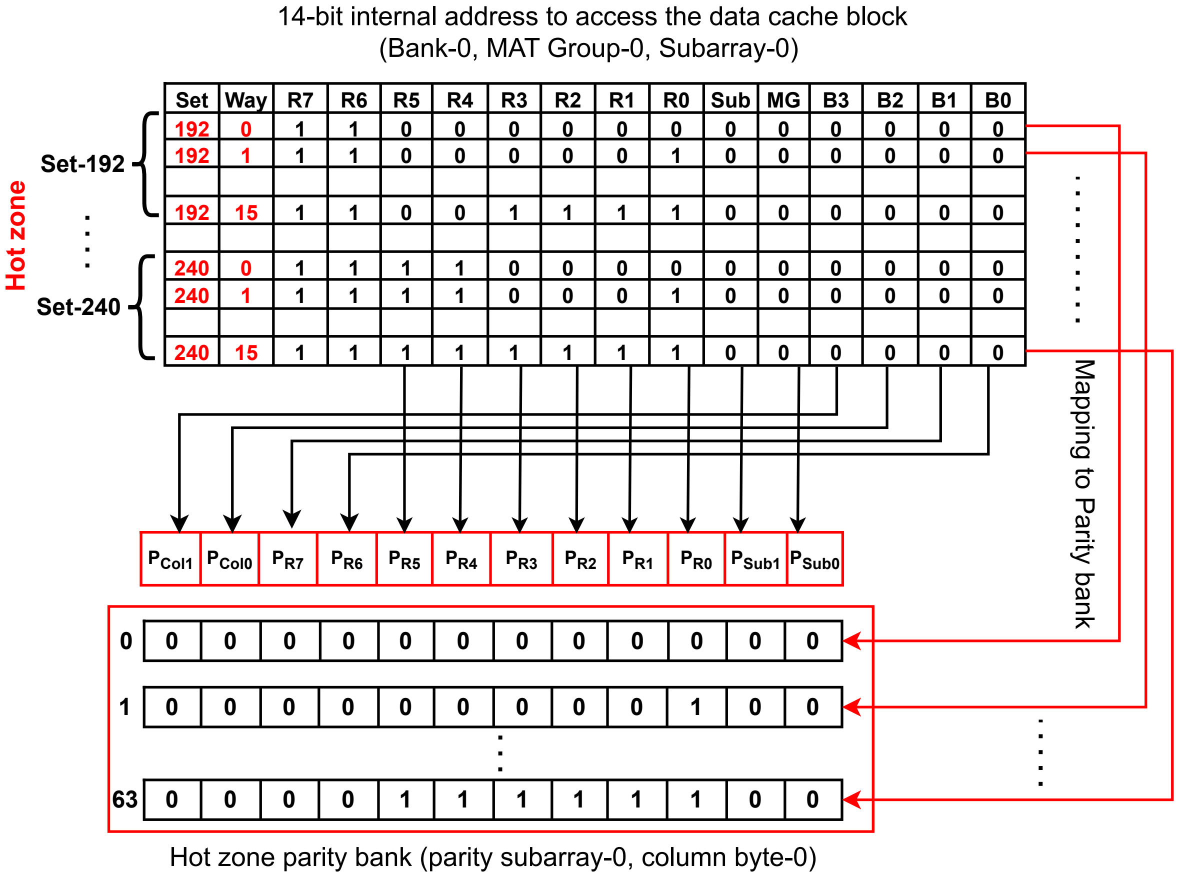
Fig. 5.7 (c) shows the structure of the proposed strategy with distinct ECC data block sizes for a three-zone configuration, where the memory subarray is divided into three zones: cold (rows near the driver), warm (middle rows of the subarray), and hot (rows farthest from the driver). The number of rows for each zone is strategically chosen to make the parity mapping process efficient while optimizing memory overhead. Additionally, using the SEC code instead of the SEC-DED code proves more efficient in this context, helping to maximize memory overhead. However, the method is still applicable to SEC-DED code with distinct data block sizes across different zones. The cold zone is protected by a (136,128) SEC code, requiring 32-bit parity ($P_C = 4 \times 8$). The warm-zone uses a (71,64) SEC code with 56-bit parity ($P_W = 8 \times 7$), while the hot zone employs a (38,32) SEC code requiring 96-bit parity ($P_H = 16 \times 6$).

As outlined earlier, a uniform 4-byte parity, corresponding to the cold zone, can be applied to all cache blocks. In the warm zone, an additional 4-byte parity (XP_W) is required for the remaining four ECC data blocks in a 512-bit cache block and must be mapped to a separate parity bank. Similarly, in the hot zone, an extra 8-byte parity (XP_H) is required and must also be mapped to the separate parity bank. The warm zone is designed to be relatively large to enable a parity bank organization similar to that of the hot zone. This architectural alignment simplifies parity mapping and enhances implementation efficiency by allowing uniform handling of parity across both zones. However, selecting the number of rows for each zone involves a trade-off in memory overhead. The number of parity subarrays for each zone can be obtained by Equation (5.2). A parity bank consisting of four subarrays (256×256) is sufficient to support the memory organization shown in Fig. 5.5, assuming 128 rows are allocated to the warm zone and 64 rows each to the hot and cold zones. The details of the parity bank addressing for the three-zone configuration are as follows:

- The parity bank addressing scheme varies between the warm and hot zones. In the warm zone parity bank, a 13-bit address is used, comprising 2-bits (P_{Sub}) for the selection of a subarray, 8-bits (P_R) for the row address, and 3-bits (P_{Col}) for the selection of columns within the row which contains the extra 4-byte parity.
- The hot zone parity bank requires a 12-bit address, which includes 2-bits (P_{Sub}) for the parity subarray, 8-bits (P_R) for the row address, and 2-bits (P_{Col}) to select the column group containing an 8-byte parity block.
- The parity bank's enable logic selects the appropriate parity bank according to the row address of the cache block being accessed, as shown in Fig. 5.11(b). If the data resides in the cold zone, the output of both enable signals remains 0, thereby disabling access to parity banks.
- The cache block data address can be mapped to the 13-bit (warm zone)/12-bit (hot zone) parity bank address by rewiring the data address bits, enabling access to the 4-byte/8-byte parity, as shown in Fig. 5.11(c).
- Warm zone: $B_2 \rightarrow P_{Col2}$, $B_1 \rightarrow P_{Col1}$, $B_0 \rightarrow P_{Col0}$, $B_3 \rightarrow P_{R7}$, $R_7 \rightarrow P_{R6}$, $R_5 \rightarrow P_{R5}$, $R_4 \rightarrow P_{R4}$, $R_3 \rightarrow P_{R3}$, $R_2 \rightarrow P_{R2}$, $R_1 \rightarrow P_{R1}$, $R_0 \rightarrow P_{R0}$, $Sub \rightarrow P_{Sub1}$, $MG \rightarrow P_{Sub0}$.
- Hot zone: $B_3 \rightarrow P_{Col1}$, $B_2 \rightarrow P_{Col0}$, $B_1 \rightarrow P_{R7}$, $B_0 \rightarrow P_{R6}$, $R_5 \rightarrow P_{R5}$, $R_4 \rightarrow P_{R4}$, $R_3 \rightarrow P_{R3}$, $R_2 \rightarrow P_{R2}$, $R_1 \rightarrow P_{R1}$, $R_0 \rightarrow P_{R0}$, $Sub \rightarrow P_{Sub1}$, $MG \rightarrow P_{Sub0}$
- Fig. 5.12 shows an example of parity address mapping for a three-zone configuration. It is mapped to Column Group-0 of the first 128 rows and 64 rows of parity Subarray-0 of the warm and hot zone parity banks, respectively.

For ECC encoding and decoding, four (136,128) SEC units are sufficient to cover the cold zone. In the warm zone, these units can be reused for the first four 64-bit data blocks with zero-padding to align with the data bit width of the (136,128) ECC unit, while the remaining four blocks require dedicated (71,64) SEC units. In the hot zone, the (136,128) and (71,64) SEC units are reused for the first eight 32-bit blocks via zero-padding, and the remaining eight blocks are handled by dedicated (38,32) SEC units. Furthermore, there is no latency penalty when using (136,128) or (71,64) SEC units for smaller data blocks, such as 32-bit or 64-bit, as each encoding and decoding operation can be completed in a single cycle. Further details are discussed in Sec.

Implementing more than two zones offers certain advantages but also introduces some design complexity. While increasing the number of zones can reduce uniform parity storage overhead, it also demands a dedicated ECC encoder and decoder module for each zone. Despite this, the ECC logic hardware overhead remains relatively small compared to the total memory area and becomes negligible as memory scales.


14-bit internal address to access the data cache block
(Bank-0, MAT Group-0, Subarray-0)
Hot zone
Set-192
Set-240
Mapping to Parity bank
Hot zone parity bank (parity subarray-0, column byte-0)

Set	Way	R7	R6	R5	R4	R3	R2	R1	R0	Sub	MG	B3	B2	B1	B0
192	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
192	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0
192	15	1	1	0	0	1	1	1	1	0	0	0	0	0	0
240	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0
240	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0
240	15	1	1	1	1	1	1	1	1	0	0	0	0	0	0

	P_{Col1}	P_{Col0}	P_{R7}	P_{R6}	P_{R5}	P_{R4}	P_{R3}	P_{R2}	P_{R1}	P_{R0}	P_{Sub1}	P_{Sub0}
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	1	0	0
63	0	0	0	0	1	1	1	1	1	1	0	0

Figure 5.10.: Demonstration of the hot zone address mapping from Mat Group-0 and Subarray-0 within Bank-0 to the corresponding parity bank in case of a two-zone configuration

Moreover, a multi-zone architecture requires parity banks with varying word-level access granularity to accommodate the additional parity associated with each zone. These factors must be carefully considered to fully leverage the practical benefits of fine-grained zone partitioning. As a result, limiting the number of zones offers an optimal trade-off between reliability, memory efficiency, and architectural complexity, making it a more practical and scalable solution. Additionally, configuring the number of rows in each zone as a power of two helps streamline the parity mapping process and improves implementation efficiency.

5.4.2. Hybrid ECC strategy using multi-bit ECC

In this variant, the proposed strategy implements DEC or DEC-TED (ECC_2) logic only in the hot zone, while the other zones are protected with the ECC_1 while having different ECC data block sizes, as shown in Fig. 5.13. This confines the higher storage overhead of ECC_2 to the hot zone, avoiding a large, uniform parity-bit overhead across the entire array.

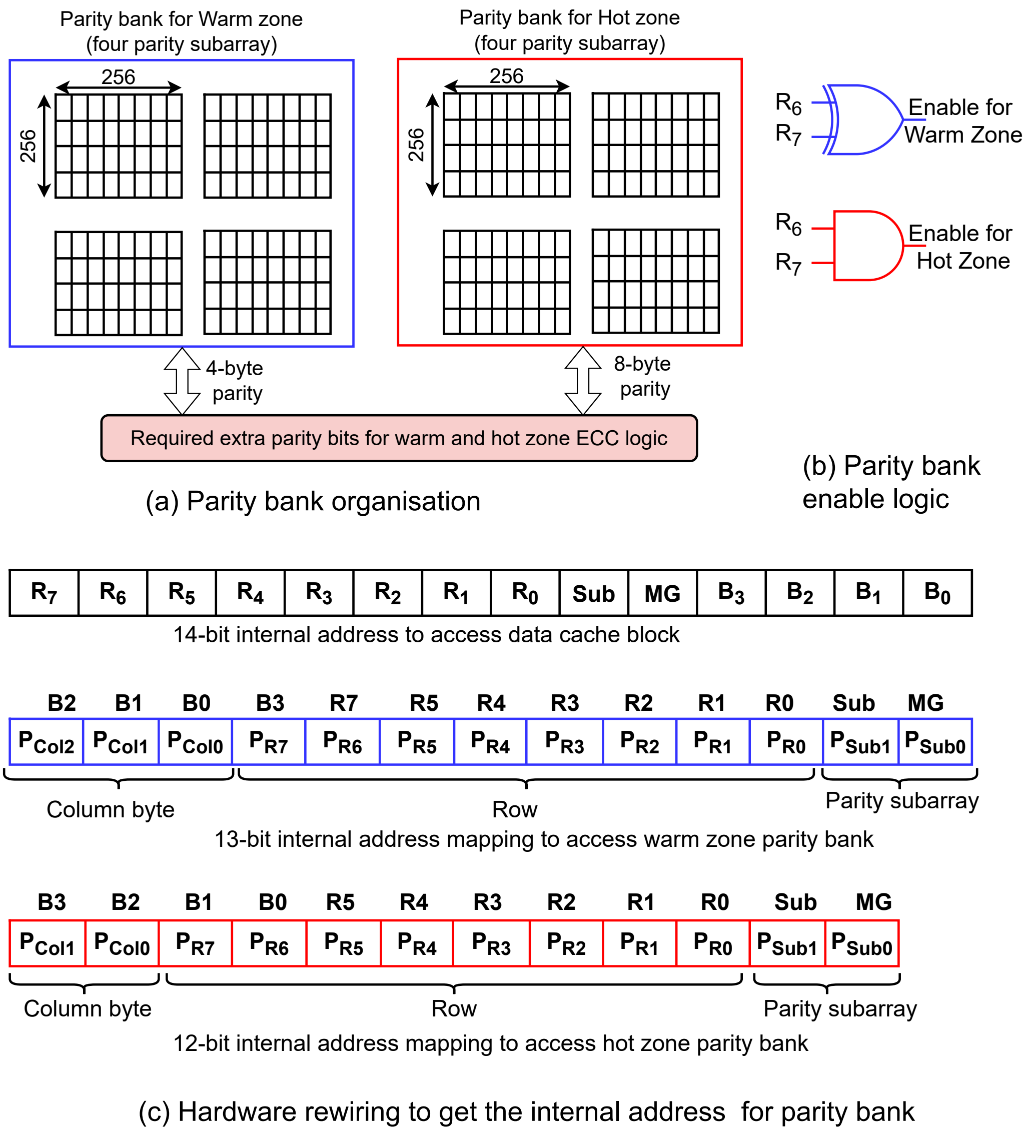
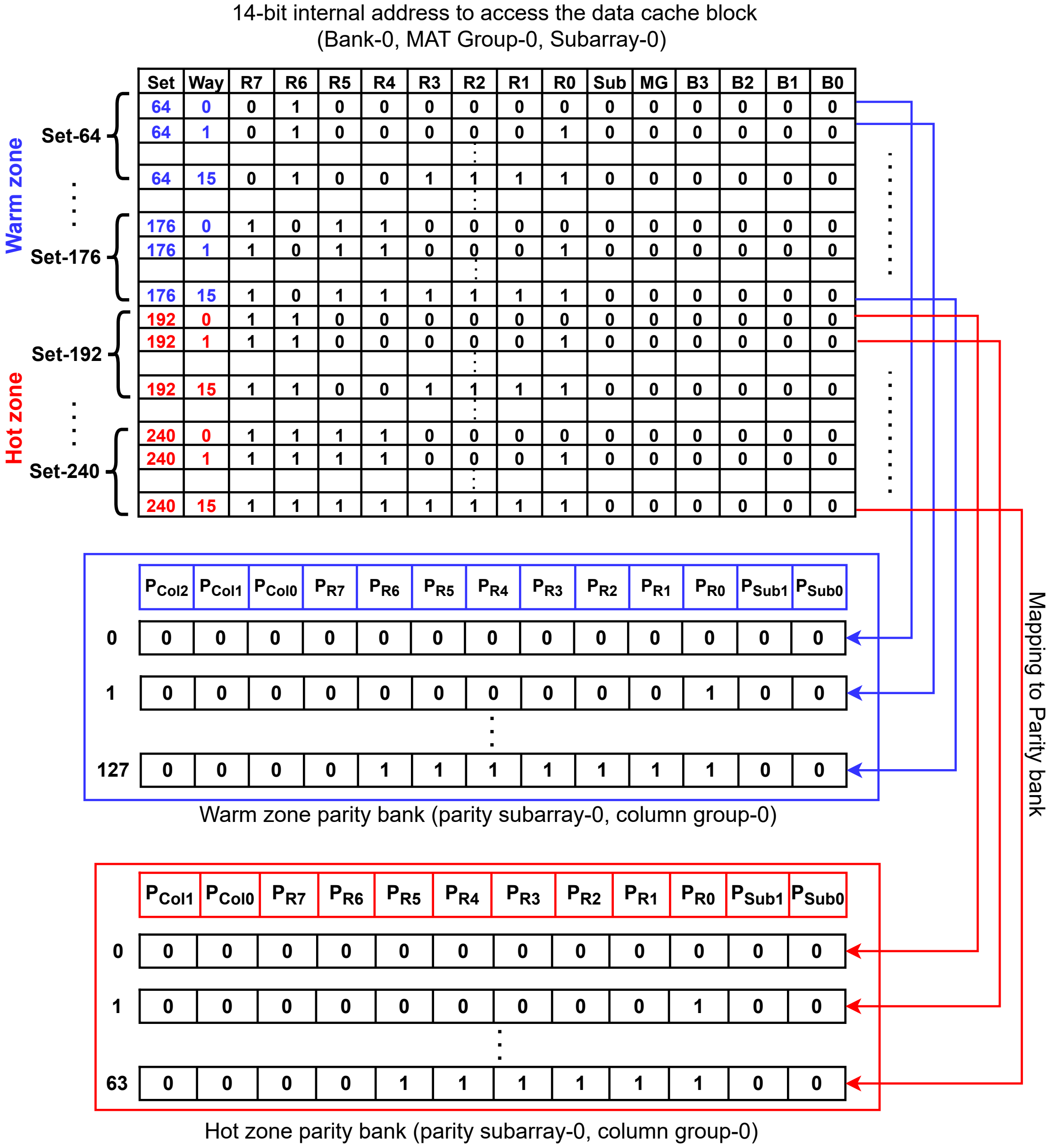

Parity bank for Warm zone
(four parity subarray)
256
256
Parity bank for Hot zone
(four parity subarray)
256
256
4-byte parity
8-byte parity
Required extra parity bits for warm and hot zone ECC logic
(a) Parity bank organisation
R_6
R_7
Enable for Warm Zone
R_6
R_7
Enable for Hot Zone
(b) Parity bank enable logic
R_7 R_6 R_5 R_4 R_3 R_2 R_1 R_0 Sub MG B_3 B_2 B_1 B_0
14-bit internal address to access data cache block
B2 B1 B0 B3 R7 R5 R4 R3 R2 R1 R0 Sub MG
P_{Col2} P_{Col1} P_{Col0} P_{R7} P_{R6} P_{R5} P_{R4} P_{R3} P_{R2} P_{R1} P_{R0} P_{Sub1} P_{Sub0}
Column byte
Row
Parity subarray
13-bit internal address mapping to access warm zone parity bank
B3 B2 B1 B0 R5 R4 R3 R2 R1 R0 Sub MG
P_{Col1} P_{Col0} P_{R7} P_{R6} P_{R5} P_{R4} P_{R3} P_{R2} P_{R1} P_{R0} P_{Sub1} P_{Sub0}
Column byte
Row
Parity subarray
12-bit internal address mapping to access hot zone parity bank
(c) Hardware rewiring to get the internal address for parity bank

Figure 5.11.: Address mapping logic for extra parity bits associated with the warm and hot zone to a designated parity bank in case of a three-zone configuration. The wiring logic described is applicable to any memory configuration. However, the specific wiring details will depend on the memory configuration

5.4.2.1. Two-zone configuration

The hot zone is protected using a (79,64) DEC-TED code, as illustrated in Fig. 5.13(a), which requires 120 bits of parity (8×15). In contrast, the cold zone uses the (72,64) SEC-DED, which requires 64 bits of parity for a 512-bit cache block. The (79,64) DEC-TED code necessitates extra parity storage in the hot zone. The 60-bit (4×15) parity corresponding to the first four 64-bit data blocks can be mapped to the available 64-bit space, while the remaining 60 bits can be mapped to a parity bank, considering it as an 8-byte parity, similarly to the discussion presented in Section 5.4.1. A similar design structure is also applicable for the (78,64) DEC code, which requires 14 bits (~2 bytes) of parity per 64-bit.


14-bit internal address to access the data cache block
(Bank-0, MAT Group-0, Subarray-0)
Warm zone
Set-64
Set-176
Hot zone
Set-192
Set-240
Mapping to Parity bank
Warm zone parity bank (parity subarray-0, column group-0)
Hot zone parity bank (parity subarray-0, column group-0)

Set	Way	R7	R6	R5	R4	R3	R2	R1	R0	Sub	MG	B3	B2	B1	B0
64	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
64	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0
							⋮								
64	15	0	1	0	0	1	1	1	1	0	0	0	0	0	0
							⋮								
176	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0
176	1	1	0	1	1	0	0	0	1	0	0	0	0	0	0
							⋮								
176	15	1	0	1	1	1	1	1	1	0	0	0	0	0	0
192	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
192	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0
							⋮								
192	15	1	1	0	0	1	1	1	1	0	0	0	0	0	0
							⋮								
240	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0
240	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0
							⋮								
240	15	1	1	1	1	1	1	1	1	0	0	0	0	0	0

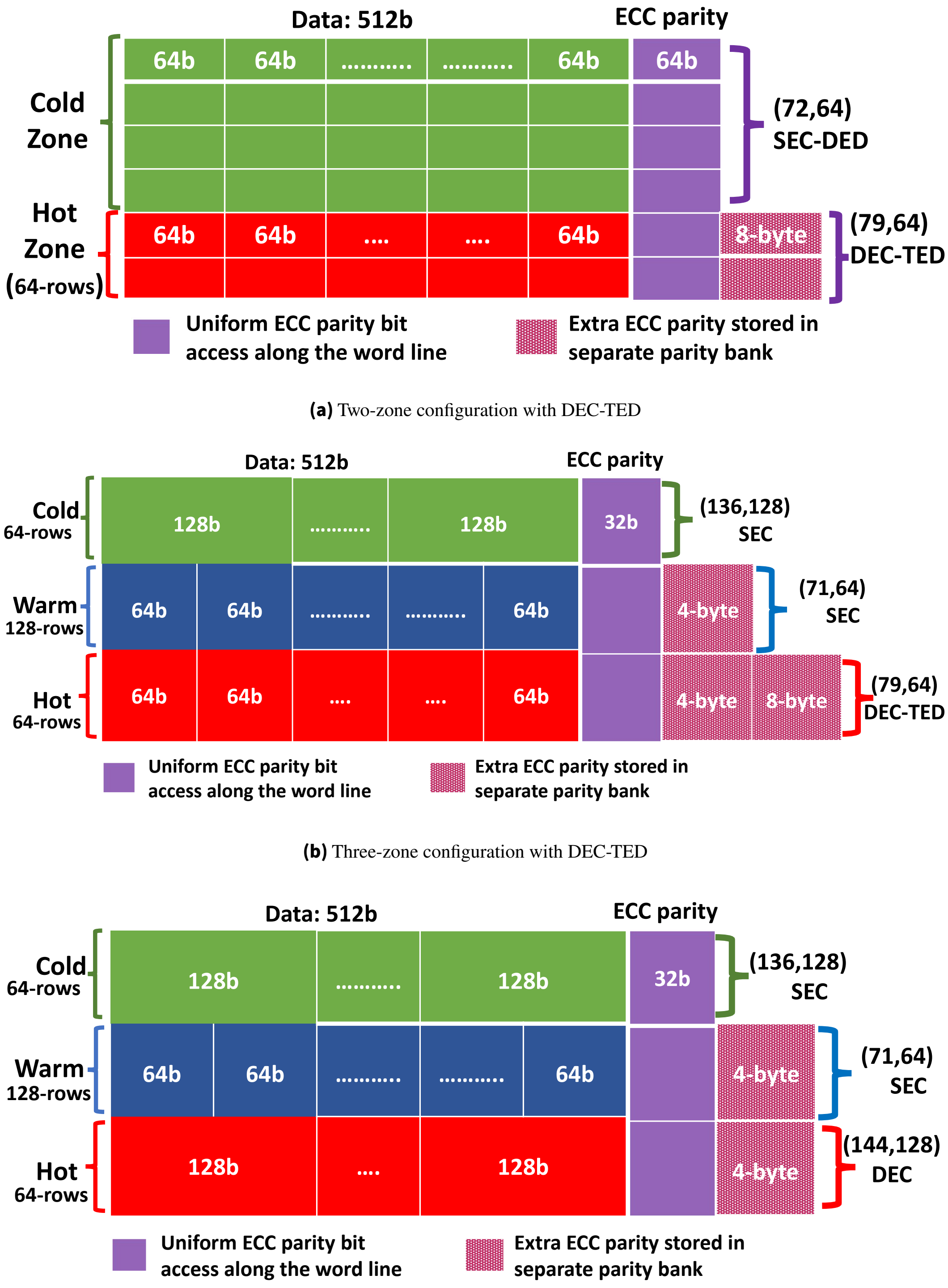
	P_{Col2}	P_{Col1}	P_{Col0}	P_{R7}	P_{R6}	P_{R5}	P_{R4}	P_{R3}	P_{R2}	P_{R1}	P_{R0}	P_{Sub1}	P_{Sub0}
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0	0
							⋮						
127	0	0	0	0	1	1	1	1	1	1	1	0	0

	P_{Col1}	P_{Col0}	P_{R7}	P_{R6}	P_{R5}	P_{R4}	P_{R3}	P_{R2}	P_{R1}	P_{R0}	P_{Sub1}	P_{Sub0}
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	1	0	0
							⋮					
63	0	0	0	0	1	1	1	1	1	1	0	0

Figure 5.12.: Demonstration of warm and hot zone address mapping from Mat Group-0 and Subarray-0 within Bank-0 to the parity bank in case of three-zone configuration

The read and write procedures are depicted in Fig. 5.14. During a read operation, the row address is first checked to see whether it falls within the hot or cold zone. Accordingly, it enables the SEC-DED decoding or DEC-TED/DEC decoding, as illustrated in Fig. 5.14 (a). Similarly, during a write operation, if the address belongs to the hot zone, the data is encoded using the DEC-TED encoder to produce the parity output; otherwise, SEC-DED encoding is performed, as shown in Fig. 5.14 (b). In this case, due to the differing error correction capacities of the codes, separate decoding logic is required.

The extra memory overhead associated with the hot zone can be avoided using a slightly larger block size (128 bits) with a DEC code, i.e., (144,128) code. In this design, the cold zone is protected by a



(a) Two-zone configuration with DEC-TED

(b) Three-zone configuration with DEC-TED

(c) Three-zone configuration with DEC

Figure 5.13.: The proposed location-aware error correction strategy that employs a hybrid ECC strategy with multi-bit ECC in the hot zone. The cache/word line size is 512 bits (64 bytes). The cold zone consists of rows near the driver, and the hot zone includes the rows farthest from the driver. *b* stands for bits

(72,64) SEC-DED code, while the hot zone employs a (144,128) DEC BCH code. Notably, using 128-bit blocks for DEC requires 64 bits of parity (4 × 16), which matches the parity requirement of four (72,64) SEC-DED blocks for a 512-bit cache line. This configuration enhances reliability in the hot zone without incurring extra memory overhead, as further discussed in Section 5.5. However, while the (144,128) DEC approach is memory-efficient, it offers lower correction granularity compared to (79,64) DEC-TED code and involves more complex decoding compared to applying DEC-TED/DEC on smaller 64-bit blocks.

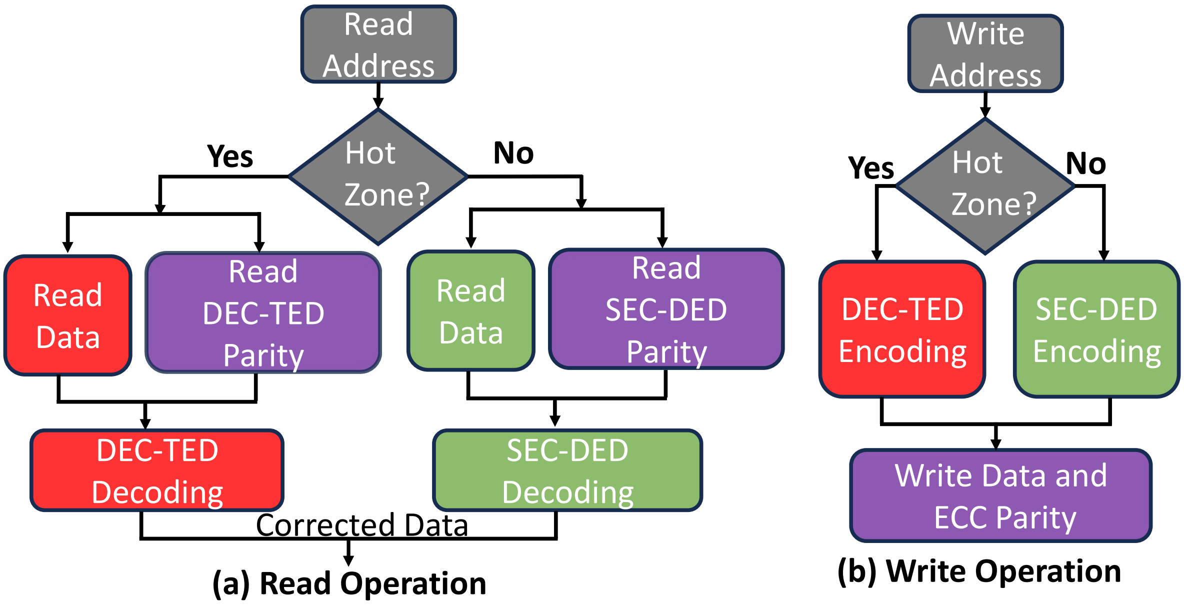


Figure 5.14.: Memory read and write operation in case of proposed strategy with strong multi-bit ($t = 2$) ECC in the hot zone

5.4.2.2. Three-zone configuration

The multi-bit ECC can also be applied to the hot zone in three-zone scenarios. However, selecting an appropriate block size and allocating rows per zone is crucial for optimizing memory overhead and maintaining symmetric parity bank organization. Here, we show two cases of extending the proposed strategy to three zones with multi-bit ECC, as shown in Fig. 5.13. In the first case, as shown in Fig. 5.13(b), the hot zone is protected with (79,64) DEC-TED code, whereas other zones are protected with SEC with distinct data block sizes. For the cold zone, a (136,128) SEC code is applied, requiring 32 bits of parity per cache block. This determines the uniform ECC parity overhead of 32 bits across all blocks. In the warm zone, a (71,64) SEC code is used, which requires seven parity bits (~byte) per 64-bit block, so 8 bytes total for a 512-bit cache block. Of this, 4 bytes can be provided from the uniform parity storage, while the remaining 4 bytes can be stored in a separate parity bank.

Similarly, for the hot zone, (79,64) DEC-TED requires a 15-bit parity bit (~2 bytes) per 64-bit data block. Therefore, out of the 16-byte parity for a 512-bit cache block, 12 bytes of parity need to be stored in a separate parity bank. The hot zone parity can even be divided into two banks operating simultaneously to serve the 12-byte parity request; one operates at a 4-byte word access, and the other operates at an 8-byte word access, ensuring the full utilization of storage inside the parity bank. The warm-zone parity bank would require a four-parity subarray, considering 128 rows under the warm zone. The parity banks of the hot zone would require six subarrays: four subarrays in the bank that operate at an 8-byte word level and two subarrays in the bank that operate at the 4-byte word level, considering 64 rows under the hot zone.

Furthermore, efficient parity mapping with reduced memory overhead can be achieved using multi-bit ECC with a 128-bit block size. As shown in Fig. 5.13(c), applying a (144,128) DEC adds 2 bytes of parity, totaling 8 bytes for a 512-bit line. In this case, 4 bytes can be accommodated in the 36-bit uniform parity storage, while the remaining 4 bytes are stored in a separate parity bank. The warm and hot zones require an additional 4 bytes of parity per cache block, allowing them to operate at a 4-byte word-level access. This common granularity allows the sharing of the same parity bank among the zones, further improving mapping. For ECC encoding and decoding, four (136,128) and (71,64) SEC units are sufficient to cover the cold and warm zones, whereas the hot zone requires a dedicated DEC unit for each block. These strategies can also be implemented using SEC-DED with uniform parity overhead of 36 bits instead of 32 bits, based on the (137,128) SEC-DED used for the cold zone. Similarly, in Section 5.4.2.2(1), SEC-DED can be applied uniformly across all zones with a uniform storage overhead of 36 bits. Additionally, the hot zone would incur extra memory overhead, as the (39,32) SEC-DED needs more parity bits than the (38,32) SEC code.

5.5. Results and discussions

5.5.1. Simulation setup

We consider an example 1 MB memory organization with 16 banks, as illustrated in Fig. 5.1 and Table 5.3. The STT-MRAM macro serves as the last-level (L2) cache [15]. However, the proposed strategy is generalizable and can be applied to other memory organizations. To evaluate the memory reliability metrics, we assess the memory word/cache block error rate (BER:P_{BER}: probability of cache block failure), i.e., the probability of having no more than t errors when ECC with t-correction capability is employed. For n-bits block, under a bit error rate (P_b) protected by t-bit error correcting code, the BER can be computed by Equation (5.4). However, when a large word/cache line is divided into several smaller blocks, the block error rate can instead be computed using Equation (5.5), where the cache line is split into multiple blocks (n_B), each protected by its own dedicated (n, k) ECC.

We also evaluate the memory overhead associated with the proposed location-aware error correction strategy and provide a comparative analysis against the uniform ECC memory overhead. Additionally, we report the ECC encoding/decoding overhead at the $22nm$ technology node. Furthermore, we assessed system performance, measured in terms of instructions per cycle (IPC), under the proposed strategy by simulating various realistic workloads from the SPEC CPU2017 benchmark suite [244]. The benchmarks were selected based on multiple criteria: first, to avoid choosing applications with similar behavior, as suggested in prior work [250]; and second, to include a mix of memory-intensive and compute-intensive benchmarks [251]. The simulation details for evaluation of system performance are provided in Table 5.3. The gem5 simulator [252] is used to produce the L2 cache traces, and the most representative workload segment is extracted through the Simpoint method [253].

$$P_S(n,t) = \sum_{i=0}^{t} \binom{n}{i} P_{bit}^{i} (1 - P_{bit})^{n-i} \tag{5.3}$$

$$P_{BER}(n,t) = 1 - P_S(n,t) \tag{5.4}$$

$$P_{BER}(n,t) = 1 - \prod_{j=1}^{n_B} P_{S_j}(n,t) \tag{5.5}$$

Table 5.3.: Simulation details for system performance estimation.

Benchmarks	602.gcc,649.fotonik3d,605.mcf,654.rom, 623.xalancbmk, 638.imagick,625.x264, 628.pop2,641.leela,607.cactus
CPU	Out-of-Order (OoO) and Clock: 2.1GHz
L1I/SRAM	16 kB, Associativity:2, Read/Write cycles: 2/2
L1D/SRAM	16 kB, Associativity:4, Read/Write cycles: 2/2
L2/STT-MRAM	1MB, 16 Bank, Associativity-16, Read/Write cycles: 12/21
Cache block	512 bits (64 bytes)
Physical address	40-bits
Subarray	$N_{Row} \times N_{Column}$: 256 × 256

5.5.2. Impacts of bit error rate and varying ECC data block size on block error rate

Fig. 5.15 illustrates how increasing the bit error rate (P_b) impacts the memory cache/word BER (P_{BER}) for a 512-bit cache block protected with SEC-DED Hamming codes (t=1 error correction capability). It can be observed that as P_{bit} increases, the cache P_{BER} increases significantly, almost following a square law relation. For instance, when the bit error rate increases from P_b to $2P_b$, i.e., two times, the P_{BER} almost increases to ~4×. This highlights that even a slight increase in bit error rate can significantly affect memory block reliability, which is valid for ECC with an even smaller data block size.

Fig. 5.16 illustrates the impact of ECC data block size on cache BER. As the block size decreases, the BER drops significantly. In particular, reducing the ECC data block size from 512 bits to 32 bits can decrease the BER by up to 11.5×, as smaller data block sizes allow for a greater number of independent error corrections. Applying ECC with a smaller data block size in the hot zone will increase the number of independent error corrections (single error correction per block) in a 512-bit cache line, as shown in Table 5.2. Applying SEC/SEC-DED ECC per 32 bits would enable 16 independent error corrections, as it divides the cache block into 16 blocks and a single error correction in each block, compared to the eight error corrections in the case of ECC per 64 bits. In the hot zone, reducing the ECC data block size from 64 bits to 32 bits can lower the cache BER by up to ~1.75× in the hot zone, as shown in Fig. 5.16.

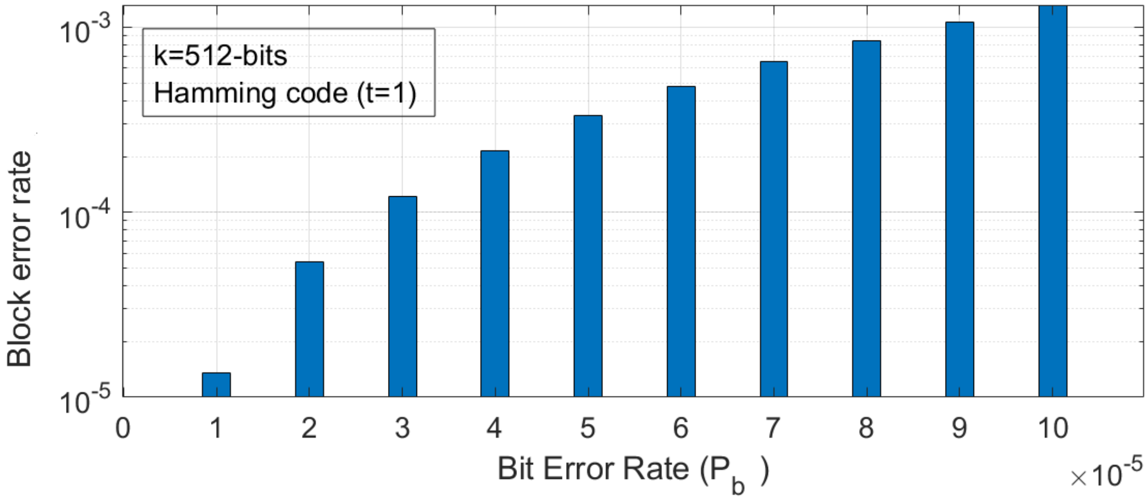


Figure 5.15.: Impact of bit error rate (P_b) on the block error rate (P_{BER})

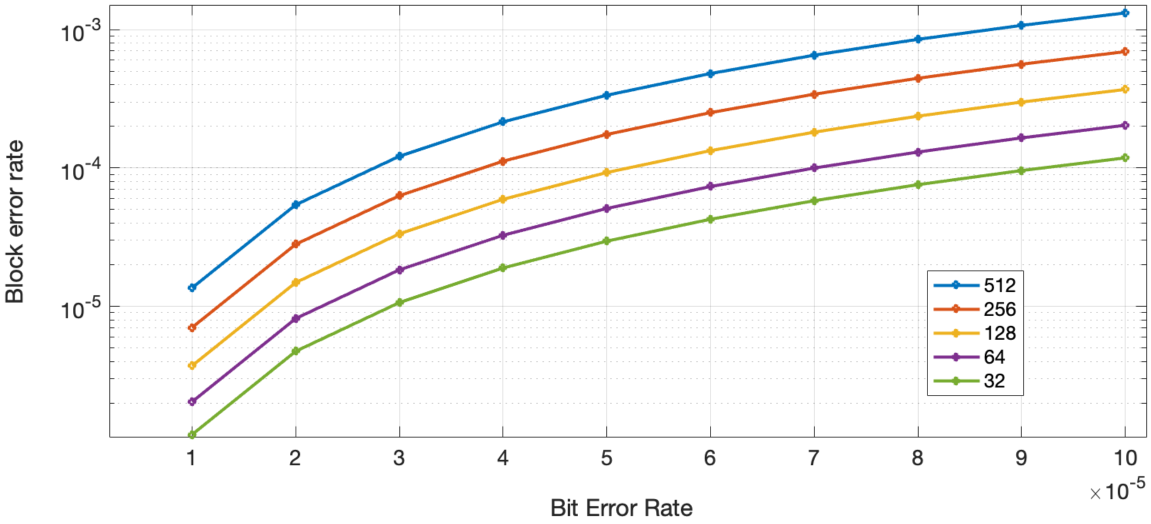


Figure 5.16.: Impact of data block size on the block error rate (P_{BER})

5.5.3. Evaluation of the location-aware error correction strategy using SEC/SEC-DED Hamming code (ECC_1) with distinct data block sizes

Fig. 5.17 illustrates the average BER reduction compared to the standard (72,64) SEC-DED code, considering $N_{\text{Row}H}$ = 64 rows assigned to the hot zone of the memory subarray and varying bit error rates in the hot zone (×Pbit). The average BER reduction is almost similar in the case of 2-Zone and 3-Zone. Even when the BER in the hot zone is five times higher ($5P_{\text{bit}}$), the proposed strategy can reduce the average BER by up to 37.5%. When the BER increases to even $10P_{\text{bit}}$, the average BER can still be reduced by up to 41% (43% in 3-Zone) compared to the (72,64) SEC-DED code. The improvement in the average BER is primarily due to the use of smaller ECC block sizes in the hot zone, specifically the (39,32) SEC-DED code. The average BER reduction will remain consistent even when the warm zone is subjected to a higher bit error rate than the cold zone. This is because the proposed strategy also applies the standard SEC/SEC-DED Hamming code per 64-bit block in the warm zone, which preserves the relative BER improvement.

Table 5.4 presents the ECC parity bit memory overhead associated with the proposed strategy with distinct ECC data block sizes across different zones. The extra memory overhead ('Extra MO') corresponds to the additional ECC parity bits required for the cache blocks mapped to the warm and hot zones, beyond the uniform ECC memory overhead. The cumulative total memory overhead is reported as the 'Total MO'. In case of 2-Zone configuration, while considering N_{Row_H}=64, the proposed strategy can reduce the memory overhead to only 15.62% (12.5%+3.12%), compared with 21.87% memory overhead caused by uniform (39,32) SEC-DED ECC for all the cache block (32.14% reduction in ECC parity bits). Similarly, in the case of a 3-Zone configuration using SEC with distinct ECC data block size, the memory overhead can be reduced to 12.5%, compared to the uniform (38,32) SEC code for all the cache blocks (33.33% reduction in ECC parity bits).

The ECC parity bit reduction is achieved while maintaining the same level of BER in the hot zone. This is because the proposed strategy applies the SEC/SEC-DED code per 32-bit data block. This enables sixteen independent error corrections (one per 32-bit block within a 512-bit cache line), effectively doubling the eight error corrections provided by the conventional (72,64) code. There is a trade-off between the

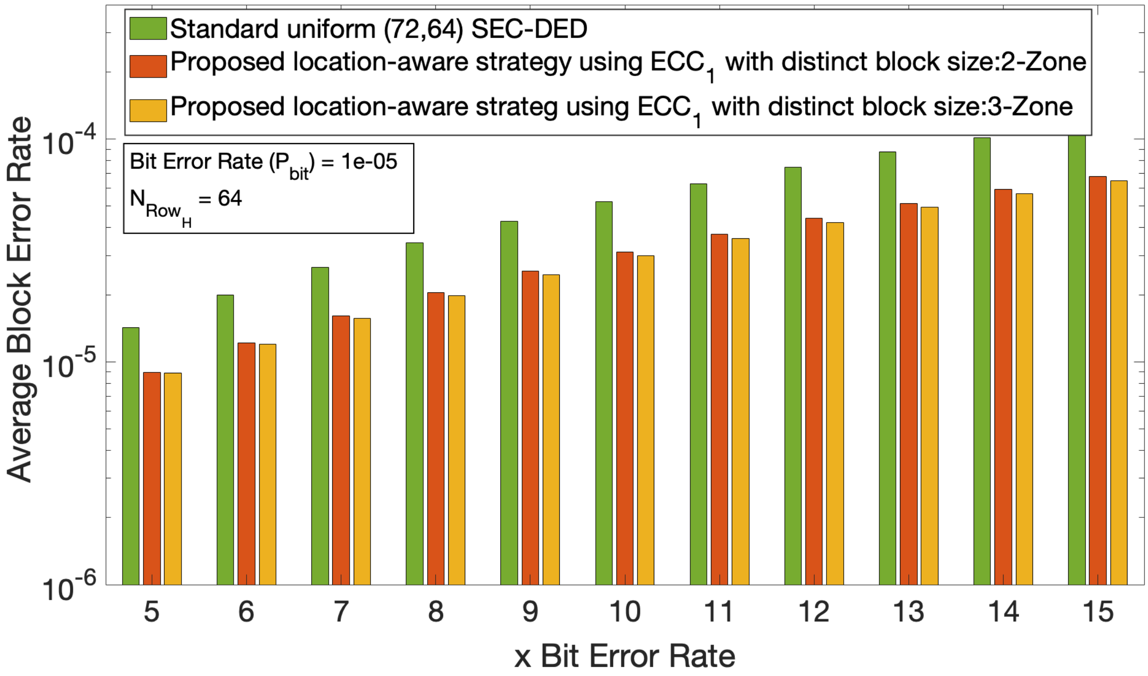


Figure 5.17.: Average cache block error rate reduction vs multiplicands of bit error rate ($\times P_b$ in the hot zone while employing SEC/SEC-DED with distinct ECC data block sizes compared to the standard (72,64) SEC-DED Hamming code. N_{Row_H} represents the number of rows under the hot zone

number of rows assigned to the hot zone and the additional memory overhead. For example, in a 2-Zone, a memory overhead reduction of up to 35.7%, can be achieved with N_{Row_H}=32 rows in the hot zone, compared to 32.14% reduction with N_{Row_H}=64, but this comes at the cost of a slightly increased average BER. However, as discussed earlier, choosing N_{Row_H}=64 offers a balanced trade-off between reliability and memory overhead, enabling the efficient parity bank addressing or mapping process, making it a reasonable choice for defining the hot zone.

Table 5.4.: Trend of memory overhead (MO) in the proposed strategy using ECC_1 with distinct data block sizes. N_{Row_H}=64 number of rows under the hot zone, N_{Sub_P} represents the number of extra parity subarrays. Total MO = Uniform MO + Extra MO

Proposed Strategy (Using distinct ECC data block size)					(72,64) SEC-DED	(39,32) SEC-DED	(38,32) SEC
#Zone	Uniform MO	N_{Sub_P}	Extra MO	Total MO	MO	MO	MO
2-Zone	12.5%	4	3.12%	15.62%	12.5%	21.87%	18.75%
3-Zone	6.25%	8	6.25%	12.5%			

Table 5.5 gives the hardware overhead per k-bit block for ECC_1. In a 2-zone configuration, the same eight (72, 64) encoder/decoder can be used by zero-padding to cover eight 32-bit blocks out of 16 data blocks. While the remaining eight 32-bit blocks can be covered with dedicated eight (39,32) encoder/decoder logic. In a 3-Zone configuration, the cold zone requires a dedicated (136,128) SEC ECC unit for each data block within the 512-bit cache line. For the warm zone, the four blocks can reuse the (136,128) ECC units through zero-padding, while the remaining four blocks are protected using dedicated (71,64) ECC units. In the hot zone, the eight blocks are covered using a combination of (136,128) and (71,64) ECC units, and the remaining eight blocks are protected by dedicated (38,32) SEC ECC units.

The hardware overhead associated with the encoding and decoding logic of ECC modules is low due to the reuse of logic resulting from their simple encoding/decoding process. This can result in an area overhead under 0.7% and 0.8% for 2-Zone, and 3-Zone, respectively, normalized to the data cache area estimated by NVSim [256]. Table 5.6 presents the hardware overhead of control logic, which corresponds to identifying the zone, routing the 512-bit cache data to the appropriate ECC encoder during encoding, and retrieving the corrected 512-bit data from the corresponding ECC decoder during decoding. The computational overhead is negligible, with no tangible impact on the clock frequency, which in turn does not affect the data path, resulting in minimal performance overhead. The details regarding performance overhead are discussed later.

The hardware overhead associated with the control logic is under 0.15% and 0.25% for 2-Zone and 3-Zone, respectively, normalized to the data cache array estimated by NVSim [256]. While comparing to the uniform (39,32) SEC-DED case, the additional area overhead (ECC + Control Logic) for both the zoning cases is negligible ($< 0.3\%$ for the 2-Zone and $< 0.5\%$ for the 3-Zone, normalized to the data cache array estimated by NVSim [256]).

Moreover, the power-delay product (PDP/energy = $Power \times Latency$) of the different components of the proposed strategy is minimal, i.e., a tiny fraction of pJ, as shown in Table 5.5 and 5.6. The energy overhead associated with ECC and control logic has a negligible impact, accounting for less than 0.75% during encoding and within 0.5% during decoding in the case of a 2-Zone (together, it results in under 0.6%). Similarly, in the case of the 3-Zone, it accounts for under 0.8% during encoding and 0.6% during decoding. Together, these result in under 0.7%, normalized to the cache access energy as estimated by NVSim [256].

Table 5.5.: Hardware overhead for SEC/SEC-DED (ECC_1) Hamming code for different data block size (k)

(a) SEC-DED Hamming code

k	Encoding			Decoding		
	Area	Latency	PDP	Area	Latency	PDP
32	$70\mu m^2$	$100ps$	$0.0508pJ$	$155\mu m^2$	$180ps$	$0.1389pJ$
64	$134\mu m^2$	$140ps$	$0.1221pJ$	$254\mu m^2$	$220ps$	$0.2166pJ$

(b) SEC Hamming code

k	Encoding			Decoding		
	Area	Latency	PDP	Area	Latency	PDP
32	$56\mu m^2$	$95ps$	$0.0322pJ$	$111\mu m^2$	$180ps$	$0.0891pJ$
64	$115\mu m^2$	$130ps$	$0.0812pJ$	$223\mu m^2$	$220ps$	$0.2006pJ$
128	$235\mu m^2$	$170ps$	$0.1694pJ$	$448\mu m^2$	$260ps$	$0.3968pJ$

Table 5.6.: Hardware overhead on top of ECC logic in case of the proposed strategy using ECC_1 with distinct data block sizes

Proposed	Data Encoding			Data Decoding		
Strategy	Area	Latency	PDP	Area	Latency	PDP
2-Zone	241 μm^2	100 ps	0.2952 pJ	500 μm^2	100 ps	0.5745 pJ
3-Zone	507 μm^2	100 ps	0.6895 pJ	1142 μm^2	100 ps	1.3580 pJ

We also assess how the computational overhead affects the data path, which can impact system performance measured by IPC. The system performance remains unaffected due to the use of a low-latency SEC/SEC-DED decoder. There is no reduction in IPC compared to the baseline ECC_1 per 64-bit data, as shown

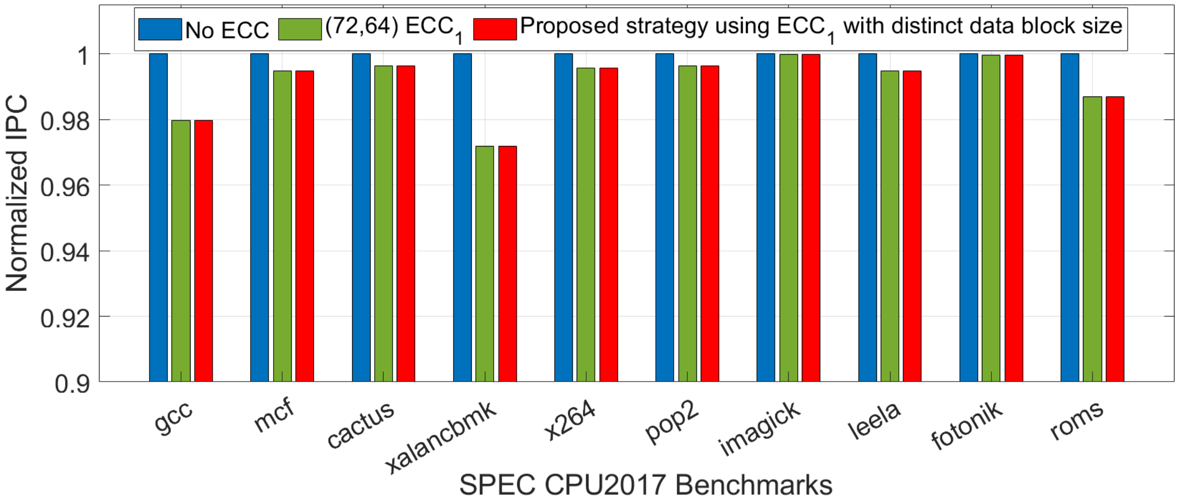


Figure 5.18.: Normalized IPC under the proposed strategy using ECC_1 with distinct data block sizes. ECC_1: SEC/SEC-DED code

in Fig. 5.18, which shows the system performance in terms of normalized IPC. Even when compared to a configuration without ECC (despite ECC protection always being present in most memories), the average performance overhead remains negligible (~1%). To evaluate system performance, we consider an additional cycle for each ECC encoding and decoding phase, which is sufficient, as the SEC-DED encoding and decoding latency is lower than the clock cycle.

5.5.4. Evaluation of hybrid ECC strategy with strong multi-bit ECC (ECC_2) in the hot zone

Using ECC_1 with smaller data blocks significantly increases the number of independent error corrections while having a moderate impact on cache failure probability. The cache BER can be further reduced by using DEC logic (t=2) in the hot zone. Fig. 5.19 shows that applying (79,64) DEC-TED to the hot zone maintains a BER in the hot zone nearly equal to (72,64) SEC-DED at the original bit error rate (P_{bit}), even when considering the bit error rate fifteen times (15×P_{bit}). Fig. 5.20 shows the average BER reduction using the proposed strategy with ECC_2, while considering different bit error rates (×Pb) for the hot zone. As shown in Fig. 5.20, even with a tenfold increase in the bit error rate (10P_b) in the hot zone, the proposed strategy using DEC-TED with N_{Row_H}=64 can reduce the BER by up to 31× (25.4× in 3-Zone). This reduction becomes even more significant under scenarios with a higher bit error rate.

Protecting the hot zone with (79,64) DEC-TED requires extra memory overhead, as illustrated in the Table. 5.7 (‘Extra MO’). However, the total memory overhead (‘Total MO’) is still much lower (33.5% reduction in parity with N_{Row_H}=64) than applying uniform DEC-TED, which requires a 23.44% memory overhead for ECC parity bits. However, the (144, 128) DEC code in the hot zone can be used to avoid the extra storage associated with the (79, 64) DEC-TED code, as discussed in Section 5.4.2. As shown in Fig. 5.19, the (144,128) DEC code can achieve nearly a similar block error rate in the hot zone, even with a ten times higher bit error rate (10×P_{bit}) in the hot zone while maintaining the same 12.5% overhead as the (72,64) SEC-DED. The proposed strategy using the (144,128) DEC code is efficient in terms of memory overhead but offers lower bit error resiliency and a higher block error rate compared to using the (79,64) DEC-TED. This is because the (144,128) DEC code performs only eight error correction (two per 128-bit), whereas the (79,64) code offers sixteen error correction (two per 64-bit data).

The hardware overhead associated with the additional ECC module required in the proposed multi-bit ECC is minimal on top of the uniform ECC_2 configuration. Furthermore, Table 5.9 details the extra control logic required for zone identification and data routing during ECC encoding and decoding. Overall, the area overhead associated with the additional ECC and control logic is below 0.65% for a 2-Zone configuration, whereas it is under 0.85% for a 3-Zone configuration, both normalized to the data cache array estimated by NVSim [256].

As shown in Table 5.8, the (144,128) DEC incurs higher decoding overhead (3.1×) due to its complex decoding logic compared to the (79,64) DEC-TED. Thus, the two multi-bit ECC variants of the proposed strategy present a trade-off among memory overhead, error resiliency, block error rate reduction, and ECC decoding overhead. The implementation of ECC_2 configuration is based on latency-efficient double-error-correction logic following the methods in [177], [257].

Table 5.7.: Memory overhead (MO) in case of the proposed hybrid ECC (ECC_2 in hot zone and ECC_1 in other zones). N_{Row_H}: 64 number of rows under hot zone, N_{Sub_P}: Number of extra parity subarrays. Total MO = Uniform MO + Extra MO

Proposed Strategy (Using ECC_2 in the hot Zone)						(72,64) SEC-DED	(144,128) DEC	(79,64) DEC-TED
ECC in Hot zone	#Zone	Uniform MO	N_{Sub_P}	Extra MO	Total MO	MO	MO	MO
(79,64) DEC-TED	2-Zone	12.5%	4	3.12%	15.62%	12.50%	12.50%	23.44%
	3-Zone	6.25%	10	7.81%	14.06%			
(144,128) DEC	2-Zone	12.5%	-	-	12.50%			
	3-Zone	6.25%	6	4.69%	10.94%			

Table 5.8.: ECC encoding and decoding overhead for ECC_2

ECC	k	Encoding			Decoding		
		Area	Latency	PDP	Area	Latency	PDP
DEC-TED	64	$202\mu m^2$	$170ps$	$0.2476pJ$	$2303\mu m^2$	$600ps$	$0.8844pJ$
DEC	128	$393\mu m^2$	$190ps$	$0.3768pJ$	$7169\mu m^2$	$700ps$	$1.4827pJ$

Table 5.9.: Hardware overhead on top of ECC logic in the proposed hybrid ECC strategy with multi-bit ECC in the hot zone

Proposed Strategy	Data Encoding			Data Decoding		
	Area	Latency	PDP	Area	Latency	PDP
2-Zone	473 μm^2	100 ps	0.5833 pJ	973 μm^2	100 ps	1.0264 pJ
3-Zone	556 μm^2	100 ps	0.7391 pJ	1242 μm^2	100 ps	1.4780 pJ

The power-delay product (PDP/energy), resulting from the various logic modules, is shown in Table 5.5, 5.8, and 5.9. The proposed strategy incurs minimal energy overhead in addition to the uniform ECC configuration; the energy overhead associated with additional ECC and control logic accounts for less than 0.7% during encoding and within 0.5% during decoding in the case of a 2-Zone (together, it results under 0.5%). Similarly, in the case of the 3-Zone, it accounts for under 0.75% and 0.6% during encoding and decoding, respectively. Together, this results in around 0.6%, normalized to the cache access energy as estimated by NVSim [256]. Additionally, the control logic also performs input gating for the ECC module, which is not in use; this can further minimize the run-time dynamic energy. Furthermore, the proposed strategy does not increase the energy overhead associated with the parity bits. By applying ECC in a non-uniform manner, the total number of parity bits is decreased relative to a uniform ECC, thereby lowering the energy.

Applying multi-bit ECC uniformly is also inefficient due to high decoding latency, which can impact the data path and ultimately degrade system performance. However, limiting it only to the required

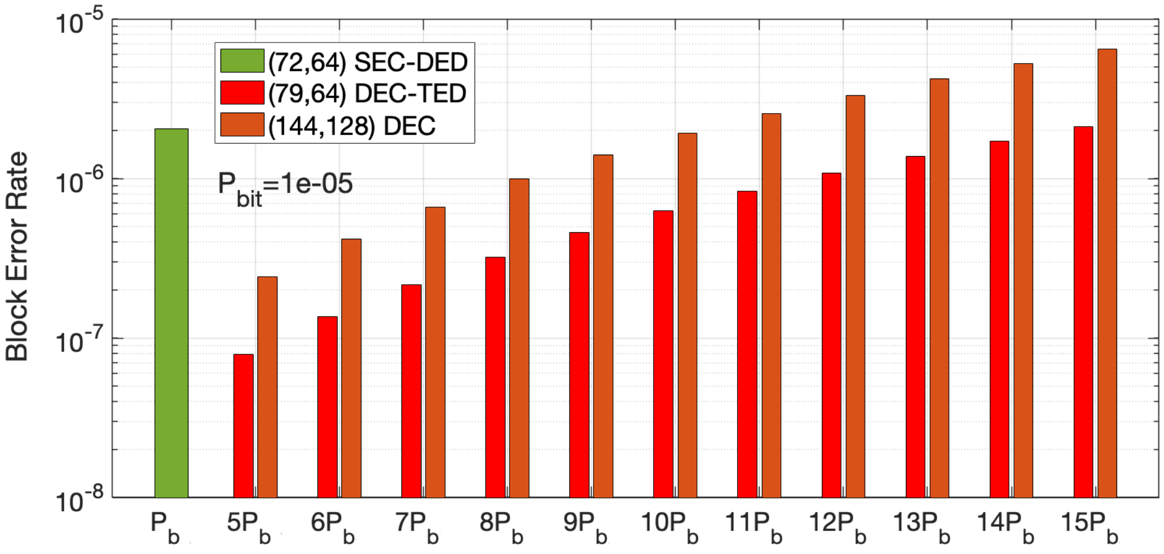


Figure 5.19.: Bit error rate resiliency with multi-bit ECC in the hot zone.

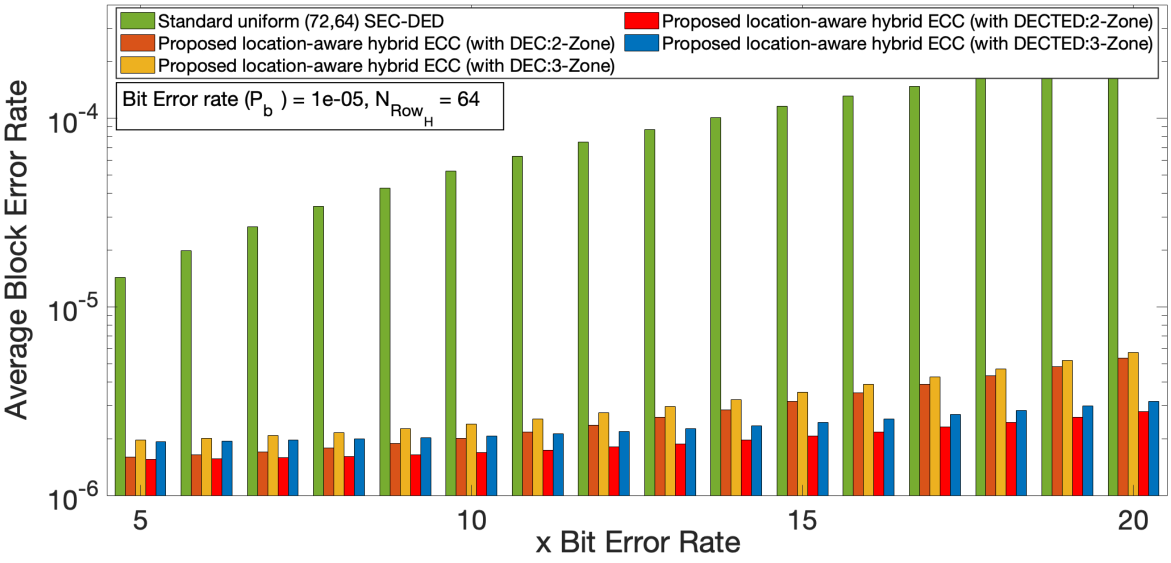
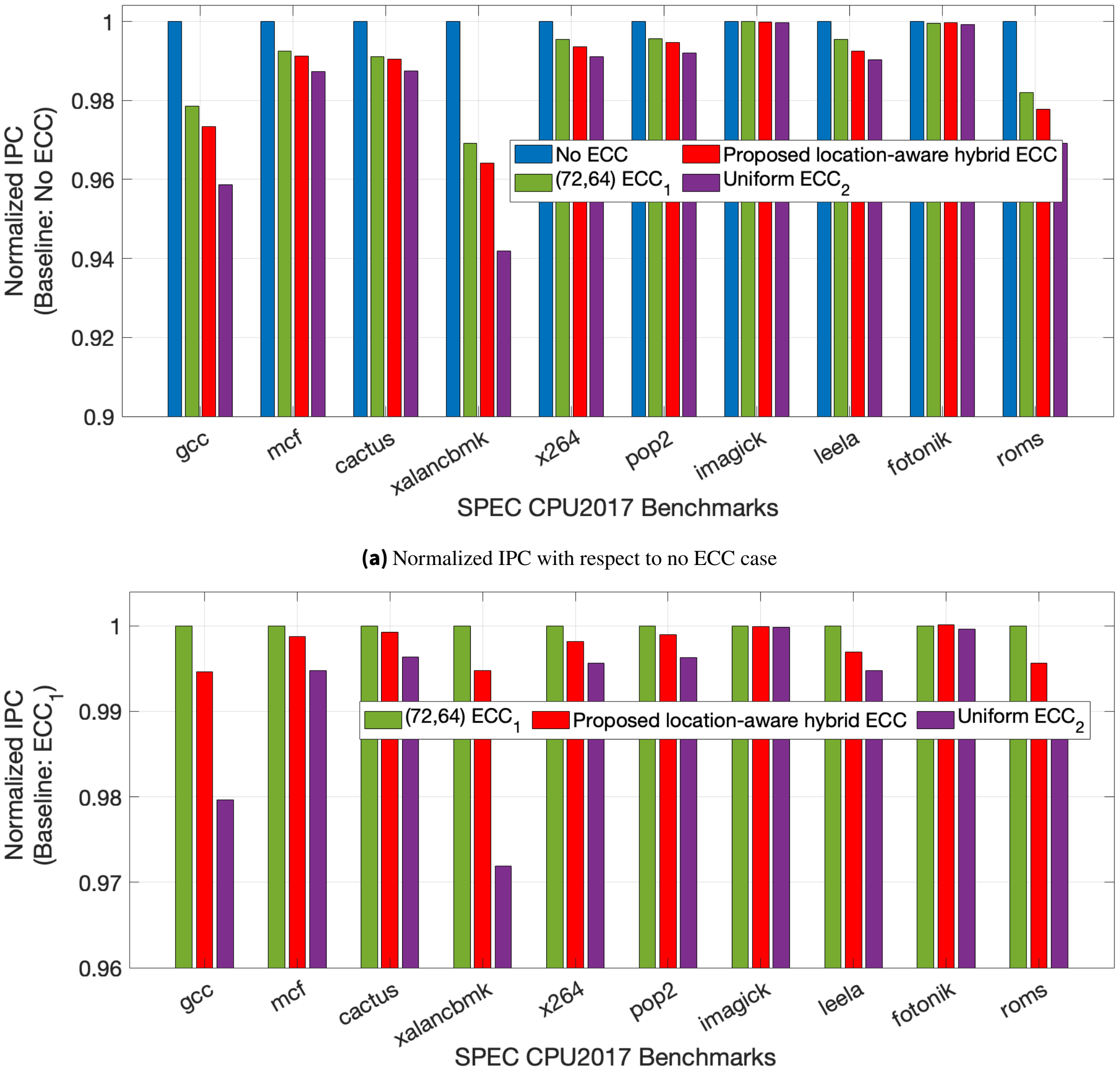


Figure 5.20.: Average block error rate reduction vs multiplicands of bit error rate (×P_b in the hot zone using the proposed hybrid ECC strategy with ECC_2 in the hot zone. ECC_2: (79,64) DEC-TED / (144,128) DEC.

zone not only reduces memory overhead but also incurs minimal performance overhead. Fig. 5.21 shows the normalized IPC. The proposed approach with selective DEC-TED/DEC maintains an average performance overhead under 1.25%, even compared to the no ECC case. However, the memory is at least protected with ECC_1, the proposed strategy incurs negligible performance overhead compared to the baseline ECC_1, as it maintains the average performance overhead within 0.25%. This is achieved by introducing additional ECC decoding cycle latency only for cache blocks mapped to the hot zone, rather than applying it uniformly.

Furthermore, concerning the feasibility of macro design for the proposed strategy, we have ensured that each parity subarray in the parity bank is sized to match the data subarray. This approach helps in fixing



(a) Normalized IPC with respect to no ECC case

(b) Normalized IPC with respect to (72,64) ECC_1

Figure 5.21.: Normalized IPC under proposed hybrid ECC strategy with ECC_2 in the hot zone. ECC_2: (79,64) DEC-TED / (144,128) DEC code.

the extra parity bank height or width, depending on how the subarrays are arranged, and supporting a rectangular macro structure. In a three-zone scenario, the number of subarrays in the additional parity bank is the same as the number of subarrays in the data banks. In the two-zone scenario, the size of the extra parity bank is smaller, as it requires only four subarrays. However, the height/width of this extra parity bank can be made the same as the data bank by rearranging the subarray placement, enabling efficient physical design. The number of subarrays can also be fine-tuned to have a specific number of subarrays in the parity bank, as it depends on the number of rows considered under the different zones. Additionally, the STT-MRAM process is CMOS-compatible, making it a viable option for embedded cache applications. Even small irregularities can be effectively managed during the final implementation of the chip. In practice, supporting logic and analog circuitry can be incorporated in the residual area near the extra parity banks — for example, cache policy control logic, charge pumps, or on-chip LDOs, and ECC/control logic. This co-integration helps to alleviate the impact of any aspect-ratio mismatches on the effective die size.

5.6. Summary

As technology scales down, interconnect parasitics become increasingly significant, potentially degrading bit-cell reliability due to voltage drops caused by interconnect resistance. In this chapter, we proposed an efficient, location-aware non-uniform error correction scheme to mitigate reliability degradation in STT-MRAM caused by increasing interconnect parasitics. Our strategy divides the memory subarray into different zones based on their distance from the driver, with each zone employing different error correction strengths. We have introduced two variants: one that applies a distinct block size for SEC/SEC-DED across different zones, and another that employs strong multi-bit ECC (DEC/DEC-TED) for the hot zone, while SEC/SEC-DED protects the other zones. The proposed strategy enhances memory reliability while minimizing uniform memory overhead by applying stronger error correction only where it is most needed. Further, by presenting a comprehensive cross-layer analysis, from technology to application, the proposed strategy improves memory reliability and has no adverse impact on system-level performance. The proposed method is highly adaptable and can be applied to other memory technologies with similar reliability aspects in the subarray interconnect.

6. Checksum-based Error Correction in Memristive Crossbar-based CiM Architecture for NN Acceleration

The memory wall issue in von Neumann architecture limits the performance of emerging applications, such as deep learning-based applications that operate on very large amounts of data. More recent advancements in computing, such as CiM, are potential solutions to the memory wall problem. The emerging resistive non-volatile memories, also known as memristors, are the most suitable choice for CiM realization, also referred to as a memristive crossbar. The MVM is one of the most frequently performed operations in deep learning hardware accelerators. The crossbar array structure, with memristive devices as its building block, has an inherent capability to perform energy-efficient MVM. However, memristive devices also suffer from various non-idealities and a limited number of stable states, as discussed in Chapter 2. Therefore, the reliability and, in turn, inference accuracy of the deep learning application are negatively impacted. In this chapter, we present a checksum-based error correction method for a memristive crossbar for reliable MVM computation. In this chapter, we focus on a memristive crossbar made of multi-bit resistive NVM devices. In the first strategy, we propose an scaled checksum-based low-overhead single-column error correction in a memristive crossbar. The proposed methodology alleviates the problem of storing the checksum value into multiple columns of the crossbar due to the limited number of stable levels of memristive devices. The number of extra columns required for storing the checksum value is reduced. We discuss two types of checksums: one based on ABFT, and the other on the Hamming code. The second strategy introduces the ECC-based checksum strategy for block error correction in the memristive crossbar, aiming to protect groups of columns simultaneously.

6.1. Introduction

Deep learning algorithms have recently gained significant importance in many areas, ranging from mobile applications to data centers [268]. These algorithms involve numerous MVM operations on massive datasets. The data transfer between the processor and memory is becoming one of the most critical performance and energy bottlenecks in conventional computing paradigms [10]–[12].. The CiM paradigm, consisting of a memristive crossbar array, is one potential solution to overcome these challenges. Emerging resistive NVMs (also known as memristive devices) are prominent choices for CiM realization [133]–[141]. The memristive crossbar array is utilized for energy-efficient analog computation of MVM, a core operation for NN accelerators. Many recent works demonstrate that non-volatile memristive devices, specifically multilevel cells such as RRAM, are promising for implementing the CiM paradigm [103], [133]–[137]. The memristive crossbar can be used for energy-efficient analog computation of MVM for deep learning acceleration. However, the functionality is severely affected by defects or non-idealities caused in memristive devices. These include stuck-at-faults, device-to-device variations, cycle-to-cycle variations, write failure, and RTN [97], [98], [104]–[107].

Several works in the literature target defect mitigation techniques for MVM using memristive crossbars [232]–[238]. The works in [237] and [238] focus on checksum-based fault-tolerant and error correction strategies for memristive crossbar realized by multi-bit resistive NVMs. The extended algorithmic-based fault tolerance (X-ABFT) for RRAM-based computing, which utilizes the traditional ABFT and a test vector to extract a signature to mitigate stuck-at faults, has been presented by [237]. The majority voting and Hamming code-based checksum techniques for error correction in RRAM-based matrix operation have been presented in [238]. In this chapter, we present two distinct checksum-based strategies for error correction in the memristive crossbar. The first strategy focuses on single-column error correction, which aims to store the checksum value efficiently. The second strategy is block error correction, designed to protect a group of columns in the crossbar simultaneously.

6.1.1. Scaled checksum-based single column error correction

The digital values are stored by programming the memristive cell in a particular resistive state. The number of stable levels stored by memristive cells is limited due to variations in device resistance from the target programmed value [61]. The limited number of stable levels impacts the maximum value stored by the memristive cell. Due to this limitation, checksum-based techniques require more than one memristive cell to store such a large checksum value, necessitating the use of several checksum columns in a memristive crossbar array, which results in significant memory overhead. These technological constraints have not been addressed in the existing checksum-based technique. We propose a low-overhead, scaled checksum-based single-column error correction technique to overcome the bottleneck of storing checksums into multiple columns of a memristive crossbar. First, we manipulate the data values in the crossbar such that the checksum values become a multiple of the chosen scaling factor. In this way, we only need to store a small factor for the checksum value, which requires only a single memristive cell. The scaled checksum values can be recovered by a left shift operation at the ADC output. However, this data manipulation has a severe negative impact on the inference accuracy. Therefore, we introduce checksum-aware training of NN to recover the accuracy loss caused by manipulation, which incurs no additional computation or storage overhead during inference, yet still achieves comparable accuracy to the original for the MNIST, Fashion-MNIST, CIFAR-10, and Veg-15 datasets.

6.1.2. Checksum-based block error correction

In this strategy, we propose an efficient online block error correction methodology that can correct any number of errors in one of the blocks of a memristive crossbar containing multiple columns, enabling it as a multi-column error correction technique. The proposed method is inspired by fault-tolerant integer parallel MVM, which combines the idea of self-checking capability (checksum) and a linear block code ECC, such as a Hamming code [269]. The proposed strategy can also be extended to perform adaptive error correction, allowing the ratio of data columns to parity checksum columns to be adjusted at runtime based on the fault rate, thereby enabling the optimal use of the crossbar array. In this block error correction strategy, we do not employ storing each checksum in a single column, as this would limit the capability of multi-column error correction and impose excessive restrictions on the values of the weight matrix. However, the proposed strategy requires a smaller code dimension, which can improve the efficiency by reducing the number of checksum columns while enabling multi-column error correction. In this strategy, we employ the Hamming code to get the checksum for the crossbar, which is used to compute the signature for error detection and correction. However, a similar strategy can also be employed by other coding techniques. We also analyze the tradeoff associated with the number of columns protected at a time (a block of crossbar) and the additional required checksum column.

6.2. Analysis of checksum for memristive crossbar

6.2.1. Weighted and Non-weighted Checksum

Fig. 6.1 shows the basic memristive crossbar array used in CiM architecture for parallel MVM. The memristive crossbar can be utilized as an accelerator module for MVM operations in an analog manner. The conductance (resistance) values (G) of memristive cells represent the synaptic weights W of NN. The bit-line (BL) current represents the summation of the multiplication of input vector (v) and conductance matrix (G), i.e., $I_N = \sum_{i=1}^{M} V_i G_{iN}$, i.e. MVM output. In the memristive crossbar, the output of an MVM is taken column-wise. So, it is not possible to get the row checksum and the output simultaneously. Instead of using a row checksum, an extra checksum column can be added to extract the signature for fault detection and error correction. The ABFT technique uses a combination of non-weighted and weighted checksums to extract the signature for error correction [179], [237], [270]. Fig. 6.2 shows an $M \times N$ memristive crossbar with two additional non-weighted and weighted checksum columns on top of crossbar data columns. The non-weighted ($G_{C_{i1}}$) and weighted ($G_{C_{i2}}$) checksum for the i^{th} ($i = 1, 2...M$) row can be computed as follows:

$$G_{C_{i1}} = \sum_{j=1}^{N} G_{ij} \quad \text{and} \quad G_{C_{i2}} = \sum_{j=1}^{N} W_{f_j} G_{ij} \tag{6.1}$$

Where W_{f_j} represents the weight factor, and G_{ij} is the quantized conductance value (quantized weight value of the given weight matrix). In the rest of the paper, G is also treated as the weight matrix of NN. The signature for the error correction is computed by taking the ratio of the weighted to the non-weighted deviation and is given by $A_{faulty} = S_2/S_1$, where $S_1 = I_{C_1} - \sum_{j=1}^{N} I_j$ is the deviation and $S_2 = I_{C_2} - \sum_{j=1}^{N} W_{f_j} I_j$ is the weighted deviation. The correction can be done by subtracting the S_1 from the faulty output current whose column address is indicated by A_{faulty}. Fault detection and correction are performed after the ADC in the digital domain.

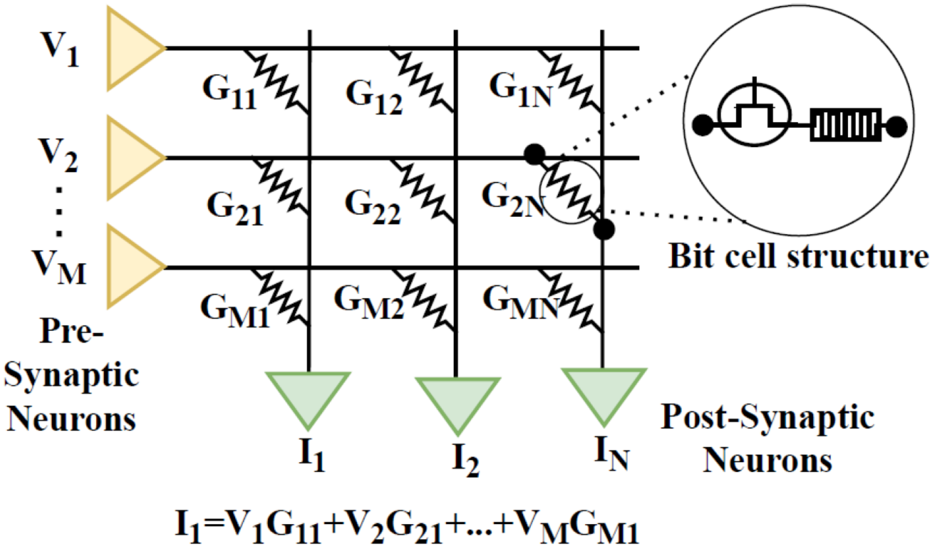


Figure 6.1.: Memristive crossbar

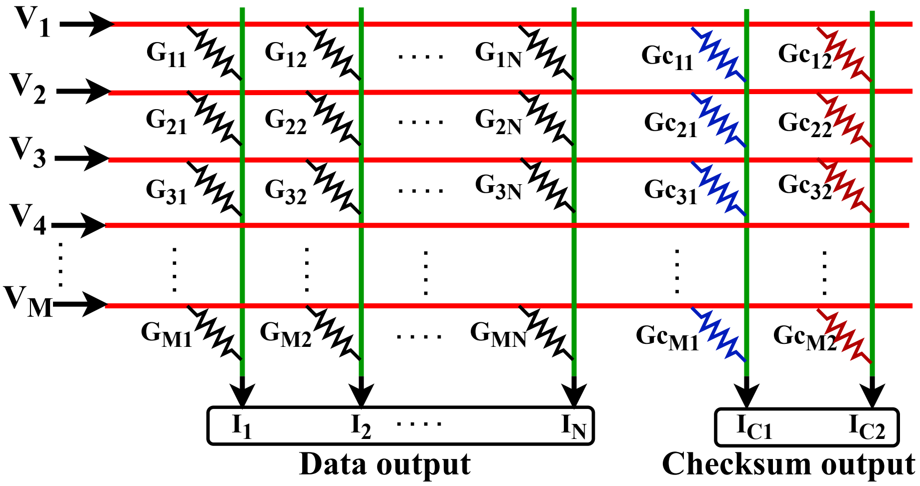


Figure 6.2.: M × N memristive crossbar with two checksum columns

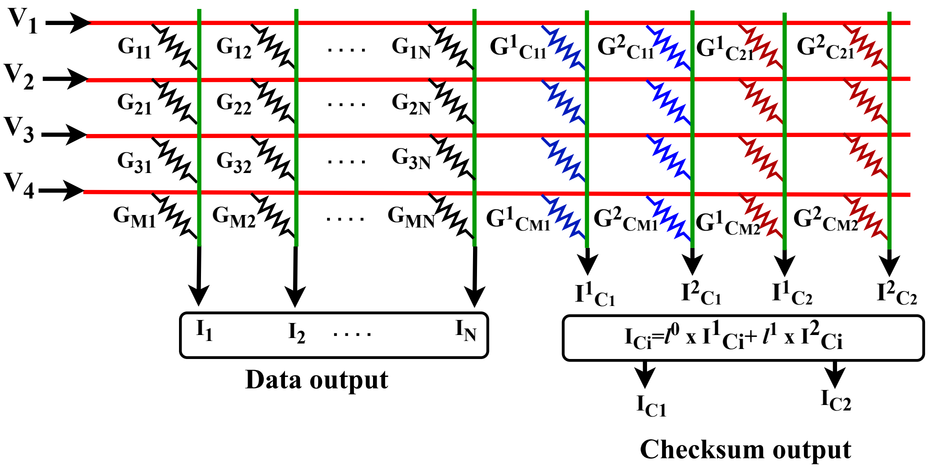


Figure 6.3.: M × N memristive crossbar with multiple checksum columns

Consider an $M \times N$ crossbar with each cell storing a b-bit value, thus $l = 2^b$ different levels for each memristive cell and a linear weight factor ($W_{f_j} = W_{f_1}, W_{f_2}...W_{f_N} = 1, 2...N$) for the weighted checksum calculation. The maximum value of the checksum for each row can be calculated as follows:

$$G_{C_{i1}}^{\max} = \sum_{j=1}^{N} (2^b - 1),$$
$$G_{C_{i2}}^{\max} = \sum_{j=1}^{N} W_j \, (2^b - 1). \tag{6.2}$$

$$G_{C_{i1}}^{\max} = N \, (2^b - 1),$$
$$G_{C_{i2}}^{\max} = \frac{N(N+1)}{2} \, (2^b - 1) \tag{6.3}$$

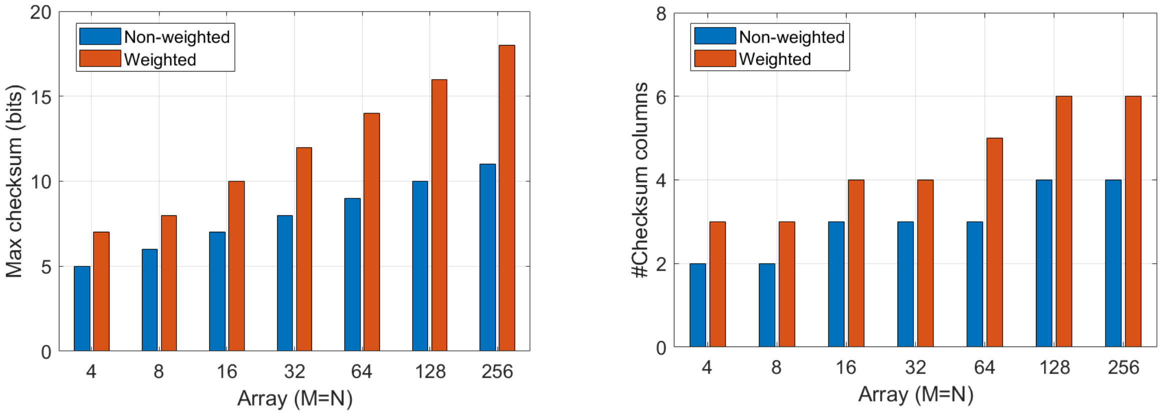
It is not possible to store a checksum value into a single memristive cell due to a limited number of stable states. The weighted column slicing needs to be used to store the checksum value into multiple columns [271]. If the memristive can store l different levels, the output current of each column of a slice would also be weighted by the power of l based on its column address. In general, the number of extra columns in the case of a non-weighted and weighted checksum is given by Equation 6.4. As an example, consider a 64 × 64 crossbar with each cell storing a 3-bit value. The maximum value of a non-weighted checksum is 448, i.e., a 9-bit value, and the maximum value of a weighted checksum is 14560, i.e., a 14-bit value. Therefore, the non-weighted and weighted checksums need to be sliced into three and five columns, respectively, instead of single columns each. The value of non-weighted (I_{C_1}) and weighted checksum (I_{C_1}) current after the slicing are given by Equation 6.5. If the maximum checksum value ($G_C^{\max}$) is a power of 2, then first add 1, and then perform the log_2 operation to avoid any discrepancy in the bit width of the maximum checksum value, because for any b-bit precision, the range is 0 to 2^b-1.

$$\begin{aligned} N_{\text{nw}} &= \left\lceil \frac{\log_2(G_{C_{i1}}^{\max})}{b} \right\rceil, \\ N_w &= \left\lceil \frac{\log_2(G_{C_{i2}}^{\max})}{b} \right\rceil \end{aligned} \tag{6.4}$$

$$I_{C_1} = l^0 \times I_{C_{11}} + l^1 \times I_{C_{12}} \tag{6.5}$$

$$I_{C_2} = l^0 \times I_{C_{21}} + l^1 \times I_{C_{22}} + l^2 \times I_{C_{23}} \tag{6.6}$$

Storing the checksum requires multiple columns due to the limited number of stable states of memristive cells, resulting in significant memory overhead. Fig. 6.4(a) shows the maximum value of the checksum in the case of non-weighted and weighted checksums. In the case of non-weighted and weighted checksums, the value of the weighted checksum is more severe as it increases drastically with the array size. As the array size increases, the maximum checksum value also increases, which imposes the requirements of extra columns into the crossbar as shown in Fig. 6.4(b).



(a) Maximum checksum value

(b) Number of checksum columns

Figure 6.4.: ABFT checksum characteristics for b = 3 bits/cell

6.2.2. Linear Block Code ECC-based Checksum

A checksum technique based on the linear block code ECC involves a non-weighted checksum (algebraic sum of information bits) of the data to determine the parity, rather than using the XOR operation. In the case of a conventional memory system, the linear block code, such as the Hamming code, is used to protect the binary data by adding extra parity bits to the information bits. These parity bits are computed by performing the XOR operation on the information or data bits. In contrast to memory applications, in the case of a crossbar for CiM implementation with multi-bit memory elements, parity is computed by the algebraic sum of data columns instead of the XOR operation. The parity check matrix (H) of ECC determines which data participates in the calculation of the checksum. The selection of the parity check matrix depends on the size of the crossbar, particularly the number of columns.

A 4×4 crossbar array encoded with a $(7, 4)$ Hamming code, having three additional checksum columns (also referred to as parity columns), is shown in Fig. 6.5. The checksum computation for parity is based on the parity check matrix (H). The parity check matrix has two sub-matrices: one is a non-identity matrix (H_N), and the other is an identity matrix (H_I), as shown in Equation (6.7). The location of one in a non-identity matrix determines which part of the crossbar matrix participates in the checksum computation to obtain the parity columns. The checksum value for all three parity columns using $(7, 4)$ code can be calculated as per Eq. (6.8). In general, the checksum for each parity is computed according to the pseudocode in Algorithm 1. This demonstrates the checksum computation as a simple matrix-vector multiplication, where each row value is multiplied by the parity check matrix to get the checksum.

$$H = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{6.7}$$

$$\begin{aligned} G_{C_{i1}} &= G_{i1} + G_{i2} + G_{i3} \\ G_{C_{i2}} &= G_{i1} + G_{i2} + G_{i4} \\ G_{C_{i3}} &= G_{i1} + G_{i3} + G_{i4} \end{aligned} \tag{6.8}$$

Algorithm 1 Checksum computation based on Hamming code

```
Input: Number of columns in crossbar ($N$)
Output: Checksums $G_{C_{ip}}$ for each parity $p$
Select the $(n, k)$ ECC code, $k = N$
Get the corresponding parity check matrix ($H$)
Extract the non-identity part $H_{NI}$ from $H$                ▷ $H = [I\ H_{NI}]$
for $p = 1$ to $P$ do                                          ▷ for each parity $p$
    $h \leftarrow$ row $p$ of $H_N$                            ▷ $h = H_{NI}[p, 1:N]$
    for $i = 1$ to $M$ do                                      ▷ for each row of $G$
        $G_{C_{ip}} \leftarrow G[i, 1:N] \cdot h^{\mathrm{T}}$   ▷ dot product / checksum for parity $p$
    end for
end for
```

Fig. 6.5 shows the 4 × 4 array with three checksum columns for three parity using the (7,4) Hamming code. Here, the assumption is that a single cell per row is sufficient to store the checksum value, resulting in a single column for each parity. However, as discussed previously, this is not practically true because memristive cells have a limited number of stable states, which would impose more than one column for each parity. In Fig. 6.5, instead of three checksum columns, six checksum columns are required as the checksum for each parity needs at least two crossbar columns due to the limited stable state of the memristive cell, as shown in Fig. 6.6. This fundamental technological constraint has not been addressed in the existing linear coding-based checksum for performing error correction in a memristive crossbar.

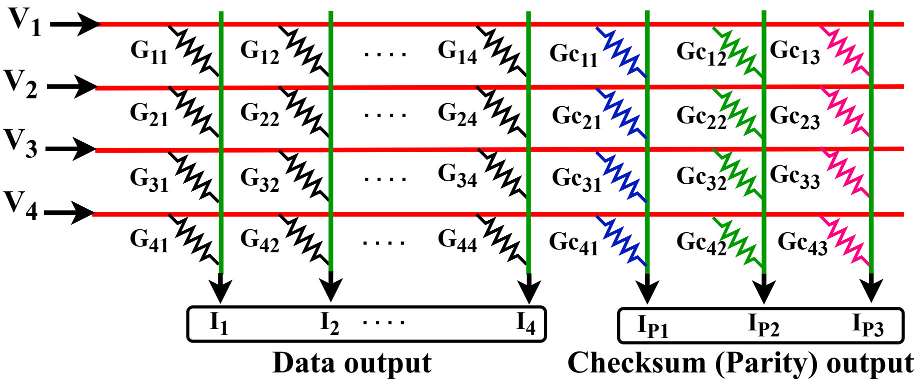


Figure 6.5.: 4 x 4 memristive crossbar with Hamming code-based checksum

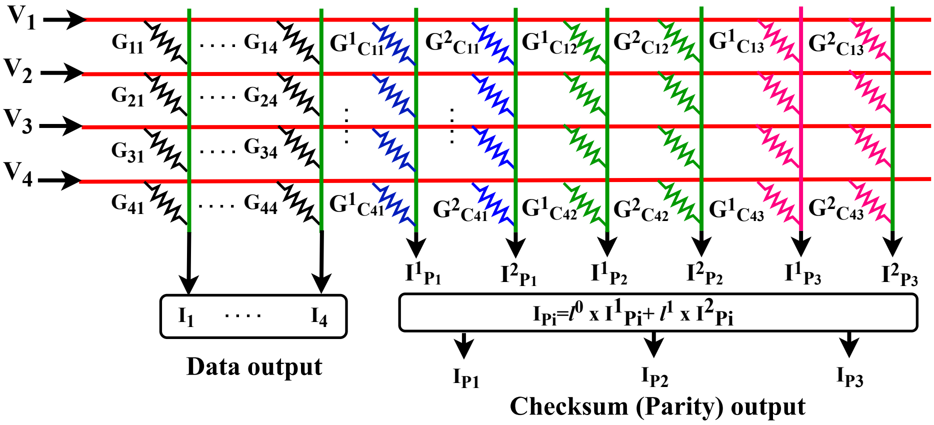
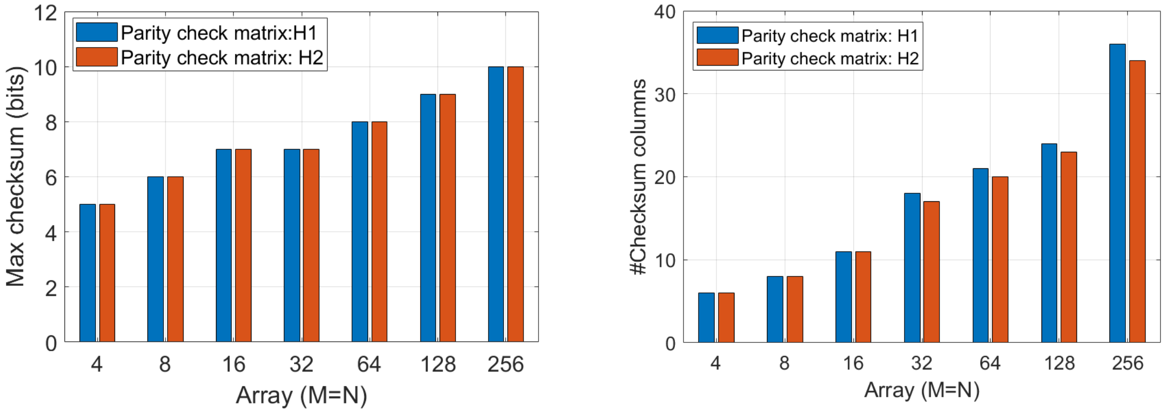


Figure 6.6.: 4 x 4 memristive crossbar with Hamming code-based checksum

Fig. 6.7(a) shows the maximum value of the checksum in the case of the Hamming code-based checksum. In the case of the Hamming code, the checksum values shown in Figure 6.7(a) correspond to the parity having a maximum checksum value. As the size of the array increases, the maximum checksum value also rises. This growth necessitates the addition of extra columns in the crossbar, as shown in Figure 6.7(b). This is due to the fact that the more entries in the columns of the parity check matrix involved in checksum computation, ultimately leading to increased parity checksum values and a higher demand for checksum columns. The specific parity matrix is decided based on the number of columns of the array. For a given column count, different constructions (e.g., generation method or underlying polynomial) may yield slightly different matrices. However, these differences have a minimal impact on the required number of checksum columns, as shown in Fig. 6.7, since the resulting checksum bit widths do not differ substantially in this respect. The details of the parity check matrix are illustrated in the Appendix.



(a) Maximum checksum value

(b) Number of checksum columns

Figure 6.7.: ABFT checksum characteristics for $b = 3$ bits/cell

6.3. Scaled checksum-based single-column error correction

In this strategy, we propose a scaled checksum-based run-time single-column error correction method for a memristive crossbar array. The checksum computation used in this work is based on ABFT, and linear block ECC, such as the Hamming code. The checksum value is the sum of multiple weight values, resulting in a large number that cannot be stored in a single cell and therefore requires multiple cells to store. We propose a method to scale the large checksum value so that it can be stored in a single memristor cell, significantly reducing memory overhead.

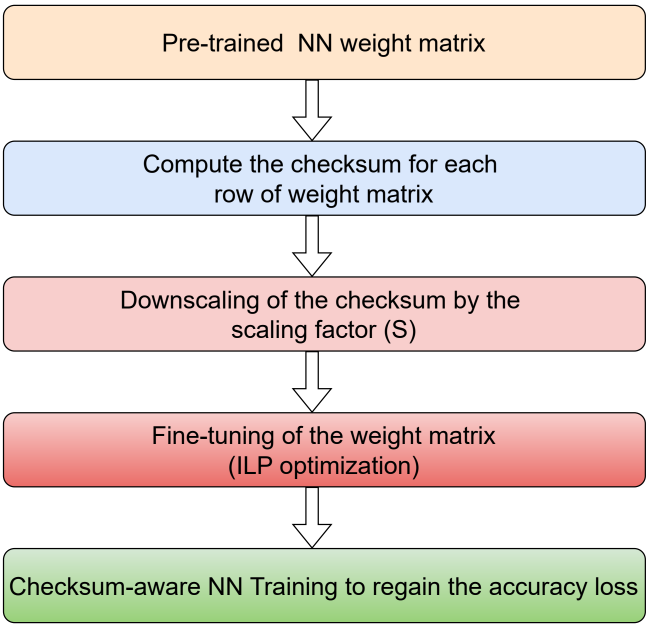

Pre-trained NN weight matrix
Compute the checksum for each row of weight matrix
Downscaling of the checksum by the scaling factor (S)
Fine-tuning of the weight matrix (ILP optimization)
Checksum-aware NN Training to regain the accuracy loss

Figure 6.8.: Overview of checksum scaling process

The overall steps of the proposed checksum scaling strategy are highlighted in Fig. 6.8, and these steps are illustrated as follows:

- First, we compute the checksum for each row of the pre-trained weight matrix. Afterward, we perform the downscaling of the checksum of the trained weight matrix using a scaling factor; that is, each checksum value is divided by the scaling factor, so we only need to store a small factor (the quotient part obtained during scaling).
- However, each checksum value is not always divisible by the scaling factor, i.e., a non-zero remainder exists after the scaling, which may introduce inaccuracy in error detection and correction. Hence, we propose fine-tuning the trained weight matrix, where the weight values are adjusted so that the obtained checksums are divisible by the scaling factor.
- The manipulation of the weight matrix may cause a severe impact on NN accuracy. We propose a checksum-aware training of NN to regain the NN inference accuracy degradation due to the manipulation of the weight matrix.

6.3.1. Down scaling of checksum value

In the case of checksum storage, the checksum value must be sliced into multiple columns due to the limited number of stable states available in the memory device. Therefore, the purpose of scaling down the checksum value is to reduce it to $2^b - 1$ or less, so that it can be easily stored in a single memristive cell with b-bit precision. This will allow us to store each checksum in a single column of the crossbar only. The scaling factor is chosen in such a way that it should be a power of 2 so that it is easy to get the original checksum value by a left shift operation at the output of the ADC in the digital domain. The calculation of the scaling factor for different cases is discussed next.

6.3.1.1. Scaling factors for non-weighted and weighted checksum

In the case of ABFT-based checksum strategy, two kinds of checksums, non-weighted and weighted, are used to extract the signature for error detection and correction. The maximum values of non-weighted and weighted checksums are computed using the Expressions (6.2) and (6.3). The scaling term associated with the non-weighted and weighted checksum (N and $\frac{N\times(N+1)}{2}$) needs to be scaled down to get the checksum value equal to or less than $2^b - 1$. The non-weighted and weighted checksum scaling conditions can be obtained using Equation (6.9) and (6.10). The non-weighted (S_{nw}) and weighted (S_w) scaling factor can be chosen approximately equal to the power of 2 nearest to N and $\frac{N\times(N+1)}{2}$, respectively.

$$\frac{N(2^b - 1)}{S_{nw}} \leq 2^b - 1 \tag{6.9}$$

$$\frac{N(N+1)}{2}\frac{(2^b - 1)}{S_w} \leq 2^b - 1 \tag{6.10}$$

6.3.1.2. Scaling factors for Hamming code-based checksum

In the case of a Hamming code-based checksum, scaling factor computation can be done using a parity check matrix (H). The number of ones in each row of the non-identity matrix of H defines the maximum value of the checksum value for each parity. The parity check matrix would differ for different array sizes, as Hamming codes would vary. The Hamming code dimension (n, k) can be determined based on the number of columns (N) of the weight matrix, which serves as a data block for the Hamming code, where n is the code length and k is the data block size. The following condition must be satisfied when computing the code dimension, which is given as by Equation 6.11 [172], where $k = N$ and $p = n - k$ represent the number of parties. The logic for computing the scaling factor in the case of the Hamming code-based checksum is summarized as pseudocode in Algorithm 2. For example, the parity check matrix for the (7,4) Hamming code is shown in Equation (6.7). To get the scaling factor for the first parity, we need to count the number of ones in the first row (excluding the identity matrix), i.e., 3, so the nearest power of 2 for the value 3 is 4, so we can choose a scaling factor equal to 4. The logic for computing the scaling factor in the case of the Hamming code-based checksum is summarized as pseudocode in Algorithm 2.

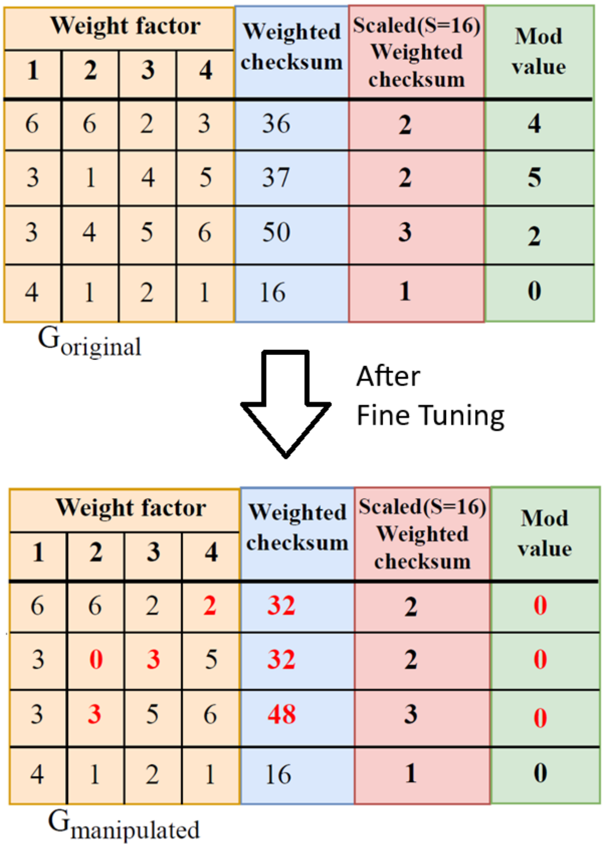
$$k \leq 2^p - p - 1 \tag{6.11}$$

Algorithm 2 Scaling factor computation for Hamming code

```
Input: Number of columns in crossbar ($N$)
Output: Scaling factor $S_H$ for each parity
Select the $(n, k)$ ECC code, $k = N$
Get the corresponding parity check matrix ($H$)
Extract the non-identity matrix ($H_{NI}$) from $H$
for $p = 1$ to $N_P$ do                              ▷ for each parity (1 to $N_P$)
  $count = 0$                                        ▷ count number of $1's$
    for $j = 1$ to $N$ do                            ▷ for column size of $H_{NI}$ (1 to $N$)
        if $H_{NI}(p, j) == 1$ then
            $count = count + 1$
        else
            count = count
        end if
    end for
    $S_H(p)$ = Power of 2 nearest to $count$         ▷ Scaling factor
end for
```

6.3.2. Fine-tuning of the weight matrix

When a scaling factor reduces the checksum value, there can be instances where the checksum is not exactly divisible by the scaling factor. This can result in a non-zero residual remainder, i.e, mod value is not zero after scaling. This remainder can lead to errors during the decoding process. Because the residual would not be recovered during decoding by the left shift operation, this would make the signature for error detection and correction incorrect. To ensure error-free decoding, the checksum value must be divisible by the scaling factor, resulting in a zero remainder (or modulus) after scaling. Practically, it is not always the case that the remainder is zero, as illustrated by Fig. 6.9. It demonstrates an example of a 4 × 4 weight matrix with an additional weighted checksum column, scaled down by a scaling factor of $S = 16$. It can be seen in this example that not all the mod values are equal to zero after scaling the weighted checksum with a scaling factor $S = 16$. This may cause an error while extracting the original checksum value by left-shifting the scaled value at the ADC output. To avoid this error and get a zero mod value after checksum scaling, we need to adjust the values of the weight matrix to ensure the mod value is zero for each checksum. This is illustrated by the example shown in Fig. 6.9.



Weight factor				Weighted checksum	Scaled(S=16) Weighted checksum	Mod value
1	2	3	4			
6	6	2	3	36	2	4
3	1	4	5	37	2	5
3	4	5	6	50	3	2
4	1	2	1	16	1	0

$G_{original}$

After
Fine Tuning

Weight factor				Weighted checksum	Scaled(S=16) Weighted checksum	Mod value
1	2	3	4			
6	6	2	2	32	2	0
3	0	3	5	32	2	0
3	3	5	6	48	3	0
4	1	2	1	16	1	0

$G_{manipulated}$

Figure 6.9.: Example showing fine-tuning of the weight matrix

Given an original weight matrix (G^o), we need to obtain a new manipulated weight matrix (G^m) such that it satisfies the constraints of checksum scaling with minimal changes in the weight values. In other words, the checksum value of the newly obtained manipulated matrix is a multiple of the corresponding scaling factor. To do this, we have formulated the required task as an integer linear programming problem to satisfy the constraints of checksum scaling. For any scaling factor S, the possible mod value can be 1 to $S - 1$, and mod value computation can be done as follows:

$$mod = remainder(\frac{\text{Checksum value}}{S}) \tag{6.12}$$

We have two types of checksums: one is non-weighted, and the other is weighted. In the case of a Hamming code-based checksum, the concept of a non-weighted checksum is also used to compute the checksum for each parity. The constraints for integer linear programming to make the mod value zero after scaling the checksum of the i_{th} row of the weight matrix are given by expression (6.13). Where G_c^o and G_c^m represent the checksum value for the original and manipulated weight matrix, respectively, mod^o represents the mod value after checksum scaling in the case of the original weight matrix. The newly obtained checksum value G_c^m is a multiple of the scaling factor S. The fine-tuning of the weight matrix is summarized as pseudocode in Algorithm 3.

$$\begin{cases} G_{c_i}^m = G_{c_i}^o - mod_i^o, & \text{if } mod_i^o \leq S/2 \\ G_{c_i}^m = G_{c_i}^o - mod_i^o + S, & \text{if } mod_i^o > S/2 \end{cases} \tag{6.13}$$

Algorithm 3 Fine-tuning of the weight matrix

```
Input: $G_{original}(G^o)$ and $mod^o$
Output: $G_{manipulated}(G^m)$
Objective: Minimize $[G^m - G^o]$
for $i = 1$ to $M$ do                                        ▷ for row (1 to $M$)
    for $j = 1$ to $S - 1$ do                                ▷ for mod value (1 to $S - 1$)
        if $mod_i^o = j$ then
            Manipulate $i_{th}$ row of $G^o$ using integer linear
            programming solver to satisfy the checksum
            scaling constraints as follows:
            if $mod_i^o \leq S/2$ then
                $G_{c_i}^m = G_{c_i}^o - mod_i^o$
            else
                $G_{c_i}^m = G_{c_i}^o - mod_i^o + S$
            end if
        else
            Keep $i_{th}$ row as it is                       ▷ i.e., $mod_i^o = 0$
        end if
    end for
end for
```

6.3.3. Checksum-aware NN training

The fine-tuning of the weight matrix manipulates the weight values to satisfy the checksum scaling requirement. However, this will impact the inference accuracy of the trained NN. To mitigate this accuracy degradation, we propose a checksum-aware training of the NN. At the algorithmic level, the typical computation of NN with overall L layers can be represented as a linear transformation of input $\mathbf{x_i}$ followed by batch normalization layer (BatchNorm) [272] and an element-wise nonlinear activation function $\phi(\cdot)$, e.g.,

$$\mathcal{Z}_i = \phi_i(\frac{G_i^o\mathbf{x}_i + \mathbf{b}_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \times \alpha + \beta), \forall i \in [1 \dots L]. \tag{6.14}$$

Where G^o represents the original weight matrix (without manipulation), μ and σ are the mean and variance of the batch normalization layer. α and β are the learnable parameter of BatchNorm layer and ϵ is a small constant.

The manipulated weight matrix is denoted as G^m. The resulting weighted sum $y' \neq y$ is different due to weight manipulation, leading to a higher value of the loss function, $\mathbf{E}' > \mathbf{E}$, where $\mathbf{E}$ is the previously optimized value of a pre-trained NN. The effect of error due to manipulating the entries of the weight matrix needs to be suppressed by optimizing $\mathbf{E}'$ using the backpropagation algorithm. Otherwise, the inference accuracy can degrade to a very low value, corresponding to a randomly initialized NN. Therefore, we introduce an additional parameter Γ with dimension N for the manipulated layers of the NN. Here, N denotes the number of neurons in the layer or the number of data columns in the crossbar. The overall computation of a layer during re-training can be described as:

$$\mathcal{Z}_i = \phi_i(\frac{(G_i^m\mathbf{x}_i + \mathbf{b}_i) \times \Gamma - \mu}{\sqrt{\sigma^2 + \epsilon}} \times \alpha + \beta), \forall i \in [1 \dots L]. \tag{6.15}$$

During retraining, the non-manipulated parameters of the NNs and the Γ parameter are optimized through the backpropagation algorithm, while the manipulated parameters are kept frozen.

Since introducing additional parameters increases both computation and memory overhead during inference, we proposed to squash the Γ parameter into the batch normalization parameter. We utilize the fact that during inference, batch normalization is constant and can be simplified into the following:

$$\mathcal{Z}_i = \phi_i(\frac{(G_i^m\mathbf{x}_i + \mathbf{b}_i) \times \Gamma \times \alpha - \mu \times \alpha}{\sqrt{\sigma^2 + \epsilon}} + \beta), \forall i \in [1 \dots L]. \tag{6.16}$$

$$\mathcal{Z}_i = \phi_i(\frac{(G_i^m\mathbf{x}_i + \mathbf{b}_i) \times \alpha' - \mu'}{\sqrt{\sigma^2 + \epsilon}} + \beta), \forall i \in [1 \dots L]. \tag{6.17}$$

Where $\alpha' = \Gamma \times \alpha$ and $\mu' = mu \times \alpha$ are the new scale and mean of the batch normalization layer after simplification, it follows that during inference, the proposed checksum-aware training does not introduce any additional memory or computation overhead.

6.3.4. Decoding logic for error detection and correction

The decoding scheme would vary slightly compared to the existing Hamming code, which performs decoding based on Hamming distance, and the implementation logic mainly employs XOR-based operations. Whereas, in the case of a Hamming code-based checksum, it would employ algebraic distance-based decoding, so the XOR operation is replaced by the algebraic checksum operations. The proposed downscaling of the checksum does not disturb the existing Hamming code-based decoding structure [238]. Instead, it would avoid the use of additional shift and add operations to obtain the checksum value from multiple columns, where checksum values were stored using column slicing operations. Fig. 6.10 illustrates the decoding logic of the Hamming code-based checksum for the first column of the 4 × 4 crossbar (Fig. 6.5). The output is flagged as erroneous if all syndrome values in rows with a 1 in their parity-check column are equal and non-zero, while all others are zero. The error magnitude is given by the syndrome corresponding to the row with 1 in the parity check matrix (H), and it can be directly added or subtracted from the received output to get the corrected one.

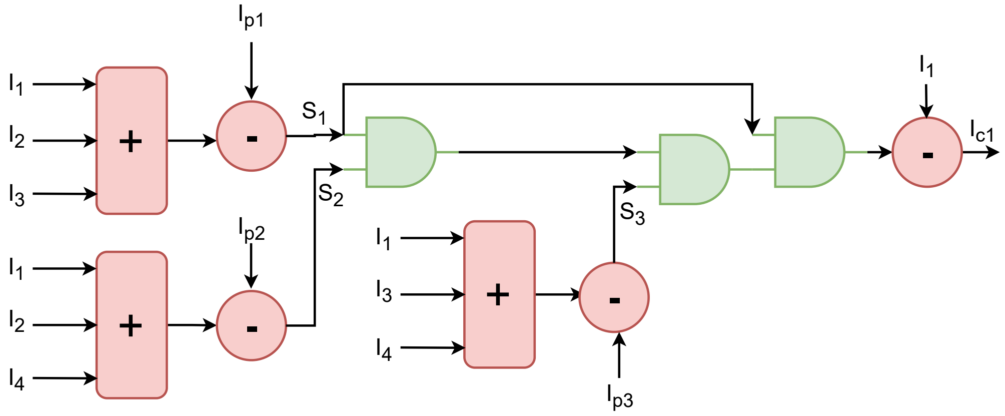


Figure 6.10.: Decoding of column current: I_1 for 4 x 4 memristive crossbar with Hamming code-based checksum

6.4. Results and discussions

6.4.1. Simulation setup

This section presents a detailed analysis of the simulation setup and an evaluation of the proposed scaled checksum-based strategy for single-column error correction in a memristive crossbar. We have considered a $b = 3$ bit per memristive cell used in a crossbar to store weight as well as to perform MVM computation. In the ABFT-based checksum, we have considered the downscaling of the weighted checksum, which is more severe than the non-weighted one. The linear weight factor ($W_f = 1, 2, 3, ...N$) is considered for weighted checksum encoding, which is used along with the non-weighted checksum. In the case of linear block code ECC, we used a Hamming code-based checksum. The checksum computation or the checksum encoding process can be done offline. The checksum computation or encoding process can be performed offline, as it only needs to be done once on a pre-trained weight matrix. Because the value of the weight matrix is fixed during the inference. So, the checksum value can be calculated offline, and then the memristive devices can be easily tuned or programmed to the particular checksum value. The integer linear programming solver for pre-trained NN weight manipulation is implemented using MATLAB.

The proposed approach is evaluated in terms of the number of additional checksum columns required. For the Hamming code-based checksum, we also assess the performance comparison in terms of decoding overhead. The implementation was performed using Verilog HDL and synthesized in Synopsys Design Compiler using the GF 22nm library. To evaluate the proposed method on deep learning applications, we have trained a 3-bit quantized 4-layer multi-layer perceptron with hidden layers containing 32, 64, 128, and 256 neurons for the MNIST and Fashion-MNIST datasets. Additionally, we trained a CNN on the CIFAR-10 and Veg-15 datasets using a ResNet-18 topology. The Veg-15 is a vegetable classification dataset comprising 21,000 real-world images of size 224×224 and 15 different classes [273]. The hidden layers are only encoded as they contain a large number of parameters, which is ~99.5% of total parameters for the ResNet-18 topology. We exclude down-sampling, as well as the first and last layers of each NN model. We have used the quantization algorithm proposed in [274]. The checksum-aware NN training is performed after the NN weight manipulation. This is because training the NN from scratch by including the checksum constraints imposes huge penalties, as it needs to satisfy both functional tasks and checksum constraints. This results in a drastic drop in accuracy, even for very small datasets such as MNIST.

6.4.2. Performance evaluation

Table 6.1 presents the various scaling factors for different crossbar array sizes ($M \times N$), specifically for non-weighted (S_{nw}), weighted (S_w), and Hamming code-based (S_H) checksums. The scaling factor calculation depends on the number of data columns in the crossbar and the bit width or precision (b) of the bit cell used in the crossbar configuration. In the cases of weighted and non-weighted checksums, a single scaling factor is sufficient for each type, as only one checksum is involved in each case. However, for Hamming code-based checksums, multiple scaling factors are necessary for each array size due to the varying number of elements involved in the checksum computation for each parity. For example, for a 32×32 matrix, the $(38, 32)$ code is used, which requires six parities. The scaling factor for the first parity is 8, the second and third require a scaling factor of 16, and the remaining parities need a scaling factor of 32. Similarly, for other array sizes, different scaling factors can be used for different parity cases to ensure the single-column checksum is satisfied for each parity.

Table 6.1.: Scaling factor for different checksums

Array	S_{nw}	S_w	S_H
32×32	32	512	8, 16, 32
64×64	64	2048	8, 32, 64
128×128	128	8192	8, 64, 128
256×256	256	32768	16, 128, 256

Table 6.2.: Number of checksum columns for ABFT-based checksum

Array size	Existing ABFT [237]	Proposed scaled ABFT
32×32	7	4
64×64	8	4
128×128	10	5
256×256	10	5

Table 6.3.: Number of checksum columns required for different array sizes

Array size	Number of Parity	Existing Hamming code-based checksum [238]	Proposed scaled Hamming code-based checksum
32×32	6	17 (18)	6
64×64	7	20 (21)	7
128×128	8	23 (24)	8
256×256	9	34 (36)	9

() corresponds to parity check matrix: H_1

TABLE 6.2 shows the comparative analysis of the existing ABFT-based checksum and the proposed strategy in terms of the number of checksum columns (N_c) required for different array sizes. The proposed technique requires fewer additional columns compared to existing ABFT-based checksums, thereby significantly reducing memory overhead. As an example, for the 128×128 and 256×256 array, the number of checksum columns is reduced from 10 to 5, resulting in a 50% reduction in memory overhead. The ABFT is a software-based fault-tolerant technique that involves an integer division operation to compute a signature for error detection and correction. Implementing the integer division operation on hardware is very costly in terms of parameters such as area, power, and latency. To accelerate the decoding process, the signature computation for fault detection and error correction can be done in a separate arithmetic logic unit (ALU) in a near-memory computing style [237].

TABLE 6.3 presents a comparative analysis of the existing Hamming code-based checksum and the proposed approach in terms of the number of checksum columns (N_c). The proposed scaled checksum-based strategy requires only a single column per parity, which significantly reduces storage requirements. In contrast, the existing Hamming code-based strategy required multiple columns for each parity, resulting in a larger number of columns compared to the number of parity bits. The existing approach may require roughly up to ~3× to ~4× number of columns compared to the proposed strategy, depending on the array size, specifically the number of data columns in an array. While the number of checksum columns may slightly differ for various parity checksum matrices (with roughly a one to two-column variation), the proposed approach is independent of the parity check matrix for the same code, as it only requires one column per parity. In this way, the number of checksum columns equals the number of parity.

Table 6.4 and Fig. 6.11 (normalized) represents the decoding overhead comparison for different array sizes in the case of the Hamming code-based checksum. The proposed strategy employs a similar decoding structure to the existing approach [238]. However, the proposed approach eliminates the need for additional shift and add operations to retrieve the original checksum value from the sliced columns in the crossbar, which are performed during the checksum encoding process. This will result in slightly lower decoding overhead, as the proposed method incurs a smaller area overhead and power in the decoding circuits. This is expected because the proposed method does not require column slicing to store the large checksum value; therefore, no additional hardware overhead (shift and add operation) is needed to obtain the final current checksum value. There is minimal impact on latency, and we used similar timing information during the design synthesis for both cases. Synthesis optimizes the design with nearly the same latency. When considering the latency in terms of cycles, the latency would remain the same.

Table 6.4.: Decoding overhead comparison for different array sizes

Array size	Existing Hamming code-based checksum [238]			Proposed scaled Hamming code-based checksum		
	Area (μm^2)	Latency (ps)	Power (mW)	Area (μm^2)	Latency (ps)	Power (mW)
32×32	1780	691	2.073	1649	670	1.96
64×64	3842	750	5.93	3668	750	5.64
128×128	8294	972	9.93	8003	925	9.69
256×256	17831	1000	21.52	17425	998	20.95

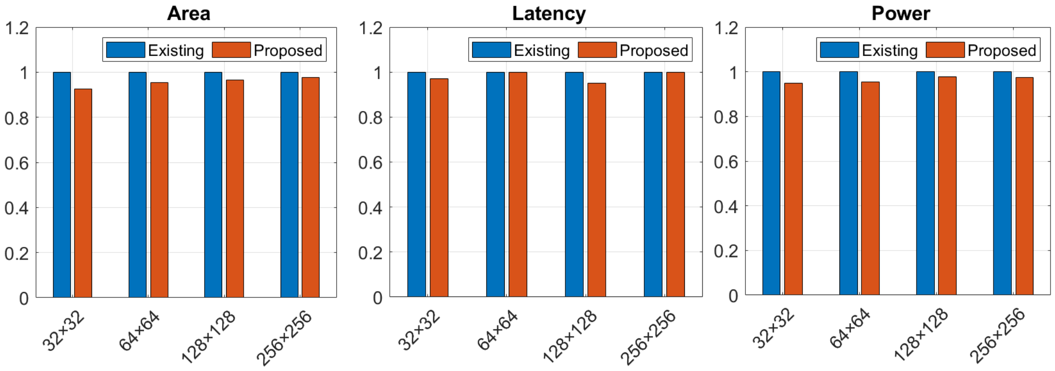


Figure 6.11.: Normalized decoding overhead compared to the existing Hamming code-based checksum

TABLE 6.5, 6.6, and 6.7 show the NN inference accuracy using the proposed scaled checksum-based method for different datasets, respectively. These tables include the original accuracy (baseline), accuracy after fine-tuning the weight matrix, and accuracy after checksum-aware training. The fine-tuning of the weight matrix results in degradation in inference accuracy. It is expected that the accuracy degradation will be more pronounced in the case of the Hamming code-based checksum, as the constraint for checksum scaling must be satisfied for each parity, which requires a significant manipulation of weight values. With the proposed checksum-aware training of the NN, the inference accuracy reaches almost the same level as the original accuracy. The NN inference accuracy after checksum-aware training is within 1% of original accuracy (baseline) for MNIST and Fashion-MNIST datasets, as shown in TABLE 6.5 and TABLE 6.6, respectively. The inference accuracy is within ~3% of original accuracy (baseline) for CNN datasets: CIFAR-10 and Veg-15, as shown in TABLE 6.7 respectively. Thus, with proposed checksum-aware training, the results show a negligible impact on inference accuracy.

Table 6.5.: Inference accuracy for MNIST dataset

N	Baseline	After fine-tuning		After checksum-aware training	
		ABFT	Hamming	ABFT	Hamming
32	95.62%	71.99%	19.74%	96.64%	95.47%
64	96.67%	40.58%	8.92%	97.17%	96.17%
128	97.05%	46.50%	10.32%	97.13%	97.33%
256	98.52%	13.54%	10.10%	98.40%	97.78%

Table 6.6.: Inference accuracy for Fashion-MNIST dataset

N	Baseline	After fine-tuning		After checksum-aware training	
		ABFT	Hamming	ABFT	Hamming
32	84.73%	77.42%	6.59%	84.96%	83.94%
64	86.31%	62.73%	10.0%	87.10%	85.53%
128	87.06%	78.32%	10.0%	87.79%	87.12%
256	88.14%	39.37%	9.56%	88.59%	87.31%

Table 6.7.: Inference accuracy for CNN datasets

Dataset	Baseline	After Fine-tuning		After checksum-aware training	
		ABFT	Hamming	ABFT	Hamming
CIFAR-10	91.63%	9.86%	9.98%	89.06%	88.00%
Veg-15	99.16%	6.66%	6.66%	96.30%	95.63%

6.5. Checksum-based block error correction in a memristive crossbar

6.5.1. Fault-tolerant structure for memristive crossbar

The basic structure of the proposed strategy is shown in Fig.6.12, in which the $M \times N$ memristive crossbar array is divided into B sub-arrays. Each sub-array is also called as sub-block and denoted as G_{sub} ($sub = 1, 2, ..., B$) having dimension $M \times N_B$, where $N_B = \frac{N}{B}$. Two redundant crossbar matrices are included with the original crossbar to enable the fault-tolerant capability, one is a sum matrix $[G_{sum}]_{M \times N_B}$, and the other is a detection matrix $[G_D]_{M \times N_D}$, respectively. The main crossbar array ($[G] = [G_1 G_2 ... G_B]$) and the redundant crossbar array (G_{sum} and G_D together) are also called the data array and checksum array, respectively. The data output of crossbar $[I_{G_{sub}}]_{1 \times N_P}$ ($sub = 1, 2...B$) is compared with the detection output $[I_D]_{1 \times N_D}$ to detect the faulty output. The computation of the correct output vector $[I_{c_{sub}}]_{1 \times N_B}$can be done by subtracting the correct vectors among ($[I_{G_1}], [I_{G_2}], ...[I_{G_B}]$) from the sum matrix $[I_{sum}]$. The assumption is that any one of the sub-blocks can be faulty at a time, so the Hamming code can be used for error detection by adding some redundant columns into the crossbar, which is represented by the detection matrix. The number of columns (N_D) of the detection matrix is a function of the number of sub-blocks, and it can be given as follows [172], [269]:

$$B = f(N_D) = 2^{N_D} - N_D - 1 \tag{6.18}$$

In other words, the number of parties in a given Hamming code denotes the number of columns of the detection matrix (N_D). For the B number of sub-blocks, the dimension of the Hamming code can be given as $(n, k) = (B + N_D, B)$. Consider an example with $B = 4$, the value of N_D would be equal to 3 (as per 2.11), so $(n, k) = (7, 4)$. The number of error correction capabilities can be determined by the number of columns in each sub-block, represented by N_{sub}. The magnitude of multi-column error correction capability varies for different code dimensions.

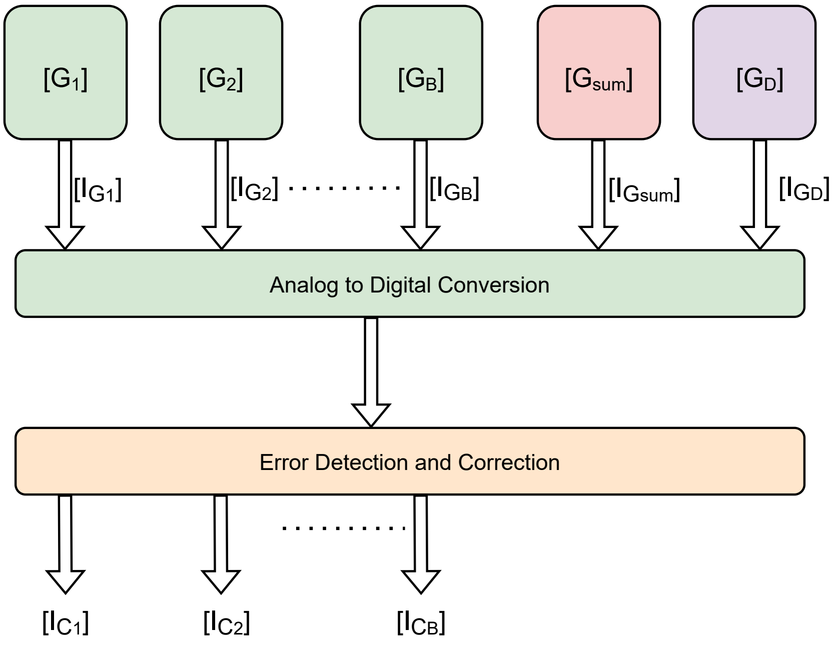


Figure 6.12.: Fault-Tolerant Structure for block error correction

6.5.2. Computation of sum matrix

The data matrix $[G]_{M\times N}$ of the crossbar is a collection of all the sub-blocks, in other words, $[G] = [G_1, G_2, ..., G_B]$. The computation of the sum matrix can be obtained by taking the sum of all the blocks of the original crossbar matrix, and it is given as follows:

$$[G_{sum}] = [G_1] + [G_2] + ... + [G_B] = \sum_{sub=1}^{B} [G_{sub}]_{M\times N_B} \tag{6.19}$$

$$\mathbf{G} \in \mathbb{Z}_{\geq 0}^{8\times 8}, \quad [G] = [\,\mathbf{G}_1 \;\; \mathbf{G}_2 \;\; \mathbf{G}_3 \;\; \mathbf{G}_4\,], \qquad \mathbf{G}_k \in \mathbb{Z}_{\geq 0}^{8\times 2}. \tag{6.20}$$

$$[G] = \begin{bmatrix} 1 & 0 & 3 & 2 & 5 & 1 & 4 & 0 \\ 2 & 1 & 4 & 0 & 6 & 2 & 5 & 1 \\ 0 & 2 & 1 & 3 & 4 & 0 & 6 & 2 \\ 3 & 1 & 0 & 2 & 7 & 3 & 5 & 1 \\ 4 & 2 & 3 & 1 & 6 & 2 & 4 & 0 \\ 5 & 0 & 2 & 1 & 7 & 1 & 3 & 2 \\ 1 & 3 & 4 & 2 & 5 & 0 & 2 & 1 \\ 0 & 1 & 3 & 4 & 2 & 5 & 6 & 7 \end{bmatrix} = \left[\underbrace{\begin{matrix} 1 & 0 \\ 2 & 1 \\ 0 & 2 \\ 3 & 1 \\ 4 & 2 \\ 5 & 0 \\ 1 & 3 \\ 0 & 1 \end{matrix}}_{\mathbf{G}_1} \;\; \underbrace{\begin{matrix} 3 & 2 \\ 4 & 0 \\ 1 & 3 \\ 0 & 2 \\ 3 & 1 \\ 2 & 1 \\ 4 & 2 \\ 3 & 4 \end{matrix}}_{\mathbf{G}_2} \;\; \underbrace{\begin{matrix} 5 & 1 \\ 6 & 2 \\ 4 & 0 \\ 7 & 3 \\ 6 & 2 \\ 7 & 1 \\ 5 & 0 \\ 2 & 5 \end{matrix}}_{\mathbf{G}_3} \;\; \underbrace{\begin{matrix} 4 & 0 \\ 5 & 1 \\ 6 & 2 \\ 5 & 1 \\ 4 & 0 \\ 3 & 2 \\ 2 & 1 \\ 6 & 7 \end{matrix}}_{\mathbf{G}_4} \right].$$

The block-wise sum (element-wise over corresponding positions) is given as follows:

$$\mathbf{G}_{\text{sum}} = \sum_{k=1}^{4} \mathbf{G}_k = \mathbf{G}_1 + \mathbf{G}_2 + \mathbf{G}_3 + \mathbf{G}_4 \in \mathbb{Z}_{\geq 0}^{8\times 2}$$

Equivalently, since each block contributes two columns, the two columns of $\mathbf{G}_{\text{sum}}$ are

$$\mathbf{G}_{\text{sum}}(:,1) = \mathbf{G}_1(:,1) + \mathbf{G}_2(:,1) + \mathbf{G}_3(:,1) + \mathbf{G}_4(:,1) = \mathbf{G}(:,1) + \mathbf{G}(:,3) + \mathbf{G}(:,5) + \mathbf{G}(:,7),$$
$$\mathbf{G}_{\text{sum}}(:,2) = \mathbf{G}_1(:,2) + \mathbf{G}_2(:,2) + \mathbf{G}_3(:,2) + \mathbf{G}_4(:,2) = \mathbf{G}(:,2) + \mathbf{G}(:,4) + \mathbf{G}(:,6) + \mathbf{G}(:,8).$$

$$\mathbf{G}_{\text{sum}} = \begin{bmatrix} 13 & 3 \\ 17 & 4 \\ 11 & 7 \\ 15 & 7 \\ 17 & 5 \\ 17 & 4 \\ 12 & 6 \\ 11 & 17 \end{bmatrix}$$

6.5.3. Computation of detection matrix

For the computation of a detection matrix, first, the checksum columns for each sub-block need to be obtained and given as follows:

$$[C_1] = \begin{bmatrix} c_1^1 \\ c_2^1 \\ \vdots \\ c_M^1 \end{bmatrix}, [C_2] = \begin{bmatrix} c_1^2 \\ c_2^2 \\ \vdots \\ c_M^2 \end{bmatrix} \ldots [C_B] = \begin{bmatrix} \sum_{j=1}^{N_B} G_{1j}^B \\ \sum_{j=1}^{N_B} G_{2j}^B \\ \vdots \\ \sum_{j=1}^{N_B} G_{Mj}^B \end{bmatrix} \tag{6.21}$$

Continuing the previous example of matrix $[G]$, the checksum column values are given as follows:

$$\mathrm{C}_1 = \begin{bmatrix} 1 \\ 3 \\ 2 \\ 4 \\ 6 \\ 5 \\ 4 \\ 1 \end{bmatrix}, \quad \mathrm{C}_2 = \begin{bmatrix} 5 \\ 4 \\ 4 \\ 2 \\ 4 \\ 3 \\ 6 \\ 7 \end{bmatrix}, \quad \mathrm{C}_3 = \begin{bmatrix} 6 \\ 8 \\ 4 \\ 10 \\ 8 \\ 8 \\ 5 \\ 7 \end{bmatrix}, \quad \mathrm{C}_4 = \begin{bmatrix} 4 \\ 6 \\ 8 \\ 6 \\ 4 \\ 5 \\ 3 \\ 13 \end{bmatrix} \tag{6.22}$$

Consider the number of sub-blocks $B = 4$, so the parity check matrix of the $(7, 4)$ Hamming code can be used to generate the columns of the detection matrix ($[G_D] = [D_1 D_2 D_3]$) and given by Equation (6.23). The mapping of each column of the sum and the detection matrix requires multiple crossbar columns due to the limited number of stable states of the memristive cell.

$$\begin{aligned} [D_1] &= [C_1] + [C_2] + [C_3] \\ [D_2] &= [C_1] + [C_2] + [C_4] \\ [D_3] &= [C_1] + [C_3] + [C_4] \end{aligned} \tag{6.23}$$

Combined checksum (C) and detection (D) computation from the same example 8×8 matrix ($[G]$) are given as follows:

$$\mathrm{G}_D = [\,\mathbf{D}_1 \;\; \mathbf{D}_2 \;\; \mathbf{D}_3\,] = \begin{bmatrix} 12 & 10 & 11 \\ 15 & 13 & 17 \\ 10 & 14 & 14 \\ 16 & 12 & 20 \\ 18 & 14 & 18 \\ 16 & 13 & 18 \\ 15 & 13 & 12 \\ 15 & 21 & 21 \end{bmatrix}$$

In general, for a given crossbar size, number of blocks, and number of detection columns, the detection matrix computation can be obtained by pseudocode shown in Algorithm 4.

Algorithm 4 Detection matrix computation based on Linear block code ECC

```
Input: Number of blocks ($B$), Number of columns in detection matrix ($N_D$)
Output: Detection Checksums $D_p$ for each parity $p$
Select the $(n, k)$ ECC code, $k = B$, $n = k + N_D$
Get the corresponding parity check matrix ($H$)
Extract the non-identity part $H_{NI}$ from $H$                ▷ $H = [I\ H_{NI}]$
for $p = 1$ to $P$ do                                           ▷ for each parity $p$
    $h \leftarrow$ row $p$ of $H_{NI}$                          ▷ $h = H_{NI}[p, 1{:}N]$
    $\mathrm{Nz}(h) \leftarrow \{\, j \in \{1, \ldots, k\} \mid h_j \neq 0 \,\}$   ▷ location of non-zero element in $h$
    $\mathrm{D}_p \leftarrow \mathbf{0}$
    for all $j \in \mathrm{Nz}(h)$ do
        $\mathrm{D}_p \leftarrow \mathrm{D} + \mathrm{C}_j$
    end for
end for
```

6.5.4. Signature computation for error detection and correction

The signature computation for error detection and correction is based on checking the deviation of crossbar blocks from the output of the detection matrix. Consider a (7,4) Hamming code as an example. The syndrome ($[S]_{1\times N_D} = [S_1 S_2 S_3]$) calculation for error detection can be given as follows:

$$\begin{aligned}
S_1 &= I_{D_1} - ([I_{G_1}] + [I_{G_2}] + [I_{G_3}]) \\
S_2 &= I_{D_2} - ([I_{G_1}] + [I_{G_2}] + [I_{G_4}]) \\
S_3 &= I_{D_3} - ([I_{G_1}] + [I_{G_3}] + [I_{G_4}])
\end{aligned} \tag{6.24}$$

As per the expressions (6.23) and (6.24), if the syndrome vectors are equal to zero, in such a case, there is no error; otherwise, depending on the values of the syndrome vectors, any one of the sub-blocks of the crossbar can be faulty, which need to be corrected. Detection and correction are performed after the ADC and in the digital domain. Each syndrome vector can be multi-bit values so that for the non-zero syndrome value, the single-bit syndrome value can be represented by logic “1”, otherwise, it is represented by logic “0”. For example, in the case of the (7,4) code, if all the values of the syndrome vector are equal to ones ($S = [111]$), it indicates that the sub-block G_1 is faulty. The error-free output of sub-block one can be obtained by subtracting the remaining correct output vectors from the output of sum matrix, e.g., $[I_{c_1}] = [I_{s_1}] - ([I_3] + [I_5] + [I_7])$. For the (n, k) Hamming code, there can be $2^{(n-k)}$ possible syndrome values to detect the location of the faults. Similarly, other possible Hamming codes can also be used to detect the error in the crossbar array.

6.5.5. Fault-tolerant structure for an example array size: 8 × 8

Fig. 6.13 shows the block error correction-based fault-tolerant structure for an 8 × 8 crossbar array using the ($n = 7, k = 4$) Hamming code technique. The data array of a crossbar is divided into four sub-blocks ($B = 4$), each containing two columns. The number of detection columns is given as $N_D = n - k = 3$, which also determines the number of syndrome symbols. The syndrome ($S = [S_1 S_2 S_3]$) computation for error detection is shown in Fig.6.14. Consider a case with $S = [111]$, which indicates the sub-block G_1 is faulty; in other words, the output current I_1 or I_2 or both can have an error depending on the location of the faults in the corresponding sub-block. The computation of corrected output value I_{c_1} and I_{c_2} are shown in Fig.6.14. Similarly, the correction can also be done for the other sub-blocks.

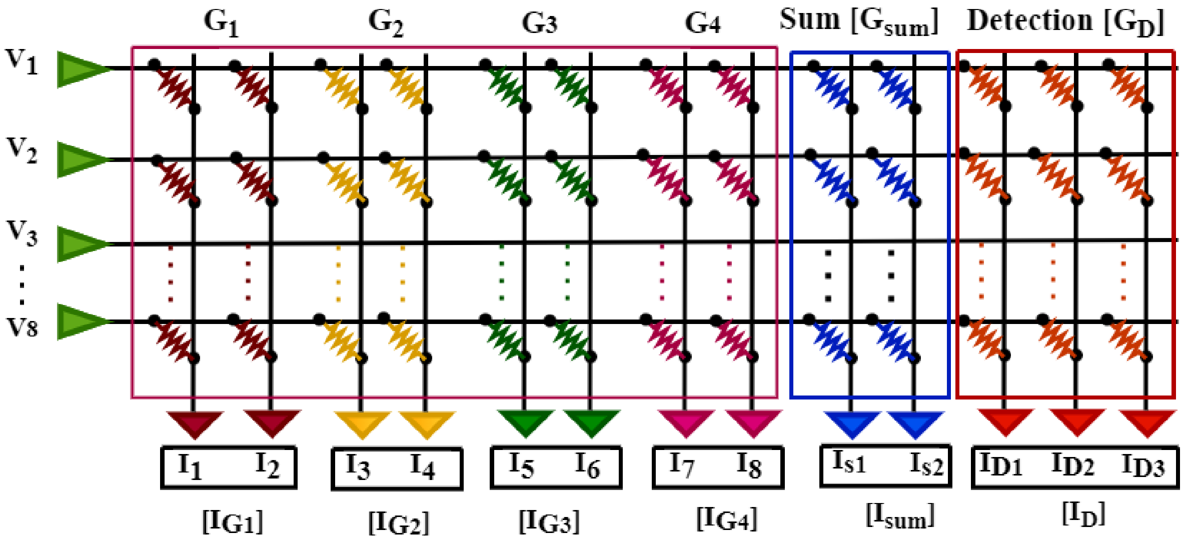


Figure 6.13.: Fault-tolerant structure for 8 × 8 crossbar with (7,4) Hamming code

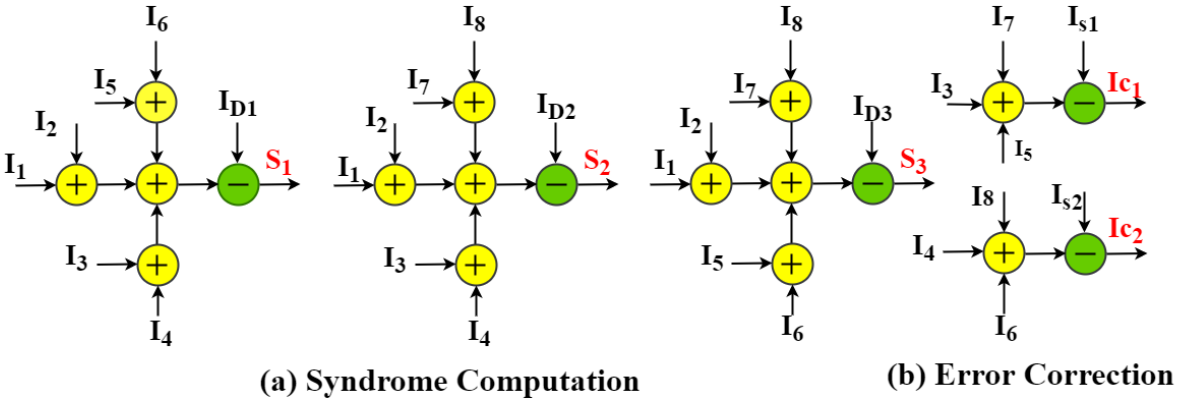


Figure 6.14.: Syndrome computation and Error Correction (sub-block 1) for 8 × 8 crossbar with (7,4) Hamming code

6.5.6. Evaluation of block error correction strategy for a memristive crossbar

This section presents a detailed performance evaluation of the proposed block error correction strategy methodology. In this work, we have considered $b = 3$ bits per memristive cell. The checksum encoding can be performed offline on the trained weight matrix, and the checksum value can be written into the memristor cells just once. Since the weight matrix values remain fixed during inference, the checksum value will not change throughout this process. The proposed methodology is evaluated in terms of the number of checksum columns required and decoding overhead. The proposed scheme was implemented using Verilog and synthesized in Synopsys Design Compiler using the GF 22 nm library. Table 6.8 lists the required Hamming codes for various array sizes, with the corresponding parity matrix (H_2) presented in the Appendix.

Table 6.9 and 6.10 represent the overhead associated with different array sizes for $N_E = 4$ and $N_E = 8$, $N_E = N_B$ represents the number of columns under one block, meaning the number of data columns that can be protected at a time. That means it can protect all the columns of that particular block. The proposed strategy employs a single block protection approach, enabling the protection of all columns within that block. The number of data columns in a block depends on the selected code dimension for a given array size. Fig. 6.15 represents the number of extra checksum columns required (N_C), number of columns protected (N_E), decoding overhead (area, power, latency) for different code dimensions in case of different array sizes. For a smaller code dimension, the strategy provides a larger number of error corrections, as the number of protected columns increases for a smaller code. However, it will also result in high memory overhead, as can be seen in Fig. 6.15, the number of required checksum columns (N_C) increases drastically. Regarding area and power overhead, larger codes will require slightly higher overhead for a given array size, as they involve more syndrome computations. For latency, there is not much difference; we use the same timing constraints for all the code dimensions for a given array, and the optimization results in nearly the same latency.

Table 6.8.: Hamming code dimension for different arrays

Crossbar array	Possible Hamming coding dimensions(n,k)
32×32	$(7,4),(12,8),(21,16)$
64×64	$(7,4),(12,8),(21,16),(38,32)$
128×128	$(7,4),(12,8),(21,16),(38,32),(71,64)$
256×256	$(7,4),(12,8),(21,16),(38,32),(71,64),(136,128)$

Table 6.9.: Decoding overhead for different array sizes in case of four columns per block (N_B)

Array	Code	$N_E = N_B$	N_C	Area (μm^2)	Latency (ps)	Power (mW)
32×32	(12,8)	4	20	2381	818	3.067
64×64	(21,16)	4	27	5599	994	7.96
128×128	(38,32)	4	30	12424	1169	12.34
256×256	(71,64)	4	39	27634	1366	27.61

Table 6.10.: Decoding overhead for different array sizes in case of eight columns per block (N_B)

Array	Code	$N_E = N_B$	N_C	Area (μm^2)	Latency (ps)	Power (mW)
32×32	(7,4)	8	25	1955	990	2.61
64×64	(12,8)	8	28	4967	998	6.51
128×128	(21,16)	8	40	11812	1400	11.41
256×256	(38,32)	8	47	26051	1498	26.091

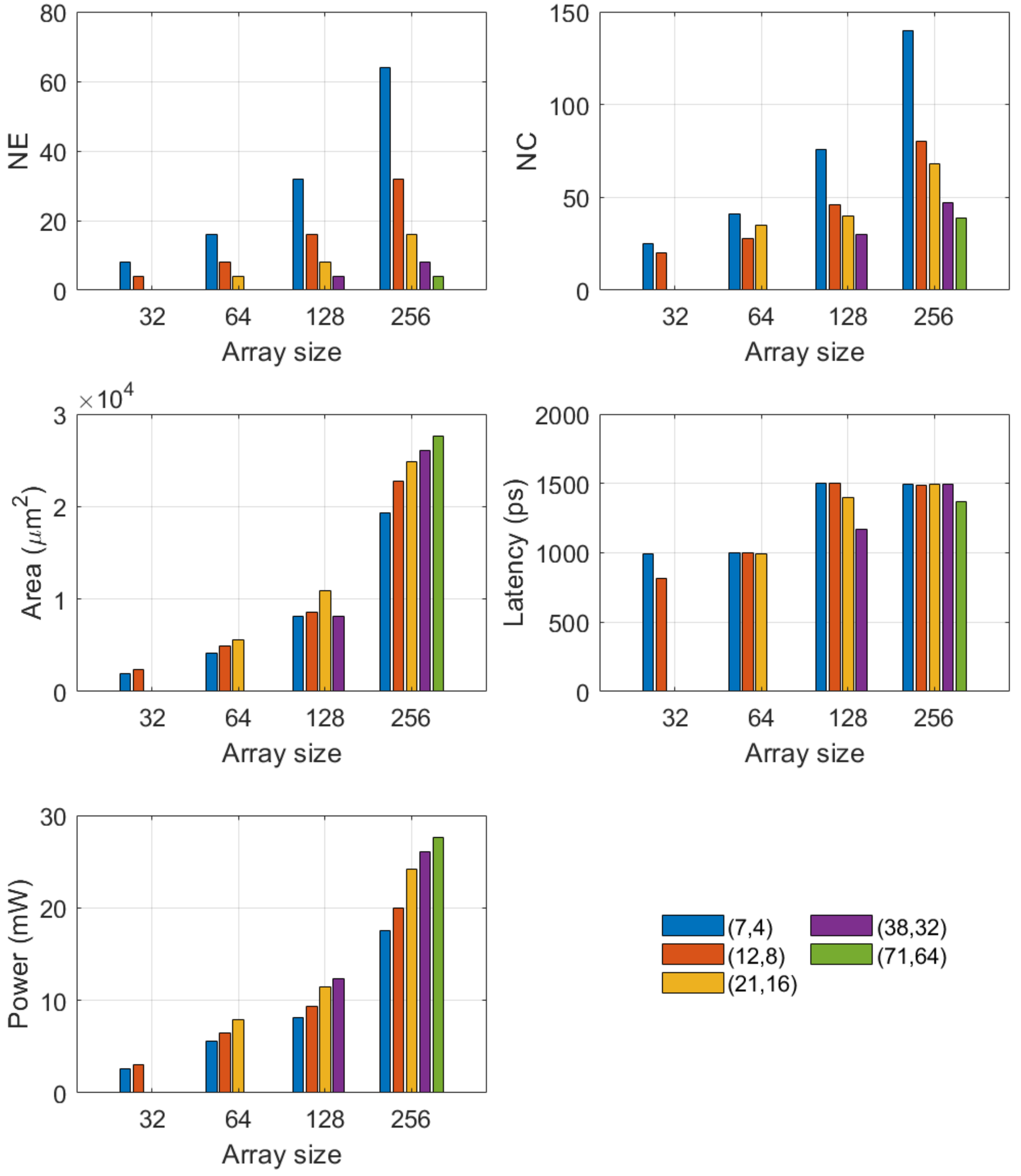


Figure 6.15.: Decoding overhead trend for different array sizes in case of block error correction

Fig. 6.16 presents a comparative analysis of the proposed block error correction with existing Hamming code checksum-based single column error correction ($t = 1$) [238], majority voting checksum-based single column error correction ($t = 1$) [238], double error correction ($t = 1$) strategies, including conventional binary OLS code [275], [276] and non-binary symbol-based OLS code [277]. Here, for the proposed strategy, two different cases are considered, one with four columns ($N_E = 4$) and another with eight columns ($N_E = 8$) in each sub-block (i.e., four and eight column correction, respectively), where as double column error correction is considered for the OLS code. Compared to Hamming code checksum-based single error correction [238], the proposed strategy can provide a larger number of error corrections with slightly higher memory overhead, as we require a slightly higher number of checksum columns.

Compared to majority voting-based checksum [238], the proposed strategy provides a larger number of error corrections with fewer checksum columns. In contrast to the majority voting-based checksum method [238], the proposed strategy delivers a higher number of error corrections while using fewer checksum columns. For the majority voting scheme, we adopt a conservative approach regarding the number of checksum columns. Although the exact construction of the parity check matrix is not detailed in [238], it is important to note that the checksums for each parity will involve summation over multiple data elements. Thus, we assume at least two columns for each parity; in reality, some parity may require more checksum columns since multiple data elements will be involved in the checksum computation. When compared to the binary OLS code, the proposed strategy is demonstrated to be more efficient due to its reduced number of checksum columns. For the OLS code, roughly the number of data elements participating for each parity is around $\sqrt{N}$ for an $M \times N$ array. Furthermore, a comparison with the symbol-based non-binary OLS code [277] reveals that, while it requires a similar number of symbols compared to the binary OLS code, it stores symbol-based parity instead of computing a checksum. This results in fewer checksum columns compared to binary OLS code, equivalent to the number of extra parities for the OLS code. For example, in the case of $M \times N$ array, the number of parity bits is equal to $2 \times t \times \sqrt{N}$, where t is the number of error correction. The OLS code is efficient at decoding, but it has significant memory overhead that would increase further with even a slight rise in error correction.

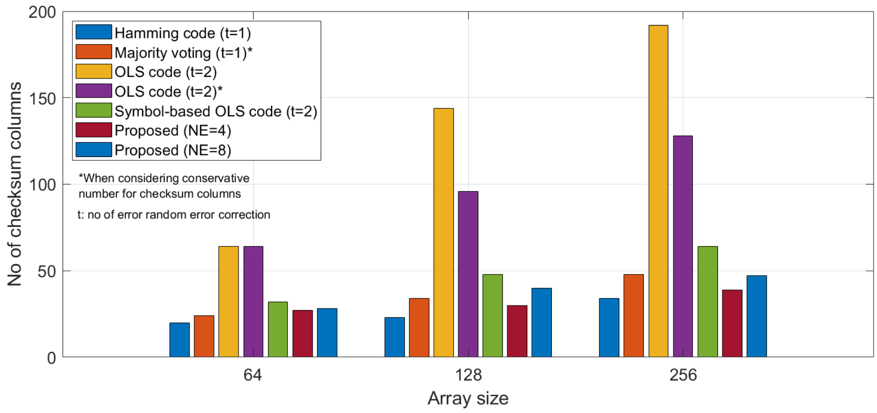


Figure 6.16.: Comparative analysis of the number of checksum columns required

6.6. Summary

In this chapter, we have developed different strategies for performing error correction in a memristive crossbar array, an architecture used to accelerate the MVM operation, a core operation of NN computation. First, we have proposed a scaled checksum-based technique for single-column error correction in the memristive crossbar, utilizing two methods: an ABFT-based checksum and a Hamming code-based checksum. In this technique, each checksum is down-scaled so that it can be stored in a single column of the crossbar, resulting in a significant reduction in the number of extra checksum columns in a crossbar. Secondly, we have also demonstrated the use of linear block code ECC, such as the Hamming code, to perform block error correction in a crossbar, where a block of the crossbar can be protected at a time, thereby enabling multi-column error correction.

7. Zero Overhead NN-ECC for Binary Weight Memories of NN Hardware Accelerators

NN have demonstrated outstanding performance in various domains, resulting in widespread deployment on diverse hardware devices. They require large memories to store the NN weight parameters, which are susceptible to numerous permanent and transient faults. Therefore, error detection and correction mechanisms with certain guarantees should be provided to ensure reliable NN operation, especially in safety-critical applications. ECCs are a common approach to protecting memories against these failures, but they impose significant memory overheads. In this chapter, we propose a zero overhead NN-ECC that employs a multi-task learning objective, integrating linear block ECC into the NN parameters during training without imposing additional memory overhead for ECC parity bits. Unlike existing methods, which selectively eliminate the storage overhead with limited error correction and necessitate specific weight distributions to utilize redundant weight bits for ECC parity, our approach is versatile. It can accommodate various ECC schemes with different correction capabilities without imposing constraints on weight distributions. In this chapter, we present a detailed approach to the ECC encoding process for NN weight memories, along with the corresponding decoding strategy. Afterward, we detailed the multi-task learning strategy to accommodate the ECC parity bits without affecting the size of the weight matrix. Finally, we discuss the performance evaluation of the proposed strategy with various ECC code dimensions, and different NN models and datasets.

7.1. Introduction

The advancements in deep learning algorithms have led to the impressive performance of NN in a wide range of tasks, including image recognition, object detection, and segmentation. NN models are deployed in various hardware devices to perform inference using trained weight parameters. The trained weights are stored in the memory system associated with the hardware accelerators. However, memory reliability plays a pivotal role, as errors within stored trained weights can severely impair NN performance [157]–[161]. These faults can be catastrophic, especially in safety-critical applications. Hence, it is crucial to improve the reliability of NN by safeguarding its weights from errors. The details about the overview of memory errors and their impacts are discussed in Chapter 2. In conventional memories, ECCs have been used to provide a guarantee for fault-free operation up to a specific fault rate. However, this comes at a considerable storage overhead, as well as encoding and decoding overhead (in terms of area, power, and latency), as discussed in Section 2.7. Several prior works in the literature target defect mitigation strategies for NNs deployed over different hardware accelerators that employ ECC as an alternative solution that guarantees the mitigation of defects by restoring the erroneous weights stored in the memory of the NN hardware accelerators [221]–[225], [227], [228], [232]–[238], [240], [241]. Some of these works [221]–[225], [227], [228], [233], [234], [236] target the binary weight memories of different NN hardware accelerators. In this chapter, we also focus on binary-weight memories.

Since NNs have millions of weight parameters, the use of ECC can lead to substantial memory overhead, as many additional parity bits also need to be accommodated on top of the original weight parameters. For example, when protecting 64-bit weight parameters, the use of Hamming code would impose an additional 7 bits in case of SEC and 8 bits in case of SEC-DED. Consequently, a few of the works aim to eliminate the ECC memory overhead for NN applications [221], [222], [227], [228]. In general, those works have certain limitations: a) require NN models with specific weight distribution, which allows a higher-order bit of each weight to store the ECC parity bits b) require a higher quantization level that already contains redundancies in LSBs or MSBs of each weight, c) have limited error detection and correction capabilities, restricting their ability to accommodate more parity bits for higher correction capabilities and d) are not able to eliminate the memory overhead for higher correction capabilities (beyond their limit). These restrictions hinder the goal of achieving zero-overhead ECC for NN applications in a generalized manner that can correct even multi-bit errors without any restrictions.

Unlike memories in general-purpose computing systems, which can store arbitrary values, the memories in NN accelerators used to store model parameters, such as trained weights, contain constrained values. This is because offline-trained weights are mapped to the weight memories, which remain fixed during inference. This provides the opportunity to embed the ECC parity bits alongside the data bits without extending the memory size. In this chapter, we propose a zero-overhead NN-ECC strategy, a generalized approach to eliminate the ECC parity storage overhead of different linear block ECC schemes for memories storing NN model parameters. The proposed zero-overhead NN-ECC strategy embeds ECC parity bits directly into the NN weights during training using a multi-task learning algorithm, without increasing the size of the original weight matrices, while achieving baseline accuracy. As a result, both the data and the parity bits are integral parts of the weight matrices and actively contribute to learning and inference tasks. In our approach, ECC encoding is done off-chip and during training. Thus, no encoding operation is required during inference, and the decoding process is the same as ECC for conventional memories. This work specifically addresses the correction of weights stored in the binary weight memories.

The overall contribution of our approach is summarized as follows:

- Incorporating ECC parity bits into the weight matrix without increasing its size so that the size of memories storing NN model parameters remains the same. That means the size of the encoded weight matrix would be equal to the size of the original data weight matrix.
- A multi-task learning algorithm that eliminates the need for specific weight distribution and higher quantization levels for embedding ECC parity into the weight matrix. A multi-task learning algorithm handles both tasks, such as embedding the parity bits into the weight matrix, as well as performing the functional task.
- Generalize the ECC parity bit embedding strategy so that it can be applied to a variety of ECC code dimensions and ECCs with different error correction capabilities. Provision for both the single and multi-bit error correction with zero memory overhead for ECC parity bits
- Elimination of extra memory overhead required to identify the parity location during decoding, as we fixed the parity placement in the weight matrix during the encoding itself.
- Capability to perform error detection and correction not only before the inference but also during the inference. As in our approach, we do not need to eliminate the parity, as it also participates in inference, because it is part of the weight matrix.

7.2. Zero overhead NN-ECC

In this work, we directly embed the ECC parity bits into the NN weight matrix, thereby eliminating the need for dedicated storage associated with ECC parity bits. Unlike ECC for memory applications in general-purpose computing systems, encoding of weight bits is *not* done during run-time using dedicated encoding hardware, but in the case of NN applications, the encoding is performed offline in software during training. The decoding follows traditional ECC logic, but with a smaller code size ($(n^p = k, k^p)$) than traditional ((n, k)). Consider a weight matrix $W_{d_1 \times d_2}$ where d_2 represents the size of the data word or the number of data columns in the memory array that stores the weight parameters. A key constraint is that the size of the weight matrix, i.e., the size of memory, remains the same after ECC encoding. The proposed encoding can also be used in the case of mapping weight to the CiM architecture with the binary memory elements. Because in such a scenario, the encoding is the same as any conventional memory encoding. The same offline encoding applies to the CiM architecture; however, the decoding strategy for the CiM architecture differs. In the case of CiM, Hamming distance is not applicable for decoding. Instead, algebraic-distance-based decoding is used for CiM architecture [233], [234], [240].

7.2.1. Encoding of weight matrix using ECC

In a typical ECC design for fault-tolerant NN applications, the weight bits of the NN serve as a data word, and the computation of parity information involves performing modulo-2 multiplication between the weight bits and the generator matrix (G) of the EECC scheme. The output of the weighted sum operation of NN is taken column-wise, so the redundant-parity bits can be added as extra columns in a weight matrix. The number of columns in the weight matrix decides the size of the data word, and accordingly, the coding dimension can be chosen for error detection and correction. Let us have a weight matrix W of a linear layer l with shape $d_1 \times d_2$. The encoded weight matrix, $[W_c]_{d_1 \times n} = [W|W_p]$ with the code (n, k), is defined by Equation 7.1. Where, $[W_p]_{d_1 \times (n-k)}$ contains the parity information, $k = d_2$ is the length of the data word, $[W_c]_{d_1 \times n}$ is the encoded weight matrix with $n >> d_2$.

$$W_c = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1d_2} & w_{p_{11}} & \cdots & w_{p_{1,n-k}} \\ w_{21} & w_{22} & \cdots & w_{2d_2} & w_{p_{21}} & \cdots & w_{p_{2,n-k}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ w_{d_1 1} & w_{d_1 2} & \cdots & w_{d_1 d_2} & w_{p_{d_1 1}} & \cdots & w_{p_{d_1,n-k}} \end{bmatrix} \tag{7.1}$$

We designed a new ECC with dimensions (n^p, k^p) to meet the given requirement of maintaining the memory size even after ECC encoding. The generator matrix (G) of the newly obtained ECC (n^p, k^p) is used to encode the weight matrix $[W]_{d_1 \times d_2}$ to give an encoded weight matrix $[W_c^{'}]_{d_1 \times n^p}$ whose shape is equal to the original weight matrix W. The newly encoded weight matrix, $[W_c^{'}]_{d_1 \times n^p} = [W_k^{'}|W_p^{'}]_{d_1 \times n^p}$, contains both data and parity information, where $n^p = d_2$. To ensure that the ECC covers the entire weight matrix, the shape of $W_c^{'}$ is kept equal to the original weight matrix W. The computation of $W_c^{'}$ is done by first selecting a k^p number of columns from W, defined as $W_k^{'}$. Then the parity matrix $W_p^{'}$ is computed by modulo-2 matrix multiplication of $W_k^{'}$ and the generator matrix (G) of the ECC code (n^p, k^p). When dealing with a weight matrix with many columns, it is possible to enhance fault tolerance by segmenting the large weight matrix column-wise into distinct sub-matrices and applying ECC to each of these sub-weight matrices.

$$W_c^{'} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1k^p} & w_{p_{11}} & \dots & w_{p_{1,n^p-k^p}} \\ w_{21} & w_{22} & \dots & w_{2k^p} & w_{p_{21}} & \dots & w_{p_{2,n^p-k^p}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ w_{d_1 1} & w_{d_1 2} & \dots & w_{d_1 k^p} & w_{p_{d_1 1}} & \dots & w_{p_{d_1,n^p-k^p}} \end{bmatrix} \tag{7.2}$$

We apply ECC per 64 weight bits, i.e., equivalent to 64 columns. However, our method also applies to other data sizes, such as 128 bits, 256 bits, and so on. We employ two widely used linear block ECC schemes: Hamming and BCH codes. Figure 7.1 illustrates the proposed NN-ECC encoding process using the Hamming code to protect 64-bit weights. In the case of conventional ECC encoding, (n=72, k=64) code is used to protect 64-bit weight requiring additional 8 parity bits. However, our proposed approach uses a more compact code size of (n=64, k=57). This ensures that the encoded weight bit size remains unchanged from the original, eliminating the memory overhead for ECC parity bits.

As an example, consider a weight matrix having dimension $d_1 \times d_2$, with $d_2 = 64$. A($n^p = 63$, $k^p = 57$) Hamming code can be used to get $W_c^{'}$ with shape $d_1 \times 63$, for single error correction. However, in the case of Hamming code, the length of the codeword can be increased by one even or odd parity bit ($n^p = 63 + 1, k^p = 57$) to enable double error detection, allowing full coverage of the weight matrix, i.e., possible to get the encoded weight matrix size equal to $d_1 \times 64$. Similarly, this can be done for a multi-bit error-correcting BCH code. With this approach, $W_c^{'}$ has the same shape as W.

In rare cases, if $n^p < d_2$, the very small left-out part (W_f with column size of $d_f = d_2 - n^p$) that does not participate in the error correction. The left-out subset of the weight matrix can be either pruned away post-training or train the NN by reducing the number of neurons such that d_2'=d_2-d_f. However, W_f can participate in NN inference depending on the embedding procedure employed. Here, $d_f = d_2 - n^p$ denotes the column size of the W_f. The code dimension, generator, and parity check matrix for encoding and decoding can generally be derived using the method specified in Algorithm 5, where d_2 is power of 2.

Algorithm 5 ECC code dimension and Generator & Parity check matrix

```
Input: Number of weight bits protected ($d_2$)
Input: Number of error correction capability: $t$
Output: ($n_p, k_p$) code and parity check matrix: $H$, generator matrix: $G$
Define temporary code ($n, k$)
$n = d_2 - 1$
$k \leftarrow n - ( \log_2(d_2) \times t)$
$n_p \leftarrow n + 1, \quad k_p \leftarrow k$
($n_p, k_p$) is extended version of ($n, k$), where $n_p = n + 1 = d_2$, and $k_p = k$
Construct generator matrix for ($n, k$) code: $[G]_{k \times n}$
Construct parity check matrix for ($n, k$) code: $[H]_{(n-k) \times n}$
Extend $[H]_{(n-k) \times n}$ to $[H]_{(n_p - k_p) \times n_p}$ by adding a row of 1s and a column of 0s (with zeros in the
    original rows and a 1 in the newly added row) for the new parity bit
Derive $[G]_{k_p \times n_p}$ from the extended $[H]_{(n_p - k_p) \times n_p}$
Encoding can also be done with $[G]$ of ($n, k$) code and append extra parity bit to satisfy ($n_p, k_p$) code
```

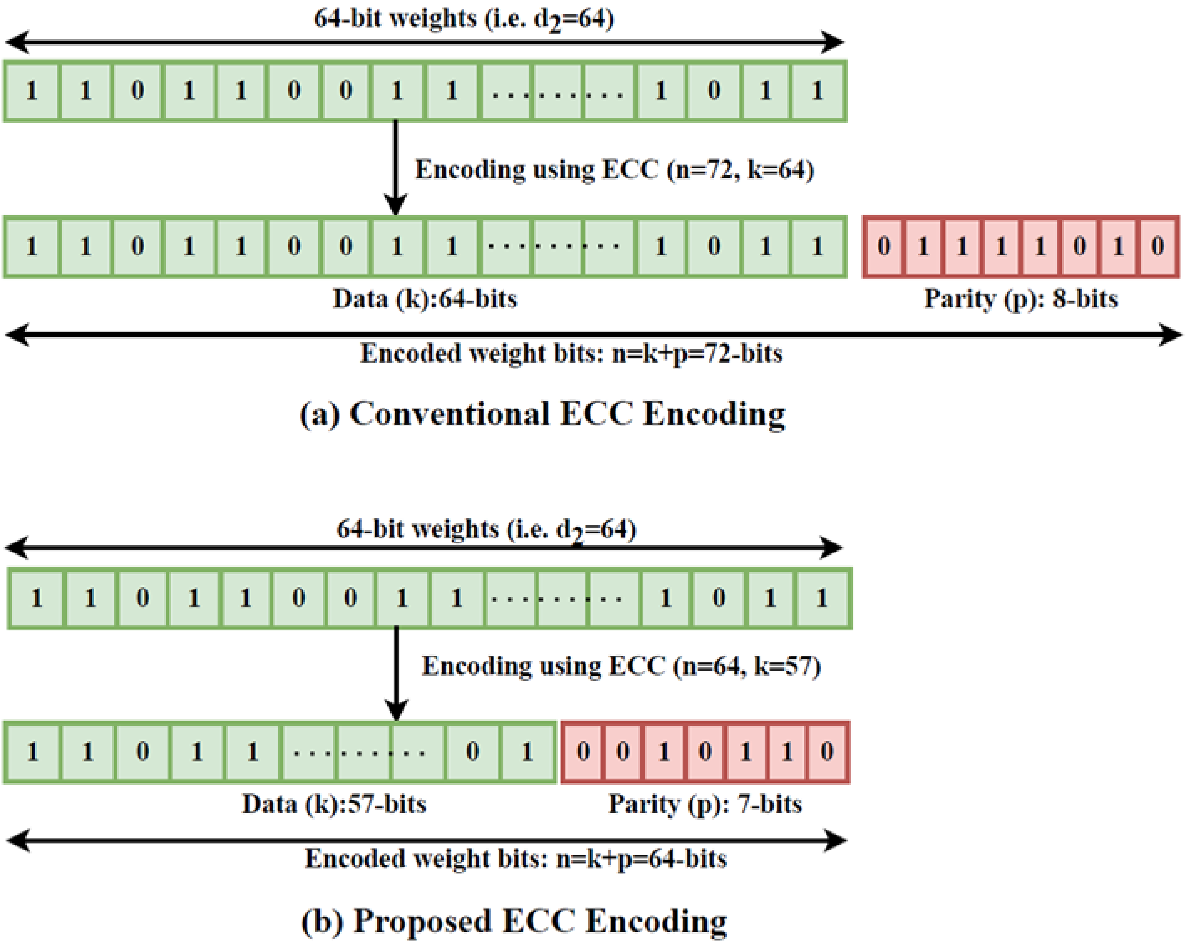

64-bit weights (i.e. d2=64)
1 1 0 1 1 0 0 1 1 .... 1 0 1 1
Encoding using ECC (n=72, k=64)
1 1 0 1 1 0 0 1 1 .... 1 0 1 1
0 1 1 1 1 0 1 0
Data (k):64-bits
Parity (p): 8-bits
Encoded weight bits: n=k+p=72-bits
(a) Conventional ECC Encoding
64-bit weights (i.e. d2=64)
1 1 0 1 1 0 0 1 1 .... 1 0 1 1
Encoding using ECC (n=64, k=57)
1 1 0 1 1 .... 0 1
0 0 1 0 1 1 0
Data (k):57-bits
Parity (p): 7-bits
Encoded weight bits: n=k+p=64-bits
(b) Proposed ECC Encoding

Figure 7.1.: Example of proposed NN-ECC encoding process showing that the size of original weights and encoded weights are equal

7.2.2. Decoding Process for Error Correction

The decoding process of the proposed strategy follows an error detection and correction approach similar to conventional ECC methods used in memory applications, and is consistent with existing techniques for reducing ECC parity overhead. However, the proposed zero overhead NN-ECC uses a smaller code size, such as ($n^p = 64, k^p = 57$) rather than ($n = 72, k = 64$) SEC-DED Hamming code, as illustrated in Fig. 7.1. This reduced code size might result in slightly reduced decoding complexity. The decoding process involves modulo-2 matrix multiplication between the encoded weight bits and the ECC parity check matrix H to calculate the syndrome vector. Subsequently, error localization and correction can be carried out based on the value of the syndrome vectors. The existing approach performs error correction before inference, as it needs to mask the parity bits before inference starts to restore the original weight bits that were occupied by the parity bits. In contrast to the existing method, our approach eliminates the need for parity masking before inference begins because both the data bits and the parity bits are inherent components of the weight matrix and actively contribute to the inference process. Due to this, the proposed method offers the advantage of concurrently performing error detection and correction during the inference process.

In the case of CiM architecture, the decoding strategy differs from conventional ECC decoding. Because multiple rows are activated simultaneously, their currents accumulate on the bit lines. This violates the Hamming distance-based decoding strategy of conventional ECC schemes. In the case of CiM, Hamming distance is not applicable for decoding. Instead, algebraic-distance-based decoding is used for CiM architecture [233], [234], [240]. The details about such strategies are discussed in Chapter 2.

7.2.3. In-training ECC embedding using multi-task learning

NN is a computing graph in which nodes represent neurons, and connections are denoted by weights. NN can be defined as a function $\mathcal{F}_\theta$ with learnable parameters θ that map a given input vector X to an output vector $\hat{Y}$. During NN training, given a dataset for a task $\mathcal{D}_{train} \subset (X_{train}, Y)$, the parameters θ are learned with a learning algorithm by minimizing the loss $\mathcal{L}$. NN learns its parameter θ for a functional task defined by the dataset D by minimizing a loss function $\mathcal{L}$. However, for the proposed zero overhead NN-ECC, the NN should learn not only functional tasks but also fault-tolerant/error correction tasks, such as incorporating the ECC parity information directly into the weight matrix without increasing its size. The fault-tolerant task can be defined as optimizing a subset of the θ that fulfills the requirement of ECC encoding without needing any additional data. Whereas the functional task is the intended task of NN, such as classification or object detection. The overall parameter for both tasks is defined as $\hat{\theta}$. In the proposed multi-task learning, we ensure that the NN learns the function task and the fault-tolerant task without performance degradation. To achieve this, we update $\hat{\theta}$, which involves both the ECC parity bits and the data parts of the weight matrix during NN training. As a result, the proposed strategy jointly learns the shared parameters $\hat{\theta}$ using the same data.

To obtain the encoded weight matrix, the generator matrix (G) of ECC must be multiplied (using the modulo-2 operation) by each row of the weight matrix. As each row of the weight matrix can be considered a data word, each row is modulo-2 multiplied by the generator matrix, resulting in the codeword. In this way, performing modulo-2 multiplication with all the rows leads to a codeword matrix that has both data and parity parts. However, the weight matrix of the convolutional layers in NN is 4D in shape $[C_{out} \times C_{out} \times F_H \times F_W]$. Here, C_{in}, C_{out}, F_H, and F_W refer to the input channels, the output channels, the height, and the width of the kernels. To satisfy the matrix-vector multiplication rule, the four-dimensional weight matrix is reshaped into a two-dimensional weight matrix of shape $[C_{out} \times (C_{in} \cdot F_H \cdot F_W)]$. Here, $\cdot$ denotes arithmetic multiplication. However, the computational cost of such an MVM can be expensive as the number of rows is significantly high. To solve this problem of MVM operation, we have reshaped the weight matrix again by inserting an additional dimension of size one into the row position to give a new shape $[C_{out} \times 1 \times (C_{in} \cdot F_H \cdot F_W)]$. This replaces MVM with matrix-matrix multiplication between the generator matrix of ECC and the weight matrix of NN.

To define the loss function during the training of the NN procedure, we have used common loss functions, such as cross-entropy for classification, to learn both functional and ECC parity embedding tasks. At the initialization of the weight, parity is embedded in the weight matrix, and it can be described by Equation 7.3. Here, the function $f_{ECC}(.)$ describes the encoding of the weight matrix. The weighted sum is performed on the encoded weight matrix during the forward pass. Since the loss is calculated with respect to the encoded weight matrix, NN optimizes both tasks.

$$\hat{W} = f_{\mathrm{ECC}}(\mathbf{W}, G) \tag{7.3}$$

During backpropagation, the NN parameters are updated with the gradient descent algorithm for the functional task. Subsequently, the NN parameter is updated deterministically using $f_{ECC}(.)$. Since the ECC parity embedding preserves the data part of the weight matrix, and the gradient is not used in the parity weight update, its gradient should not affect backpropagation for the functional task. As a result, the straight-through estimator (STE) [278] is used during forward and backward propagation. This is because parity is a function of the data part, so whenever the weight is updated, parity should update accordingly. One cannot update weight and parity together with the gradient update rule, as it will not preserve the property of the ECC parity bits. That's why STE is used during both the forward and backward passes in the case of parity update.

With a straight-through estimator, during weight update, ECC embedded weights are computed by applying the $f_{ECC}(.)$ function to their corresponding functional weight proxies. During backpropagation, a gradient is an estimation of functional weight proxies, i.e., the STE treats the gradient of the function $f_{ECC}(.)$ with respect to functional weight proxies as an identity and can be defined by Equation (7.4). Therefore, the straight-through estimator allows us to estimate the gradient of the loss with respect to the function $f_{ECC}(.)$ and update the proxies as per the definition by Equation (7.5).

$$\frac{\delta W}{\delta W'} = 1 \tag{7.4}$$

$$\frac{\delta \mathcal{L}}{\delta W'} = \frac{\delta \mathcal{L}}{\delta W} \tag{7.5}$$

The overall algorithm for training is described as follows:

Algorithm 6 Proposed Multi-task Learning Algorithm

```
Require: initial $\theta$, loss function $\mathcal{L}$, number of layers $L$, NN model $\mathcal{F}$, training dataset $\mathcal{D}_{\text{train}} \subset (X, Y)$
Require: ECC code dimension $(n^p, k^p)$, generator $(G)$ and parity-check $(H)$ matrices
Ensure: trained parameters $\hat{\theta}$ for $(n^p, k^p)$ ECC code
Initialize NN parameters
Initialize parity subset of NN parameters using $f_{\text{ECC}}(\cdot)$
for epoch in epochs do
    Sample random minibatch $\mathcal{D}_d$
    Forward pass $\mathcal{D}_d$ on NN with ECC-embedded parameter $\hat{\theta}$
    Calculate the loss $\mathcal{L}$ on the embedded ECC parameter $\hat{\theta}$
    Backpropagate and update parameters $\hat{\theta}$
    Flatten weight matrix of convolutional layers into shape $[\,C_{out} \times (C_{in}.F_H.F_W)\,]$
    Reshape flattened weight matrix into shape $[\,C_{out} \times 1 \times (C_{in}.F_H.F_W)\,]$
    Update reshaped weight matrix $\mathbf{W}'_l, \forall\, l \in [0, \ldots, L]$ using $f_{\text{ECC}}(\cdot)$
    Reshape updated weight matrix to original shape $[\,C_{out} \times C_{in} \times F_H \times F_W\,]$
end for
```

7.3. Results and discussions

7.3.1. Simulation Setup

To enable fault-tolerance in NN using error correction with ECCs, we explore the effectiveness of two widely used linear block ECCs: the Hamming code and the BCH code, both of which have t error correction capability and $t + 1$ error detection capability. We consider the BCH code with various error correction capabilities, including $t = 2$, $t = 3$, and $t = 4$, to highlight the versatility and ability of the proposed zero overhead NN-ECC to accommodate a higher number of parity bits, thereby enabling multi-bit error correction capability. The MATLAB tool is used to obtain the ECC code dimension and generator matrix. The proposed strategy is evaluated using state-of-the-art NN models with 8-bit quantized weights on the CIFAR-10 and CIFAR-100 datasets. These models, along with their number of parameters, are detailed in Table 7.1.

Table 7.1.: NN Models and Their Respective Parameter Counts

NN Model	ResNet-18	EfficientNet-B0	MobileNet-V2	ResNet-34
#Parameters	89.3×10^6	43.7×10^6	17.1×10^6	17.9×10^6

We train each model once without embedding any ECC during training and four times to embed different ECC schemes with different error correction capabilities ($t = 1$ to 4) using the proposed strategy. We have used ADAM optimization with the default setting in PyTorch and a cross-entropy loss function. The model that achieved the best accuracy in the validation dataset is used for the simulation. In the context of fault injection, bit-flip faults are introduced into the weights of NN models by randomly sampling from a uniform distribution before inference. Following the fault injection phase, error correction is executed based on ECC schemes. The fault tolerance is evaluated by observing the improvement in the NN inference accuracy with the proposed NN-ECC solution. We have performed 100 Monte Carlo (MC) simulations and reported the mean accuracy.

7.3.2. ECC dimension

The possible code dimensions (n^p, k^p) for the Hamming code ($t = 1$) and the BCH code ($t = 2, 3, 4$) with different column shapes (d_2) of the weight matrix are presented in Table 7.2. In this work, we consider protecting eight 8-bit weights, that is, a 64-bit word size for ECC. In this work, we have used the $(n^p = 64, k^p = 57)$ code for the Hamming code and $(n^p = 64, k^p = 51)$, $(n^p = 64, k^p = 45)$, and $(n^p = 64, k^p = 39)$ for the BCH code with $t = 2, 3, 4$ error correction capability, respectively, to ensure error correction per 64-bit weights. In this way, the proposed approach ensures that the dimension of the encoded weight matrix perfectly matches that of the original weight matrix, resulting in the complete elimination of additional memory cells required to store the ECC parity bits, thereby representing zero memory overhead. Furthermore, unlike our method, it avoids additional memory overhead for identifying parity bit locations. This is because we employ the concept of systematic linear block ECC, where the parity bits are appended after or before the data bits. In this way, the parity bits' position is always hard-coded, eliminating the need for extra bits to identify their location within the weight bits.

Table 7.2.: The possible Code Dimension (n^p, k^p) for Different Column size (d_2) of the Weight Matrix

ECC	Hamming (t=1)	BCH (t=2)	BCH (t=3)	BCH (t=4)
d_2	(n^p, k^p)	(n^p, k^p)	(n^p, k^p)	(n^p, k^p)
64	(64, 57)	(64, 51)	(64, 45)	(64,39)
128	(128, 120)	(128, 113)	(128, 106)	(128, 99)
256	(256, 247)	(256, 239)	(256, 231)	(256, 223)

7.3.2.1. Impact on baseline accuracy due to ECC Encoding

It is essential to assess how the proposed zero-overhead NN-ECC encoding techniques, based on multi-task learning, impact the NN baseline accuracy. The ECC encoding should not disturb the baseline accuracy. Table 7.3 and 7.4 show the impact of the proposed zero-overhead NN-ECC techniques based on multi-task learning on NN baseline accuracy. Our method achieves accuracy comparable to the baseline across diverse NN topologies and various ECC configurations in the CIFAR-10 and CIFAR-100 datasets. Furthermore, the encoding using the multi-bit ECC, such as the BCH code with $t = 4$, also shows a negligible drop in accuracy, less than 1% for the CIFAR-10 dataset and up to $\sim 2\%$ for the CIFAR-100 dataset. This is achieved despite the fact that BCH code requires a large number of parity bits, posing challenges in simultaneously meeting functional tasks and embedding ECC parity bits into the weight matrix without increasing the size of the weight matrix.

Table 7.3.: Impact on Baseline accuracy due to ECC Encoding (CIFAR-10).

Topology	NN Inference Accuracy (%)				
	Baseline	Proposed NN-ECC			
		Hamming (t=1)	BCH (t=2)	BCH (t=3)	BCH (t=4)
ResNet-18	92.05%	92.03%	92.27%	92.44%	92.50%
EfficientNet-B0	90.19%	89.95%	90.30%	89.44%	89.20%

Table 7.4.: Impact on baseline accuracy due to ECC encoding (CIFAR-100).

Topology	NN Inference Accuracy (Top-1 %, Top-5%)				
	Baseline	Proposed NN-ECC			
		Hamming (t=1)	BCH (t=2)	BCH (t=3)	BCH (t=4)
ResNet-34	(72.5, 91.5)	(72.3, 91.4)	(72.3, 91.5)	(71.6, 90.9)	(71.6, 90.4)
MobileNet-V2	(67.7, 89.3)	(68.0, 88.3)	(66.2, 89.1)	(67.3, 88.8)	(65.6, 88.7)

7.3.2.2. Impact on fault-tolerance

Fig. 7.2 and Fig. 7.3 show the result of random fault simulation for different NN models with CIFAR-10 and CIFAR-100 datasets. The baseline accuracy drops very sharply, even at a very low fault rate. However, the accuracy remains stable up to a certain fault rate when errors are corrected using Hamming and BCH codes. As the error correction capability increases, the sustainability of NN against the number of injected faults also increases significantly. The increment in the fault tolerance limit almost doubles with a 1-bit increase in error correction capability. For example, the fault tolerance limit of the BCH code with t=2, t=3, and t=4 is almost 2×, 4×, and 8× the Hamming code (t = 1), respectively, as shown in Fig. 7.2 and Fig. 7.3. In EfficientNet-B0, the improvement is minor, as there was not much reduction in baseline accuracy, as shown in Fig. 7.3.

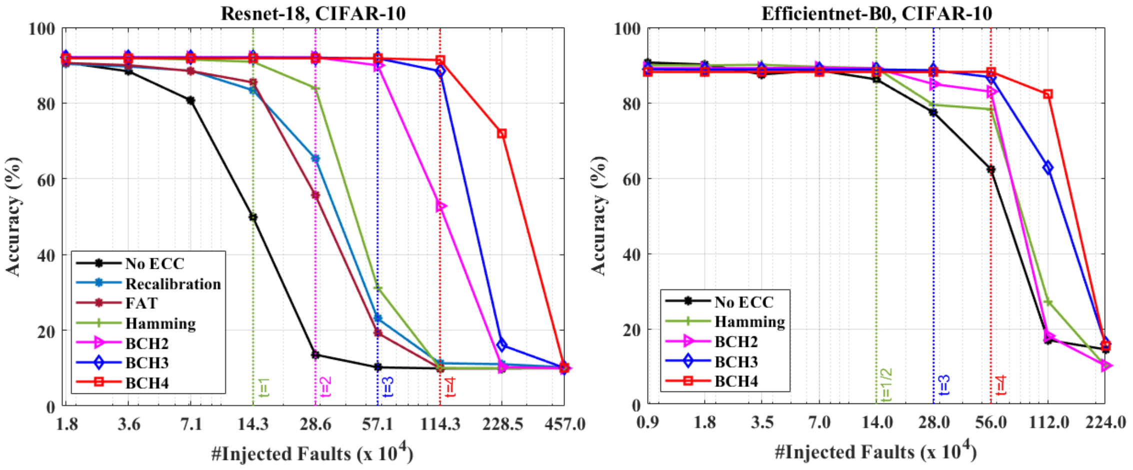


Figure 7.2.: NN inference accuracy on CIFAR-10 Datasets. The vertical dotted line indicates the fault-tolerance limits with t number of error corrections.

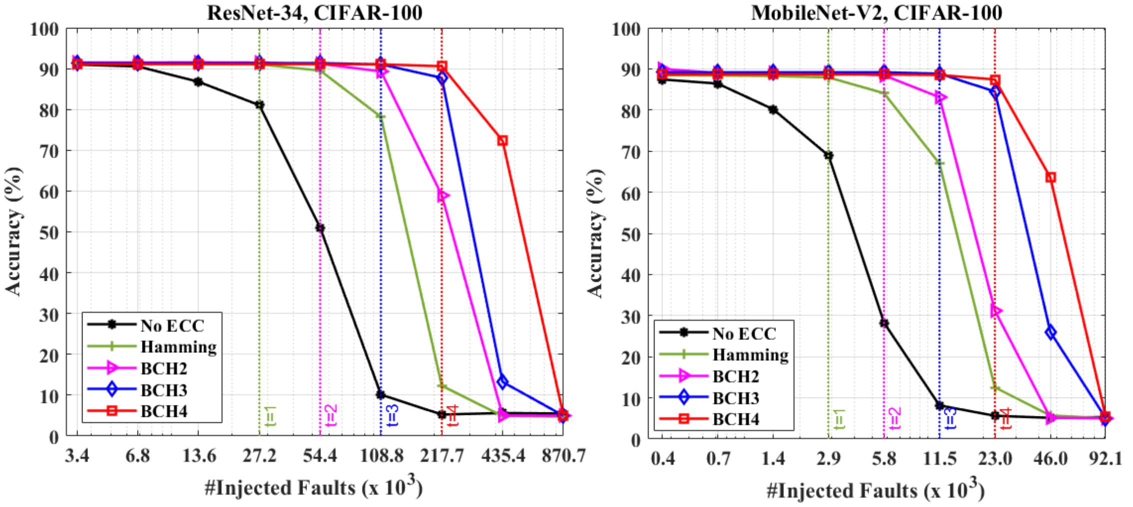


Figure 7.3.: NN inference accuracy (Top-5) on CIFAR-100 Datasets. The vertical dotted line indicates the fault-tolerance limits with t number of error corrections. The fault-tolerance trend was similar in the case of Top-1 accuracy.

Furthermore, we also demonstrate a case of the superiority of the proposed zero-overhead NN-ECC against fault-aware training (FAT) and recalibration-based fault-tolerant strategy [218], as shown in Figure 7.2 (ResNet-18 on CIFAR-10). We consider a case of FAT and recalibration for a specific fault rate (7.1×10^4 number of faults), and we observe a noticeable reduction in starting accuracy. Subsequently, the accuracy experiences a significant decline after reaching the designated fault rate, as shown in Figure 7.2. These strategies recover the accuracy *approximately* up to the designated fault rate, and they are computationally intensive, which becomes even more pronounced with higher fault rates. Furthermore, they may lead to a greater loss in starting accuracy at higher fault rates.

7.3.3. Comparative analysis

The proposed zero-overhead NN-ECC offers notable benefits over existing state-of-the-art ECC parity overhead reduction techniques [221], [222], [227], [228], as highlighted in Table 7.5. The works in [221], [222] are based on weight nulling, which detects errors based on even or odd parity. The LSB bit of the q-bit quantized weight is used as the parity. An error is detected when an odd number of bit errors occur for a weight, and this erroneous weight is replaced with a zero value. However, this corrective action results in information loss since the erroneous weights are replaced with zeros, which may reduce the NN performance. This approach focuses primarily on error detection, lacking explicit error correction. In contrast to these techniques, the proposed strategy has the capability of error detection and correction, including multi-bit error correction, without incurring any ECC parity bit memory overhead. Similarly, the work in [227] relies on redundant bits for an 8-bit quantized weight. A modified training is proposed to ensure weight distribution in a specific range. This allows a second higher-order bit (b_6) of each weight to store the parity bits of the SEC-DED ECC. However, this method lacks multi-bit error correction capability and is not suitable for other types of ECCs. Moreover, this approach demands a large number of modified training iterations to achieve the specific weight distribution, potentially resulting in significant computational overhead. These challenges become even more pronounced when addressing scenarios involving multi-bit error correction.

Our approach does not require a specific symmetric weight distribution and a higher quantization level to store the parity. This is because we do not need weights to have some redundancy in their MSB or LSB bits for storing parity information. Instead, we employ multi-task learning to embed the parity into the weight matrix without expanding its dimensions. Also, the proposed ECC encoding, based on multi-task learning, does not degrade the baseline inference accuracy ($\sim 2\%$), even for BCH codes with $t = 4$ error correction capability, which requires a very high number of parity bits. On the contrary, the work in [228] can incur a drop in accuracy of up to $\sim 4\%$ due to the ECC encoding even for $t = 2$ error correction, which might worsen with asymmetry in weight distributions. Secondly, the proposed strategy possesses the capability to perform not only single-bit but also multi-bit error correction, even more than two, with zero memory overhead, as illustrated in Table 7.6, which provides a comparative analysis of memory overhead for various ECC schemes. This level of flexibility and error correction capability was not observed in [228], where their method was limited to $t = 2$ only and was unable to maintain zero memory overhead for higher error correction capabilities, as shown in Table 7.6. This is primarily due to their inability to accommodate more parity bits. For example, the work in [228] can only accommodate 16-bit parity per 64-bit weight and can correct up to $t = 2$ errors per 64-bit weight with $\sim 0.1\%$ storage overhead (which is related to identifying the location of the parity bits during decoding). However, when considering a $t = 3$, they may require an additional $\sim 9.4\%$ of memory overhead, which would increase further for $t = 4$.

The detailed analysis of the memory overhead in [228] is as follows: 48 out of the 64 bits can be used as data bits, leaving the remaining 16 bits to store parity bits. A $(70, 48)$ BCH code can be used to perform up to $t = 3$ error correction, which requires 22 parity bits. So, an additional 6 bits of storage are required to accommodate the remaining parity bits, resulting in a memory overhead of $\sim 9.4\%$ per 64-bit weight. The same approach is also applicable to higher correction capabilities, such as $t = 4$, incurring a memory overhead of $\sim 20.3\%$.

Additionally, the proposed strategy eliminates the requirement of masking ECC parity bits before inference begins to restore the original weight bits. This is achievable because, in our method, both data and parity bits actively participate in the inference process. Thus, we have access to parity information during inference, which can offer the advantage of performing error detection and correction simultaneously with the inference process. In contrast, existing approaches [221], [222], [227], [228] require masking the ECC parity bits before inference begins to recover the original weight values. This further limits their method to perform error detection and correction before inference only. The proposed strategy also eliminates the need for additional memory overhead to identify parity locations during the decoding process, a requirement observed in [228], which may be approximately 0.1% to 0.2% of memory overhead. We adhere to the concept of systematic linear block codes, where all the parity bits are appended together before or after the data bits. In this way, we already know the location of parity bits during the decoding process. This design choice ensures that the proposed method incurs zero memory overhead while maintaining multi-bit correction capability. Lastly, with the higher correction ability of the proposed strategy, NN resilience against a large number of randomly injected faults is significantly improved, as illustrated in Figures 7.2 and 7.3. This demonstrates its adaptability to a variety of ECC schemes without loss of performance.

Table 7.5.: Factors that distinguish the proposed method from existing works

Factors	[221], [222]	[227]	[228]	Proposed
Require specific weight distribution	Yes	Yes	Yes	No
Relies on higher quantization precision	Yes	Yes	Yes	No
Limited error detection and correction	Yes	Yes	Yes	No
Require parity masking before inference	Yes	Yes	Yes	No
Require storage to identify parity	No	No	Yes	No
Accuracy degradation due to encoding	Yes	No	Yes	No

Note. “Yes”: constraint is required (a bottleneck); “No”: constraint is not required (an advantage).

Table 7.6.: Comparative analysis of memory overhead (MO) for ECC parity bits

Method	[222]		[227]		[228]		Proposed	
ECC	(n^p, k^p)	MO	(n^p, k^p)	MO	(n^p, k^p)	MO	(n^p, k^p)	MO
Hamming	(72,64)	10.9%	(64,57)	0%	(64,57)	0%	(64,57)	**0%**
BCH (t=2)	(79,64)	21.9%	(71,56)	10.9%	(64,51)	0.1%	(64,51)	**0%**
BCH (t=3)	(86,64)	32.8%	(78,56)	21.9%	(70,48)	9.4%	(64,45)	**0%**
BCH (t=4)	(93,64)	43.7%	(85,56)	32.8%	(77,48)	20.3%	(64,39)	**0%**

7.4. Summary

In this chapter, we proposed zero-overhead NN-ECC, an efficient and generalized method that embeds the ECC parity bits in the NN weight matrix during NN training, thereby eliminating the parity bit storage overhead for different ECC schemes in the NN accelerator memories that store NN model parameters. This way, we provide multi-bit error correction guarantees, as required in safety-critical AI applications. We devised a multi-task training approach for the suggested embedding that achieves the same level of inference accuracy as the baseline. The efficacy of the proposed strategy is tested by state-of-the-art codings, such as the Hamming code (single-bit ECC) and the BCH code (multi-bit ECC), to different NN models on various benchmarks. The results showed that the proposed strategy can significantly improve fault tolerance depending on the selected ECC, without introducing storage overhead. Notably, the proposed NN-ECC is versatile and applicable to diverse NNs and linear block ECC schemes.

8. Conclusions and Outlook

8.1. Conclusions

In this thesis, we have developed various low-overhead fault-tolerant strategies to improve the reliability of emerging memories in both general-purpose computing and NN computing systems. For memories in general-purpose computing systems and standalone memory systems, we have focused on STT-MRAM, which is one of the mature memory technologies among other emerging memories. First, we have improved the reliability of STT-MRAM by integrating ECC with lightweight architectural adjustments. Second, we have enhanced fault tolerance by protecting the weight memories of NN accelerators, which include CiM architectures and digital MAC-based accelerators.

In Chapter 2, we have discussed the different failure mechanisms in emerging memories and surveyed fault-tolerant/error-correction strategies for STT-MRAM and the weight memories of NN accelerators. These findings motivate the thesis direction: combining ECC-based techniques with minimal architectural modification to deliver robust, scalable protection. In chapter 3, we have presented the strategies to perform runtime hard-error correction in STT-MRAM to enhance the stuck-at fault tolerance. We have introduced a scheme that stores block-wise offsets of hard-error locations and uses these offsets during decoding to recover the correct bit values. We have incorporated experimental measurement data from manufactured STT-MRAM chips to obtain the hard error distribution, supporting the proposed hard error correction strategy. In chapter 4, we have presented the asymmetric and adaptive error correction strategy to mitigate the asymmetric failure mechanism in STT-MRAM. We have shown that the proposed technique is memory efficient and significantly enhances reliability, as measured by a last-level cache block error rate. In chapter 5, we have presented a systematic location-aware non-uniform error correction strategy to mitigate the impact of interconnect-induced reliability issues in STT-MRAM on top of random error correction. This strategy reduces the requirements for ECC parity by selectively applying stronger error correction to specific rows within the memory array, rather than applying it uniformly across the entire array.

In chapter 6, we have presented a checksum-based column error correction in CiM architecture, which consists of NVM-based memristive crossbar for reliable MVM, which is the core operation of NN computation. Firstly, we have developed a scaled checksum-based method that offers low memory overhead for single-column error correction within the memristive crossbar. We have demonstrated the effectiveness of this strategy using checksums based on ABFTand Hamming codes. Next, we have also illustrated the application of Hamming codes for block error correction in the memristive crossbar, enabling the simultaneous protection of multiple columns. In chapter 7, we have presented the zero-overhead ECC-based fault-tolerant technique to protect the binary-weight memories of NN accelerators. We have demonstrated the strategy to embed the ECC parity into the weight matrix itself without increasing the size of the weight matrix, enabling zero overhead for ECC parity bits. The proposed strategy can also apply to the binary weight memories of CiM architecture.

In conclusion, the methodologies and solutions presented in this thesis are viable alternatives towards resilient memory and computing systems. Additional research not included in this thesis [72], [73] also examines the potential of non-uniform and adaptive error correction techniques, as well as reliability improvement methods. For instance, the work in [72] investigates a non-uniform error correction strategy that enhances the fault resilience of HyperDimensional Computing (HDC), while preserving the hardware efficiency. HDC is a brain-inspired computational paradigm designed to emulate human cognitive processes. It is highly suitable for resource-constrained edge environments where efficiency is crucial. The approach utilizes statistical fault sensitivity analysis to categorize memory regions storing HDC parameters into different levels of fault sensitivity. This method identifies highly sensitive areas and distinguishes them from less critical regions that can tolerate faults without significantly impacting overall system performance. An efficient absolute summation weight-based method for fault sensitivity analysis is employed, which is adaptable to various configurations and datasets of HDC, thereby eliminating the need for time-consuming fault injections. By applying targeted ECC-based error correction strategies to only the most sensitive regions, the strategy optimizes fault correction while maintaining resource efficiency, making it a practical solution for edge environments. The work in [73] analyzes the radiation impacts in SRAM-based register files. The reliability of SRAM-based register files is critical for ensuring robust performance in high-performance computing systems. The work employs an architecture-level, adaptive location-dependent WL pulse-width modulation to mitigate radiation impact in FDSOI technology.

8.2. Outlook

This thesis advances fault-tolerant techniques to improve the reliability of emerging memories and computing systems. The methodologies are technology-agnostic and can be adapted to a range of memory technologies and computing paradigms with modest, technology-aware modifications. Looking ahead, the development of hybrid hardware–software co-design frameworks appears to be a promising area for future research, as such co-design can further balance reliability, overhead, and performance. The reliability solutions for emerging NVMs have primarily focused on CPU caches and main memory. However, with the substantial rise in GPU usage, NVMs will likely soon be utilized for caches and main memory in GPUs, which opens up opportunities to explore new reliability solutions tailored explicitly for GPU architectures. In the future, hybrid memory designs that combine the strengths of conventional CMOS memory and emerging NVMs are expected to emerge, opening up opportunities to explore new fault models and reliability solutions. Moreover, developing effective fault tolerance across diverse NN architectures is complex, highlighting the need for further exploration into architecture- and application-specific fault tolerance. In addition, the passive memristive crossbar, while offering an area efficiency compared to active crossbars, presents different testing and reliability challenges that can be explored in the future. One of the works [74] not included in this thesis covered the testing aspects of a passive memristive crossbar. Overall, the strategies developed in this thesis provide a foundation for future research on improving the reliability of next-generation memories and computing paradigms.

Part I.

Appendix

Parity check matrix for SEC Hamming code:

Method 1 (H_1): In general, for each integer $p \geq 2$, there is a code-word with length $n = 2^p - 1$ and data length $k \leq 2^p - p - 1$. For a given k bit data, if the default Hamming code is not available (especially when k is a power of 2s), then derive from a slightly larger (n, k) code dimension, which is given as follows:

Algorithm 7 Hamming Code Generator Matrix and Parity Check Matrix Construction

```
Require: Data bit width $k$
Ensure: Generator matrix $G$ and parity check matrix $H$
Get the $p$ value which satisfy: $k \leq 2^p - p - 1$
Generate preliminary Hamming matrices $(h, g)$ for $p$
$n_{delete} \leftarrow \text{rows}(g) - k$
Extract reduced generator matrix: $G \leftarrow g(1 : \text{rows}(g) - n_{delete},\ 1 : \text{cols}(g) - n_{delete})$
Let $x_1 \leftarrow \text{rows}(G), \quad x_2 \leftarrow \text{cols}(G)$
$x_{diff} \leftarrow x_2 - x_1$
Augment generator matrix: $G_1 \leftarrow [I_k \mid G(:, 1 : x_{diff})]$
Derive parity check matrix: $H_1 \leftarrow \texttt{gen2par}(G_1)$
```

Method 2 (H_2):

It is possible to generate the codeword for any given data bits from scratch by applying the principles of the Hamming code, which is shown below [173]. There are open-source simulators available that can generate the parity check matrix for the provided input data bits [279], [280].

1. All bit positions that are powers of two are reserved for parity bits: 1, 2, 4, 8, 16, 32, 64, . . .
2. All other bit positions are used for the data: 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, . . .
3. Each parity bit checks a specific sequence of data and parity positions.
    - Position 1: check 1 bit, skip 1 bit, . . . Example: (1, 3, 5, 7, 9, 11, 13, 15, . . .)
    - Position 2: check 2 bits, skip 2 bits, . . . Example: (2, 3, 6, 7, 10, 11, 14, 15, . . .)
    - Position 4: check 4 bits, skip 4 bits, . . . Example: (4, 5, 6, 7, 12, 13, 14, 15, 20, 21, . . .)
    - Position 8: check 8 bits, skip 8 bits, . . . Example: (8–15, 24–31, 40–47, . . .)
    - Position 16: check 16 bits, skip 16 bits, . . . Example: (16–31, 48–63, 80–95, . . .)
4. Each parity bit is set according to the number of ones in the positions it checks:
    - If the total number of ones is odd, set the parity bit to 1.
    - If the total number of ones is even, set the parity bit to 0.

(a) Data size: $k = 32$ **bits**

$$H_1 = \begin{matrix} 10000110001010011110100011100100100000 \\ 11000101001111010001110010010110010000 \\ 01100010100111101000111001001011001000 \\ 00110001010011110100011100100101000100 \\ 00011000101001111010001110010010000010 \\ 00001100010100111101000111001001000001 \end{matrix}$$

$$H_2 = \begin{matrix} 11111100000000000000000000000000100000 \\ 00000011111111111111100000000000010000 \\ 00000011111111000000011111110000001000 \\ 11100011110000111100011110001110000100 \\ 10011011001100110011011001101101000010 \\ 01010110101010101010110101011011000001 \end{matrix}$$

(b) Data size: $k = 64$ **bits**

$$H_1 = \begin{matrix} 10001001100010111010110110000011001101 010011100111101101000010101000000 \\ 01000100110001011101011011000001100110 101001110011110110100001010100000 \\ 00100010011000101110101101100000110011 010100111001111011010000100010000 \\ 10011000101110101101100000110011010100 111001111011010000101010110001000 \\ 01001100010111010110110000011001101010 011100111101101000010101010000100 \\ 00100110001011101011011000001100110101 001110011110110100001010100000010 \\ 00010011000101110101101100000110011010 100111001111011010000101010000001 \end{matrix}$$

$$H_2 = \begin{matrix} 1111111000 0000000000 0000000000 0000000000 0000000000 0000000000 0000100000 0 \\ 0000000111 1111111111 1111111111 1111111100 0000000000 0000000000 0000010000 0 \\ 0000000111 1111111111 1110000000 0000000011 1111111111 1110000000 0000001000 0 \\ 0000000111 1111100000 0001111111 1000000011 1111110000 0001111111 0000000100 0 \\ 1111000111 1000011110 0001111000 0111100011 1100001111 0001111000 1110000010 0 \\ 11001101100110011001100110011001100110 110011001100110110011011010000010 \\ 10101011010101010101010101010101010101 101010101010101101010110110000001 \end{matrix}$$

(c) Data size: $k = 128$ **bits**

$$H_1 = \begin{matrix} 100011100010010111000000110010010011011100100000101011011010110010110000 1111101101111010111010001000011011000111100111001100010110000000 \\ 010001110001001011100000011001001001101110010000010101101101011001011000 0111110110111101011101000100001101100011110011100110001001000000 \\ 101011011010110010110000111110110111101011101000100001101100011110011100 1100010110100100010100101010011101110110011110111111101000010000 \\ 110110001111001110011000101101001000101001010100111011101100111101111110 1001100110101000110000011101010101111100101000010011111100010000 \\ 111000100101110000001100100100110111001000001010110110101100101100001111 1011011110101110100010000110110001111001110011000101101000001000 \\ 011100010010111000000110010010011011100100000101011011010110010110000111 1101101111010111010001000011011000111100111001100010110100000100 \\ 001110001001011100000011001001001101110010000010101101101011001011000011 1110110111101011101000100001101100011110011100110001011000000010 \\ 000111000100101110000001100100100110111001000001010110110101100101100001 1111011011110101110100010000110110001111001110011000101100000001 \end{matrix}$$

$$H_2 = \begin{matrix} 1111111100 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000010 000000 \\ 0000000011 1111111111 1111111111 1111111111 1111111111 1111111111 1111111111 1000000000 0000000000 0000000000 0000000000 0000000000 0000000001 000000 \\ 0000000011 1111111111 1111111111 1111111111 0000000000 0000000000 0000000000 0111111111 1111111111 1111111111 1100000000 0000000000 0000000000 100000 \\ 0000000011 1111111111 1111000000 0000000000 1111111111 1111110000 0000000000 0111111111 1111111000 0000000000 0011111111 1111111000 0000000000 010000 \\ 1000000011 1111110000 0000111111 1100000000 1111111100 0000001111 1111000000 0111111110 0000000111 1111100000 0011111111 0000000111 1111000000 001000 \\ 0111100011 1100001111 0000111100 0011110000 1111000011 1100001111 0000111100 0111100001 1110000111 1000011110 0011110000 1111000111 1000111000 000100 \\ 0110011011 0011001100 1100110011 0011001100 1100110011 0011001100 1100110011 0110011001 1001100110 0110011001 1011001100 1100110110 0110110100 000010 \\ 0101010110 1010101010 1010101010 1010101010 1010101010 1010101010 1010101010 1101010101 0101010101 0101010101 0110101010 1010101101 0101101100 000001 \end{matrix}$$

(d) Data size: $k = 256$ **bits**

$H_1 =$ [illegible]

$H_2 =$ [illegible]

To obtain the extended SEC-DED Hamming code, an extra parity bit can be added to the codeword obtained by SEC. This parity bit is calculated by taking the XOR of all the bits (data and parity both).

Parity check matrix for DEC BCH code:

(a) Data size: $k = 64$ **bits**

$$H = \begin{matrix}
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{matrix}$$

(b) Data size: $k = 128$ **bits**

$$H = \begin{matrix}
0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{matrix}$$

To obtain the extended DEC-TED BCH code, an extra parity bit can be added to the codeword obtained by DEC. This parity bit is calculated by taking the XOR of all the bits (data and parity both).

Bibliography

[1] Wikipedia. “Apple silicon”. (2025), [Online]. Available: `https://en.wikipedia.org/wiki/Apple_silicon`.

[2] K. Rupp, *42 Years of Microprocessor Trend Data*, `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`, Blog post, Feb. 2018.

[3] H.-S. Wong, D. Frank, P. Solomon, C. Wann, and J. Welser, “Nanoscale cmos”, *Proceedings of the IEEE*, vol. 87, no. 4, pp. 537–570, 1999. DOI: `10.1109/5.752515`.

[4] J. Hutchby, G. Bourianoff, V. Zhirnov, and J. Brewer, “Extending the road beyond cmos”, *IEEE Circuits and Devices Magazine*, vol. 18, no. 2, pp. 28–41, 2002. DOI: `10.1109/101.994856`.

[5] S.-H. Lee, “Technology scaling challenges and opportunities of memory devices”, in *2016 IEEE International Electron Devices Meeting (IEDM)*, 2016, pp. 1.1.1–1.1.8. DOI: `10.1109/IEDM.2016.7838026`.

[6] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, “Design of ion-implanted mosfet’s with very small physical dimensions”, *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974. DOI: `10.1109/JSSC.1974.1050511`.

[7] B. Hoefflinger, “Itrs: The international technology roadmap for semiconductors”, *Chips 2020: A Guide to the Future of Nanoelectronics, The Frontiers Collection*, pp. 161–, Jan. 2012. DOI: `10.1007/978-3-642-23096-7_7`.

[8] R. Chau, B. Doyle, S. Datta, J. Kavalieros, and K. Zhang, “Integrated nanoelectronics for the future”, *Nature Materials*, vol. 6, no. 11, pp. 810–812, Nov. 2007. DOI: `10.1038/nmat2014`.

[9] L. Atzori, A. Iera, and G. Morabito, “The internet of things: A survey”, *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010, ISSN: 1389-1286. DOI: `10.1016/j.comnet.2010.05.010`. [Online]. Available: `https://doi.org/10.1016/j.comnet.2010.05.010`.

[10] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious”, *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995, ISSN: 0163-5964. DOI: `10.1145/216585.216588`. [Online]. Available: `https://doi.org/10.1145/216585.216588`.

[11] K. Olukotun and L. Hammond, “The future of microprocessors”, *ACM Queue*, vol. 3, Sep. 2005. DOI: `10.1145/1095408.1095418`.

[12] A. Zeng, K. Rose, and R. Gutmann, “Memory performance prediction for high-performance microprocessors at deep submicrometer technologies”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 9, pp. 1705–1718, 2006. DOI: `10.1109/TCAD.2005.858346`.

[13] J. R. Goodman, “Using cache memory to reduce processor-memory traffic”, in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ser. ISCA ’83, Stockholm, Sweden: Association for Computing Machinery, 1983, pp. 124–131, ISBN: 0897911016. DOI: `10.1145/800046.801647`. [Online]. Available: `https://doi.org/10.1145/800046.801647`.

[14] A. Ranjan, S. Venkataramani, Z. Pajouhi, R. Venkatesan, K. Roy, and A. Raghunathan, “Staxcache: An approximate, energy efficient stt-mram cache”, in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 356–361. DOI: `10.23919/DATE.2017.7927016`.

[15] T. Marinelli, J. I. G. Pérez, C. Tenllado, M. Komalan, M. Gupta, and F. Catthoor, “Microarchitectural exploration of stt-mram last-level cache parameters for energy-efficient devices”, *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 1, Jan. 2022, ISSN: 1539-9087. DOI: 10.1145/3490391. [Online]. Available: https://doi.org/10.1145/3490391.

[16] N. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan, “Leakage current: Moore’s law meets static power”, *Computer*, vol. 36, no. 12, pp. 68–75, 2003. DOI: 10.1109/MC.2003.1250885.

[17] *International technology roadmap for semiconductors (itrs): 2015 executive report*, https://www.semiconductors.org/wp-content/uploads/2018/06/0_2015-ITRS-2.0-Executive-Report-1.pdf, Online, Semiconductors Industry Association, 2015.

[18] Unknown, *As nodes advance, so must power analysis*, http://semiengineering.com/as-nodes-advance-so-must-power-analysis/, Online.

[19] S. Sakhare, M. Perumkunnil, T. H. Bao, S. Rao, W. Kim, D. Crotti, F. Yasin, S. Couet, J. Swerts, S. Kundu, D. Yakimets, R. Baert, H. Oh, A. Spessot, A. Mocuta, G. S. Kar, and A. Furnemont, “Enablement of stt-mram as last level cache for the high performance computing domain at the 5nm node”, in *2018 IEEE IEDM*, 2018, pp. 18.3.1–18.3.4. DOI: 10.1109/IEDM.2018.8614637.

[20] Y. J. Song, J. H. Lee, S. H. Han, H. C. Shin, K. H. Lee, K. Suh, D. E. Jeong, G. H. Koh, S. C. Oh, J. H. Park, S. O. Park, B. J. Bae, O. I. Kwon, K. H. Hwang, B. Seo, Y. Lee, S. H. Hwang, D. S. Lee, Y. Ji, K. Park, G. T. Jeong, H. S. Hong, K. P. Lee, H. K. Kang, and E. S. Jung, “Demonstration of highly manufacturable stt-mram embedded in 28nm logic”, in *2018 IEEE IEDM*, 2018, pp. 18.2.1–18.2.4. DOI: 10.1109/IEDM.2018.8614635.

[21] S. Ikegawa, K. Nagel, F. B. Mancoff, S. M. Alam, M. Arora, M. DeHerrera, H. K. Lee, S. Mukherjee, G. Shimon, J. J. Sun, I. Rahman, F. Neumeyer, H. Y. Chou, C. Tan, A. Shah, and S. Aggarwal, “High-speed (400mb/s) and low-ber stt-mram technology for industrial applications”, in *2022 IEEE IEDM*, 2022, pp. 10.4.1–10.4.4. DOI: 10.1109/IEDM45625.2022.10019513.

[22] Y. J. Song, J. H. Lee, H. C. Shin, K. H. Lee, K. Suh, J. R. Kang, S. S. Pyo, H. T. Jung, S. H. Hwang, G. H. Koh, S. C. Oh, S. O. Park, J. K. Kim, J. C. Park, J. Kim, K. H. Hwang, G. T. Jeong, K. P. Lee, and E. S. Jung, “Highly functional and reliable 8mb stt-mram embedded in 28nm logic”, in *2016 IEEE IEDM*, 2016, pp. 27.2.1–27.2.4. DOI: 10.1109/IEDM.2016.7838491.

[23] S. Sakhare, M. Perumkunnil, T. H. Bao, S. Rao, W. Kim, D. Crotti, F. Yasin, S. Couet, J. Swerts, S. Kundu, D. Yakimets, R. Baert, H. Oh, A. Spessot, A. Mocuta, G. S. Kar, and A. Furnemont, “Enablement of stt-mram as last level cache for the high performance computing domain at the 5nm node”, in *2018 IEEE IEDM*, 2018, pp. 18.3.1–18.3.4. DOI: 10.1109/IEDM.2018.8614637.

[24] S. H. Han, J. H. Lee, K. S. Suh, K. T. Nam, D. E. Jeong, S. C. Oh, S. H. Hwang, Y. Ji, K. Lee, K. Lee, Y. J. Song, Y. G. Hong, and G. T. Jeong, “Reliability of stt-mram for various embedded applications”, in *2021 IEEE IRPS*, 2021, pp. 1–5. DOI: 10.1109/IRPS46558.2021.9405094.

[25] V. B. Naik, K. Yamane, T. Lee, J. Kwon, R. Chao, J. Lim, N. Chung, B. Behin-Aein, L. Hau, D. Zeng, Y. Otani, C. Chiang, Y. Huang, L. Pu, S. Jang, W. Neo, H. Dixit, S. K. L. C. Goh, E. H. Toh, T. Ling, J. Hwang, J. Ting, R. Low, L. Zhang, C. Lee, N. Balasankaran, F. Tan, K. W. Gan, H. Yoon, G. Congedo, J. Mueller, B. Pfefferling, O. Kallensee, A. Vogel, V. Kriegerstein, T. Merbeth, C. Seet, S. Ong, J. Xu, J. Wong, Y. You, S. Woo, T. Chan, E. Quek, and S. Y. Siah, “Jedec-qualified highly reliable 22nm fd-soi embedded mram for low-power industrial-grade, and extended performance towards automotive-grade-1 applications”, in *2020 IEEE IEDM*, 2020, pp. 11.3.1–11.3.4. DOI: 10.1109/IEDM13553.2020.9371935.

[26] T. Y. Lee, K. Yamane, Y. Otani, D. Zeng, J. Kwon, J. H. Lim, V. B. Naik, L. Y. Hau, R. Chao, N. L. Chung, T. Ling, S. H. Jang, L. C. Goh, J. Hwang, L. Zhang, R. Low, N. Balasankaran, F. Tan, J. W. Ting, J. Chang, C. S. Seet, S. Ong, Y. S. You, S. T. Woo, T. H. Chan, and S. Y. Siah, "Advanced mtj stack engineering of stt-mram to realize high speed applications", in *2020 IEEE IEDM*, 2020, pp. 11.6.1–11.6.4. DOI: 10.1109/IEDM13553.2020.9372015.

[27] P.-H. Lee, C.-F. Lee, Y.-C. Shih, H.-J. Lin, Y.-A. Chang, C.-H. Lu, Y.-L. Chen, C.-P. Lo, C.-C. Chen, C.-H. Kuo, T.-L. Chou, C.-Y. Wang, J. J. Wu, R. Wang, H. Chuang, Y. Wang, Y.-D. Chih, and T.-Y. J. Chang, "33.1 a 16nm 32mb embedded stt-mram with a 6ns read-access time, a 1m-cycle write endurance, 20-year retention at 150°c and mtj-otp solutions for magnetic immunity", in *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, 2023, pp. 494–496. DOI: 10.1109/ISSCC42615.2023.10067837.

[28] Y.-D. Chih, Y.-C. Shih, C.-F. Lee, Y.-A. Chang, P.-H. Lee, H.-J. Lin, Y.-L. Chen, C.-P. Lo, M.-C. Shih, K.-H. Shen, H. Chuang, and T.-Y. J. Chang, "13.3 a 22nm 32mb embedded stt-mram with 10ns read speed, 1m cycle write endurance, 10 years retention at 150°c and high immunity to magnetic field interference", in *2020 IEEE International Solid-State Circuits Conference - (ISSCC)*, 2020, pp. 222–224. DOI: 10.1109/ISSCC19947.2020.9062955.

[29] S. M. Nair, "Variation analysis, fault modeling and yield improvement of emerging spintronic memories", Ph.D. dissertation, KIT-Bibliothek, 2020.

[30] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, "Ai and memory wall", *IEEE Micro*, vol. 44, no. 3, pp. 33–39, May 2024, ISSN: 0272-1732. DOI: 10.1109/MM.2024.3373763. [Online]. Available: https://doi.org/10.1109/MM.2024.3373763.

[31] S. Yin, X. Sun, S. Yu, and J.-S. Seo, "High-throughput in-memory computing for binary deep neural networks with monolithically integrated rram and 90-nm cmos", *IEEE Transactions on Electron Devices*, vol. 67, no. 10, pp. 4185–4192, 2020. DOI: 10.1109/TED.2020.3015178.

[32] P. Gu, B. Li, T. Tang, S. Yu, Y. Cao, Y. Wang, and H. Yang, "Technological exploration of rram crossbar array for matrix-vector multiplication", in *The 20th Asia and South Pacific Design Automation Conference*, 2015, pp. 106–111. DOI: 10.1109/ASPDAC.2015.7058989.

[33] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan, "Computing in memory with spin-transfer torque magnetic ram", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 3, pp. 470–483, 2018. DOI: 10.1109/TVLSI.2017.2776954.

[34] Z. He, S. Angizi, and D. Fan, "Exploring stt-mram based in-memory computing paradigm with application of image edge extraction", in *2017 IEEE International Conference on Computer Design (ICCD)*, 2017, pp. 439–446. DOI: 10.1109/ICCD.2017.78.

[35] G. W. Burr, R. M. Shelby, S. Sidler, C. di Nolfo, J. Jang, I. Boybat, R. S. Shenoy, P. Narayanan, K. Virwani, E. U. Giacometti, B. N. Kurdi, and H. Hwang, "Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element", *IEEE Transactions on Electron Devices*, vol. 62, no. 11, pp. 3498–3507, 2015. DOI: 10.1109/TED.2015.2439635.

[36] P. Mannocci, M. Farronato, N. Lepri, L. Cattaneo, A. Glukhov, Z. Sun, and D. Ielmini, "In-memory computing with emerging memory devices: Status and outlook", *APL Machine Learning*, vol. 1, no. 1, p. 010 902, Feb. 2023, ISSN: 2770-9019. DOI: 10.1063/5.0136403. eprint: https://pubs.aip.org/aip/aml/article-pdf/doi/10.1063/5.0136403/19999534/010902_1_5.0136403.pdf. [Online]. Available: https://doi.org/10.1063/5.0136403.

[37] A. Mukherjee, K. Saurav, P. Nair, S. Shekhar, and M. Lis, "A case for emerging memories in dnn accelerators", in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 938–941. DOI: 10.23919/DATE51398.2021.9474252.

[38] H. Li, M. Bhargav, P. N. Whatmough, and H.-S. Philip Wong, “On-chip memory technology design space explorations for mobile deep neural network accelerators”, in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[39] N. Sayed, “High-performance energy-efficient and reliable design of spin-transfer torque magnetic memory”, Ph.D. dissertation, KIT-Bibliothek, 2020.

[40] W. ZHAO, Y. Zhang, T. Devolder, J.-O. Klein, D. Ravelosona Ramasitera, C. Chappert, and P. Mazoyer, “Failure and reliability analysis of stt-mram”, *Microelectronics Reliability*, vol. 52, pp. 1848–1852, Sep. 2012. DOI: 10.1016/j.microrel.2012.06.035.

[41] T. Devolder, J. Hayakawa, K. Ito, H. Takahashi, S. Ikeda, P. Crozat, N. Zerounian, J.-V. Kim, C. Chappert, and H. Ohno, “Single-shot time-resolved measurements of nanosecond-scale spin-transfer induced switching: Stochastic versus deterministic aspects”, *Phys. Rev. Lett.*, vol. 100, p. 057 206, 5 2008. DOI: 10.1103/PhysRevLett.100.057206.

[42] N. Sayed, R. Bishnoi, F. Oboril, and M. B. Tahoori, “A cross-layer adaptive approach for performance and power optimization in stt-mram”, in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 791–796. DOI: 10.23919/DATE.2018.8342114.

[43] L. Wu, S. Rao, M. Taouil, E. J. Marinissen, G. Sankar Kar, and S. Hamdioui, “Impact of magnetic coupling and density on stt-mram performance”, in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020, pp. 1211–1216. DOI: 10.23919/DATE48585.2020.9116444.

[44] G. Zhang and Y. Jiang, “Modeling and enhancing magnetic immunity of STT-MRAM”, *AIP Advances*, vol. 13, no. 2, p. 025 232, Feb. 2023, ISSN: 2158-3226. DOI: 10.1063/9.0000521. eprint: https://pubs.aip.org/aip/adv/article-pdf/doi/10.1063/9.0000521/16759415/025232_1_online.pdf.

[45] C. Münch, J. Yun, M. Keim, and M. B. Tahoori, “Mbist-supported trim adjustment to compensate thermal behavior of mram”, in *2021 IEEE European Test Symposium (ETS)*, 2021, pp. 1–6. DOI: 10.1109/ETS50041.2021.9465383.

[46] P. Girard, Y. Cheng, A. Virazel, W. Zhao, R. Bishnoi, and M. B. Tahoori, “A survey of test and reliability solutions for magnetic random access memories”, *Proceedings of the IEEE*, vol. 109, no. 2, pp. 149–169, 2021. DOI: 10.1109/JPROC.2020.3029600.

[47] H. Naeimi, C. Augustine, A. Raychowdhury, S.-L. Lu, and J. Tschanz, “Sttram scaling and retention failure”, *Intel Technology Journal*, vol. 17, pp. 54–75, Jan. 2013.

[48] W. ZHAO, X. Zhao, B. Zhang, K. Cao, L. Wang, W. Kang, Q. Shi, M. Wang, Y. Zhang, Y. Wang, S. Peng, J.-O. Klein, L. Naviner, and D. Ravelosona Ramasitera, “Failure analysis in magnetic tunnel junction nanopillar with interfacial perpendicular magnetic anisotropy”, *Materials*, vol. 9, p. 41, Jan. 2016. DOI: 10.3390/ma9010041.

[49] L. Wu, S. Rao, M. Taouil, E. J. Marinissen, G. Sankar Kar, and S. Hamdioui, “Testing stt-mram: Manufacturing defects, fault models, and test solutions”, in *2021 IEEE International Test Conference (ITC)*, 2021, pp. 143–152. DOI: 10.1109/ITC50571.2021.00022.

[50] K. L. Pey, J. H. Lim, N. Raghavan, S. Mei, J. H. Kwon, V. B. Naik, K. Yamane, H. Yang, and K. Lee, “New insights into dielectric breakdown of mgo in stt-mram devices”, in *2019 Electron Devices Technology and Manufacturing Conference (EDTM)*, 2019, pp. 264–266. DOI: 10.1109/EDTM.2019.8731071.

[51] J. Lim, N. Raghavan, A. Padovani, J. Kwon, K. Yamane, H. Yang, V. Naik, L. Larcher, K. Lee, and K. Pey, "Investigating the statistical-physical nature of mgo dielectric breakdown in stt-mram at different operating conditions", in *2018 IEEE International Electron Devices Meeting (IEDM)*, 2018, pp. 25.3.1–25.3.4. doi: `10.1109/IEDM.2018.8614515`.

[52] L. Wu, M. Taouil, S. Rao, E. J. Marinissen, and S. Hamdioui, "Survey on STT-MRAM testing: Failure mechanisms, fault models, and tests", *CoRR*, vol. abs/2001.05463, 2020. arXiv: `2001.05463`. [Online]. Available: `https://arxiv.org/abs/2001.05463`.

[53] C. Münch and M. B. Tahoori, "Defect characterization of spintronic-based neuromorphic circuits", in *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2020, pp. 1–4. doi: `10.1109/IOLTS50870.2020.9159722`.

[54] D. Veksler, G. Bersuker, L. Vandelli, A. Padovani, L. Larcher, A. Muraviev, B. Chakrabarti, E. Vogel, D. C. Gilmer, and P. D. Kirsch, "Random telegraph noise (rtn) in scaled rram devices", in *2013 IEEE International Reliability Physics Symposium (IRPS)*, 2013, MY.10.1–MY.10.4. doi: `10.1109/IRPS.2013.6532101`.

[55] A. Grossi, E. Nowak, C. Zambelli, C. Pellissier, S. Bernasconi, G. Cibrario, K. El Hajjam, R. Crochemore, J. Nodin, P. Olivo, and L. Perniola, "Fundamental variability limits of filament-based rram", in *2016 IEEE International Electron Devices Meeting (IEDM)*, 2016, pp. 4.7.1–4.7.4. doi: `10.1109/IEDM.2016.7838348`.

[56] C.-Y. Chen, H.-C. Shih, C.-W. Wu, C.-H. Lin, P.-F. Chiu, S.-S. Sheu, and F. T. Chen, "Rram defect modeling and failure analysis based on march test and a novel squeeze-search scheme", *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 180–190, 2015. doi: `10.1109/TC.2014.12`.

[57] Y. Long, X. She, and S. Mukhopadhyay, "Design of reliable dnn accelerator with un-reliable reram", in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 1769–1774. doi: `10.23919/DATE.2019.8715178`.

[58] Y. Cassuto, S. Kvatinsky, and E. Yaakobi, "Write sneak-path constraints avoiding disturbs in memristor crossbar arrays", in *2016 IEEE International Symposium on Information Theory (ISIT)*, 2016, pp. 950–954. doi: `10.1109/ISIT.2016.7541439`.

[59] H. R. Mahdiani, S. M. Fakhraie, and C. Lucas, "Relaxed fault-tolerant hardware implementation of neural networks in the presence of multiple transient errors", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 8, pp. 1215–1228, 2012. doi: `10.1109/TNNLS.2012.2199517`.

[60] Y. Ibrahim, H. Wang, J. gyunyunLiu, J. Wei, L. Chen, P. Rech, K. Adam, and G. Guo, "Soft errors in dnn accelerators: A comprehensive review", *Microelectronics Reliability*, vol. 115, p. 113 969, 2020, issn: 0026-2714. doi: `https://doi.org/10.1016/j.microrel.2020.113969`.

[61] S. Wiefels, C. Bengel, N. Kopperberg, K. Zhang, R. Waser, and S. Menzel, "Hrs instability in oxide-based bipolar resistive switching cells", *IEEE Transactions on Electron Devices*, vol. 67, no. 10, pp. 4208–4215, 2020. doi: `10.1109/TED.2020.3018096`.

[62] B. Gleixner, F. Pellizzer, and R. Bez, "Reliability characterization of phase change memory", in *2009 10th Annual Non-Volatile Memory Technology Symposium (NVMTS)*, 2009, pp. 7–11. doi: `10.1109/NVMT.2009.5429783`.

[63] H. Pozidis, T. Mittelholzer, N. Papandreou, T. Parnell, and M. Stanisavljevic, "Phase change memory reliability: A signal processing and coding perspective", *IEEE Transactions on Magnetics*, vol. 51, no. 4, pp. 1–7, 2015. doi: `10.1109/TMAG.2014.2357176`.

[64] S. Hemaram, M. B. Tahoori, F. Catthoor, S. Rao, S. Couet, V. Pica, and G. S. Kar, "Soft and hard error-correction techniques in stt-mram", *IEEE Design & Test*, vol. 41, no. 5, pp. 65–82, 2024. DOI: `10.1109/MDAT.2024.3395972`.

[65] S. Hemaram, M. B. Tahoori, F. Catthoor, S. Rao, S. Couet, and G. S. Kar, "Hard error correction in stt-mram", in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 752–757. DOI: `10.1109/ASP-DAC58780.2024.10473796`.

[66] S. Hemaram, M. B. Tahoori, F. Catthoor, S. Rao, S. Couet, T. Marinelli, V. Pica, and G. S. Kar, "Asymmetric and adaptive error correction in stt-mram", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–14, 2025. DOI: `10.1109/TCAD.2025.3541188`.

[67] S. Hemaram, M. Mayahinia, M. B. Tahoori, F. Catthoor, S. Rao, S. Couet, T. Marinelli, A. Farokhnejad, and G. S. Kar, "Intera-ecc: Interconnect-aware error correction in stt-mram", in *2025 Design, Automation & Test in Europe Conference (DATE)*, 2025, pp. 1–2. DOI: `10.23919/DATE64628.2025.10992925`.

[68] S. Hemaram, T. Marinelli, M. Mayahinia, M. B. Tahoori, F. Catthoor, S. Rao, F. G. Redondo, and G. S. Kar, "Location-aware error correction for mitigating the impact of interconnects on stt-mram reliability", *IEEE Transactions on Device and Materials Reliability*, pp. 1–1, 2025. DOI: `10.1109/TDMR.2025.3627442`.

[69] S. Hemaram, S. T. Ahmed, M. Mayahinia, C. Münch, and M. B. Tahoori, "A low overhead checksum technique for error correction in memristive crossbar for deep learning applications", in *2023 IEEE 41st VLSI Test Symposium (VTS)*, 2023, pp. 1–7. DOI: `10.1109/VTS56346.2023.10140077`.

[70] S. Hemaram, M. Mayahinia, and M. B. Tahoori, "Adaptive block error correction for memristive crossbars", in *2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2022, pp. 1–6. DOI: `10.1109/IOLTS56730.2022.9897817`.

[71] S. T. Ahmed, S. Hemaram, and M. B. Tahoori, "Nn-ecc: Embedding error correction codes in neural network weight memories using multi-task learning", in *2024 IEEE 42nd VLSI Test Symposium (VTS)*, 2024, pp. 1–7. DOI: `10.1109/VTS60656.2024.10538886`.

[72] M. S. Roodsari, S. Hemaram, and M. B. Tahoori, "Non-uniform error correction for hyperdimensional computing edge accelerators", in *2025 IEEE European Test Symposium (ETS)*, 2025, pp. 1–6. DOI: `10.1109/ETS63895.2025.11049622`.

[73] Z. Zhang, S. Hemaram, M. Mayahinia, C. Weis, N. Wehn, M. Tahoori, S. Nassif, G. Tshagharyan, G. Harutyunyan, and Y. Zorian, "Analysis and mitigation of radiation effects in sram-based register files", in *2025 IEEE European Test Symposium (ETS)*, 2025, pp. 1–4. DOI: `10.1109/ETS63895.2025.11049612`.

[74] S. M. Siddaramu, M. Mayahinia, S. Hemaram, S. Ozev, and M. Tahoori, "Testing of passive memristive crossbars in ai hardware accelerators", in *2025 IEEE ATS/ITC-Asia (Accepted)*, 2025, pp. 1–6.

[75] A. V. Khvalkovskiy, D. Apalkov, S. Watts, R. Chepulskii, R. S. Beach, A. Ong, X. Tang, A. Driskill-Smith, W. H. Butler, P. B. Visscher, D. Lottis, E. Chen, V. Nikitin, and M. Krounbi, "Basic principles of stt-mram cell operation in memory arrays", *Journal of Physics D: Applied Physics*, vol. 46, no. 7, p. 074 001, 2013. DOI: `10.1088/0022-3727/46/7/074001`. [Online]. Available: `https://dx.doi.org/10.1088/0022-3727/46/7/074001`.

[76] X. Fong, Y. Kim, R. Venkatesan, S. H. Choday, A. Raghunathan, and K. Roy, "Spin-transfer torque memories: Devices, circuits, and systems", *Proceedings of the IEEE*, vol. 104, no. 7, pp. 1449–1488, 2016. DOI: `10.1109/JPROC.2016.2521712`.

[77] S. Salehi, D. Fan, and R. F. Demara, “Survey of stt-mram cell design strategies: Taxonomy and sense amplifier tradeoffs for resiliency”, *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, Apr. 2017, ISSN: 1550-4832. DOI: 10.1145/2997650. [Online]. Available: https://doi.org/10.1145/2997650.

[78] J. Yang, P. Wang, Y. Zhang, Y. Cheng, W. Zhao, Y. Chen, and H. H. Li, “Radiation-induced soft error analysis of stt-mram: A device to circuit approach”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 380–393, 2016. DOI: 10.1109/TCAD.2015.2474366.

[79] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing”, *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004. DOI: 10.1109/TDSC.2004.2.

[80] M. Peercy and P. Banerjee, “Fault tolerant vlsi systems”, *Proceedings of the IEEE*, vol. 81, no. 5, pp. 745–758, 1993. DOI: 10.1109/5.220905.

[81] J. Sosnowski, “Transient fault tolerance in digital systems”, *IEEE Micro*, vol. 14, no. 1, pp. 24–35, 1994. DOI: 10.1109/40.259897.

[82] G. Buja and R. Menis, “Dependability and functional safety: Applications in industrial electronics systems”, *IEEE Industrial Electronics Magazine*, vol. 6, no. 3, pp. 4–12, 2012. DOI: 10.1109/MIE.2012.2207815.

[83] R. Baumann, “Soft errors in advanced computer systems”, *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005. DOI: 10.1109/MDT.2005.69.

[84] C. L. Chen and M. Y. Hsiao, “Error-correcting codes for semiconductor memory applications: A state-of-the-art review”, *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 124–134, 1984. DOI: 10.1147/rd.282.0124.

[85] W. K. Huang, Y.-N. Shen, and F. Lombardi, “New approaches for the repairs of memories with redundancy by row/column deletion for yield enhancement”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 3, pp. 323–328, 1990. DOI: 10.1109/43.46807.

[86] C.-L. Su, Y.-T. Yeh, and C.-W. Wu, “An integrated ecc and redundancy repair scheme for memory reliability enhancement”, in *20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT’05)*, 2005, pp. 81–89. DOI: 10.1109/DFTVS.2005.18.

[87] M. Le Gallo and A. Sebastian, “An overview of phase-change memory device physics”, *Journal of Physics D: Applied Physics*, vol. 53, no. 21, p. 213 002, Mar. 2020. DOI: 10.1088/1361-6463/ab7794. [Online]. Available: https://dx.doi.org/10.1088/1361-6463/ab7794.

[88] M. Le Gallo, A. Athmanathan, D. Krebs, and A. Sebastian, “Evidence for thermally assisted threshold switching behavior in nanoscale phase-change memory cells”, *Journal of Applied Physics*, vol. 119, no. 2, p. 025 704, Jan. 2016, ISSN: 0021-8979. DOI: 10.1063/1.4938532. eprint: https://pubs.aip.org/aip/jap/article-pdf/doi/10.1063/1.4938532/15177001/025704_1_online.pdf. [Online]. Available: https://doi.org/10.1063/1.4938532.

[89] S. Eilert, M. Leinwander, and G. Crisenza, “Phase change memory: A new memory enables new memory usage models”, in *2009 IEEE International Memory Workshop*, 2009, pp. 1–2. DOI: 10.1109/IMW.2009.5090604.

[90] A. Chen, “A review of emerging non-volatile memory (nvm) technologies and applications”, *Solid-State Electronics*, vol. 125, pp. 25–38, 2016, Extended papers selected from ESSDERC 2015, ISSN: 0038-1101. DOI: https://doi.org/10.1016/j.sse.2016.07.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0038110116300867.

[91] M. Mayahinia, “Reliability modeling and mitigation in advanced memory technologies and paradigms”, Ph.D. dissertation, KIT-Bibliothek, 2024.

[92] A. L. Lacaita and D. Ielmini, “Reliability issues and scaling projections for phase change non volatile memories”, in *2007 IEEE International Electron Devices Meeting*, 2007, pp. 157–160. DOI: 10.1109/IEDM.2007.4418890.

[93] T. Kwon, M. Imran, and J.-S. Yang, “Reliability enhanced heterogeneous phase change memory architecture for performance and energy efficiency”, *IEEE Transactions on Computers*, vol. 70, no. 9, pp. 1388–1400, 2021. DOI: 10.1109/TC.2020.3009498.

[94] G. Navarro, “Reliability analysis of embedded Phase-Change Memories based on innovative materials”, Theses, Université de Grenoble, Dec. 2013. [Online]. Available: https://theses.hal.science/tel-01061792.

[95] G. Lama, G. Bourgeois, M. Bernard, N. Castellani, J. Sandrini, E. Nolot, J. Garrione, M. Cyrille, G. Navarro, and E. Nowak, “Reliability analysis in gete and gesbte based phase-change memory 4kb arrays targeting storage class memory applications”, *Microelectronics Reliability*, vol. 114, p. 113 823, 2020, 31st European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, ESREF 2020, ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2020.113823. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026271420304054.

[96] H. Pozidis, T. Mittelholzer, N. Papandreou, T. Parnell, and M. Stanisavljevic, “Phase change memory reliability: A signal processing and coding perspective”, *IEEE Transactions on Magnetics*, vol. 51, no. 4, pp. 1–7, 2015. DOI: 10.1109/TMAG.2014.2357176.

[97] S. Wiefels, C. Bengel, N. Kopperberg, K. Zhang, R. Waser, and S. Menzel, “Hrs instability in oxide-based bipolar resistive switching cells”, *IEEE Transactions on Electron Devices*, vol. 67, no. 10, pp. 4208–4215, 2020. DOI: 10.1109/TED.2020.3018096.

[98] C. Bengel, A. Siemon, F. Cüppers, S. Hoffmann-Eifert, A. Hardtdegen, M. von Witzleben, L. Hellmich, R. Waser, and S. Menzel, “Variability-aware modeling of filamentary oxide-based bipolar resistive switching cells using spice level compact models”, *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 12, pp. 4618–4630, 2020. DOI: 10.1109/TCSI.2020.3018502.

[99] S. Yu, “Overview of resistive switching memory (rram) switching mechanism and device modeling”, in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2014, pp. 2017–2020. DOI: 10.1109/ISCAS.2014.6865560.

[100] Y. Chen, “Reram: History, status, and future”, *IEEE Transactions on Electron Devices*, vol. 67, no. 4, pp. 1420–1433, 2020. DOI: 10.1109/TED.2019.2961505.

[101] F. Zahoor, F. A. Hussin, U. B. Isyaku, S. Gupta, F. A. Khanday, A. Chattopadhyay, and H. Abbas, “Resistive random access memory: Introduction to device mechanism, materials and application to neuromorphic computing”, *Discover Nano*, vol. 18, no. 1, p. 36, Mar. 2023. DOI: 10.1186/s11671-023-03775-y. [Online]. Available: https://doi.org/10.1186/s11671-023-03775-y.

[102] W. Wan, R. Kubendran, C. Schaefer, *et al.*, “A compute-in-memory chip based on resistive random-access memory”, *Nature*, vol. 608, pp. 504–512, 2022. DOI: 10.1038/s41586-022-04992-8. [Online]. Available: https://doi.org/10.1038/s41586-022-04992-8.

[103] S. Yu, W. Shim, X. Peng, and Y. Luo, “Rram for compute-in-memory: From inference to training”, *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 7, pp. 2753–2765, 2021. DOI: 10.1109/TCSI.2021.3072200.

[104] D. Veksler, G. Bersuker, L. Vandelli, A. Padovani, L. Larcher, A. Muraviev, B. Chakrabarti, E. Vogel, D. C. Gilmer, and P. D. Kirsch, "Random telegraph noise (rtn) in scaled rram devices", in *2013 IEEE International Reliability Physics Symposium (IRPS)*, 2013, MY.10.1–MY.10.4. DOI: 10.1109/IRPS.2013.6532101.

[105] A. Grossi, E. Nowak, C. Zambelli, C. Pellissier, S. Bernasconi, G. Cibrario, K. El Hajjam, R. Crochemore, J. Nodin, P. Olivo, and L. Perniola, "Fundamental variability limits of filament-based rram", in *2016 IEEE International Electron Devices Meeting (IEDM)*, 2016, pp. 4.7.1–4.7.4. DOI: 10.1109/IEDM.2016.7838348.

[106] C.-Y. Chen, H.-C. Shih, C.-W. Wu, C.-H. Lin, P.-F. Chiu, S.-S. Sheu, and F. T. Chen, "Rram defect modeling and failure analysis based on march test and a novel squeeze-search scheme", *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 180–190, 2015. DOI: 10.1109/TC.2014.12.

[107] W. Li, Y. Wang, H. Li, and X. Li, "Rramedy: Protecting reram-based neural network from permanent and soft faults during its lifetime", in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 2019, pp. 91–99. DOI: 10.1109/ICCD46524.2019.00020.

[108] H. Yi, S. Shiyu, D. Xiusheng, and C. Zhigang, "A study on deep neural networks framework", in *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, 2016, pp. 1519–1522. DOI: 10.1109/IMCEC.2016.7867471.

[109] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[110] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

[111] J. K. Eshraghian, M. Ward, E. O. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun, D. S. Jeong, and W. D. Lu, "Training spiking neural networks using lessons from deep learning", *Proceedings of the IEEE*, vol. 111, no. 9, pp. 1016–1054, 2023. DOI: 10.1109/JPROC.2023.3308088.

[112] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey and benchmarking of machine learning accelerators", in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–9. DOI: 10.1109/HPEC.2019.8916327.

[113] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A survey of accelerator architectures for deep neural networks", *Engineering*, vol. 6, no. 3, pp. 264–274, 2020, ISSN: 2095-8099. DOI: https://doi.org/10.1016/j.eng.2020.01.007.

[114] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators", *Commun. ACM*, vol. 63, no. 7, pp. 48–57, Jun. 2020, ISSN: 0001-0782. DOI: 10.1145/3361682. [Online]. Available: https://doi.org/10.1145/3361682.

[115] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Ai accelerator survey and trends", in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–9. DOI: 10.1109/HPEC49654.2021.9622867.

[116] S. Bavikadi, A. Dhavlle, A. Ganguly, A. Haridass, H. Hendy, C. Merkel, V. Janapa Reddi, P. Sutradhar, A. Joseph, and S. M. P D, "A survey on machine learning accelerators and evolutionary hardware platforms", *IEEE Design & Test*, vol. 39, pp. 91–116, Jun. 2022. DOI: 10.1109/MDAT.2022.3161126.

[117] P. Dhilleswararao, S. Boppu, M. S. Manikandan, and L. R. Cenkeramaddi, "Efficient hardware architectures for accelerating deep neural networks: Survey", *IEEE Access*, vol. 10, pp. 131 788–131 828, 2022. DOI: 10.1109/ACCESS.2022.3229767.

[118] C. Silvano, D. Ielmini, F. Ferrandi, L. Fiorin, S. Curzel, L. Benini, F. Conti, A. Garofalo, C. Zambelli, E. Calore, S. Schifano, M. Palesi, G. Ascia, D. Patti, S. Perri, N. Petra, D. De Caro, L. Lavagno, T. Urso, and R. Birke, "A survey on deep learning hardware accelerators for heterogeneous hpc platforms", Jun. 2023.

[119] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and scalability of gpu-based convolutional neural networks", in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010, pp. 317–324. doi: 10.1109/PDP.2010.43.

[120] H. Kimm, I. Paik, and H. Kimm, "Performance comparision of tpu, gpu, cpu on google colaboratory over distributed deep learning", in *2021 IEEE 14th International Symposium on Embedded MCSoC*, 2021, pp. 312–319. doi: 10.1109/MCSoC51149.2021.00053.

[121] K. Seshadri, B. Akin, J. Laudon, R. Narayanaswami, and A. Yazdanbakhsh, "An evaluation of edge tpu accelerators for convolutional neural networks", in *2022 IEEE International Symposium on Workload Characterization (IISWC)*, Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2022, pp. 79–91. doi: 10.1109/IISWC55918.2022.00017.

[122] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "Can fpgas beat gpus in accelerating next-generation deep neural networks?", in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, Monterey, California, USA: Association for Computing Machinery, 2017, pp. 5–14, isbn: 9781450343541. doi: 10.1145/3020078.3021740.

[123] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[dl] a survey of fpga-based neural network inference accelerators", *ACM Transactions on Reconfigurable Technology and Systems*, vol. 12, pp. 1–26, Mar. 2019. doi: 10.1145/3289185.

[124] I. Ullah, K. Inayat, J.-S. Yang, and J. Chung, "Factored radix-8 systolic array for tensor processing", in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6. doi: 10.1109/DAC18072.2020.9218585.

[125] J. J. Zhang, T. Gu, K. Basu, and S. Garg, "Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator", in *2018 IEEE 36th VLSI Test Symposium (VTS)*, 2018, pp. 1–6. doi: 10.1109/VTS.2018.8368656.

[126] M. Bavandpour, M. Mahmoodi, H. Nili, F. M. Bayat, M. Prezioso, A. Vincent, D. Strukov, and K. Likharev, "Mixed-signal neuromorphic inference accelerators: Recent results and future prospects", in *2018 IEEE International Electron Devices Meeting (IEDM)*, 2018, pp. 20.4.1–20.4.4. doi: 10.1109/IEDM.2018.8614659.

[127] H. Valavi, P. J. Ramadge, E. Nestler, and N. Verma, "A mixed-signal binarized convolutional-neural-network accelerator integrating dense weight storage and multiplication for reduced data movement", in *2018 IEEE Symposium on VLSI Circuits*, 2018, pp. 141–142. doi: 10.1109/VLSIC.2018.8502421.

[128] J. Zhang, Z. Wang, and N. Verma, "In-memory computation of a machine-learning classifier in a standard 6t sram array", *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, pp. 915–924, 2017. doi: 10.1109/JSSC.2016.2642198.

[129] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars", in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 14–26. doi: 10.1109/ISCA.2016.12.

[130] K. Roy, I. Chakraborty, M. Ali, A. Ankit, and A. Agrawal, “In-memory computing in emerging memory technologies for machine learning: An overview”, in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218505.

[131] S. Yin, X. Sun, S. Yu, and J.-S. Seo, “High-throughput in-memory computing for binary deep neural networks with monolithically integrated rram and 90-nm cmos”, *IEEE Transactions on Electron Devices*, vol. 67, no. 10, pp. 4185–4192, 2020. DOI: 10.1109/TED.2020.3015178.

[132] Y. Halawani, B. Mohammad, M. Abu Lebdeh, M. Al-Qutayri, and S. F. Al-Sarawi, “Reram-based in-memory computing for search engine and neural network applications”, *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 388–397, 2019. DOI: 10.1109/JETCAS.2019.2909317.

[133] A. Chen, “A review of emerging non-volatile memory (nvm) technologies and applications”, *Solid-State Electronics*, vol. 125, pp. 25–38, 2016, Extended papers selected from ESSDERC 2015, ISSN: 0038-1101. DOI: https://doi.org/10.1016/j.sse.2016.07.006.

[134] I. Valov, “Redox-based resistive switching memories (rerams): Electrochemical systems at the atomic scale”, *ChemElectroChem*, vol. 1, no. 1, pp. 26–36, 2014. DOI: https://doi.org/10.1002/celc.201300165.

[135] N. Kopperberg, S. Wiefels, K. Hofmann, J. Otterstedt, D. J. Wouters, R. Waser, and S. Menzel, “Endurance of 2 mbit based beol integrated reram”, *IEEE Access*, vol. 10, pp. 122 696–122 705, 2022. DOI: 10.1109/ACCESS.2022.3223657.

[136] S. Yin, X. Sun, S. Yu, and J.-S. Seo, “High-throughput in-memory computing for binary deep neural networks with monolithically integrated rram and 90-nm cmos”, *IEEE Transactions on Electron Devices*, vol. 67, no. 10, pp. 4185–4192, 2020. DOI: 10.1109/TED.2020.3015178.

[137] P. Gu, B. Li, T. Tang, S. Yu, Y. Cao, Y. Wang, and H. Yang, “Technological exploration of rram crossbar array for matrix-vector multiplication”, in *The 20th Asia and South Pacific Design Automation Conference*, 2015, pp. 106–111. DOI: 10.1109/ASPDAC.2015.7058989.

[138] S. Eilert, M. Leinwander, and G. Crisenza, “Phase change memory: A new memory enables new memory usage models”, in *2009 IEEE International Memory Workshop*, 2009, pp. 1–2. DOI: 10.1109/IMW.2009.5090604.

[139] G. W. Burr, R. M. Shelby, S. Sidler, C. di Nolfo, J. Jang, I. Boybat, R. S. Shenoy, P. Narayanan, K. Virwani, E. U. Giacometti, B. N. Kurdi, and H. Hwang, “Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element”, *IEEE Transactions on Electron Devices*, vol. 62, no. 11, pp. 3498–3507, 2015. DOI: 10.1109/TED.2015.2439635.

[140] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan, “Computing in memory with spin-transfer torque magnetic ram”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 3, pp. 470–483, 2018. DOI: 10.1109/TVLSI.2017.2776954.

[141] Z. He, S. Angizi, and D. Fan, “Exploring stt-mram based in-memory computing paradigm with application of image edge extraction”, in *2017 IEEE International Conference on Computer Design (ICCD)*, 2017, pp. 439–446. DOI: 10.1109/ICCD.2017.78.

[142] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, “Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars”, in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 14–26. DOI: 10.1109/ISCA.2016.12.

[143] S. T. Ahmed, M. Hefenbrock, C. Münch, and M. B. Tahoori, “Neuroscrub+: Mitigating retention faults using flexible approximate scrubbing in neuromorphic fabric based on resistive memories”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2022. DOI: 10.1109/TCAD.2022.3205872.

[144] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, “Tetris: Scalable and efficient neural network acceleration with 3d memory”, *SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 751–764, Apr. 2017, ISSN: 0163-5964. DOI: 10.1145/3093337.3037702.

[145] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, “Processing-in-memory for energy-efficient neural network training: A heterogeneous approach”, in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 655–668. DOI: 10.1109/MICRO.2018.00059.

[146] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li, “Smartshuttle: Optimizing off-chip memory accesses for deep learning accelerators”, in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 343–348. DOI: 10.23919/DATE.2018.8342033.

[147] A. Asad, R. Kaur, and F. Mohammadi, “A survey on memory subsystems for deep neural network accelerators”, *Future Internet*, vol. 14, no. 5, 2022, ISSN: 1999-5903. DOI: 10.3390/fi14050146.

[148] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, “Ares: A framework for quantifying the resilience of deep neural networks”, in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465834.

[149] S. Jeong, S. Kang, and J.-S. Yang, “Pair: Pin-aligned in-dram ecc architecture using expandability of reed-solomon code”, in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218745.

[150] C. Schorn, A. Guntoro, and G. Ascheid, “An efficient bit-flip resilience optimization method for deep neural networks”, in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 1507–1512. DOI: 10.23919/DATE.2019.8714885.

[151] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, “Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules”, in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 283–295. DOI: 10.1109/HPCA.2015.7056040.

[152] W.-H. Chen, K.-X. Li, W.-Y. Lin, K.-H. Hsu, P.-Y. Li, C.-H. Yang, C.-X. Xue, E.-Y. Yang, Y.-K. Chen, Y.-S. Chang, T.-H. Hsu, Y.-C. King, C.-J. Lin, R.-S. Liu, C.-C. Hsieh, K.-T. Tang, and M.-F. Chang, “A 65nm 1mb nonvolatile computing-in-memory reram macro with sub-16ns multiply-and-accumulate for binary dnn ai edge processors”, in *2018 IEEE International Solid-State Circuits Conference - (ISSCC)*, 2018, pp. 494–496. DOI: 10.1109/ISSCC.2018.8310400.

[153] K. Guo, J. Yu, X. Ning, Y. Hu, Y. Wang, and H. Yang, “Rram based buffer design for energy efficient cnn accelerator”, in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2018, pp. 435–440. DOI: 10.1109/ISVLSI.2018.00085.

[154] Y. Pan, P. Ouyang, Y. Zhao, W. Kang, S. Yin, Y. Zhang, W. Zhao, and S. Wei, “A multilevel cell stt-mram-based computing in-memory accelerator for binary convolutional neural network”, *IEEE Transactions on Magnetics*, vol. 54, no. 11, pp. 1–5, 2018. DOI: 10.1109/TMAG.2018.2848625.

[155] V. Joshi, M. Le Gallo, S. Haefeli, *et al.*, “Accurate deep neural network inference using computational phase-change memory”, *Nature Communications*, vol. 11, no. 2473, 2020. DOI: https://doi.org/10.1038/s41467-020-16108-9.

[156] A. Mukherjee, K. Saurav, P. Nair, S. Shekhar, and M. Lis, "A case for emerging memories in dnn accelerators", in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 938–941. DOI: `10.23919/DATE51398.2021.9474252`.

[157] F. Su, C. Liu, and H.-G. Stratigopoulos, "Testability and dependability of ai hardware: Survey, trends, challenges, and perspectives", *IEEE Design & Test*, vol. 40, no. 2, pp. 8–58, 2023. DOI: `10.1109/MDAT.2023.3241116`.

[158] C. Torres-Huitzil and B. Girau, "Fault and error tolerance in neural networks: A review", *IEEE Access*, vol. 5, pp. 17 322–17 341, 2017. DOI: `10.1109/ACCESS.2017.2742698`.

[159] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications", in *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[160] F. S. Hosseini, F. Meng, C. Yang, W. Wen, and R. Cammarota, "Tolerating defects in low-power neural network accelerators via retraining-free weight approximation", *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, Sep. 2021, ISSN: 1539-9087. DOI: `10.1145/3477016`.

[161] J. J. Zhang, T. Gu, K. Basu, and S. Garg, "Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator", in *2018 IEEE 36th VLSI Test Symposium (VTS)*, 2018, pp. 1–6. DOI: `10.1109/VTS.2018.8368656`.

[162] A. Orgenci, G. Dundar, and S. Balkur, "Fault-tolerant training of neural networks in the presence of mos transistor mismatches", *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 48, no. 3, pp. 272–281, 2001. DOI: `10.1109/82.924069`.

[163] E. M. El Mhamdi and R. Guerraoui, "When neurons fail", in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 1028–1037. DOI: `10.1109/IPDPS.2017.66`.

[164] B. Salami, O. S. Unsal, and A. C. Kestelman, "On the resilience of rtl nn accelerators: Fault characterization and mitigation", in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018, pp. 322–329. DOI: `10.1109/CAHPC.2018.8645906`.

[165] J. E. R. Condia, F. F. dos Santos, M. S. Reorda, and P. Rech, "Combining architectural simulation and software fault injection for a fast and accurate cnns reliability evaluation on gpus", in *2021 IEEE 39th VLSI Test Symposium (VTS)*, 2021, pp. 1–7. DOI: `10.1109/VTS50974.2021.9441044`.

[166] A. Chaudhuri, J. Talukdar, F. Su, and K. Chakrabarty, "Functional criticality analysis of structural faults in ai accelerators", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 12, pp. 5657–5670, 2022. DOI: `10.1109/TCAD.2022.3166108`.

[167] C. Liu, M. Hu, J. P. Strachan, and H. Li, "Rescuing memristor-based neuromorphic design with high defects", in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017, pp. 1–6. DOI: `10.1145/3061639.3062310`.

[168] N. Khoshavi, C. Broyles, Y. Bi, and A. Roohi, "Fiji-fin: A fault injection framework on quantized neural network inference accelerator", in *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2020, pp. 1139–1144. DOI: `10.1109/ICMLA51294.2020.00183`.

[169] Y. Toca-Díaz, R. Hernández Palacios, R. Gran Tejero, and A. Valero, "Flip-and-patch: A fault-tolerant technique for on-chip memories of cnn accelerators at low supply voltage", *Microprocessors and Microsystems*, vol. 106, p. 105 023, 2024, ISSN: 0141-9331. DOI: `https://doi.org/10.1016/j.micpro.2024.105023`.

[170] H. Stratigopoulos, T. Spyrou, and S. Raptis, "Testing and reliability of spiking neural networks: A review of the state-of-the-art", in *2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2023, pp. 1–8. DOI: 10.1109/DFT59622.2023.10313541.

[171] A. Aviziens, "Fault-tolerant systems", *IEEE Transactions on Computers*, vol. C-25, no. 12, pp. 1304–1312, 1976. DOI: 10.1109/TC.1976.1674598.

[172] S. Lin and D. Costello, *Error Control Coding: Fundamentals and Applications* (Computer applications in electrical engineering series). Prentice-Hall, 1983, ISBN: 9780132837965.

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

[174] M. Y. Hsiao, "A class of optimal minimum odd-weight-column sec-ded codes", *IBM J. Res. Dev.*, vol. 14, no. 4, pp. 395–401, Jul. 1970, ISSN: 0018-8646. DOI: 10.1147/rd.144.0395.

[175] S. Nabipour, J. Javidan, and R. Drechsler, "Trends and challenges in design of embedded bch error correction codes in multi-levels nand flash memory devices", *Memories - Materials, Devices, Circuits and Systems*, vol. 7, p. 100 099, 2024, ISSN: 2773-0646. DOI: https://doi.org/10.1016/j.memori.2024.100099.

[176] D. Strukov, "The area and latency tradeoffs of binary bit-parallel bch decoders for prospective nanoelectronic memories", in *2006 Fortieth Asilomar Conference on Signals, Systems and Computers*, 2006, pp. 1183–1187. DOI: 10.1109/ACSSC.2006.354942.

[177] R. Naseer and J. Draper, "Parallel double error correcting code design to mitigate multi-bit upsets in srams", in *ESSCIRC 2008 - 34th European Solid-State Circuits Conference*, 2008, pp. 222–225. DOI: 10.1109/ESSCIRC.2008.4681832.

[178] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations", *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984. DOI: 10.1109/TC.1984.1676475.

[179] J.-Y. Jou and J. Abraham, "Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures", *Proceedings of the IEEE*, vol. 74, no. 5, pp. 732–741, 1986. DOI: 10.1109/PROC.1986.13535.

[180] W. Kang, W. Zhao, L. Yang, J.-O. Klein, Y. Zhang, and D. Ravclosona, "One-step majority-logic-decodable codes enable stt-mram for high-speed working memories", in *2014 IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2014, pp. 1–6. DOI: 10.1109/NVMSA.2014.6927189.

[181] B. Li, Y. Pei, and W. Wen, "Efficient low-density parity-check (ldpc) code decoding for combating asymmetric errors in stt-ram", in *2016 IEEE Computer Society Annual Symposium on VLSI*, 2016, pp. 266–271. DOI: 10.1109/ISVLSI.2016.9.

[182] Z. Mei, K. Cai, and B. Dai, "Polar codes for spin-torque transfer magnetic random access memory", *IEEE Transactions on Magnetics*, vol. 54, no. 11, pp. 1–5, 2018. DOI: 10.1109/TMAG.2018.2846786.

[183] H. Farbeh, H. Kim, S. G. Miremadi, and S. Kim, "Floating-ecc: Dynamic repositioning of error correcting code bits for extending the lifetime of stt-ram caches", *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3661–3675, 2016. DOI: 10.1109/TC.2016.2557326.

[184] X. Guo, M. N. Bojnordi, Q. Guo, and E. Ipek, "Sanitizer: Mitigating the impact of expensive ecc checks on stt-mram based main memories", *IEEE Transactions on Computers*, vol. 67, no. 6, pp. 847–860, 2018. DOI: 10.1109/TC.2017.2779151.

[185] M. Hadizadeh, E. Cheshmikhani, and H. Asadi, “Stair: High reliable stt-mram aware multi-level i/o cache architecture by adaptive ecc allocation”, in *2020 DATE*, 2020, pp. 1484–1489. DOI: 10.23919/DATE48585.2020.9116550.

[186] S. M. Seyedzadeh, R. Maddah, A. Jones, and R. Melhem, “Leveraging ecc to mitigate read disturbance, false reads and write faults in stt-ram”, in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 215–226. DOI: 10.1109/DSN.2016.28.

[187] S. Mittal, J. S. Vetter, and L. Jiang, “Addressing read-disturbance issue in stt-ram by data compression and selective duplication”, *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 94–98, 2017. DOI: 10.1109/LCA.2016.2645207.

[188] E. Cheshmikhani, H. Farbeh, and H. Asadi, “Enhancing reliability of stt-mram caches by eliminating read disturbance accumulation”, in *2019 DATE*, 2019, pp. 854–859. DOI: 10.23919/DATE.2019.8714946.

[189] B. Wu, B. Zhang, Y. Cheng, Y. Wang, D. Liu, and W. Zhao, “An adaptive thermal-aware ecc scheme for reliable stt-mram llc design”, *IEEE Transactions on VLSI Systems*, vol. 27, no. 8, pp. 1851–1860, 2019. DOI: 10.1109/TVLSI.2019.2913207.

[190] E. Cheshmikhani, H. Farbeh, and H. Asadi, “3rset: Read disturbance rate reduction in stt-mram caches by selective tag comparison”, *IEEE Transactions on Computers*, vol. 71, no. 6, pp. 1305–1319, 2022. DOI: 10.1109/TC.2021.3082004.

[191] A. Haj Aboutalebi, E. C. Ahn, B. Mao, S. Wu, and L. Duan, “Mitigating and tolerating read disturbance in stt-mram-based main memory via device and architecture innovations”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 12, pp. 2229–2242, 2019. DOI: 10.1109/TCAD.2018.2878166.

[192] W. Wen, M. Mao, X. Zhu, S. H. Kang, D. Wang, and Y. Chen, “Cd-ecc: Content-dependent error correction codes for combating asymmetric nonvolatile memory operation errors”, in *2013 IEEE/ACM ICCAD*, 2013, pp. 1–8. DOI: 10.1109/ICCAD.2013.6691090.

[193] Z. Azad, H. Farbeh, A. M. H. Monazzah, and S. G. Miremadi, “An efficient protection technique for last level stt-ram caches in multi-core processors”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1564–1577, 2017. DOI: 10.1109/TPDS.2016.2628742.

[194] Z. Azad, H. Farbeh, A. M. H. Monazzah, and S. G. Miremadi, “Aware: Adaptive way allocation for reconfigurable eccs to protect write errors in stt-ram caches”, *IEEE Transactions on Emerging Topics in Computing*, vol. 7, no. 3, pp. 481–492, 2019. DOI: 10.1109/TETC.2017.2701880.

[195] X. Wang, M. Mao, E. Eken, W. Wen, H. Li, and Y. Chen, “Sliding basket: An adaptive ecc scheme for runtime write failure suppression of stt-ram cache”, in *2016 DATE*, 2016, pp. 762–767.

[196] I. Alam, S. Pal, and P. Gupta, “Compression with multi-ecc: Enhanced error resiliency for magnetic memories”, in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS ’19, Washington, District of Columbia, USA: Association for Computing Machinery, 2019, pp. 85–100, ISBN: 9781450372060. DOI: 10.1145/3357526.3357533. [Online]. Available: https://doi.org/10.1145/3357526.3357533.

[197] M. A. Qureshi, J. Park, and S. Kim, “Sale: Smartly allocating low-cost many-bit ecc for mitigating read and write errors in stt-ram caches”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 6, pp. 1357–1370, 2020. DOI: 10.1109/TVLSI.2020.2977131.

[198] N. Sayed, R. Bishnoi, and M. B. Tahoori, “Fast and reliable stt-mram using nonuniform and adaptive error detecting and correcting scheme”, *IEEE Transactions on VLSI Systems*, vol. 27, no. 6, pp. 1329–1342, 2019. DOI: 10.1109/TVLSI.2019.2903592.

[199] N. Sayed, F. Oboril, R. Bishnoi, and M. B. Tahoori, “Leveraging systematic unidirectional error-detecting codes for fast stt-mram cache”, in *2017 IEEE 35th VLSI Test Symposium (VTS)*, 2017, pp. 1–6. DOI: 10.1109/VTS.2017.7928937.

[200] R. Bishnoi, M. Ebrahimi, F. Oboril, and M. B. Tahoori, “Improving write performance for stt-mram”, *IEEE Transactions on Magnetics*, vol. 52, no. 8, pp. 1–11, 2016. DOI: 10.1109/TMAG.2016.2541629.

[201] B. Wu, P. Dai, Z. Wang, C. Wang, Y. Wang, J. Yang, Y. Cheng, D. Liu, Y. Zhang, W. Zhao, and X. S. Hu, “Bulkyflip: A nand-spin-based last-level cache with bandwidth-oriented write management policy”, *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 1, pp. 108–120, 2020. DOI: 10.1109/TCSI.2019.2947242.

[202] M. Talebi, A. Salahvarzi, A. M. Hosseini Monazzah, K. Skadron, and M. Fazeli, “Rocky: A robust hybrid on-chip memory kit for the processors with stt-mram cache technology”, *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2198–2210, 2021. DOI: 10.1109/TC.2020.3040152.

[203] J. Kim, M. Sullivan, E. Choukse, and M. Erez, “Bit-plane compression: Transforming data for better compression in many-core architectures”, in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 329–340. DOI: 10.1109/ISCA.2016.37.

[204] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “Base-delta-immediate compression: Practical data compression for on-chip caches”, in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’12, Minneapolis, Minnesota, USA: Association for Computing Machinery, 2012, pp. 377–388, ISBN: 9781450311823. DOI: 10.1145/2370816.2370870. [Online]. Available: https://doi.org/10.1145/2370816.2370870.

[205] W. Kang, L. Zhang, W. Zhao, J.-O. Klein, Y. Zhang, D. Ravelosona, and C. Chappert, “Yield and reliability improvement techniques for emerging nonvolatile stt-mram”, *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 5, no. 1, pp. 28–39, 2015. DOI: 10.1109/JETCAS.2014.2374291.

[206] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, “Use ecp, not ecc, for hard failures in resistive memories”, in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA ’10, Saint-Malo, France: Association for Computing Machinery, 2010, pp. 141–152, ISBN: 9781450300537. DOI: 10.1145/1815961.1815980.

[207] S. Swami, P. M. Palangappa, and K. Mohanram, “Ecs: Error-correcting strings for lifetime improvements in nonvolatile memories”, *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, Dec. 2017, ISSN: 1544-3566. DOI: 10.1145/3151083.

[208] N. H. Seong, D. H. Woo, V. Srinivasan, J. A. Rivers, and H.-H. S. Lee, “Safer: Stuck-at-fault error recovery for memories”, in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’43, USA: IEEE Computer Society, 2010, pp. 115–124, ISBN: 9780769542997. DOI: 10.1109/MICRO.2010.46.

[209] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez, “Free-p: Protecting non-volatile memory against both hard and soft errors”, in *2011 IEEE 17th International Symposium on HPCA*, 2011, pp. 466–477. DOI: 10.1109/HPCA.2011.5749752.

[210] A. Das and N. A. Touba, “Online correction of hard errors and soft errors via one-step decodable ols codes for emerging last level caches”, in *2019 IEEE Latin American Test Symposium (LATS)*, 2019, pp. 1–6. DOI: 10.1109/LATW.2019.8704568.

[211] M. K. Qureshi, “Pay-as-you-go: Low-overhead hard-error correction for phase change memories”, in *2011 44th Annual IEEE/ACM MICRO*, 2011, pp. 318–328.

[212] J. Deng, Y. Fang, Z. Du, Y. Wang, H. Li, O. Temam, P. Ienne, D. Novo, X. Li, Y. Chen, and C. Wu, “Retraining-based timing error mitigation for hardware neural networks”, in *2015 Design Automation and Test in Europe Conference and Exhibition (DATE)*, 2015, pp. 593–596.

[213] J.-Y. Hu, K.-W. Hou, C.-Y. Lo, Y.-F. Chou, and C.-W. Wu, “Rram-based neuromorphic hardware reliability improvement by self-healing and error correction”, in *2018 IEEE International Test Conference in Asia (ITC-Asia)*, 2018, pp. 19–24. DOI: 10.1109/ITC-Asia.2018.00014.

[214] J. J. Zhang, T. Gu, K. Basu, and S. Garg, “Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator”, in *2018 IEEE 36th VLSI Test Symposium (VTS)*, 2018, pp. 1–6. DOI: 10.1109/VTS.2018.8368656.

[215] C. Schorn, A. Guntoro, and G. Ascheid, “An efficient bit-flip resilience optimization method for deep neural networks”, in *2019 Design Automation and Test in Europe Conference and Exhibition (DATE)*, 2019, pp. 1507–1512. DOI: 10.23919/DATE.2019.8714885.

[216] M. Sadi and U. Guin, “Test and yield loss reduction of ai and deep learning accelerators”, *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 41, no. 1, pp. 104–115, Jan. 2022, ISSN: 0278-0070. DOI: 10.1109/TCAD.2021.3051841. [Online]. Available: https://doi.org/10.1109/TCAD.2021.3051841.

[217] J. J. Zhang, K. Basu, and S. Garg, “Fault-tolerant systolic array based accelerators for deep neural network execution”, *IEEE Design and Test*, vol. 36, no. 5, pp. 44–53, 2019. DOI: 10.1109/MDAT.2019.2915656.

[218] L.-H. Tsai, S.-C. Chang, Y.-T. Chen, J.-Y. Pan, W. Wei, and D.-C. Juan, “Robust processing-in-memory neural networks via noise-aware normalization”, *arXiv preprint arXiv:2007.03230*, 2020.

[219] S. T. Ahmed, M. Hefenbrock, C. Münch, and M. B. Tahoori, “Neuroscrub+: Mitigating retention faults using flexible approximate scrubbing in neuromorphic fabric based on resistive memories”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[220] U. Zahid, G. Gambardella, N. J. Fraser, M. Blott, and K. Vissers, “Fat: Training neural networks for reliable inference under hardware faults”, in *2020 IEEE International Test Conference (ITC)*, 2020, pp. 1–10. DOI: 10.1109/ITC44778.2020.9325249.

[221] M. Qin, C. Sun, and D. Vucinic, “Robustness of neural networks against storage media errors”, *arXiv preprint arXiv:1709.06173*, 2017.

[222] S. Burel, A. Evans, and L. Anghel, “Zero-overhead protection for cnn weights”, in *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2021, pp. 1–6. DOI: 10.1109/DFT52944.2021.9568363.

[223] A. Lotfi, S. Hukerikar, K. Balasubramanian, P. Racunas, N. Saxena, R. Bramley, and Y. Huang, “Resiliency of automotive object detection networks on gpu architectures”, in *2019 IEEE International Test Conference (ITC)*, 2019, pp. 1–9. DOI: 10.1109/ITC44170.2019.9000150.

[224] B. Salami, O. S. Unsal, and A. C. Kestelman, “Evaluating built-in ecc of fpga on-chip memories for the mitigation of undervolting faults”, in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2019, pp. 242–246. DOI: 10.1109/EMPDP.2019.8671543.

[225] K. Huang, P. H. Siegel, and A. A. Jiang, “Functional error correction for reliable neural networks”, in *2020 IEEE International Symposium on Information Theory (ISIT)*, 2020, pp. 2694–2699. DOI: 10.1109/ISIT44484.2020.9174137.

[226] M. Sabih, F. Hannig, and J. Teich, “Fault-tolerant low-precision dnns using explainable ai”, in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2021, pp. 166–174. DOI: 10.1109/DSN-W52860.2021.00036.

[227] H. Guan, L. Ning, Z. Lin, X. Shen, H. Zhou, and S.-H. Lim, “In-place zero-space memory protection for cnn”, *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[228] S.-S. Lee and J.-S. Yang, “Value-aware parity insertion ecc for fault-tolerant deep neural network”, in *2022 DATE*, IEEE, 2022, pp. 724–729.

[229] E. Ozen and A. Orailoglu, “Sanity-check: Boosting the reliability of safety-critical deep neural network applications”, in *2019 IEEE 28th Asian Test Symposium (ATS)*, 2019, pp. 7–75. DOI: 10.1109/ATS47505.2019.000-8.

[230] K. Zhao, S. Di, S. Li, X. Liang, Y. Zhai, J. Chen, K. Ouyang, F. Cappello, and Z. Chen, “Ft-cnn: Algorithm-based fault tolerance for convolutional neural networks”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1677–1689, 2021. DOI: 10.1109/TPDS.2020.3043449.

[231] F. F. d. Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, “Analyzing and increasing the reliability of convolutional neural networks on gpus”, *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, 2019. DOI: 10.1109/TR.2018.2878387.

[232] Q. Lou, T. Gao, P. Faley, M. Niemier, X. S. Hu, and S. Joshi, “Embedding error correction into crossbars for reliable matrix vector multiplication using emerging devices”, in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED ’20, Boston, Massachusetts: Association for Computing Machinery, 2020, pp. 139–144, ISBN: 9781450370530. DOI: 10.1145/3370748.3406583.

[233] W. Li, J. Read, H. Jiang, and S. Yu, “Mac-ecc: In-situ error correction and its design methodology for reliable nvm-based compute-in-memory inference engine”, *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 12, no. 4, pp. 835–845, 2022. DOI: 10.1109/JETCAS.2022.3223031.

[234] B. Crafton, S. Spetalnick, J.-H. Yoon, W. Wu, C. Tokunaga, V. De, and A. Raychowdhury, “Cim-secded: A 40nm 64kb compute in-memory rram macro with ecc enabling reliable operation”, in *2021 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, 2021, pp. 1–3. DOI: 10.1109/A-SSCC53895.2021.9634742.

[235] B. Feinberg, S. Wang, and E. Ipek, “Making memristive neural network accelerators reliable”, in *2018 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2018, pp. 52–65. DOI: 10.1109/HPCA.2018.00015.

[236] C. Li, R. M. Roth, C. Graves, X. Sheng, and J. P. Strachan, “Analog error correcting codes for defect tolerant matrix multiplication in crossbars”, in *2020 IEEE International Electron Devices Meeting (IEDM)*, 2020, pp. 36.6.1–36.6.4. DOI: 10.1109/IEDM13553.2020.9371978.

[237] M. Liu, L. Xia, Y. Wang, and K. Chakrabarty, “Fault tolerance for rram-based matrix operations”, in *2018 IEEE International Test Conference (ITC)*, 2018, pp. 1–10. DOI: 10.1109/TEST.2018.8624687.

[238] A. Das and N. A. Touba, “Selective checksum based on-line error correction for rram based matrix operations”, in *2020 IEEE 38th VLSI Test Symposium (VTS)*, 2020, pp. 1–6. DOI: 10.1109/VTS48691.2020.9107606.

[239] M. Davey and D. MacKay, “Low density parity check codes over gf(q)”, in *1998 Information Theory Workshop (Cat. No.98EX131)*, 1998, pp. 70–71. DOI: 10.1109/ITW.1998.706440.

[240] R. M. Roth, “Fault-tolerant dot-product engines”, *IEEE Transactions on Information Theory*, vol. 65, no. 4, pp. 2046–2057, 2019. DOI: 10.1109/TIT.2018.2869794.

[241] Y. Hu, K. Cheng, Z. Zhang, R. Wang, Y. Wang, and R. Huang, "Error correction scheme for reliable rram-based in-memory computing", in *2021 5th IEEE Electron Devices Technology Manufacturing Conference (EDTM)*, 2021, pp. 1–3. DOI: 10.1109/EDTM50988.2021.9420944.

[242] C.-T. Huang, C.-F. Wu, J.-F. Li, and C.-W. Wu, "Built-in redundancy analysis for memory yield improvement", *IEEE Transactions on Reliability*, vol. 52, no. 4, pp. 386–399, 2003. DOI: 10.1109/TR.2003.821925.

[243] S. M. Nair, R. Bishnoi, and M. B. Tahoori, "A comprehensive framework for parametric failure modeling and yield analysis of stt-mram", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 7, pp. 1697–1710, 2019. DOI: 10.1109/TVLSI.2019.2904197.

[244] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark", in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18, Berlin, Germany: Association for Computing Machinery, 2018, pp. 41–42, ISBN: 9781450356299. DOI: 10.1145/3185768.3185771.

[245] X. Bi, Z. Sun, H. Li, and W. Wu, "Probabilistic design methodology to improve run-time stability and performance of stt-ram caches", in *2012 IEEE/ACM ICCAD*, 2012, pp. 88–94.

[246] Y. Zhang, X. Wang, Y. Li, A. Jones, and Y. Chen, "Asymmetry of mtj switching and its implication to stt-ram designs", in *2012 DATE*, 2012, pp. 1313–1318. DOI: 10.1109/DATE.2012.6176695.

[247] Y. Zhang, W. Zhao, G. Prenat, T. Devolder, J.-O. Klein, C. Chappert, B. Dieny, and D. Ravelosona, "Electrical modeling of stochastic spin transfer torque writing in magnetic tunnel junctions for memory and logic applications", *IEEE Transactions on Magnetics*, vol. 49, no. 7, pp. 4375–4378, 2013. DOI: 10.1109/TMAG.2013.2242257.

[248] H. Helgert and R. Stinaff, "Shortened bch codes (corresp.)", *IEEE Transactions on Information Theory*, vol. 19, no. 6, pp. 818–820, 1973. DOI: 10.1109/TIT.1973.1055099.

[249] Y. Liu, C. Ding, and C. Tang, "Shortened linear codes over finite fields", *IEEE Transactions on Information Theory*, vol. PP, pp. 1–1, Jun. 2021. DOI: 10.1109/TIT.2021.3087082.

[250] S. Song, Q. Wu, S. Flolid, J. Dean, R. Panda, J. Deng, and L. K. John, "Experiments with spec cpu 2017: Similarity, balance, phase behavior and simpoints", Laboratory for Computer Architecture, Department of Electrical and Computer Engineering, The University of Texas at Austin, Tech. Rep., 2018.

[251] S. Singh and M. Awasthi, "Memory centric characterization and analysis of spec cpu2017 suite", in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19, Mumbai, India: Association for Computing Machinery, 2019, pp. 285–292, ISBN: 9781450362399. DOI: 10.1145/3297663.3310311.

[252] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator", *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011, ISSN: 0163-5964. DOI: 10.1145/2024716.2024718.

[253] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior", *SIGPLAN Not.*, vol. 37, no. 10, pp. 45–57, Oct. 2002, ISSN: 0362-1340. DOI: 10.1145/605432.605403.

[254] L. Zhang, T. Li, and T. Endoh, "Small area and high throughput error correction module of stt-mram for object recognition systems", *IEEE Transactions on Industrial Informatics*, vol. 20, no. 5, pp. 7777–7786, 2024. DOI: 10.1109/TII.2024.3362373.

[255] W. Wen, Y. Zhang, Y. Chen, Y. Wang, and Y. Xie, "Ps3-ram: A fast portable and scalable statistical stt-ram reliability/energy analysis method", *IEEE TCAD*, vol. 33, no. 11, pp. 1644–1656, 2014. DOI: 10.1109/TCAD.2014.2351581.

[256] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory", *IEEE TCAD*, vol. 31, no. 7, pp. 994–1007, 2012. DOI: 10.1109/TCAD.2012.2185930.

[257] M. McCartney, "Sram reliability improvement using ecc and circuit techniques", *Ph.D. dissertation, Dept. Elect. Comput. Eng., Carnegie Mellon Univ., Pittsburgh, PA, USA, 2014.*, 2018.

[258] R. Brain, "Interconnect scaling: Challenges and opportunities", in *2016 IEEE International Electron Devices Meeting*, 2016, pp. 9.3.1–9.3.4. DOI: 10.1109/IEDM.2016.7838381.

[259] G. Bonilla, N. Lanzillo, C.-K. Hu, C. Penny, and A. Kumar, "Interconnect scaling challenges, and opportunities to enable system-level performance beyond 30 nm pitch", in *2020 IEEE IEDM*, 2020, pp. 20.4.1–20.4.4. DOI: 10.1109/IEDM13553.2020.9372093.

[260] K. Park and H. Simka, "Advanced interconnect challenges beyond 5nm and possible solutions", in *2021 IEEE International Interconnect Technology Conference (IITC)*, 2021, pp. 1–3. DOI: 10.1109/IITC51362.2021.9537552.

[261] V. B. Naik, J. H. Lim, K. Yamane, J. Kwon, B.-A. B., N. L. Chung, S. K, L. Y. Hau, R. Chao, C. Chiang, Y. Huang, L. Pu, Y. Otani, S. Jang, N. Balasankaran, W. P. Neo, T. Ling, J. W. Ting, H. Yoon, J. Mueller, B. Pfefferling, O. Kallensee, T. Merbeth, C. Seet, J. Wong, Y. S. You, S. Soss, T. H. Chan, and S. Y. Siah, "Extended mtj tddb model, and improved stt-mram reliability with reduced circuit and process variabilities", in *2022 IEEE International Reliability Physics Symposium (IRPS)*, 2022, 6B.3-1-6B.3–6. DOI: 10.1109/IRPS48227.2022.9764563.

[262] Y.-C. Shih, C.-F. Lee, Y.-A. Chang, P.-H. Lee, H.-J. Lin, Y.-L. Chen, K.-F. Lin, T.-C. Yeh, H.-C. Yu, H. Chuang, Y.-D. Chih, and J. Chang, "Logic process compatible 40nm 16mb, embedded perpendicular-mram with hybrid-resistance reference, sub-μ a sensing resolution, and 17.5ns read access time", in *2018 IEEE Symposium on VLSI Circuits*, 2018, pp. 79–80. DOI: 10.1109/VLSIC.2018.8502260.

[263] Y. Wang, H. Cai, L. A. d. B. Naviner, Y. Zhang, X. Zhao, E. Deng, J.-O. Klein, and W. Zhao, "Compact model of dielectric breakdown in spin-transfer torque magnetic tunnel junction", *IEEE Transactions on Electron Devices*, vol. 63, no. 4, pp. 1762–1767, 2016. DOI: 10.1109/TED.2016.2533438.

[264] W. Gallagher, E. Chien, T.-W. Chiang, J.-C. Huang, M.-C. Shih, C. Wang, C.-H. Weng, S. Chen, C. Bair, G. Lee, Y.-C. Shih, C.-F. Lee, P.-H. Lee, R. Wang, K. H. Shen, J. J. Wu, W. Wang, and H. Chuang, "22nm stt-mram for reflow and automotive uses with high yield, reliability, and magnetic immunity and with performance and shielding options", in *2019 IEEE International Electron Devices Meeting (IEDM)*, 2019, pp. 2.7.1–2.7.4. DOI: 10.1109/IEDM19573.2019.8993469.

[265] K. Rho, K. Tsuchida, D. Kim, Y. Shirai, J. Bae, T. Inaba, H. Noro, H. Moon, S. Chung, K. Sunouchi, J. Park, K. Park, A. Yamamoto, S. Chung, H. Kim, H. Oyamatsu, and J. Oh, "23.5 a 4gb lpddr2 stt-mram with compact 9f2 1t1mtj cell and hierarchical bitline architecture", in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 396–397. DOI: 10.1109/ISSCC.2017.7870428.

[266] T. Y. Lee, K. Yamane, Y. Otani, D. Zeng, J. Kwon, J. H. Lim, V. B. Naik, L. Y. Hau, R. Chao, N. L. Chung, T. Ling, S. H. Jang, L. C. Goh, J. Hwang, L. Zhang, R. Low, N. Balasankaran, F. Tan, J. W. Ting, J. Chang, C. S. Seet, S. Ong, Y. S. You, S. T. Woo, T. H. Chan, and S. Y. Siah, "Advanced mtj stack engineering of stt-mram to realize high speed applications", in *2020 IEEE IEDM*, 2020, pp. 11.6.1–11.6.4. DOI: 10.1109/IEDM13553.2020.9372015.

[267] Y.-C. Shih, C.-F. Lee, Y.-A. Chang, P.-H. Lee, H.-J. Lin, Y.-L. Chen, C.-P. Lo, K.-F. Lin, T.-W. Chiang, Y.-J. Lee, K.-H. Shen, R. Wang, W. Wang, H. Chuang, E. Wang, Y.-D. Chih, and J. Chang, "A reflow-capable, embedded 8mb stt-mram macro with 9ns read access time in 16nm finfet logic cmos process", in *2020 IEEE IEDM*, 2020, pp. 11.4.1–11.4.4. DOI: 10.1109/IEDM13553.2020.9372115.

[268] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[269] Z. Gao, Q. Jing, Y. Li, P. Reviriego, and J. A. Maestro, "An efficient fault-tolerance design for integer parallel matrix–vector multiplications", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 1, pp. 211–215, 2018. DOI: 10.1109/TVLSI.2017.2755765.

[270] J.-Y. Jou and J. Abraham, "Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures", *Proceedings of the IEEE*, vol. 74, no. 5, pp. 732–741, 1986. DOI: 10.1109/PROC.1986.13535.

[271] M. A. Zidan and W. D. Lu, "Chapter 9 - vector multiplications using memristive devices and applications thereof", in *Memristive Devices for Brain-Inspired Computing*, ser. Woodhead Publishing Series in Electronic and Optical Materials, S. Spiga, A. Sebastian, D. Querlioz, and B. Rajendran, Eds., Woodhead Publishing, 2020, pp. 221–254, ISBN: 978-0-08-102782-0. DOI: https://doi.org/10.1016/B978-0-08-102782-0.00009-5.

[272] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", in *International conference on machine learning*, pmlr, 2015, pp. 448–456.

[273] M. I. Ahmed, S. Mahmud Mamun, and A. U. Zaman Asif, "Dcnn-based vegetable image classification using transfer learning: A comparative study", in *2021 5th International Conference on Computer, Communication and Signal Processing (ICCCSP)*, 2021, pp. 235–243. DOI: 10.1109/ICCCSP52374.2021.9465499.

[274] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks", *arXiv preprint arXiv:1805.06085*, 2018.

[275] M. Y. Hsiao, D. C. Bossen, and R. T. Chien, "Orthogonal latin square codes", *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 390–394, 1970. DOI: 10.1147/rd.144.0390.

[276] A. Sánchez-Macián, F. Garcia-Herrero, and J. Maestro, "Reliability of 3d memories using orthogonal latin square codes", *Microelectronics Reliability*, vol. 95, pp. 74–80, 2019, ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2019.03.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026271418310230.

[277] K. Namba and F. Lombardi, "Non-binary orthogonal latin square codes for a multilevel phase charge memory (pcm)", *IEEE Transactions on Computers*, vol. 64, no. 7, pp. 2092–2097, 2015. DOI: 10.1109/TC.2014.2346182.

[278] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation", *arXiv preprint arXiv:1308.3432*, 2013.

[279] http://www.ecs.umass.edu/ece/koren/FaultTolerantSystems/simulator/Hamming/HammingCodes.html, 2025.

[280] http://www.mathaddict.net/hamming.htm, 2025.

List of Figures

List of Tables

Acronyms

ABFT Algorithm-based Fault Tolerance.

ADC Analog-to-Digital.

AI Artificial Intelligence.

AMS analog and mixed-signal.

ASIC Application Specific Integrated Circuit.

BCH Bode-Chaudhuri-Hocquenghem.

BECP Block Error Correction Pointer.

BER Block Error Rate.

BL Bit Line.

CiM Computation-in-Memory.

CMOS Complementary Metal-Oxide Semiconductor.

CNN Convolutional Neural Network.

CP Correction Pointer.

DAC Digital-to-Analog.

DEC Double Error Correction.

DEC-TED Double Error Correction-Tripple Error Detection.

DNN Deep Neural Network.

DRAM Dynamic Random Access Memory.

ECC Error-Correcting Code.

ECP Error Correction Pointer.

ECS Error Correction String.

FPGA Field Programmable Gate Array.

GPU Graphic Processing Unit.

HDC HyperDimensional Computing.

HRS High Resistance State.

IPC Instructions Per Cycle.

LDPC Low-Density Parity Check.

LLC Last Level Cache.

LRS Low Resistance State.

LSB Least Significant Bit.

MAC Multiply and Accumulate.

MBECP Multi-Block Error Correction Pointer.

MC Monte Carlo.

MSB Most Significant Bit.

MTJ Magnetic Tunnel Junction.

MTTF Mean Time To Failure.

MVM Matrix-Vector Multiplication.

NN Neural Network.

NVM Non-Volatile Memory.

OLS Orthogonal Latin Square.

OS-MLD One-Step Majority-Logic Decodable.

PCM Phase-Change Material.

PCRAM Phase Change Random Access Memory.

PE Processing Element.

RR Redundancy Repair.

RRAM Resistive Random Access Memory.

RT Retention Time.

RTN Random Telegraph Noise.

SEC Single Error Correction.

SEC-DED Single Error Correction-Double Error Detection.

SIMD Single Instruction Multiple Data.

SL Source Line.

SoC System-on-Chip.

SRAM Static Random Access Memory.

STT-MRAM Spin Transfer Torque Magnetic Random Access Memory.

TMR Tunneling Magnetoresistance.

TPU Tensor Processing Unit.

VLSI Very Large Scale Integration.

WL Word Line.