PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

MARCO POKORSKI STEFANI

DYNAMIC FAULT TOLERANT MECHANISM FOR MEMORY CONTROLLERS

Porto Alegre
2023

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

DYNAMIC FAULT TOLERANT MECHANISM FOR MEMORY CONTROLLERS

MARCO POKORSKI STEFANI

Doctoral Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science.

Advisor: Prof. Dr. César Marcon
Co-Advisor: Prof. Dr. Jarbas Silveira

Porto Alegre, 2023

Ficha Catalográfica

P761d Pokorski Stefani, Marco

Dynamic Fault Tolerant Mechanism for Memory Controllers / Marco Pokorski Stefani. – 2023.
148.
Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. César Augusto Missio Marcon.
Coorientador: Prof. Dr. Jarbas Silveira.

1. Memory Controller. 2. Fault Tolerance. 3. Dynamic Error Correction Code. I. Missio Marcon, César Augusto. II. Silveira, Jarbas. III. , . IV. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

MARCO POKORSKI STEFANI

DYNAMIC FAULT TOLERANT MECHANISM FOR MEMORY CONTROLLERS

This Doctoral Thesis has been submitted in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul.

Sanctioned on August 24th, 2023.

COMMITTEE MEMBERS:

Prof. Dr. Avelino Francisco Zorzo (PPGCC/PUCRS)

Prof. Dr. Eduardo Augusto Bezerra (PPGEEL/UFSC)

Prof Dr. Márcio Eduardo Kreutz (PPgSC/UFRN)

Prof. Dr. Jarbas Silveira (PPGETI/UFC - Co-advisor)

Prof. Dr. César Marcon (PPGCC/PUCRS - Advisor)

“I thought I knew a lot of, but in the end, it was just one small piece from the puzzle.”
(Deyth Banger)

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Prof. Dr. César Marcon for allowing me to pursue my Ph.D. at PUCRS. His unwavering commitment to academic excellence, ethical values, and selfless guidance has been a constant source of inspiration and motivation throughout my doctoral journey.

I would also like to acknowledge my family for their support and encouragement. Their selfless sacrifices and endless love have been the driving force behind my success. I am grateful for the countless weekends, holidays, and evenings they devoted to helping me pursue my academic goals.

Lastly, I would like to thank God for granting me the strength, perseverance, and courage to achieve this personal milestone. Without His grace and blessings, none of this would have been possible.

MECANISMO DINÂMICO DE TOLERÂNCIA A FALHAS PARA CONTROLADORAS DE MEMÓRIA

RESUMO

Erros de memória podem causar falhas, vulnerabilidades de segurança, corrupção e perda de dados que são inaceitáveis para servidores. Esses problemas impulsionam a construção de um projeto de arquitetura de memória robusta. As controladoras de memória podem atenuar esses erros empregando um Código de Correção de Erro (ECC) nos fluxos de gravação e leitura de dados. Fatores ambientais e tecnológicos implicam diferentes probabilidades de erro, impedindo definir em tempo de projeto qual ECC é mais eficaz e eficiente a ser utilizado. Este trabalho propõe um mecanismo tolerante a falhas atuando como gerenciador de codificação e decodificação da controladora de memória. Este mecanismo define dinamicamente o ECC para cada bloco de memória, seguindo como critério a taxa de erro capturada em tempo de execução e a eficácia dos ECCs implementados na controladora. Blocos de memória com uma alta taxa de erro podem ser recodificados para um ECC de alta eficácia e vice-versa. Resultados experimentais mostram que nossa proposta alcança elevada eficácia na correção de erros com alta eficiência energética. Adicionalmente, desenvolvemos a ferramenta Absimth para analisar a eficácia e eficiência da proposta que emprega mecanismos de gerenciamento dinâmico de tolerância a falhas. A ferramenta Absimth permite a modelagem e verificação de hardware/software em diversas granularidades, desde aplicativos armazenados na memória até o sistema operacional, incluindo processos de codificação e decodificação de diversos ECCs, habilitando comparar a eficácia e eficiência das soluções propostas em inúmeros cenários.

Palavras-chave: Tolerância a falhas, Memória confiável, Código de Correção de Erro (ECC), ECC dinâmico, Controlador de Memória.

DYNAMIC FAULT TOLERANT MECHANISM FOR MEMORY CONTROLLERS

ABSTRACT

Memory errors can cause failures, security vulnerabilities, corruption, and data loss, which are unacceptable for server systems. These problems push the construction of a robust computing memory architecture design. Memory controllers can mitigate these errors by employing an Error Correction Code (ECC) in the data write and read flows. Environmental and technological factors imply different error probabilities, preventing defining at design time which ECC is most effective and efficient to be used. This work proposes a fault-tolerant mechanism acting as a memory controller encoding and decoding manager. This mechanism dynamically defines the ECC for each memory block, following as criteria the error rate captured at runtime and the ECCs efficacy implemented in the controller. Memory blocks with a high error rate can be recoded to a high efficacy ECC and vice versa. Experimental results show that our proposal achieves high error correction efficacy with high energy efficiency. Additionally, we developed the Absimth tool to analyze the efficacy and efficiency of the proposal that employs dynamic fault tolerance management mechanisms. Absimth enables hardware/software modeling and verification in various granularity levels, from in-memory applications to the operating system, including encoding and decoding processes that employ ECCs, enabling comparing the efficacy and efficiency of the proposed solutions in uncountable scenarios.

Keywords: Fault Tolerance, Reliable Memory, Error Correcting Code (ECC), Dynamic ECC, Memory controller.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

1D	One-Dimensional
2D	Two-Dimensional
5EC6ED	5-Error Correction, 6-Error Detection
ADRAM	Asynchronous Dynamic Random-Access Memory
AES-DP	Adaptive ECC Scheme for Dynamic Protection
ARCC	Adaptive Reliability Chipkill Correct
ASCII	American Standard Code for Information Interchange
BCH	Bose–Chaudhuri–Hocquenghem
BIRA	Built-In Redundancy Analysis
BISR	Memory Built-In Self-Repair
BIST	Built-In Self-Test
CDRAM	Cache Dynamic Random-Access Memory
CP	Correction Prediction
CPT	Correction Prediction Table
CPU	Central Processing Unit
DDPC	Double Double-Pin Correcting
DDR SDRAM	Double-Data-Rate Synchronous Dynamic Random-Access Memory
DECTED	Double Error Correction - Triple Error Detection
DFTM	Dynamic Fault Tolerance Module
DFMC	Dynamic Fault-Tolerant Memory Controller
DIMM	Dual Inline Memory Module
DMC	Decimal Matrix Code
DMR	Double Memory Redundancy
DPC	Double Pin Correcting
DPSR	Double Parity Single Redundancy
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
ECC	Error Correction Code
ECC-P	ECC Parity
EDC	Error Detection Code
eDRAM	embedded DRAM
E-ECC	Erasure and Error Correction Codes
eMRSC	extended Matrix Region Selection Code
EPC	Extended Product Code

FIT	Failures In Time
GEC	Global Error Correction
GF	Galois Field
HPC	High-Performance Computing
HVDD	Horizontal-Vertical-Double-Bit Diagonal
JEDEC	Joint Electron Device Engineering Council
LAE-FTL	Lifetime Adaptive ECC in NAND Flash Page Management
LSB	Least Significant Bit
LED	Local Error Detection
LLC	Last Level Cache
LOT-ECC	LOcalized and Tiered ECC
LPC	Line Product Code
MBU	Multiple Bits Upset
MC	Mixed Code
MLC	Multi-Level Cell
ML-ECC	Multi-line ECC
MM-ECC	Memory Mapped ECC
MSB	Most Significant Bit
MRSC	Matrix Region Selection Code
MTTF	Mean Time To Failure
NAE	Number of Accumulated Errors
NER	Number of Errors to Reduce
OLS	Orthogonal Latin Square
OOB	Out-Of-Band
OPC	Octuple Pin Correcting
P/E	Program/Erase
PBA	Physical Block Address
PC	Product Code
PCoSA	Product Code for Space Applications
PERC	Punctured ECC Recovery Cache
PmC2	Parity-Based Mono-Copy Cache
PPA	Physical Page Address
QDPC	Quadruple Double Pin Correcting
QPC	Quadruple Pin Correcting
RAM	Random Access Memory
RDRAM	Rambus Dynamic Random-Access Memory
S2E	Straightforward 2D-ECC

SDRAM	Synchronous Dynamic Random-Access Memory
SECDED	Single Error Correction – Double Error Detection
SER	Soft Error Rate
SH	Static Hamming
SIMM	Single Inline Memory Module
SL	Static LPC
SLC	Single-Level Cell
SO-DIMM	Small Outline - Dual in-line Memory Module
SPC	Single Pin Correcting
SPC-TPD	Single Pin Correcting - Triple Pin Detecting
SPEC	Standard Performance Evaluation Corporation
SRAM	Static Random Access Memory
SSCDSD	Single Symbol Correction – Double Symbol Detect
SSD	Solid State Drive
STVQ	Single-Tier Virtual Queuing
R+W	Read and Write
RAPM	Reconfigurable ECC for Adaptive Protection of Memory
RBER	Raw Bit Error Rate
PE	Processing Element
RS	Reed Solomon
PARSEC	Princeton Application Repository for Shared-Memory Computers
T1EC	First Tier Error Code
T2EC	Second Tier Error Code
TAP	Test Access Port
TCS	Threshold Cycle Size
TMR	Triple Memory Redundancy
VC-ECC	Variable Capability ECC
VF-ECC	Virtualized and Flexible ECC
VL-ECC	Variable data-Length ECC
WE	Without ECC

CONTENTS

1 INTRODUCTION

High-capacity and scalable memories[1] are crucial in computer systems [1]. Both industry and university have been investigating technologies to increase memory density, thus, decreasing area consumption and raising the number of cells on each chip for increasing storage capacity and operating frequency [2], as evidenced in Figure 1.

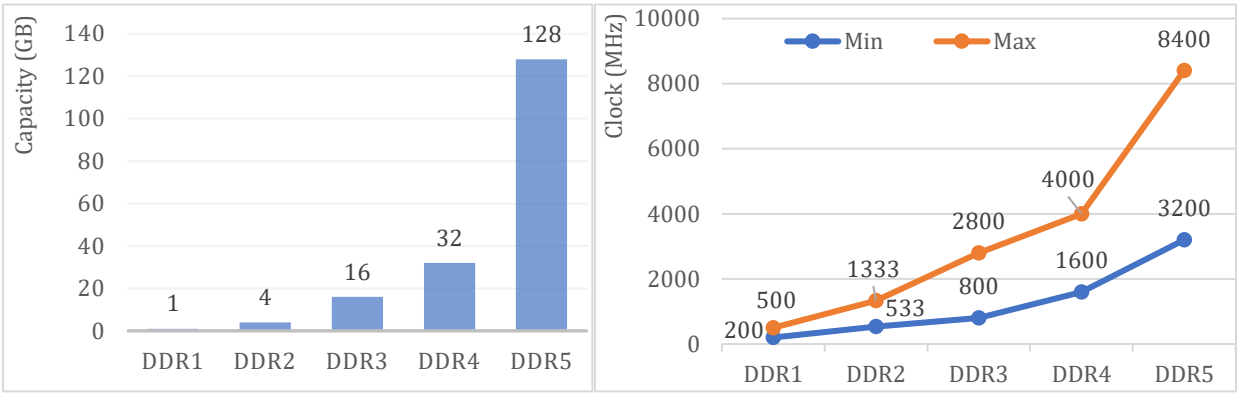


Figure 1. Memory capacity Evolution by GB module and clock in MHz [Author].

Reliable hardware is essential for a wide range of projects, including High-Performance Computing (HPC) and storage projects [3]. Memory is prone to failure due to magnetic fields, radiation, and natural stress from physical components; the reliability of memories is particularly sensitive to manufacturing process variations, environmental conditions, or wear, being errors commonly found in servers [4][5]. Google noted 70000 *Failures In Time/Mb* (FIT/Mb)[2] in storage systems, whose 8% of memory modules are affected per year of operation [3]. At the same time, Facebook reported monthly memory errors on 2.5% of its servers [6]. Additionally, advances in memory research can also help improve energy efficiency while providing some resilient degree [7].

The main reliability challenge of memory systems is mitigating performance issues and energy losses while guaranteeing the correct data written. Adding an Error Correction Code (ECC), e.g., Hamming [8] and Chipkill [9], enables mitigating problems caused by temporary errors. Reliability degrades over time, requiring data re-reading and rewriting through scrubbing techniques before the number of errors exceeds the ECC efficacy [10]. Memory controllers focus on data throughput as their primary role; consequently, they depend exclusively on error correction techniques by encoding/decoding circuits, leaving other complex fault-tolerant methods to the Operating System (OS).

[1] Scalable memories refer to computer memory systems that can easily adapt, growing or shrinking the capacity as needed to meet changing computational demands without requiring a complete overhaul of the system [128].
[2] FIT - Number of failures expected at one billion per device-hour of operation. For example, 1000 devices for 1 million hours. The FIT to MTBF ratio can be expressed as MTBF = 1,000,000,000 x 1 / FIT.

The memory controller manages the encoding and decoding circuits, defining ECCs for protecting data from the memory modules based on the error rate of memory blocks. Commercial servers use a static fault-tolerance approach, which maintains the encoding of memory blocks throughout the system operation. While each memory controller can have a different ECC, the usual procedure for commercial servers is to have all memory modules with the same ECC defined according to operating requirements.

This work improves the design of high-tech memory controllers delivering higher reliability to servers. We proposed the Dynamic Fault-Tolerant Memory Controller (DFMC) that enables each memory block to have specific encoding, dynamically adjusting the ECC of each block according to the error rate and system requirements. DFMC enables executing applications with controllable error correction rates and mitigating operational costs such as energy consumption and latency. Although the proposed approach applies to any memory technology, this work focuses on systems compatible with standard hardware that uses Double-Data-Rate Synchronous Dynamic Random-Access Memory (DDR SDRAM).

1.1 Motivation

Businesses increasingly depend on big data[3] server resources, reinforcing the need to avoid data loss, and the memory component is crucial for mitigating temporarily stored data losses.

Memory errors can happen due to physical factors such as voltage difference, impact, temperature variation, radiation, magnetic variations, electrical fluctuations, electromagnetic interference from the computer itself, quality failure, or stress over time [3]. Errors caused by logical factors can occur when writing data differently than initially intended. Since bits retain their programmed value as an electrical charge, there are many potential causes for errors. Sorin [11] classifies these errors as (i) *transient* if it occurs once and does not persist - temporarily. This error can occur due to electromagnetic emissions coming from the computer itself, electrical fluctuation, natural magnetic variations, or radiation; and (ii) *permanent* if it cannot be corrected and can occur due to various aspects such as temperature fluctuations, energy overload stress, hardware impact, poor handling, manufacturing defects, and physical degradation of the component.

Govindavajhala and Appel [12] exemplify that memory errors can be used to attack and gain access to virtual machines. Furthermore, Memory errors can cause crashes, transcription errors, corrupted or lost data, and security vulnerabilities. Any data loss or

[3] *Big Data* is the area of knowledge that studies how to study, analyze, and obtain information from data sets that are too large to apply by traditional systems.

transcription error is unacceptable on servers because they contain sensitive information, such as medical or financial data. Moreover, the error incidence increases with the memory size and the system usage time, requiring fault tolerance techniques that reduce errors to a manageable number [3].

ECC use increases reliability; on the other hand, an ECC circuit spends more energy on encoding and decoding, requiring more storage area, implying higher latency and more memory access for reading and writing [13].

Hwang, Stefanovici, and Schroeder [14] describe that errors are not evenly distributed in memory; some parts of the memory chip and physical addresses experience a higher error rate than others. Their description suggests that it is crucial to have a dynamic approach for assessing the memory state and measuring the coding quality. In other words, at runtime, the controller can select an ECC suitable for the error rate of each memory block and, therefore, guarantee the fault tolerance required by the application, meet reliability, and provide efficiency by saving energy and maintaining performance.

1.2 Problem Statement and Thesis Contributions

Memory is prone to failure by factors such as magnetic fields, radiation, and natural stress from physical components [4][5]; therefore, commercial memories like DDR SDRAM implement ECC-based mechanisms to provide fault tolerance. These mechanisms are configurable according to the architecture requirements where the memory is used; this configuration range goes from the absence to using ECCs with high correction power. Figure 2 illustrates the memory organization, focusing on fault tolerance aspects.

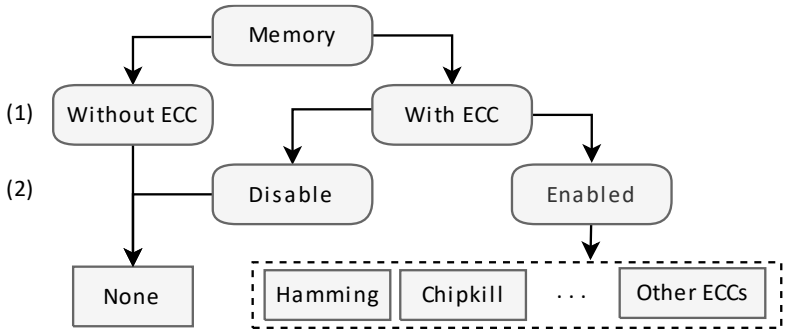


Figure 2. Memory organization example concerning fault tolerance aspects [Author].

Level (1) differentiates between memories with and without ECC mechanisms. Memories implemented with ECC contain an extra chip, delivering more bits per word than memory without ECC; e.g., a 64-bit memory without ECC has eight chips, with ECC has an additional chip, totaling nine chips delivering 72 bits.

Level (2) shows that memories encompassing ECCs are programmed to enable or

disable this fault-tolerant mechanism. When ECC is disabled, the memory operates similarly to non-ECC memory, reducing latency and energy consumption with the penalty of not correcting or detecting errors. Enabling the fault tolerance operation makes the memory reach a certain degree of error correction efficacy and operation cost; e.g., Hamming provides less efficacy and greater efficiency than Chipkill. Commercial memories provide a single and static ECC configuration; i.e., only one ECC can be chosen, and once the ECC is selected, it must remain until the system restarts. Therefore, the tradeoffs between efficacy and efficiency cannot be changed during system operation.

This work proposes a dynamic fault tolerance mechanism that analyzes the state of the memory block to define the ECC configuration at runtime, thus, mitigating data reliability and considering requirements for energy saving and runtime optimization. Figure 3 shows the fault tolerance memory organization proposed with the main elements to carry out a dynamic approach. Memory commercial approaches are single-ECC and static, while our proposal is multi-ECC and dynamic; i.e., each part of the memory can have one type of ECC, and these ECCs can be changed at runtime, as described in levels (3) and (4). Both single- and multi-ECC approaches can enable or disable the fault tolerance mechanisms. The approaches differ mainly in two aspects: (i) the multi-ECC fault-tolerance proposed here provides a finer memory granularity for ECC selection, and (ii) the dynamic approach can selectively change the ECC of a given memory module at runtime, according to pre-established requirements.

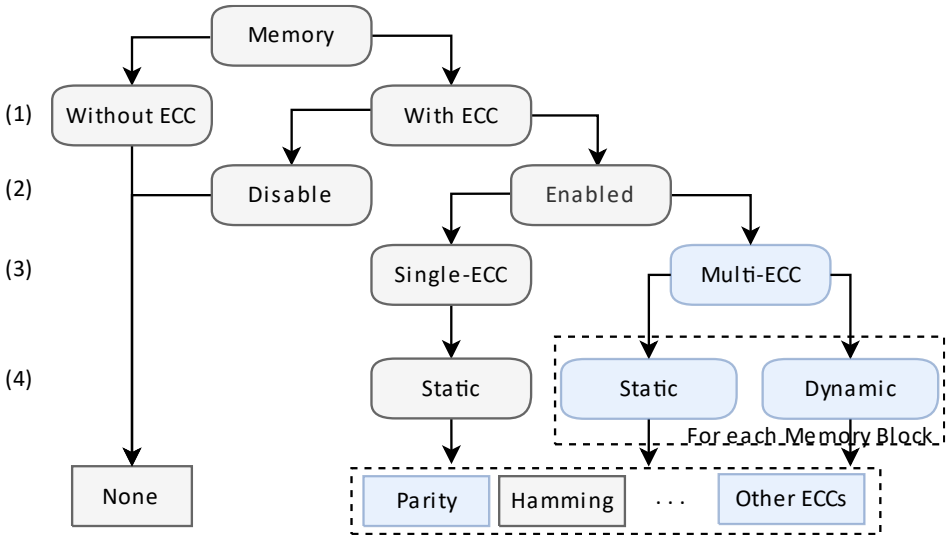


Figure 3. Fault-tolerant memory organization example encompassing the commercial (in gray) and proposed (in blue) approaches [Author].

Memory errors tend to happen at the same or close address [3][15]; however, this tendency is not explored by the ***Single-ECC*** approach employed on commercial memory controllers, as they use the same encoding procedure for the entire memory. Fault-tolerant

techniques that logically subdivide memory enable us to explore ECCs according to the error incidence in each memory subdivision, i.e., a ***Multi-ECC*** memory approach. Logical memory subdivisions can encompass ECCs with different error correction rates and operating costs. A low-susceptible to-error subdivision can use a low-efficacy ECC, e.g., Hamming, which consumes less energy and does not compromise the overall memory performance. Conversely, a high-susceptible to error subdivision can employ a high-efficacy ECC with higher operating costs, e.g., Chipkill.

We cite the reliability study of Google servers by Schroeder, Pinheiro, and Weber [3] to illustrate one of the advantages of our proposal. This study was performed over 2.5 years, covering multiple vendors, DRAM capacities, and technologies; it displays that 91.78% of memories did not present operating errors, and the total number of memories with corrected and uncorrected errors was greater than 8% and fewer than 0.22%, respectively. Additionally, among the 0.22% of memories that did not have an error corrected, 70% to 80% had an error corrected in a previous observation period, indicating that errors do not occur randomly; i.e., the error occurrence is strongly correlated - there is a high probability of subsequent errors occurring at the same or close location to previous errors. Although this reliability study [3] is based on memory technologies that are more than a decade old, e.g., DDR2, it indicates the importance of employing a dynamic and specific distribution approach for ECCs for each memory module, mainly when the scaling down of DDR amplifies the device vulnerability to potential faults induced by radiation effects, leaving a 90/80nm in DDR2 [16] to 14nm in DDR5 [17]. This approach enables us to meet the reliability required for the system with low operating costs; e.g., 91.78% of memory modules could operate with low-cost ECCs, while 8.22% would operate with more complex ones.

1.3 Document Structure

In addition to this section that contextualizes, motivates, and introduces the work, this Thesis contains six more sections. Chapter 2 explains a macro view of the main elements of a memory controller, their roles, and how they are correlated. Chapter 3 presents some works of memory controller design, illustrating trends and allowing us to understand research gaps for fulfilling the requirements of the current memory technologies. Chapter 4 describes the proposed memory controller’s macroblocks. Chapter 5 describes Absimth, a tool used to design and validate the proposed architecture. Chapter 6 demonstrates the efficacy and efficiency of the proposed memory controller when executing a set of experiments. Finally, Chapter 7 concludes the work and discusses future works.

2 THEORETICAL REFERENCE

This chapter comprises scientific fundamentals employed in this Thesis, such as fault, error, and failure events, the reason for their occurrence, and mechanisms used to mitigate these undesirable events in memories.

2.1 Fault, Error, and Failure Concepts

Avizienis et al. [18] explain that a service is correct when it implements the functionality defined for the system, with a **failure** occurring when the service deviates from its definition. Let service be a sequence of states; then, a **service failure** means that at least one state deviates from its definition, and this deviation is called an **error**. The cause of an **error** is called a **fault**, which can be internal or external to the system. The presence of an **internal fault** enables an **external fault** to harm the system, causing **errors** and possibly subsequent **failures**. In most cases, a fault first causes an **error** in the service state of a component that is part of the system's internal state, and the external state is not immediately affected.

Let an AND logic gate with two inputs be an element that composes the system to be evaluated; then, a grounded input (i.e., a stuck at 0) is a latent **fault**. An **error** happens when the two inputs assume the logical value one since the output value results in 0 (due to the stuck at 0) instead of one. In turn, a **service failure** happens if the system decision differs from the specification due to the wrong result of the AND gate; otherwise, the **error** remains latent.

This work targets memory systems; therefore, we focus on memory faults that cause errors and, consequently, memory failure. Logical or physical factors are the sources of errors [19]. Logical factors occur when writing data differently than initially intended. Physical factors include voltage difference, impact, temperature variation, radiation, magnetic variations, electrical fluctuations, electromagnetic interference, and stress over time [3]. In [20], the authors classify faults into three types according to their durations:

i. **Permanent fault** caused by a physical event that affects all system life, like an undesired short or open circuit; for this reason, it is only corrected by changing the hardware. Due to functioning or fabrication issues, a permanent fault occurs mainly due to three different causes [18]: (a) Manufacturing and Design Time Faults that come from errors in the design or manufacturing process and are manifested as stuck at 0/1 and signal delay. (b) Wearout Mechanisms influenced by the system aging. Negative-Bias temperature instability, hot

carrier injection, time-dependent dielectric breakdown, and electro-migration are some of the mechanisms that produce this fault. All cited mechanisms induce at the beginning intermittent faults that become permanent faults. (c) Process Variations due to manufacturing variability, such as non-perfect doping, causing random differences among transistors of the same chip;

ii. **Intermittent fault** that sporadically appears at irregular intervals. Intermittent faults are often considered early indicators of potential permanent faults;

iii. **Transient fault** occurs randomly, mainly due to charged particle emissions [21]. The fault is manifested by one or more bitflips or computation errors and occurs for a small amount of time.

Within the context of this Thesis, we are interested in dealing with *transient faults*, employing mechanisms that recover the correct state of the system upon detecting a fault.

2.2 Single Event Effect (SEE)

The Integrated Circuit (IC) scaling down increases the computational power of the system but makes the devices more susceptible to a potential fault caused by radiation effects. One of the most common failure-causing effects in electronic circuits is a Single Event Effect (SEE), an electrical disturbance that causes a change in the operation of a circuit. The passage of charged particles at the transistor junctions can induce SEEs. The transistor behavior depends on the ion charge at the impact time. Figure 4 illustrates how a highly charged ion affects the junction of a transistor [22]: (a) when crossing the junction, the ion generates a cylindrical beam of highly charged electron-hole pairs; (b) the load imbalance induces the creation of a temporary funnel; (c) the funnel is broken, and the remaining ions are equilibrated by diffusion.

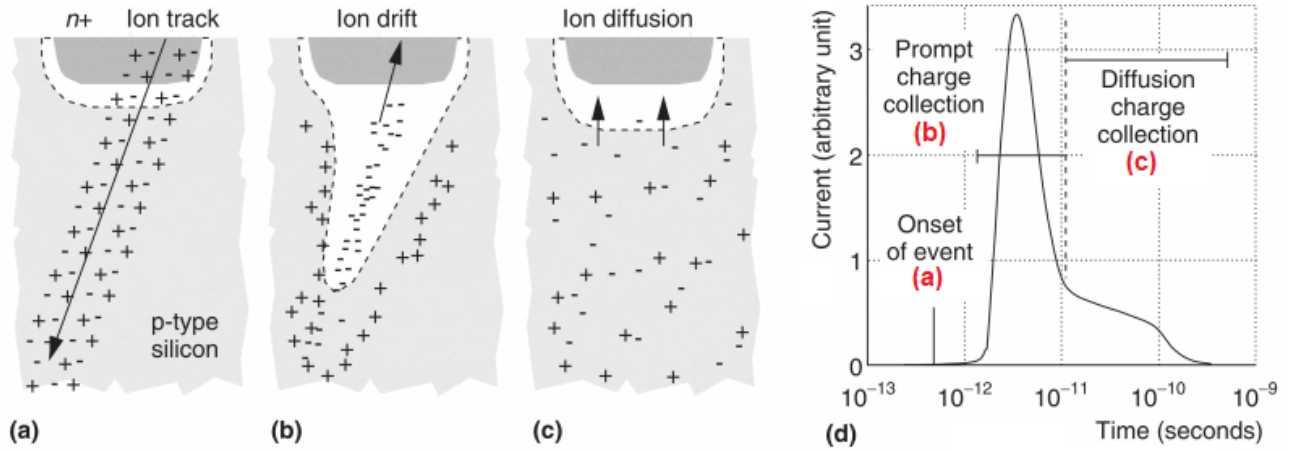


Figure 4. Effect of a charged particle passing over a transistor junction. (a) Cylindrical track of electron-hole pairs; (b) funnel extending the depletion region; (c) diffusion dominating the collection process; (d) resulting current pulse (based on [22]).

Although SEEs frequently occur in space applications due to solar radiation and cosmic rays [23] at the ground level, alpha particles and neutrons can modify the system state. Alpha is the most encountered particle; neutrons are usually separated into categories according to their charges. The collision of these particles in electronic devices causes transient and sometimes permanent faults. Therefore, a transient fault is a subset of a SEE [24] classified into:

i. Single Bit Upset (SBU) - when occurs a single event bitflip in a single cell [25], which is also used as a synonym for Single Event Upset (SEU);
ii. Multiple Cell Upset (MCU) - when a single event changes two or more memory cells [26];
iii. Multiple Bit Upset (MBU) - when a single event flips the content of two or more cells in the same word [27];
iv. Single Event Transient (SET) - when a single event causes a voltage failure in a circuit [28];
v. Single Event Functional Interrupt (SEFI) - when a single event causes functionality loss due to disturbance of registers, clocks, reset, and others [29];
vi. Single Event Latch-up (SEL) - when a single event causes a non-normal high current and requires a power reset [30].

2.2.1 SEEs in Memories

Memories are sensitive to radiation, making a SEE a constant threat to systems exposed to charged particles. Designers of these systems need to know the most likely SEEs to mitigate operational problems. Memory errors have been analyzed in detail for decades; studies show that a significant fraction of these errors repeatedly occurs at the same address, line, column, chip, or rank [31][32][33]. Furthermore, faults tend to be clustered [14], having strong correlations in space and time.

Studies in [14] demonstrate that memory errors increase with time. An analysis of failures on DRAMs performed over 15 months showed that although the failure rate due to transient errors increases slightly, the failure rate due to permanent errors is higher at the beginning and becomes almost the same as transient failures around the sixth to eighth month [32].

Transistor size has been reduced, maintaining the Moore law prediction [34] and creating the challenge called scaling faults [35]. The shrinking of transistor sizes directly impacts hardware sensitivity to temporary errors, increasing MCU for the newest technologies [36]. Usually, a SEE is linked to an SBU; however, Figure 5 illustrates the

growing presence of MBU when the technology is shrunk. For example, in SRAMs under 40nm, more than 40% of particle strikes result in MBUs [20].

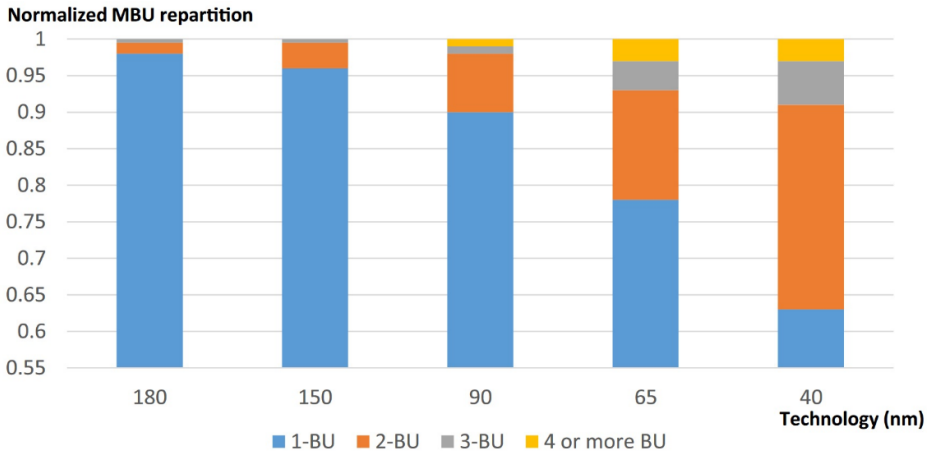


Figure 5. MBU percentages by technology nodes in nm for SRAMs [20].

Figure 6 exemplifies the CMOS technological advance reducing the size of square cells by half, and consequently, the area is reduced to a quarter. Considering the same memory region and particle energy, the reduction of transistors increases the possibility of the same SEE generating more MCUs. Additionally, the scaling technology reduces the capacitances associated with each transistor and the threshold voltage, increasing the bitflip occurrence.

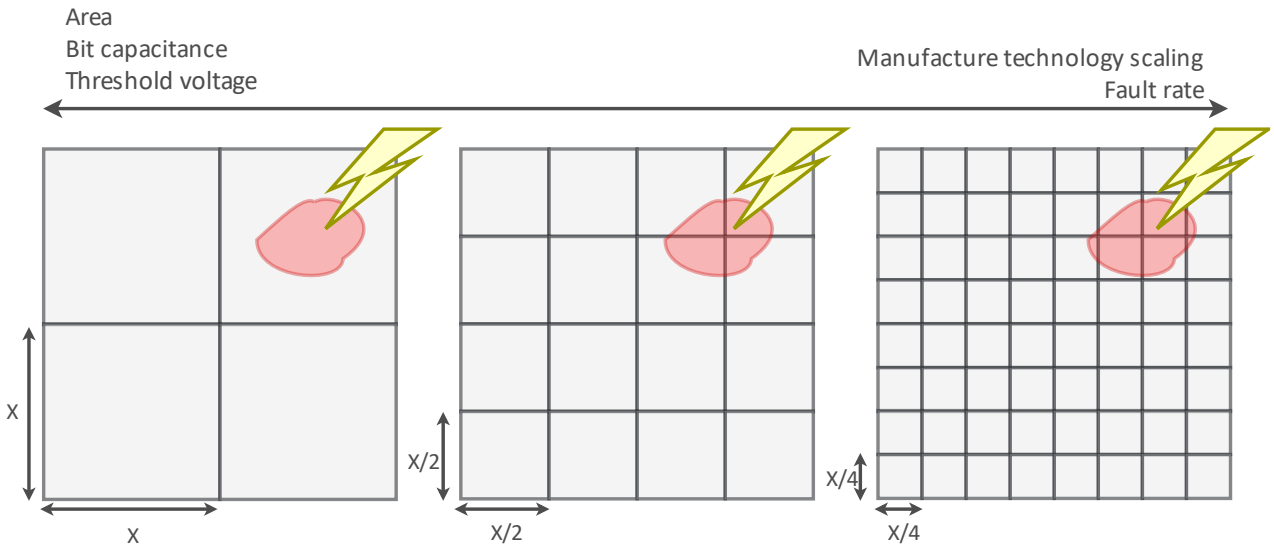


Figure 6. Impact of radiation on memory errors due to CMOS technology scaling; x is used to correlate the dimension measure of each square cell [Author].

Finally, Figure 7 shows an experiment by Gracia-Morán et al. [37]; the authors simulate several SEEs on 45nm memories, employing the radiation level in the terrestrial environment. Their experiment shows that although SBU is predominant, MBUs represented approximately half of the occurrences, with double and triple errors being the cases above 10% of all simulations.

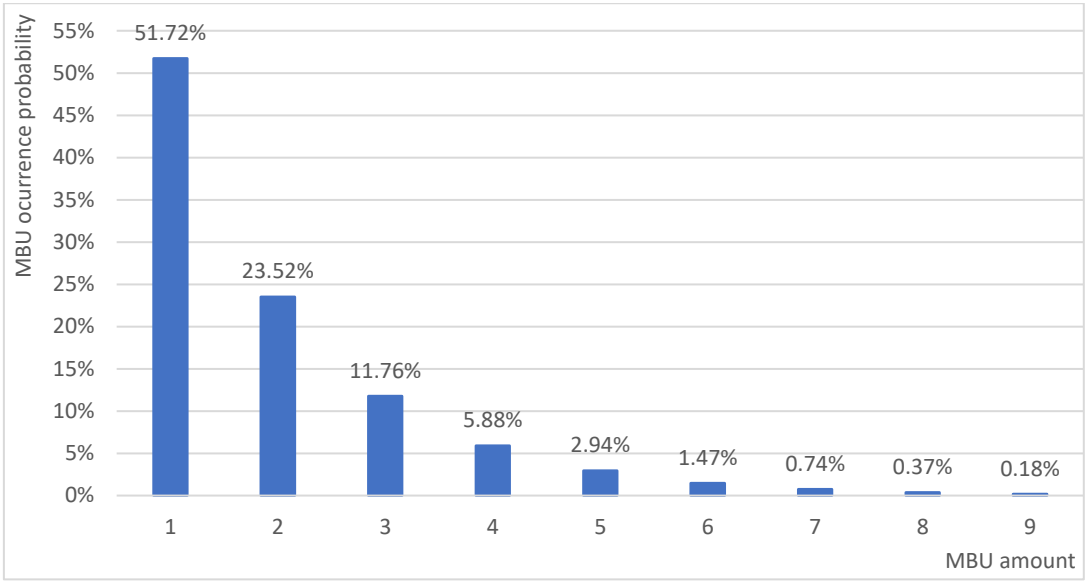


Figure 7. Impact of radiation on memory errors due to CMOS technology scaling [Author].

2.3 Techniques Employed to Mitigate SEEs in Memories

Businesses increasingly depend on Big Data server resources, reinforcing the need to avoid data loss. Memory is one of the crucial components of mitigating the loss of temporarily stored data. Reliability techniques are used at different system levels to preserve system functionality, having the goals of preventing, forecasting, tolerating, or correcting faults [38]; this section shows some of these techniques applied to memory systems.

2.3.1 Physical Bit Interleaving

The bit interleaving principle splits the data bits among words - in different memory modules or address spaces. This approach transforms MBUs into SBUs, increasing the error correction probability. However, more than one line is accessed during a reading and writing operation, and bit-extraction operations must be made to obtain the desired word. Usually, two interleaved words must be accessed, consuming more energy and latency [39]. Figure 8 exemplifies a bit-interleaving on the same line.

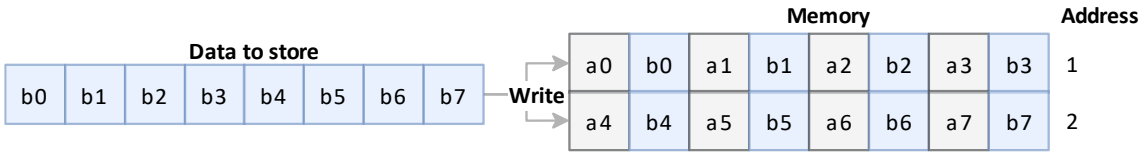


Figure 8. Bit-interleaving technique [20].

2.3.2 Memory Scrubbing

Memory scrubbing consists of continuously accessing data in memory using an error correction circuit; therefore, error cells are continuously corrected. This approach reduces the probability of clustered errors that prevent the correction circuit from reaching

high efficacy, reducing the probability of the system failing [20].

Scrubbing is an active defense against uncorrectable MCU; however, it consumes significant energy and bandwidth since the error detecting and correction circuit is contained in the memory controller, having to transport the data from memory to the controller [3]. Consequently, memory controllers that perform scrubbing are only in environments that demand as close to absolute reliability as possible, such as memory systems operating in extreme environments or large servers.

Schroeder, Pinheiro, and Weber [3] describe that half of the scenarios evaluated in the Google servers have the scrubber active, operating at a rate of 1GB every 45 minutes. This reinforces the article o Mukherjee et al. [40], which affirms that large memories need scrubbing to reduce the temporal double-bit error rate to a tolerable range.

2.3.3 Built-In Self-Test (BIST) and Built-In Self-Repair (BISR) in Memory

BIST, described for the first time by Marinescu [41], is a circuit integrated into a memory that executes test patterns to verify errors. This circuit typically uses a multiplexer ahead of memory and adds spare rows and columns, affecting performance and area size. The memory is repaired during the tests, storing faulty addresses and returning these addresses after the test [42]. Several test algorithms can be used in BIST to detect faults, such as a bit with a fixed value, failure in the address decoder, or transition failure [43]. Figure 9 illustrates a BIST architecture compounded by a controller, an address and data generator, and a comparator [44][45].

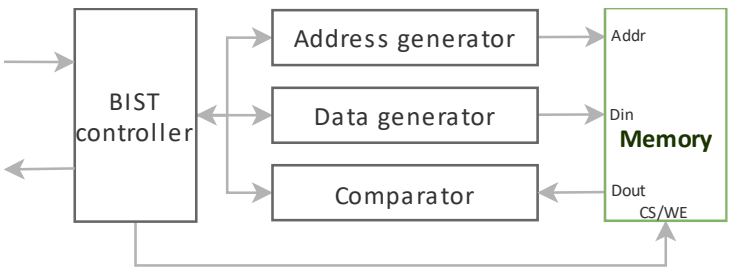


Figure 9. BIST base architecture [43].

The memory storage capacity has increased four times yearly, enabling it to meet the requirements of various IoT devices [43]; however, an automated test strategy is needed as density increases to reduce operating time and cost. Therefore, memory incorporates a self-test and repair circuit called memory Built-In Self-Repair (BISR). Figure 10 exemplifies a BISR architecture comprising a Test Access Port (TAP), BIST, Built-In Redundancy Analysis (BIRA), and a Fuse Controller.

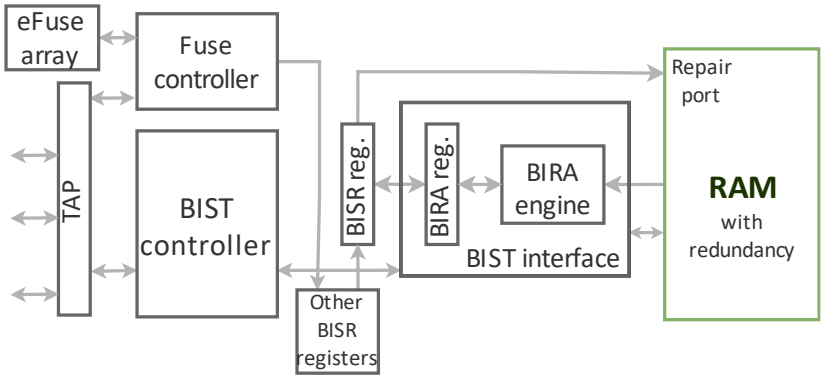


Figure 10. Example of a macro architecture of a BISR architecture [43].

BISR is implemented in three steps: (i) the BIST circuit analyzes faults; (ii) the BIRA circuit determines the type of repair to be done; and (iii) the Fuse controller receives from BIRA the circuits which must be disconnected for repair, reconfiguring the operation [46].

2.3.4 Double or Triple Memory Redundancy (DMR/TMR)

The DMR technique consists of duplicating the stored data and using an extra circuit to compare both data allowing us to decide if they are correct [20]. This approach does not fix the data directly but enables a memory management mechanism to perform a readback to find the correct data; for example, in case of a mismatch of a cache level using DMR, the cache level just below can provide the correct data.

The TMR technique consists of tripling the stored data and using an extra circuit to select the correct data in case of discordance [20]. Figure 11 exemplifies a TMR approach that acts during the read operation when a voter decides the correct value among the three available, and the majority determines the correct one.

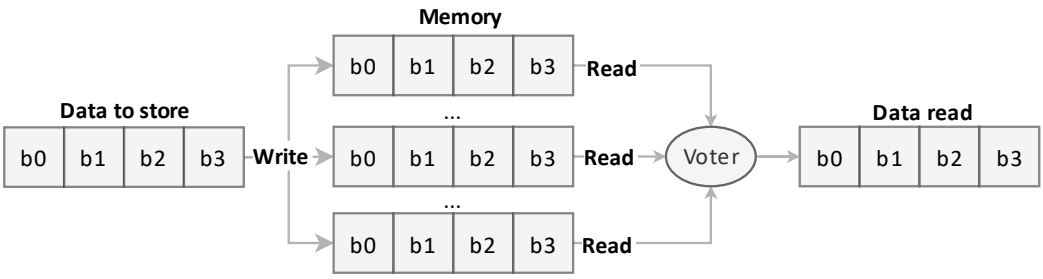


Figure 11. Example of the TMR technique applied to a 4-bit word (based on [20]).

Memory areas where data are stored in DMR and TMR approaches should be separated enough to consider a particle strike modifying only one stored data. With the hypothesis of the distance enough between redundant memory areas, DMR allows error detection, and TMR allows error detection and correction.

The main disadvantage of this memory technique is its memory space usage. Other solutions have been developed to address specific needs for robustness, replicating

different hardware parts. However, these solutions go with a rise in cost and complexity, and sometimes, a single erroneous bit makes an entire part of the memory unusable. With process variations increasing, the solution reaches its limits [47].

2.4 Error Correction Code (ECC)

ECC is a composition of a codeword and an encoding/decoding algorithm. The codeword includes data and check bits jointly evaluated by the ECC algorithm to enable data error detection/correction.

ECC is a technique to provide reliability emphasized in this Chapter because it is applied in the reliable memory controller explored in this Thesis.

2.4.1 Hamming Distance

The Hamming distance is the number of positions where the corresponding bits of two vectors of equal length differ. It is a metric regarding the minimum number of errors required to transform the content of one vector into another. Hamming distance allows us to know the ECC limiting capacity of correcting and detecting errors.

Let d be the Hamming distance, then Equation 1 and Equation 2 calculate the maximum number of errors that a code can correct EC or detect ED based on d [48].

$$\text{Equation 1.} \quad EC = \left\lfloor \frac{d-1}{2} \right\rfloor$$

$$\text{Equation 2.} \quad ED = d - 1$$

These equations are exclusive, i.e., EC or ED, but not EC and ED simultaneously. The simultaneity relationship among EC, ED, and d implies using Equation 3 instead of Equation 2. Consequently, if an application targets correcting and detecting errors simultaneously, ED is reduced [49].

$$\text{Equation 3.} \quad ED = d - EC - 1$$

2.4.2 Parity Code

The error detection procedure of the first memories was done through parity bits added to each byte [50]. The parity technique adds one bit to the memory line to compute the number of 1s or 0s stored. For example, for an even parity code, the parity encoder counts the number of data bits in 1; if this number is even, the engine puts 0 in the parity bit; otherwise, it puts 1. When reading the data, the parity decoder checks if the number of bits in 1 is even; if not, it informs that the data is corrupted.

On the one hand, a parity error is detectable only when an odd number of bits is changed to the opposite value – i.e., 1 to 0, or vice-versa. On the other hand, an even inversion of bits is masked since the decoding algorithm applies XOR (or exclusive) logic among all codeword bits. Fortunately, SBU – an odd error case – is the most common error pattern for corrupting data [3][15]. Table 1 exemplifies an even parity validation.

Table 1. Example of even parity with a codeword encompassing a Data byte (Data) and a Parity bit (P) [Author].

Codeword		Number of 1s in the codeword	Is right?
Data	P		
00000000	0	0	Yes
10000001	1	3	No
11100000	1	4	Yes
10000000	1	2	Yes

The parity code is not actually an ECC but an Error Detection Code (EDC); since the Hamming distance is 2, the code can only detect SBU, not correct them. However, the parity code is usually presented with ECCs, mainly because many ECCs are the parity composite with other codes. Additionally, the parity code composition also results in an ECC, as in some 2D ECCs [49].

Figure 12 illustrates the read and write operation; on the write operation, an XOR is performed among all bits, and the result is added to the stored bits. The same is made for the read operation, comparing the stored parity bit with the generated one.

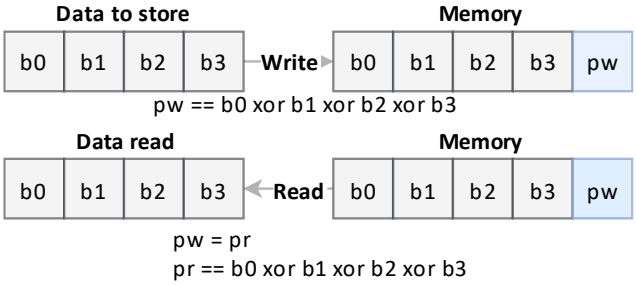


Figure 12. Parity technique [20].

2.4.2.1 Parity-based mono-Copy Cache (PmC2)

Alouani et al. in [51] propose combining the double memory redundancy and parity to create the PmC2 technique. During write operations, the parity bit generated is stored together with data in just one location. On a read operation, the parity bit of the value read is compared to the parity bit stored; if it is equal, read the original area; otherwise, the value taken is stored redundantly. This technique, illustrated in Figure 13, is a trade-off between a single parity bit and redundancy; it uses the detection power of parity bits and redundancy to correct a detected error.

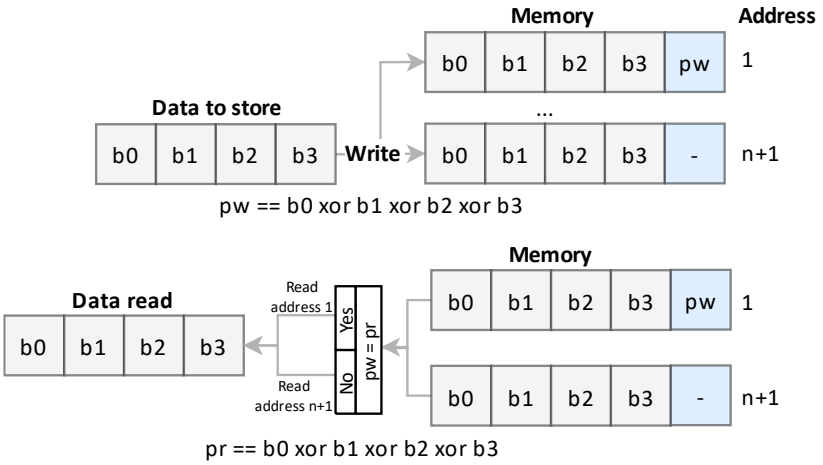


Figure 13. PmC2 technique [20].

2.4.2.2 Double Parity bit Single Redundancy (DPSR)

Chabot et al. in [20] proposed DPSR; they observed that it is improbable to find 2-bit upsets with a gap between the two flipped bits. Regarding the work in [36], it represents less than 2% of 2-bit-upsets, which makes less than 0.6% of total observed patterns for 40nm SRAM technology. Moreover, in the case of a 3-bit-upset, the only pattern that may lead to corruption in this solution is when three horizontally aligned bits are flipped, and the probability for this pattern is less than 0.028%. DPSR proposal consists of having two parity bits formed by one bit of distance, as demonstrated in *pw0* and *pw1*; when one parity fails, read the partial redundancy information stored in the other block, as illustrated in Figure 14.

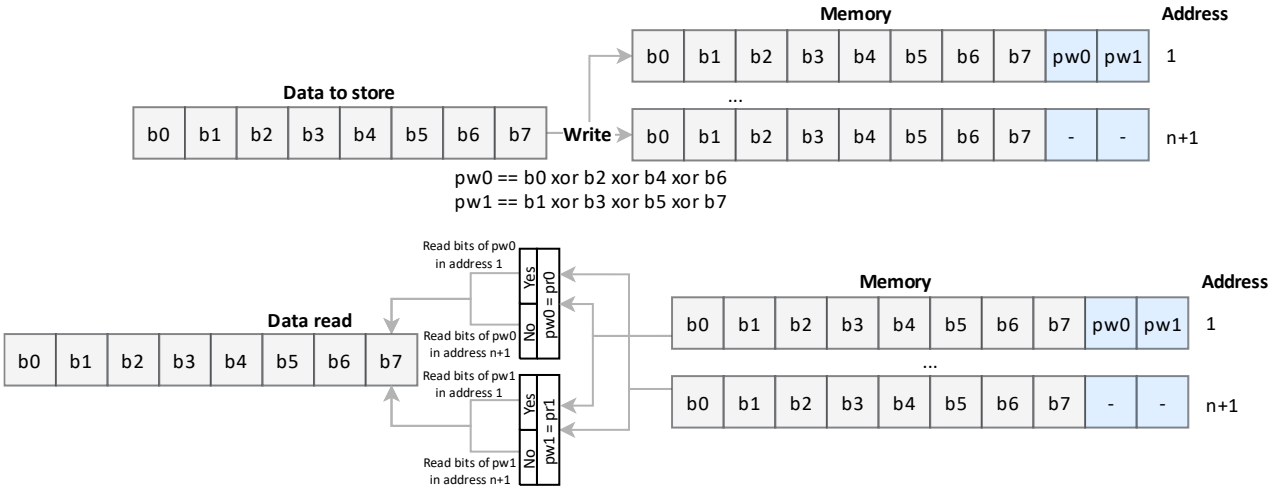


Figure 14. High-level description of DPSR technique [20].

Based on thorough fault injection experiments, DPSR shows promising results; It detects and corrects more than 99.6% of encountered MBU and has an average time overhead of less than 3%.

2.4.3 Hamming Code

Hamming was one of the first ECCs developed for application in computer systems

intending to correct simple errors since the Hamming distance is 3 [49]. Let M be the data vector and k be the number of check bits; then, N is the codeword vector, and $|N| = |M| + k$ is the number of the codeword bits, such that Hamming code is represented by $Ham(|M|, |N|)$ with the following fundamentals [50]:

- Hamming results from a linear system of equations and a truth table to identify the error position.
- A Hamming Code needs to respect Inequations 1 and 2.

Inequation 1. $2^k \geq k + |M| + 1$

Inequation 2. $2^{|M|} \geq \frac{2^{|N|}}{|N|+1}$

Hamming encoding and decoding use identity, generation, and verification matrices, illustrated in the following $Ham(4,7)$ example.

Equation 4 and Equation 5 describe, respectively, the squared identity matrix I_{2^k-k-1} of order $2^k - k - 1$ and the matrix Q encompassing the data addresses 011, 101, 110, and 111 in case of error occurrence.

Equation 4. $$I_{2^k-k-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 5. $$Q = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Equation 6 displays the generating matrix G employed to start the Hamming encoding, and Equation 7 represents $G(4,7)$.

Equation 6. $$G = [\, I_{2^k-k-1}\, Q]$$

Equation 7. $$G(4,7) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Equation 8 describes the linear transformation that multiplies the data vector $M(1,4)$ and matrix $G(4,7)$ to encode $Ham(4,7)$ into the codeword $N(1,7)$.

Equation 8. $N = M \times G$

For instance, Equation 9 illustrates that the data vector $M = [1000]$ encoded with

$Ham(4,7)$ results in the codeword $N = [1000011]$. Note that the first four bits [1000] are data, and the remaining three bits [011] are redundancy (check bits).

Equation 9. $$N = [1 \quad 0 \quad 0 \quad 0] \times \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} = [1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1]$$

Decoding is the reverse of the encoding process, which requires verifying whether the check bits have the same values as obtained in the encoding process.

Let Q^T be the transposed matrix of Q and I_k be the squared identity matrix of order k, respectively computed by Equation 10 and Equation 11, then, Equation 12 computes the redundancy matrix H.

Equation 10. $$Q^T = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

Equation 11. $$I_k = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Equation 12. $$H = [Q^T I_k] = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Let N' be the codeword read in the decoding process such that $N' = N$ in case of error absence, then, N'^T be the transposed matrix of N', as Equation 13 displays, then, Equation 14 multiply H by N'^T to compute the check bit syndrome vector $S[s_0 \; s_1 \; s_2]$. Note that this example produces $S = [000]$ since the codewords employed in the encoding (N') and decoding (N) processes are equal, meaning error absence.

Equation 13. $$N'^T = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Equation 14. $$S = H \times N'^T = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = [0 \quad 0 \quad 0]$$

Table 2 displays that the decoding process does not detect errors when $S = [000]$; otherwise, the codeword N contains one error, and the error bit position is defined by S, as described by Equation 15.

Equation 15. $Error\ position = 4 \times c_0 + 2 \times c_1 + c_2$

Table 2. Relation between the check bit syndrome vector and codeword error bit [Author].

Codeword	S	Error position	Error bit
$N = [\ \underset{3}{d_0}\ \underset{5}{d_1}\ \underset{6}{d_2}\ \underset{7}{d_3}\ \underset{4}{c_0}\ \underset{2}{c_1}\ \underset{1}{c_2}\ ]$ $data\ bits \quad check\ bits$	[000]	0	Ø
	[001]	1	$[\ c_2\ ]$
	[010]	2	$[\ c_1\ ]$
	[011]	3	$[\ d_0\ ]$
	[100]	4	$[\ c_0\ ]$
	[101]	5	$[\ d_1\ ]$
	[110]	6	$[\ d_2\ ]$
	[111]	7	$[\ d_3\ ]$

Despite the matrixial codeword computation, the check bit vector is easily calculated by XOR logic ($\oplus$) operation, as Equation 16 to Equation 18 shows.

Equation 16. $c_0 = d_1 \oplus d_2 \oplus d_3$

Equation 17. $c_1 = d_0 \oplus d_2 \oplus d_3$

Equation 18. $c_2 = d_0 \oplus d_1 \oplus d_3$

Additionally, S is also computed by $\oplus$ operations between the stored check bit vector $C[c_0\ c_1\ c_2]$ and the recomputed one $C'[c'_0\ c'_1\ c'_2]$, as Equation 19 to Equation 21 display.

Equation 19. $s_0 = c_0 \oplus c'_0$

Equation 20. $s_1 = c_1 \oplus c'_1$

Equation 21. $s_2 = c_2 \oplus c'_2$

For instance, considering the storage codeword $N' = [1\mathbf{1}00011]$, whose check bits $C = [011]$ where computed considering $M = [1000]$, consequently producing $N = [1000011]$. When recomputing the stored $M' = [1\mathbf{1}00]$ results $C' = [110]$. Applying Equation 19 to Equation 21 results $S = [101]$.

2.4.4 Extended Hamming Code

The Extended Hamming ECC is formed by adding a parity bit, increasing the Hamming distance to 4, which means the code can fix one error and catch two errors.

Therefore, Extended Hamming belongs to the Single Error Correction - Double Error Detection (SECDED) class of ECCs [52]. The Extended Hamming code is represented by $ExHam(|M|, |N^*|)$, such that the $|N^*| = |N| + 1$. Besides the check-bit syndrome vector S, $ExHam(|M|, |N^*|)$ includes the parity syndrome δ, computed by applying XOR between the stored parity bit p and the most recent computed p', as described by Equation 22.

Equation 22. $\delta = p \oplus p'$

Let $\boldsymbol{s}$ be the OR logic (∨) among all bits of S, described by Equation 23; then, Table 3 describes the error type according to the combination of both syndromes.

Equation 23. $\boldsymbol{s} = s_0 \vee s_1 \vee s_2$

Table 3. Type of error according to the combination of syndromes [Author].

$\boldsymbol{s}$	δ	Error type
0	0	No error
0	1	Simple error in Parity
1	0	Double error
1	1	Simple error

Employing the same example of Section 2.4.3 with $M = [1000]$ results in $N^* = [10000111]$ since the parity is computed considering all data and check bits, as illustrated by Equation 24.

Equation 24. $p = d_0 \oplus d_1 \oplus d_2 \oplus d_3 \oplus c_0 \oplus c_1 \oplus c_2$

In the occurrence of two errors in $N^* = [\mathbf{01}000111]$, the parity syndrome δ is zeroed, and $\boldsymbol{s}$ is one ($S = [110]$). The combination of δ and $\boldsymbol{s}$ indicates a double error detection; however, the error positions are unknown; thus, N^* cannot be fixed.

The limitation of only detecting two errors without the ability to correct them is often incompatible with critical system requirements. Therefore, a Double Error Correction - Triple Error Detection (DECTED) ECC [20][53] is a natural evolution of a SECDED with a penalty of adding redundancy bits and raising the algorithmic complexity for data encoding/decoding.

On the one hand, the integrated circuit scaling makes them more susceptible to MBU occurrence, demanding higher efficacy ECCs, sometimes with more power than DECTED codes. On the other hand, linear ECCs demand long sequences of redundancy to reach high error correction rates, mainly for large data words. Therefore, the Two-Dimensional (2D) ECCs emerge as an efficacy solution to MBUs occurrence since its format

cross-combine low-cost codes to reach similar error correction rates than huge linear ECCs without overloading the codeword with redundancy bits.

2.4.5 Traditional Error Correction Codes

Over the years, several ECCs have gained notoriety for specific applications, such as Golay Binary Code (GBC) [54] for space operations, Reed Soloman (RS) [55] for data transmission in telecommunications, and Bose–Chaudhuri–Hocquenghem (BCH) [56] and Chipkill [57] for data storage.

The Voyager 1 and 2 spacecraft use GBC, an ECC that encodes 12-bit data into a 24-bit word [54]. This code can fix any 3-bit error and detect up to 7-bit errors.

Wilkerson et al. [56] explain that BCH codes are a large class of ECCs that can correct highly concentrated and widely scattered MBUs [58]. Each BCH is a linear block code defined over a finite Galois Field GF(2^m) [59], with a generator polynomial, where 2^m represents the maximum number of codeword bits. The encoding logic takes *k*-bit input data word *d* and uses a pre-defined encoder matrix *G* to generate the corresponding codeword *u (u = d × G).* Since BCH is a systematic code, the original *k*-bit data is retained in the codeword and is followed by *r*-check bits. The decoding logic employs syndrome generation, error classification, and error correction to detect and correct errors in the stored codeword. Previous studies have shown that the decoding procedure of multi-bit BCH is arduous, the complexity overgrows as the number of bit corrections increase, and the error correction is the most complex and time-consuming procedure [60].

RS is a powerful ECC based on BCH, widely used in many digital systems such as DVDs and space mission communication systems [55]. RS is a systematic way of building codes to detect and correct random errors. Adding t-check symbols to the data, RS can detect any combination of up to *t* wrong symbols and correct up to *t/2* symbols.

Chipkill is a symbol-based ECC for Single Symbol Correction – Double Symbol Detection (SSCDSD); it is an advanced form of computer memory error checking that corrects up to 4-bit errors [57]. It has been used in many cases since large-scale systems, including Google server farms [3], numerous supercomputers [31][14], and article proposes [61][62][63][64]. Initially, the Chipkill circuit was located on the memory module, preventing the use of standard memory modules. Later, this circuit was placed either on the Northbridge chipset of the computer motherboard or within the Central Processing Unit (CPU), enabling the SDRAM standard use, like DDR3 memory modules. Chipkill operates on 4-bit nibbles, called symbols; if just one symbol is erroneous, it can correct all bits of the symbol. If more than one symbol is erroneous, it can only detect the error. Chipkill is most effective if used

with memory modules built with 4-bit width chips - each symbol corresponds to one chip. Thus, multi-bit errors are isolated in a single chip or symbol because of the physical distance between the chips. Chipkill is less effective with ×8 chips because a multi-bit error could then straddle a symbol boundary, thus causing a double-symbol error that would only be detected. For a 64-bit word, Chipkill consists of 32 4-bit symbols for data and four 4-bit symbols for checking, totaling 144 bits. When the computer stores a number in memory, it calculates values for the check symbol and stores them along with the number.

2.4.6 Two-Dimensional Error Correction Codes (2D-ECCs)

A 2D-ECC is characterized by having data and/or redundancy bits in two dimensions, typically named row and column. This definition allows including any 1D-ECC physically organized in rows and columns in the 2D-ECC class; therefore, Freitas et al. [49] subdivide 2D-ECC into four classes: (i) Straightforward 2D-ECC (S2E) - a code organized in 2D physical structure, but correcting errors with 1D-algorithms; (ii) Product Code (PC) - an ECC treated as a product of two codes enabling to build long codes based on small ones; (iii) Extended Product Code (EPC) - a special case of PC that uses more than one code per row and/or column; and (iv) Mixed Code (MC) - a 2D-ECC containing at least one bit of data or redundancy whose change implies encoding both dimensions but cannot be classified as PC or EPC. Our main interest here is the PC class since the Line Product Code (LPC) [65], one of the codes used in this Thesis, derives from PC.

Let α and β be the number of columns composing the data and redundancy areas, and let ϑ and ε be the number of rows composing the data and redundancy areas, respectively, such that $\gamma = \alpha + \beta$ and $\theta = \vartheta + \varepsilon$. Then, each PC row and column is encoded using the $C_1(\gamma, \alpha, d_1)$ and $C_2(\theta, \vartheta, d_2)$ codes, respectively, forming the $C_1 \times C_2$ code, and any bitflip in the data region disturbs the row and column of the corresponding bit. Figure 15(a) illustrates the basic PC structure. Also, PC adds a region containing check bits of check bits, increasing the Hamming distance and, consequently, the code correction potential [50].

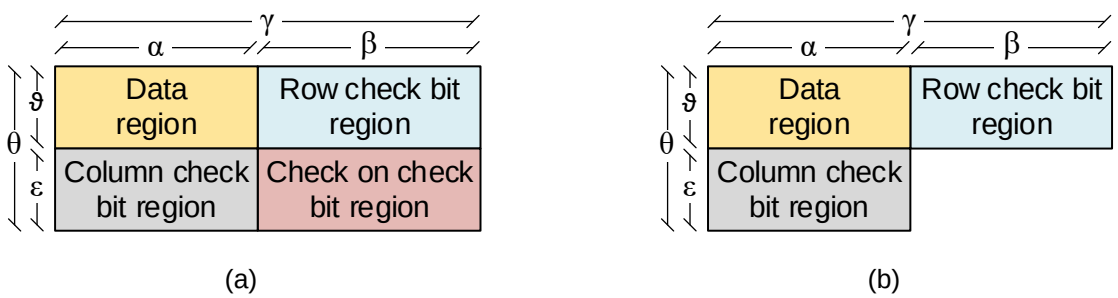


Figure 15. Basic structures of (a) PC and (b) modified PC [49].

Equation 25 computes the Hamming distance of a PC d_{PC} by multiplying the

distances of each 1D-ECC that compose PC; note that this large Hamming distance results from hierarchical check bits combination [49].

Equation 25. $d_{PC} = d_1 \times d_2$

Equation 25 demonstrates that PC increases the theoretical correction and detection capabilities, but this increase also raises the redundancy costs, implying more area and energy consumption. To mitigate these costs, some authors proposed the *modified PC* to reduce the associated redundancy costs, which do not have the check bits of the check bits. Figure 15(b) illustrates the structure of a modified PC, and Equation 26 displays its minimum distance calculation d_{mPC} [49].

Equation 26. $d_{mPC} = d_1 + d_2 - 1$

There are several well-known 2D ECC like Matrix [66], Decimal Matrix Code (DMC) [67], Horizontal-Vertical-Double-Bit Diagonal (HVDD) [68], Product Code for Space Applications (PCoSA) [69], extended Matrix Region Selection Code (eMRSC) [70] and Line Product Code (LPC) [65].

Argyrides et al. [66] presented the Matrix code based on a matrix codeword format for correcting adjacent errors. The proposal fixes up to 16 adjacent bitflips for every 64-bit data. In [71], the same authors present a method that divides the data bits in matrix form and applies the Hamming code to generate the check bits; this method can detect and correct errors of up to five bits on a row. Guo et al. [67] introduced DMC, which mitigates soft memory errors. DMC determines all possible combinations of adjacent errors and randomly alters the word to maximize the combination of adjacent bits based on the proposal of Dutta et al. [72]. The DMC method divides the correction codes and data bits into symbols arranged in an *n* × *m* matrix for detecting and correcting multiple errors on each row. Rahman et al. [68] proposed the HVDD method based on parities for each row, column, and diagonal, enabling to correction of up to 3-bit errors with a low additional cost. Freitas et al. [69] proposed PCoSA, an ECC that employs Hamming and parity in rows and columns. They evaluated PCoSA with Matrix, CLC, RM, and PBD ECCs, demonstrating that PCoSA reaches 100% error detection for up to 3 bits; the other codes reached lower detection rates; Matrix code achieved the worst performance, detecting only 16% for seven bitflips. Silva et al. [70] introduced eMRSC, an ECC with low implementation cost that can be configured in several formats; for instance, eMRSC(32, 3, 64) and eMRSC(32, 3, 56) formats that require different redundancy and consequently result in different synthesis costs and error efficacy. Freitas et al. [65] proposed LPC, a modified PC that uses Extended Hamming in both rows

and columns to implement reliable memories; since this Thesis employs LPC in the experimental results, the following section explains LPC in detail.

2.4.7 Line Product Code (LPC)

LPC [65][73] is a modified Product Code used in this work. Figure 16 shows the LPC codeword for enclosing 16 data bits; this codeword encompasses a 4×4 data matrix (D); a 4×3 row check-bit matrix (CR); a 4×1 row parity matrix (PR); a 3×4 column check-bit matrix (CC); and a 1×4 column parity matrix (PC). Rows and columns implement Extended Hamming using CR and PR matrices, and CC and PC matrices, respectively.

D_0	D_1	D_2	D_3	CR_0	CR_1	CR_2	PR_0
D_4	D_5	D_6	D_7	CR_3	CR_4	CR_5	PR_1
D_8	D_9	D_{10}	D_{11}	CR_6	CR_7	CR_8	PR_2
D_{12}	D_{13}	D_{14}	D_{15}	CR_9	CR_{10}	CR_{11}	PR_3
CC_0	CC_1	CC_2	CC_3				
CC_4	CC_5	CC_6	CC_7				
CC_8	CC_9	CC_{10}	CC_{11}				
PC_0	PC_1	PC_2	PC_3				

Figure 16. LPC structure encompassing five regions of bits: data (D), row-check (CR), column-check (CC), row-parity (PR), and column-parity (PC) [65].

Let q be an auxiliary variable to compose the indices of the LPC elements, then, Equation 27 to Equation 29 computes the CR matrix, and Equation 30 calculates the PR matrix.

$$\text{Equation 27.} \quad CR_{3q} = D_{4q} \oplus D_{4q+1} \oplus D_{4q+3} \qquad \forall\, 0 \le q \le 3$$

$$\text{Equation 28.} \quad CR_{3q+1} = D_{4q} \oplus D_{4q+2} \oplus D_{4q+3} \qquad \forall\, 0 \le q \le 3$$

$$\text{Equation 29.} \quad CR_{3q+2} = D_{4q+1} \oplus D_{4q+2} \oplus D_{4q+3} \qquad \forall\, 0 \le q \le 3$$

$$\text{Equation 30.} \quad PR_q = D_{4q} \oplus D_{4q+1} \oplus D_{4q+2} \oplus D_{4q+3} \oplus CR_{3q} \oplus CR_{3q+1} \oplus CR_{3q+2} \qquad \forall\, 0 \le q \le 3$$

Employing the same auxiliary variable q, Equation 31 to Equation 33 calculates the CC matrix, and Equation 34 computes the PC matrix.

$$\text{Equation 31.} \quad CC_{3q} = D_q \oplus D_{q+4} \oplus D_{q+12} \qquad \forall\, 0 \le q \le 3$$

$$\text{Equation 32.} \quad CC_{3q+1} = D_q \oplus D_{q+8} \oplus D_{q+12} \qquad \forall\, 0 \le q \le 3$$

$$\text{Equation 33.} \quad CC_{3q+2} = D_{q+4} \oplus D_{q+8} \oplus D_{q+12} \qquad \forall\, 0 \le q \le 3$$

$$\text{Equation 34.} \quad PC_q = D_{4q} \oplus D_{q+4} \oplus D_{q+8} \oplus D_{q+12} \oplus CC_q \oplus CC_{q+4} \oplus CC_{q+8} \qquad \forall\, 0 \le q \le 3$$

When reading a memory location, the same equations used to calculate CR, PR, CC, and PC are used to create an analogous check and parity bit structure. This new structure

is represented by the same symbols concatenated with “r” to indicate that it is a recalculated structure, i.e., sCR, sPR, sCC, and sPC.

Let q be an auxiliary variable to compose the syndrome indices; then, Equation 35 to Equation 38 computes the check and parity bit syndromes.

Equation 35. $$sCR_q = \left(CR_{3q} \oplus rCR_{3q}\right) \vee \left(CR_{3q+1} \oplus rCR_{3q+1}\right) \vee \left(CR_{3q+2} \oplus rCR_{3q+2}\right) \quad \forall\, 0 \leq q \leq 3$$

Equation 36. $$sCC_q = \left(CC_q \oplus rCC_q\right) \vee \left(CC_{q+7} \oplus rCC_{q+7}\right) \vee \left(CC_{q+14} \oplus rCC_{q+14}\right) \quad \forall\, 0 \leq q \leq 3$$

Equation 37. $$sPR_q = PR_q \oplus rPR_q \quad \forall\, 0 \leq q \leq 3$$

Equation 38. $$sPC_q = PC_q \oplus rPC_q \quad \forall\, 0 \leq q \leq 3$$

Figure 17 illustrates the same LPC codeword as Figure 16, adding the auxiliary structures used in the LPC encoding process. The arrows in Figure 17 help to understand the elements that encompass the auxiliary structures.

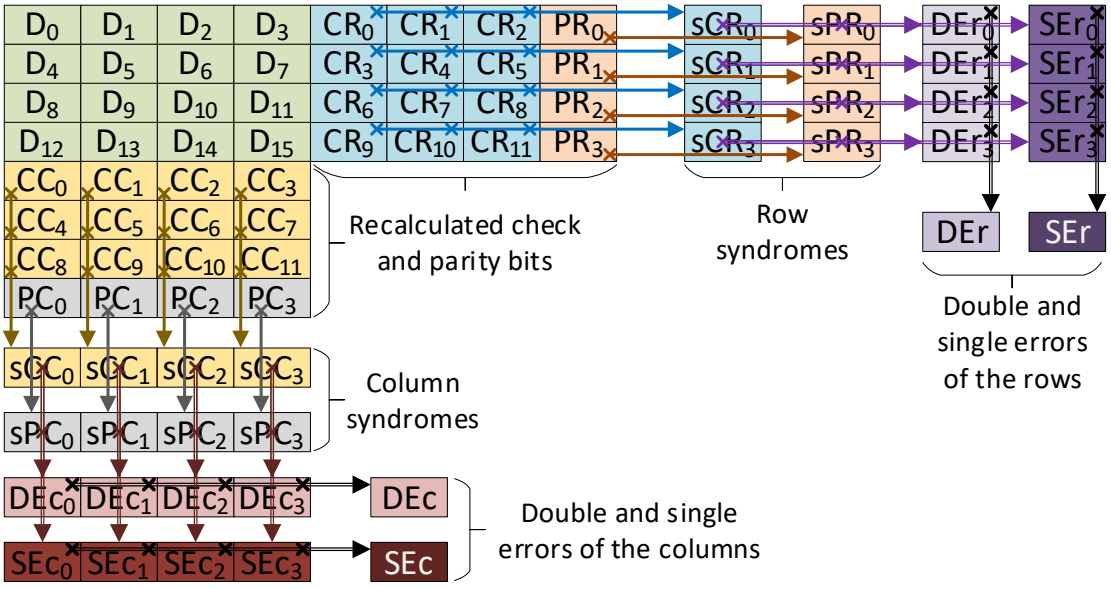


Figure 17. Graphical representation of LPC codeword and the auxiliary structures sCR, sPR, DEr, SEr, CC, PC, sCC, sPC, DEc, and SEc [65].

The SE and DE structures are employed to calculate the occurrence of single and double errors, respectively. Let q be an auxiliary variable to compose the SE and DE indices; then, Equation 39 and Equation 40 calculate the single and double error occurrence in each row q, respectively; analogously, Equation 41 and Equation 42 calculate the single and double error occurrence in each column q, respectively. The SE and DE calculations correspond to the application of Table 3 of Section 2.4.4 (Extended Hamming Code).

Equation 39. $$SEr_q = \left([sCR_q, sPR_q] = [1,1]\right) ? 1 : 0 \quad \forall\, 0 \leq q \leq 3$$

Equation 40. $DEr_q = ([sCR_q, sPC_q] = [1,0])\ ?\ 1 : 0 \qquad \forall\ 0 \leq q \leq 3$

Equation 41. $SEc_q = ([sCC_q, sPC_q] = [1,1])\ ?\ 1 : 0 \qquad \forall\ 0 \leq q \leq 3$

Equation 42. $DEc_q = ([sCC_q, sPR_q] = [1,0])\ ?\ 1 : 0 \qquad \forall\ 0 \leq q \leq 3$

Additionally, SEr, DEr, SEc, and DEc are the sum of single and double errors on rows and columns; these variables are computed by Equation 43 to Equation 46.

Equation 43. $$SEr = \sum_{q=0}^{3} SEr_q \qquad \forall\ 0 \leq q \leq 3$$

Equation 44. $$DEr = \sum_{q=0}^{3} DEr_q \qquad \forall\ 0 \leq q \leq 3$$

Equation 45. $$SEc = \sum_{q=0}^{3} SEc_q \qquad \forall\ 0 \leq q \leq 3$$

Equation 46. $$DEc = \sum_{q=0}^{3} DEc_q \qquad \forall\ 0 \leq q \leq 3$$

As described in Section 2.4, an ECC is the composition of a bit structure and the application of an encoding and decoding algorithm. The LPC authors proposed more than one algorithm with the same codeword, such as double error and iterative corrections [73]. This Thesis employs a less complex algorithm to reduce power dissipation, energy consumption, latency, and area. The adopted LPC algorithm uses only cross and simultaneous decoding of all rows and columns through the equations described in this section. This approach allows us to correct any combination of data and check bit errors that result in rows and columns with single errors. Although the correction potential is reduced compared to more elaborate LPC versions, the experiments of this work showed the efficacy of the proposed LPC algorithm, reaching error correction rates of up to 2.3 times higher compared to other Hamming-based algorithms.

2.5 Main Memory Organization

The main memory is usually built with Random Access Memory (RAM), a device that temporarily stores and reads data providing fast hardware access. The Joint Electron Device Engineering Council[4] (JEDEC) [74] standardizes commercial memories like Dynamic Random Access Memory (DRAM), which is widely used due to its low cost and high density. Figure 18 displays a memory controller connected with two DRAMs in a Dual In-line Memory

[4] JEDEC is responsible for standardizing semiconductor engineering, representing all areas of the US electronics industry. JEDEC aims at product interoperability, benefiting the industry and, ultimately, analyzing consumers by decreasing product release time on the market and reducing development costs [74].

Module (DIMM)[5].

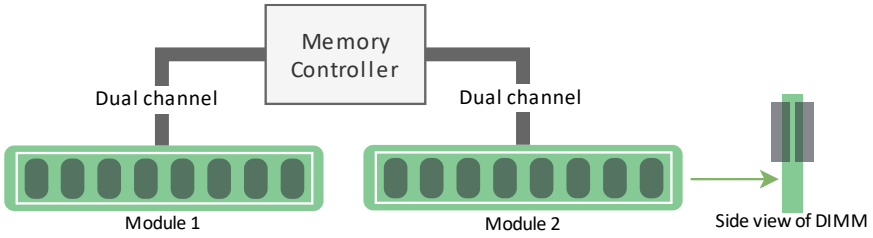


Figure 18. The communication architecture between the memory controller and DRAM DIMM [76].

Each bit of memory is stored in a single transistor and capacitor (cell) [75]; these memory cells are organized in arrays with rows and columns, where each column can deliver *n* bits. A DRAM can be classified by its number of memory arrays; for example, an *x8 DRAM* indicates a DRAM organized in an eight-memory array format. A set of memory arrays creates a bank that forms a group of banks and, subsequently, a chip. The number of RAM chips depends on whether the memory supports an ECC mechanism. Memories with and without ECC support use nine and eight chips, respectively. Figure 19 illustrates that this group of chips forms a rank, and one or more ranks perform a memory module.

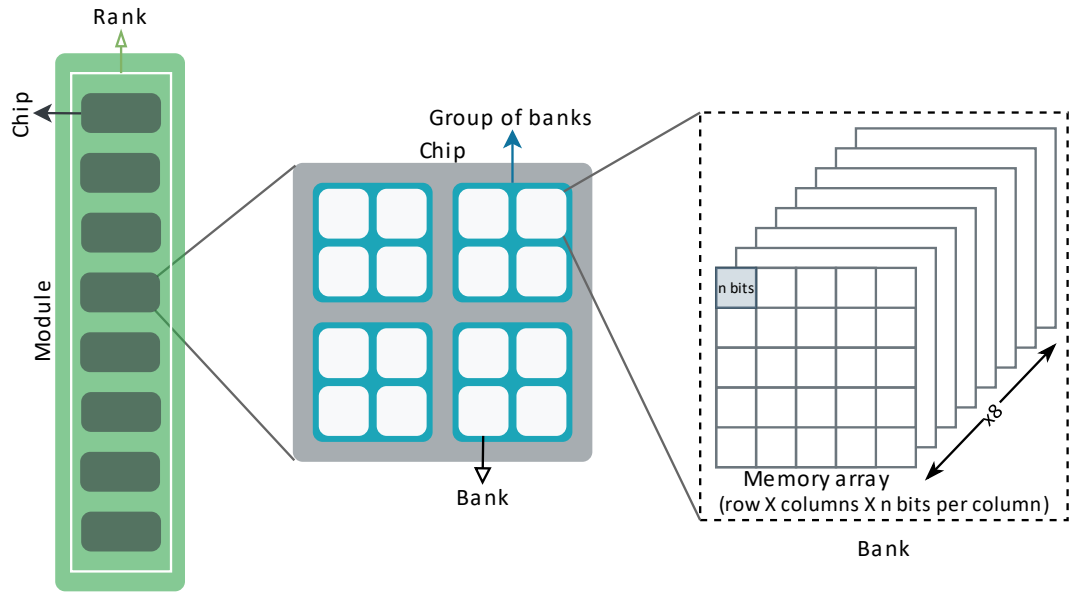


Figure 19. Memory base architecture [76].

Commercial memories that include an ECC use an extra chip for each module to add check bits; although expensive, this additional protection is vital, especially for programs with extensive data [77]. ECCs employ sequences of check bits together with the data, generating a codeword whose coding allows identifying and correcting one or more error positions; e.g., a 7-bit ECC can tolerate an error in a 64-bit data array. However, using ECC implies encoding and decoding data, wasting energy and area, and implying computation

[5] The Dual In-line Memory Module (DIMM) has memory modules and connectors on both sides, requiring a dual channel to communicate with the memory controller.

delay. Due to the verification process, memories that include ECCs are usually between 1% and 2% slower than memories without ECC [78].

2.6 Memory Controller Fundamentals

A memory controller is a system that manages memory operations, consisting of an interface to a memory client, called Processing Element (PE), like processors, hardware accelerators, and other peripherals. These PEs send their transactions to the memory controller, which stores them in separate queues. Memory controller architectures can be logically organized into Frontend and Backend blocks [79]. The Frontend block is independent of the memory technology; it provides an interface for each PE supplying buffer, enabling reading and writing memory requests and responses. The Backend block organization depends on the memory technology; it interfaces the requests and responses of the Frontend block with the target memory. Figure 20 presents an abstract organization of the main elements of a generic memory controller and their relationships.

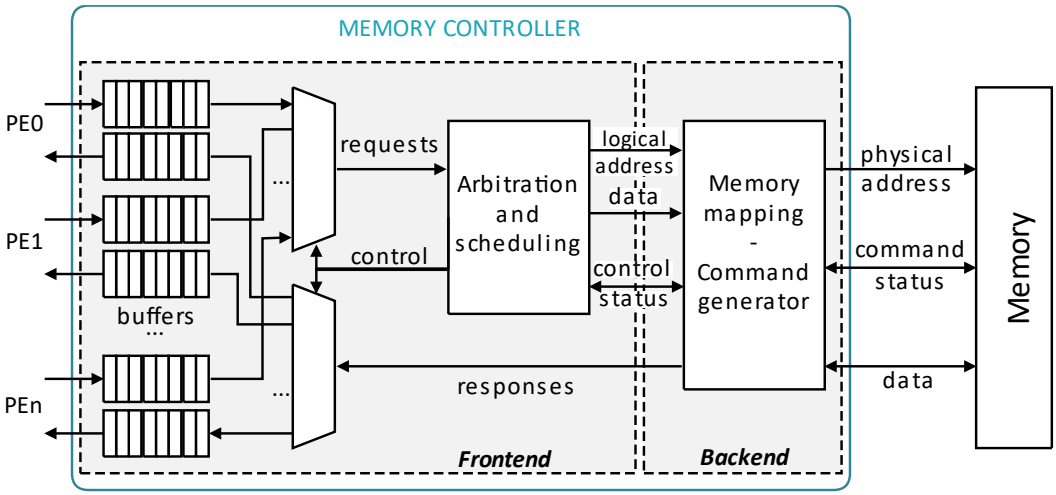


Figure 20. Abstract organization of a generic memory controller [Author].

The *Arbitration and scheduling* module executes transaction policies to define which request communicates first with memory since it is crucial in meeting latency requirements and real-time operations. Goossens et al. [79] explain that the *Arbitration and scheduling* module can be designed following a static or dynamic approach for reaching properties such as predictability, fairness, and flexibility, which are sometimes contradictory; e.g., fairness and flexibility are reduced to attain predictable transactions. A static memory controller schedules predictable transactions at design time, a vital requirement for real-time systems. However, the static approach requires well-known and specified data traffic; thus, bandwidth and latency are generally designed for the worst-case scenario. In dynamic memory controllers, scheduling requests are flexible and ordered at runtime, allowing for dealing with

strategies to improve performance and save energy but reducing predictability.

The memory controller Backend is usually composed of two modules – *Memory mapping* and *Command generator*. The *Memory mapping* module decodes the logical memory address provided by PEs into bank, row, and column, performing the physical address. The *Memory mapping* module can also implement access patterns, e.g., sequential or interleaving, that impact performance operation and energy consumption. The *Command generator* module produces commands customized for a particular memory technology to handle specific protocols that encompass timing, power, and bus-width requirements.

According to the requirements of the target application, a memory controller encloses other modules with characteristics that make them dependent on or independent of the memory technology, thus being implemented in the Frontend or Backend of the memory controller. This is the case of the project proposed in this work, which considers the reliability requirement; details of this module are described in Chapter 4.

Some of the techniques presented in this chapter are better used in the memory controller; the memory controller has access to all the memories and data before they are delivered to the CPU. For example, Jian and Kumar [80] present a technique for efficient memory error resilience for multi-channel memory systems, using parity and double memory redundancy to evaluate the memory. Duwe, Jin, and Kumar [81] employ BIST on the memory controller to evaluate the entire memory at each time to find any fault bit, stamping the area with the status, for when the data was read, perform a weak or strong ECC. Paul et al. [82] present a protection scheme that changes the ECC based on the number of errors; the proposed uses the BCH with interleaving to apply different lengths of BCH depending on the number of faults. Chapter 3 extensively demonstrates memory controllers with fault tolerance, which is the base of this thesis.

3 RELATED WORK

Memory reliability has received increased attention from the architecture community, with mutual interest between the academy and industry. Several works highlight the importance of memory reliability, but just a few propose memory controller approaches [64] with dynamism. Furthermore, most approaches are restricted to the cache hierarchy, leaving many innovation opportunities for the main memory controller [83]. This opportunity becomes even more evident in the fault tolerance scope since memory controllers manage many data of several application requirements.

This chapter positions our proposal by presenting and discussing schemes for main-memory controller reliability. Additionally, we split this chapter into static and dynamic reliability schemes. In static reliability, only one ECC can be chosen at the same level, and once selected, the ECC must remain the same until the entire system restarts. In dynamic reliability, each part of the memory can have a different type of ECC, and any ECC can be selected at runtime.

3.1 Static Error Protection Schemes for Memory Controllers

This section covers memory controllers that employ a static scheme to provide reliability at the cache level, main memory, or the entire memory hierarchy.

3.1.1 Choosing an Error Protection Scheme for a Microprocessor's L1 Data Cache [84]

Sadler and Sorin presented the Punctured ECC Recovery Cache (PERC) [84] that employs EDC on L1, and ECC for L1 and L2, as illustrated in Figure 21.

Each 32-bit word has seven bits for ECC and one for providing single-bit error detection (EDCp). L1 always checks EDCp and only executes the ECC in case of error occurrence; the PERC mechanism corrects the error reading of the data block by applying the error correction codeword (ECCp) from the PERC structure. Next, and in the case of non-error detection, the mechanism provides the corrected data word to the processor, stores the corrected data and EDCp in L1, and computes and stores ECCp in PERC structure.

In the case of L2 data written, the PERC mechanism reads the data word and EDCp from L1 and ECCp from the PERC structure. Subsequently, the mechanism writes this combined block into L2 and concatenates EDCp and ECCp in ECCnp. In case of an L1 miss, the PERC mechanism reads the corresponding block from L2, placing the data word

and EDCp in L1 and ECCp in the PERC structure.

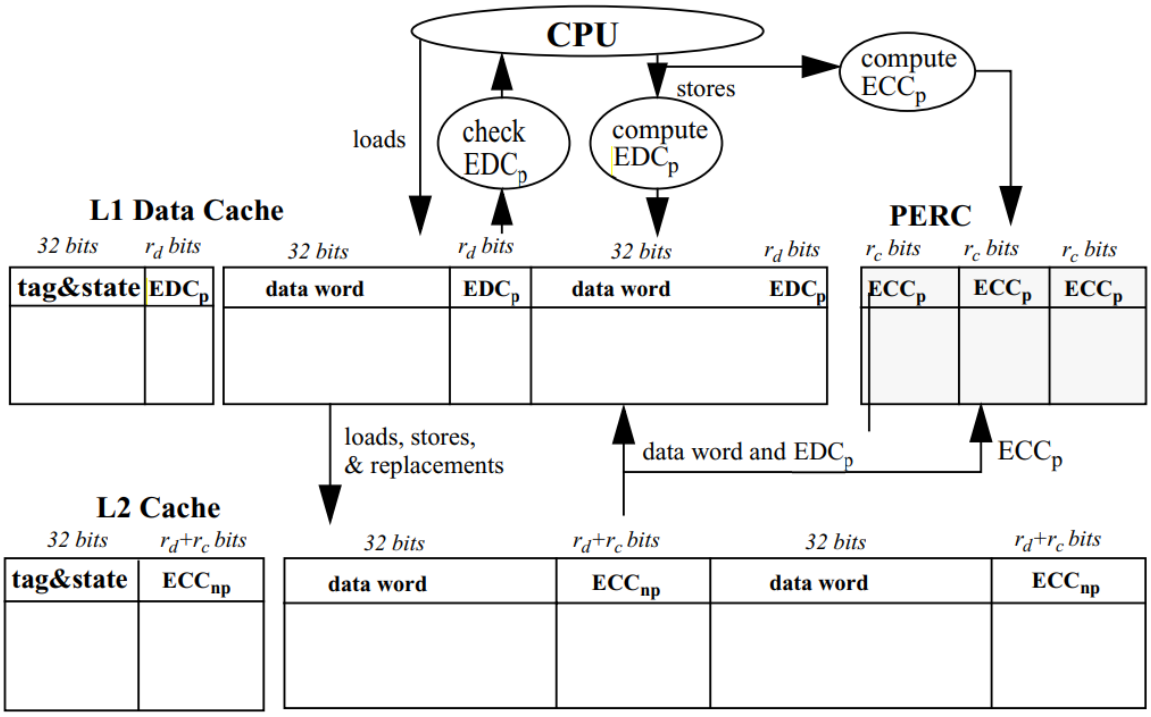


Figure 21. A simplified example of cache hierarchy with PERC [84].

The experiment demonstrates that the EDC model performs similarly to the unprotected scheme, increasing by 1% when checking the single-bit error detection and 10% when executing the ECC; besides, the power dissipation is increased by 24.9%, on average.

3.1.2 Memory Mapped ECC: Low-cost Error Protection for Last Level Caches [61]

Yoon and Eraz [61] present the Memory Mapped ECC (MM-ECC), a two-tiered protection mechanism to ensure the Last Level Cache (LLC) data integrity and achieve protection at a low cost. The First Tier Error Code (T1EC) accesses every read with a low latency and energy ECC decode, using parity for error detection and Hamming to fix it. The Second Tier Error Code (T2EC) is stored in a mapped DRAM region to be cacheable in the LLC; T2EC uses a DECTED to provide a strong error correction capability, but costly to compute and store.

When T1EC detects an error, the correction mechanism attempts to correct the error using T1EC, if possible. If T1EC cannot correct the error found, the cache line is clean, re-fetching the line from DRAM with the corresponding energy and latency penalty. T2EC maps the physical DRAM addresses to avoid the address translation when reading and writing. Figure 22 shows the operation in LLC with MM-ECC.

MM-ECC minimizes the need for a dedicated SRAM by maintaining ECC information within the memory hierarchy and reusing existing cache storage and control.

Compared to the MAXn [85], the area and power savings are significant, 15% and 8%, respectively, while performance is degraded by only 1.3% on average and less than 4%.

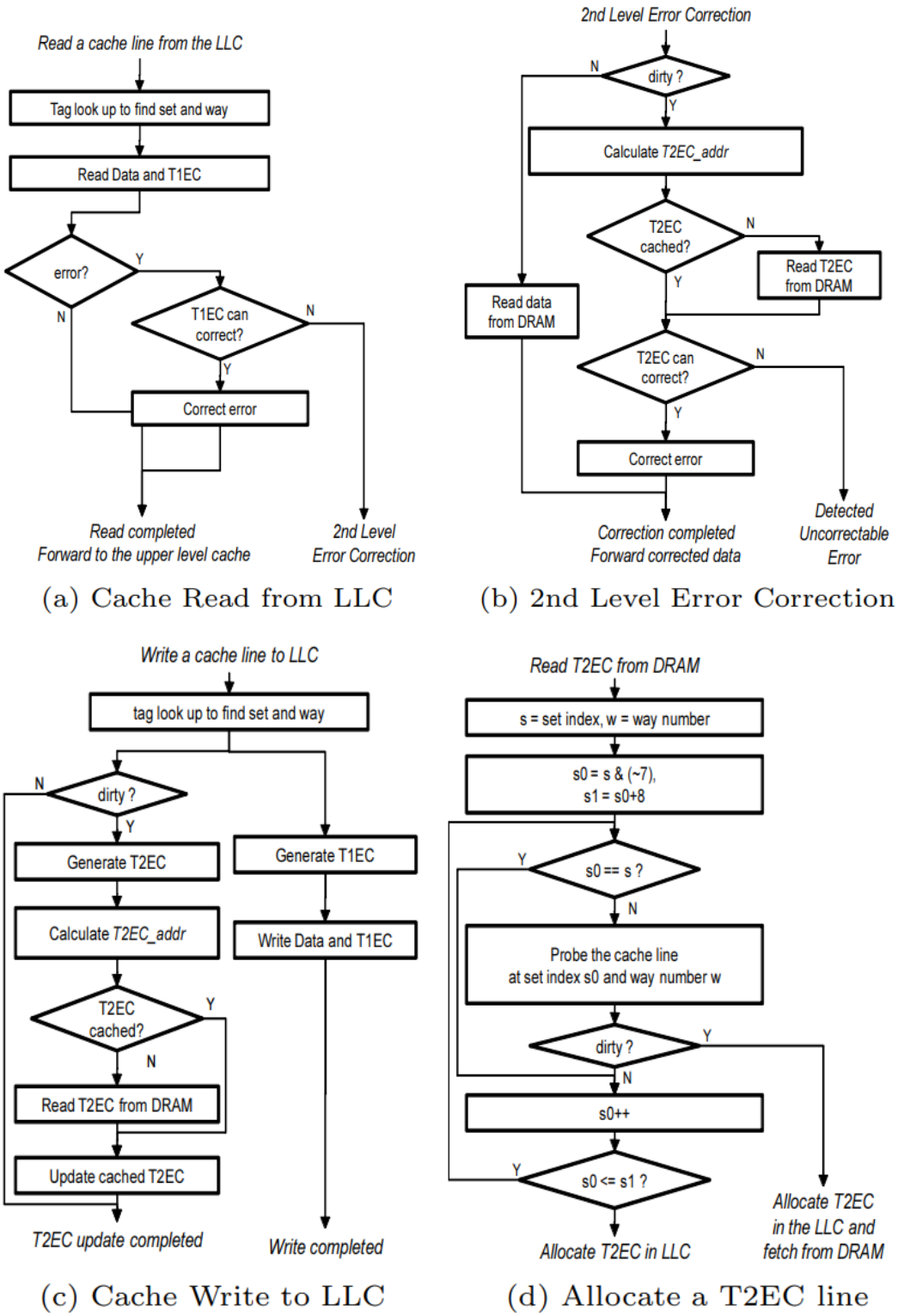


Figure 22. Operations in LLC with Memory Mapped ECC [61].

3.1.3 Reducing Cache Power with Low-cost, Multi-bit Error-Correcting Codes [56]

Technology advancements have enabled the integration of large on-die embedded DRAM (eDRAM) caches, which are significantly denser than traditional SRAMs but must be

periodically refreshed to retain data. Like SRAM, eDRAM is susceptible to devise variations, and refresh power potentially represents a significant fraction of overall system power, particularly during low-power states when the CPU is idle. Wilkerson et al. [56] proposed Hi-ECC, encompassing three techniques for reducing storage overhead, latency, and dynamic power, incorporated with multi-bit ECC to reduce refresh rate significantly. Hi-ECC avoids the decoder complexity by using strong ECCs to identify and disable cache sections with multi-bit failures while providing efficient single-bit error correction for the typical case.

The authors propose a system with 128MB eDRAM LLC, which must be refreshed every 30 microseconds leading to a significant amount of power dissipation. One alternative to reduce power dissipation is decreasing the refresh frequency; however, it requires tolerating more failures for each line. The Hi-ECC implements a 5-Error Correction, 6-Error Detection (5EC6ED) code on each 1KB line, requiring an additional 71 bits (0.87% overhead) to reduce the storage overhead. A hardware implementation of a 5EC6ED code is very complex and imposes a long decoding latency penalty. If full-strength encoding/decoding was required for every cache access, this could significantly increase cache access latency. However, the proposal leverages that error-prone cache portions can be disabled, avoiding the high decode latency during typical operations.

Figure 23 displays that the authors implemented a technique to reduce the average decoder latency by employing a simple ECC that performs one-error correction in a single cycle. However, if the line has more than one bitflip, the correction mechanism employs BCH, which has a high latency.

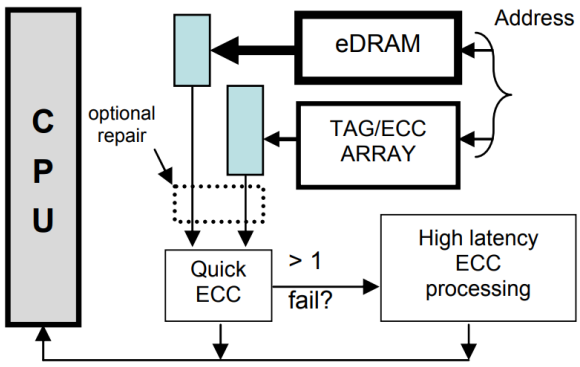


Figure 23. Block diagram for Hi-ECC [56].

The Hi-ECC challenge is reducing the dynamic power; while the L3 cache writes in 64 subblocks, the LLC operates with a 1KB line. It is necessary to perform a read-modify-write operation to modify a 64B sub-block in a 1KB line since it is needed to compute the ECC. Fortunately, as a linear code, BCH inherits the additive property of linear systems, ensuring that the ECC check bits can be updated using only the modified data block information.

Hi-ECC demonstrated a practical system for tolerating refresh-related failures in eDRAM-based caches. Hi-ECC reduces the cache refresh power by 93% compared to an eDRAM with no error correction and by 66% compared to a SECDED eDRAM and accomplished with only 2% storage overhead.

3.1.4 Virtualized ECC: Flexible Reliability in Main Memory [63]

Yoon and Eraz [63] extend the work in [62], which is an improvement of [61], presenting the Virtualized and Flexible ECC (VF-ECC) for Main Memory. These last articles explore the cooperative operating system and hardware techniques to maintain or improve error-protection levels while reducing energy consumption. The operating architecture mechanism virtualizes DRAM ECC protection and decouples redundant information mapping from data mapping. The two-tier protection developed for DRAM improves reliability guarantees and reduces energy consumption using wider-access configurations. Additionally, the mechanism provides ECC protection to systems with standard non-ECC DIMMs without changing the data mapping.

They proposed two mechanisms to achieve this: (i) An augmented virtual memory interface that allows a separate virtual-to-physical mapping for data and its associated redundant ECC information and (ii) the two-tiered protection mechanism. T1EC detects errors on every access, and T2EC is only needed when an error is detected. Figure 24 compares the traditional virtual memory (a) versus the (b) virtualized ECC architecture.

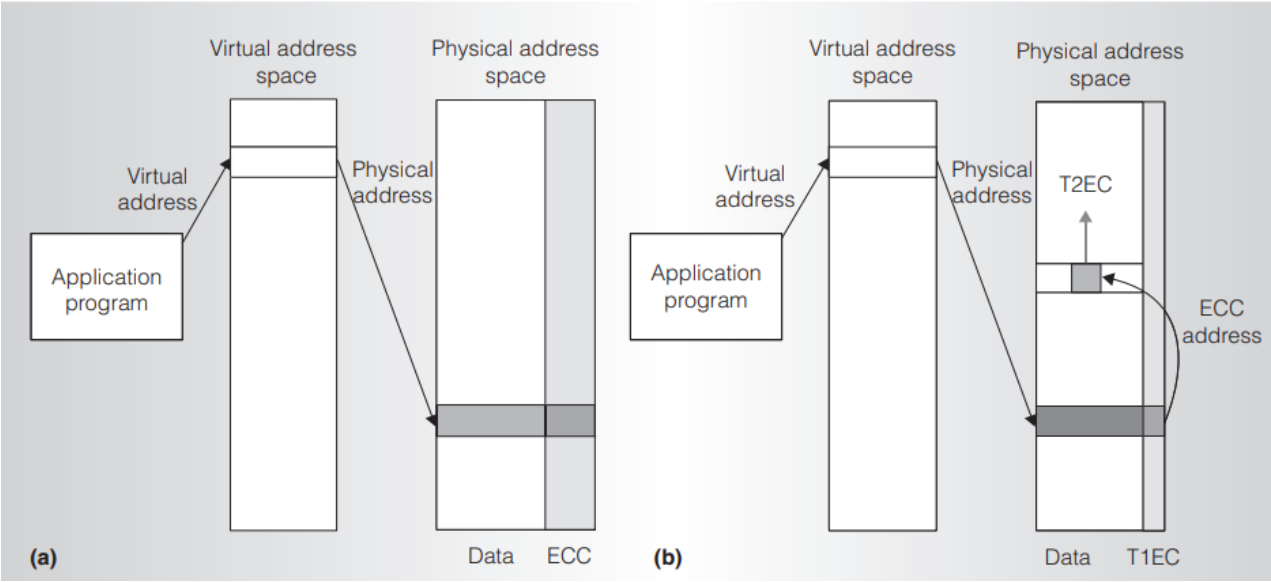


Figure 24. (a) traditional virtual memory versus the (b) virtualized ECC architecture [63].

Figure 24(a) displays that traditional virtual memory translates a virtual address from the application namespace to a physical address in DRAM. The DRAM access retrieves or writes both the data and the ECC information, which aligns with the data in the dedicated

ECC DRAM chips. On the virtualized ECC architecture, Figure 24(b) shows the OS and memory management unit maintaining the pair of mappings and ensuring that data and ECC match and are up-to-date. T2EC is rarely accessed to read the data; however, the data write requires second DRAM access, mitigated with an LLC to reduce the ECC traffic.

LLC improves the virtualized ECC performance by storing redundant information in the same physical namespace as data and contains an ECC address translation unit. Figure 25 illustrates the workflow of the Cache-DRAM Interface, where on operation 1, the memory controller fetches a data burst and its aligned T1EC from the main memory. When the information goes to LLC, the ECC is re-executed, and in case of error, the memory controller writes back the fixed information to the DRAM - operation 2. T2EC starts translating the data address to ECC address on operations 3 and 4. Errors in LLC can be ignored; the line is evicted and back to DRAM, as demonstrated in operation 5.

The proposal virtualizes an ECC with non-ECC DIMMs that only use the T2EC module; the ECC data is stored in another rank, and the ECC address is stored in LCC. The authors intend to bring dynamics to the project in the future by selecting different encodes.

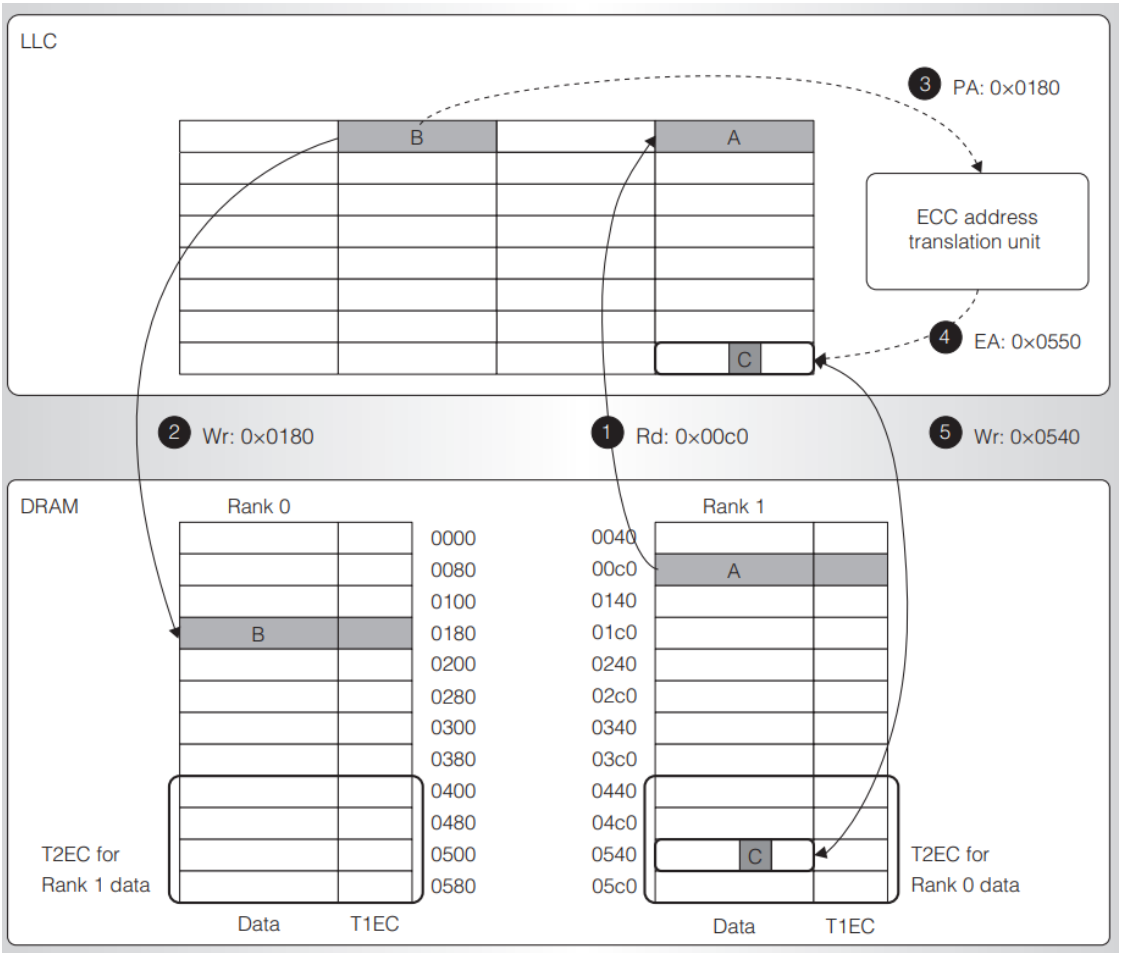


Figure 25. Dynamic RAM (DRAM) and LLC operations in a two-tiered virtualized ECC [63].

The proposal was evaluated using Chipkill, also testing the error correction for

systems that do not use ECC DIMMs. Overall, analysis of demanding SPEC[6] CPU 2006 [86] and PARSEC[7] [87][88] benchmarks indicates that the (i) performance overhead is only 1% with ECC DIMMs and less than 10% using standard Non-ECC DIMM configurations, (ii) DRAM power savings can be as high as 27%, and (iii) system energy-delay product is improved by 12% on average.

3.1.5 LOT-ECC: LOcalized and Tiered Reliability Mechanisms for Commodity Memory Systems [64]

Udipi et al. [64] describe memory reliability as a growing concern for modern servers. Memory protection mechanisms such as Chipkill imply costs like: (i) activation of many chips for each memory access, increasing power dissipation; (ii) performance reduction due to the reduced parallelism; and (iii) bandwidth waste due to the access granularity increase, challenging to use in small-bus computer systems. Therefore, the authors propose the LOcalized and Tiered ECC (LOT-ECC), displayed in Figure 26, a protection mechanism that solves these problems by separating error detection from correction, employing simple parity and checksum codes to provide fault tolerance with a straightforward implementation.

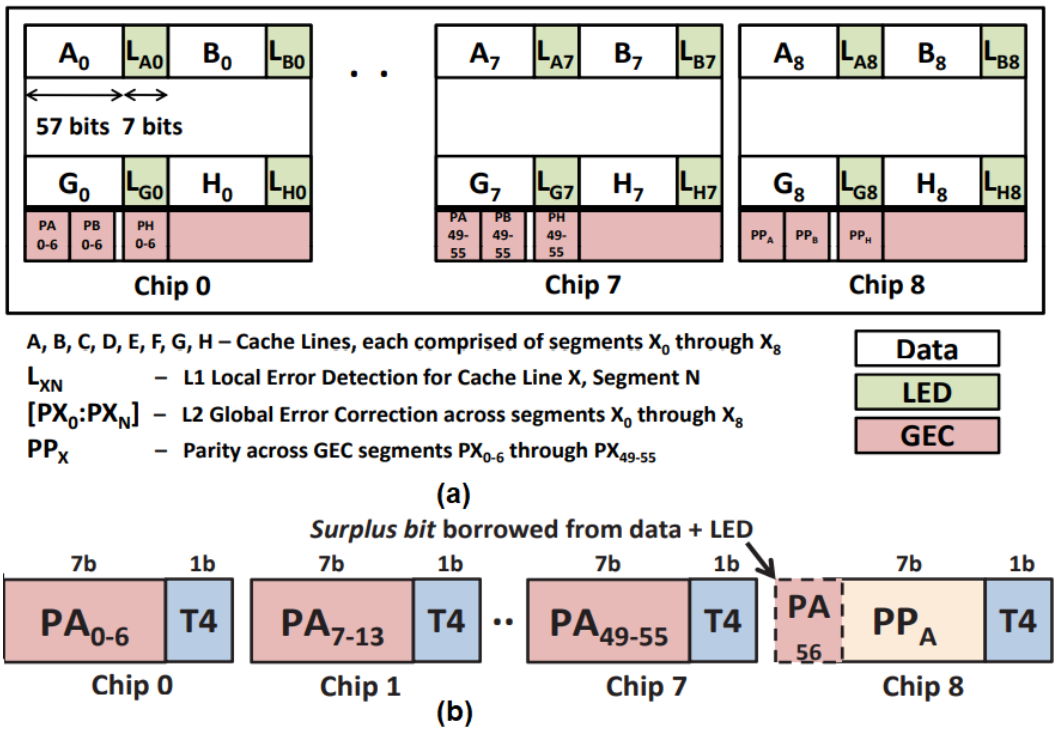


Figure 26. (a) LOT-ECC is shown with a single rank of nine ×8 DRAM chips and (b) data layout for one GEC cache line in the red-shaded GEC region [64].

LOT-ECC operates with local and global protection layers. Local Error Detection (LED) is the first layer with a 64-byte cache line composed of seven words of 64 bits each

[6] SPEC is the anacronym for Standard Performance Evaluation Corporation.
[7] PARSEC is the anacronym for Princeton Application Repository for Shared-Memory Computers.

to detect errors in every 57 bits of data through a 7-bit checksum. Global Error Correction (GEC) is the second layer of three tiers. The first tier contains a 57-bit entity, a column-wise XOR parity of the nine cache line segments with a related parity bit (T4) - e.g., cache line A has a GEC parity (PA), an XOR of data segments A0, A1, …, A8, enabling the data reconstruction from the GEC code, and this PA 7-bits has 1 bit of parity (T4). In the second tier, GEC parity is distributed for each cache line among all nine chips, where the nine chip receives a surplus bit. The last tier adds the PP_A, an XOR of eight other PA 7-bit fields, PA_{0-6}, …, PA_{49-55} to give the ability to recover the whole chip.

The authors developed an effective fault-tolerant mechanism for providing reliability guarantees. The proposed memory controller adds up to 136 bits having a storage overhead of 26.5%, where 2.5% is provided by the ninth chip on standard ECC DIMMs, and the other 14% is stored in data memory. Compared with Chipkill, they activated a few chips, reducing energy consumption and latency by up to 44.8% and 46.9%, respectively.

3.1.6 Low-power, Low-storage-overhead Chipkill Correct via Multi-line Error Correction [89]

Jian et al. [89] present the Multi-line ECC (ML-ECC) for avoiding the high overhead of error detection by replacing the error-correction method used in prior Chipkill with erasure correction. ML-ECC also provides error correction at a coarse granularity to reduce storage overhead - i.e., over groups of lines in memory.

The use of erasure correction over the original correction-check symbol per codeword on Chipkill increases the likelihood of correction, reducing error detection significantly. Erasure correction allows ML-ECC to correct single-symbol errors and detect double-symbol errors while using only nine devices per rank, considerably reducing the dynamic memory power dissipation. ML-ECC stores checksums in each rank device, sharing a common set of checksums across a group of codewords and letting every codeword belong to a column checksum group. Figure 27 shows that a column checksum group comprises a set of codewords in the same column and a set of checksums computed from these codewords. Despite ML-ECC keeping the overhead required for erasure correction and memory access, overall, ML-ECC incurs low dynamic power at low storage overhead (12.9%), with minimal reliability impact by sharing error-localization checksums. The evaluation using workloads across a wide range of memory access rates shows that ML-ECC reduces memory power dissipation by a mean of 27% and up to 38% compared to commercial Chipkill solutions.

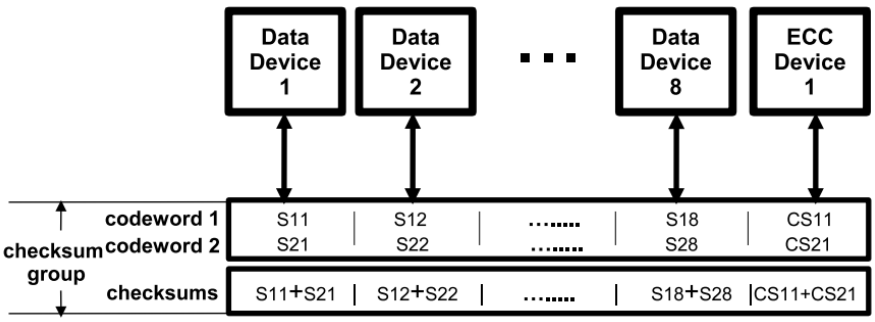


Figure 27. Checksum example; the symbol "+" represents the computation of checksums from data symbols, S is short for symbol, and CS stands for an erasure check symbol [89].

3.1.7 ECC Parity: A Technique for Efficient Memory Error Resilience for Multi-Channel Memory Systems [80]

Jian and Kumar [80] explain that servers often use a strong ECC to meet their reliability and availability requirements but imply power overheads. The authors observed that error correction typically needs to be performed for one channel at a time since memory channels are independent of one another. Based on this, they presented ECC Parity (ECC-P), an error resilience technique for multi-channel memories.

The authors proposed storing only the ECC bitwise parity of memory channels. When necessary, the actual ECC correction bits of a line in a faulty channel can be obtained by *XORing* the ECC-P line with the ECC correction bits of appropriate lines in the remaining healthy channels. The ECC correction bits are stored in memory only after a fault occurs in a channel; this protects against fault accumulation across multiple channels over time. In comparison, current systems always store every channel ECC in memory.

Figure 28 exemplifies their proposal, where *D* stands for the eight detection check symbols, *C* stands for the eight correction check symbols per line, and *C0C1C2* stands for C0 ⊕ C1 ⊕ C2. Shaded boxes represent values only calculated for the data lines but not actually stored in memory and are only used when a fault happens.

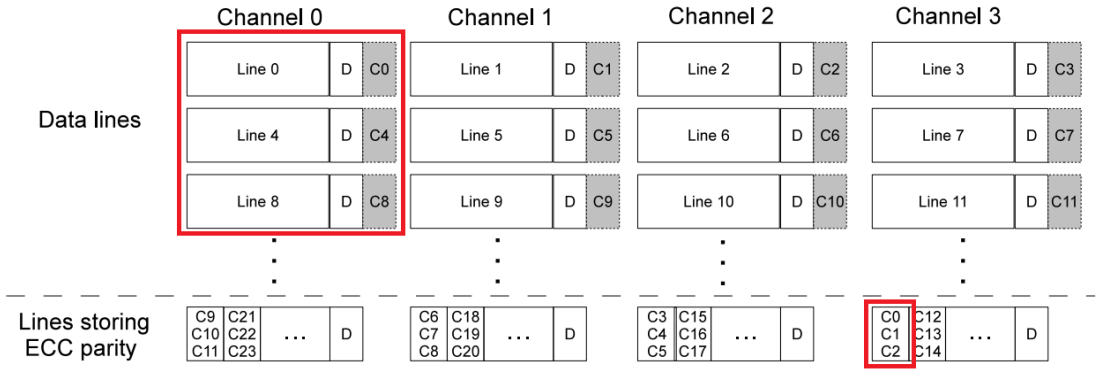


Figure 28. ECC-P example. *D* stands for the eight detection check symbols, and *C* stands for the eight correction check symbols per line, and *C0C1C2* stands for C0 ⊕ C1 ⊕ C2. Shaded boxes represent values only calculated for the data lines but not stored in memory [80].

The authors explain that ECC-P reduces memory system energy per instruction by 54.4% and 20.6%, on average, across two memory system configurations, compared to 36-device commercial Chipkill correct and commercial DIMM-kill correct, at similar or lower capacity overheads.

3.1.8 Correction Prediction: Reducing Error Correction Latency for On-chip Memories [81]

Duwe, Jian, and Kumar [81] explain that strong error correction often incurs a high latency relative to the on-chip memory access time. The authors proposed avoiding this problem with the *correction prediction* mechanism that forecasts the result of strong error correction, reducing latency and improving the L1 cache performance.

Their proposal stores all information in a Correction Prediction Table (CPT), which is much faster than accessing the L1 cache. Each entry corresponds to four words, totaling 128 bits. The CPT entry contains a *predFlag* and two Map Units, and each Map Unit can fix one bit. The *predFlag* indicates to the cache controller if it can perform the correction prediction or must proceed with the strong correction. The Map Unit contains *valid*, *location*, and *value*, where *valid* indicate if it has an error; in case of error, the address and the bit position are fulfilled. Figure 29 shows the CPT format, corresponding to four 32-bit words.

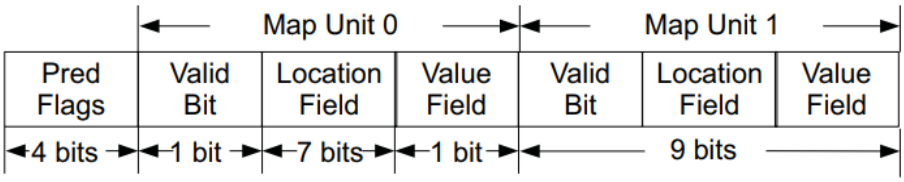


Figure 29. CPT Entry Format. Each table entry corresponds to four 32-bit words [81].

CPT has a BIST routine to test the L1 cache and populate CPT in runtime. The BIST routine first tests the two Map Units of the CPT entry, then verifies the four cached words to identify as many faulty bit locations are in the valid Map Units. If the routine finds any faulty bit, it sets to false the valid bit of the Map Unit.

When the pipeline requests any word, the L1 cache performs the normal access, reading the CPT entry corresponding to the word accessed in parallel. In the prediction case, the cache applies the weak error correction, delivers the result to the pipeline, and executes the factual correction to evaluate if a *misprediction* occurred. In case of still an error, the pipeline squashes all dependent instructions, returning the correct value to the pipeline and restarting the instruction that initiated the cache request with the correct value. When the prediction is not performed, only a strong correction is applied. Figure 30(a) illustrates the L1 cache modifications, and Figure 30(b) the prediction workflow in an L1 cache access.

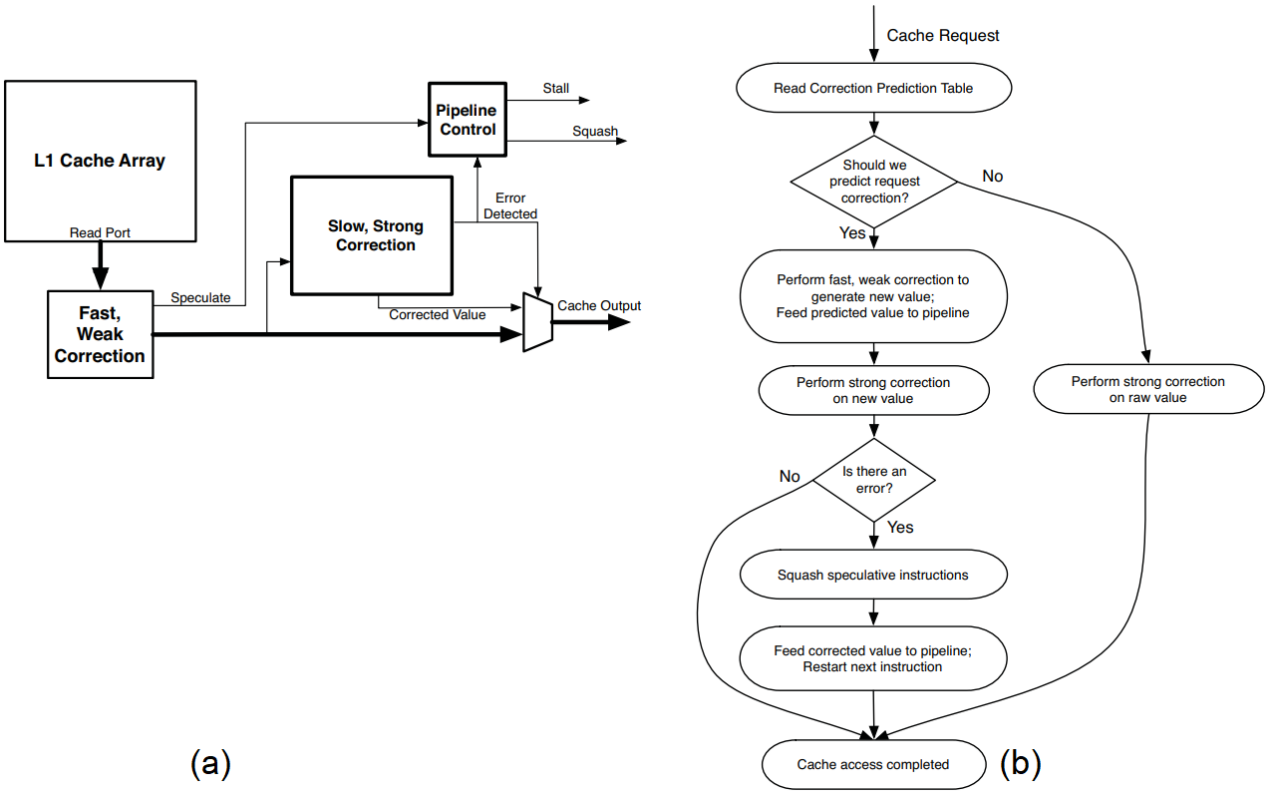


Figure 30. (a) Modifications to L1 caches; the critical path for the common case of correct prediction is in bold, (b) correction prediction for an L1 cache access [81].

Their proposal was evaluated with a 32KB SRAM, 4-way, set associative, L1 cache, which shows that the proposed implementation reduces the average access latency by 38%-52% and reduces the energy of a 2-issue in-order core by 16%-21% when compared to the strong correction alone. The *misprediction* rate is less than 0.1%, increase less than 10% area, and has less than 2.5% worst-case latency to a cache with strong error correction.

3.1.9 Bamboo ECC: Strong, Safe, and Flexible Codes for Reliable Computer Memory [90]

Large-scale systems often employ redundant storage, error checking, and correction codes to achieve reliability and higher bandwidth. Kim, Sullivan, and Erez [90] proposed the Bamboo ECC that provides significantly stronger protection than the current state-of-the-art ECC mechanisms, requiring the same or less redundant storage and off-chip bandwidth. Bamboo ECC groups per-pin data as symbols and uses an 8-bit symbol RS code to provide strong pin and error chip protection. The authors present five organizations that can meet different memory system constraints and reliability requirements.

The *Single Pin Correcting* (SPC) organization can correct a bit or a pinning error with two redundant pins. SPC requires just a quarter of the redundant storage of SECDED on a 64b data channel – i.e., 3.1% vs. 12.5%, yet it provides a better uncorrectable error rate. The second organization proposed can be used as either a *Double Pin Correcting*

(DPC) or *Single Pin Correcting - Triple Pin Detecting* (SPC-TPD) scheme. SPC-TPD has a very high detection coverage, reaching 100% of up-to-3-pin detection errors. The *Quadruple Pin Correcting* (QPC) organization provides the same Chipkill protection with equal or less redundancy. *Octuple Pin Correcting* (OPC) achieves eight pin corrections with 16 redundant pins, resulting in a 25% overhead on a 64-bit data channel or a 12.5% overhead on a 128-bit data channel. Finally, the *Double Double-Pin Correcting* (DDPC) organization can correct two ×4 errors or one ×8 error by correcting four 2-pin symbols, requiring 6.3%/12.5% storage overheads on the 128-bit data channel that is typical of wide on-package memories.

Figure 31 illustrates the current schemes used by the industry, and Figure 32 and Figure 33 show the organization proposed. The proposed organization evaluates on a DDR 64-bit data channel the SECDED against SPC-TPD and the AMD Chipkill against QPC. On the Wide-IO 128-bit data channel, was evaluated AMD Chipkill and DDPC, and doubled AMD Chipkill vs. Quadruple Double Pin Correcting (QDPC) to protect the 4-bit subarray. The results show significant error coverage and memory lifespan improvements of Bamboo ECC relative to existing SECDED, Chipkill-correct, and double-Chipkill-correct schemes.

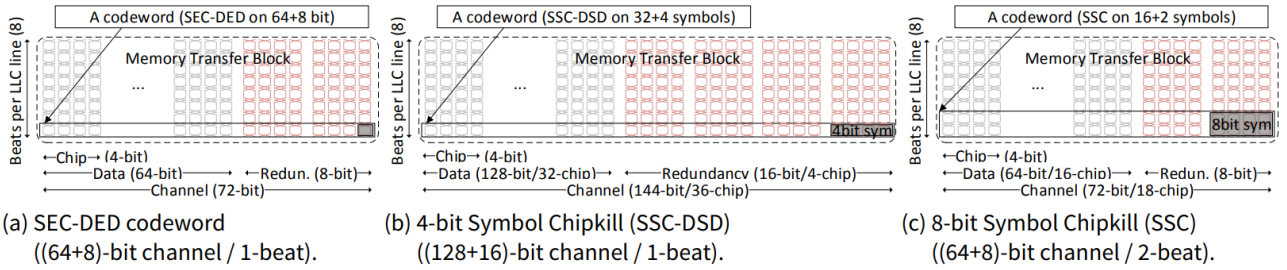


Figure 31. The codeword layout of ECC schemes that are currently in use [90].

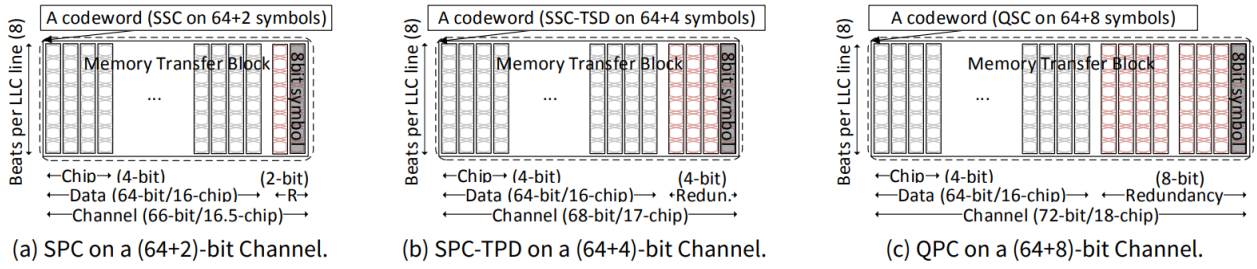


Figure 32. Bamboo ECC layouts on a 64b data channel (with an 8b burst length) [90].

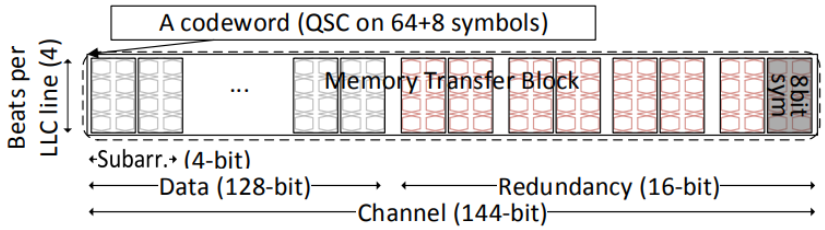


Figure 33. QDPC on a 128b data channel (2-pin ×4-beat symbols) [90].

3.1.10 Adaptive ECC Scheme for Hybrid SSD's [91]

Hsieh, Chen and Lin [91] propose an adaptive scheme with four ECC levels to enhance the Solid State Drive (SSD) reliability. Their scheme contains (i) a Multi-Level Cell (MLC) flash memory being the major data storage medium due to its lower cost and higher density; and (ii) a Single-Level Cell (SLC) flash memory to ECC storage due to the higher endurance and lower disturb fault rate compared to MLC.

The ECC encoder/decoder employs an adaptive BCH engine [92] to provide 9, 14, 19, or 24 bits error correction capability per 512 bytes according to the error rate of an MLC page. The 4-bit ECC protects the ECC generated by the adaptive ECC encoder/decoder. After ECC has been generated, the 4-bit ECC encoder generates the ECC parity. Both ECC codes are written into the SLC chip. The ECC mapping scheme maintains a table to keep track of the mapping between the data and the corresponding ECC.

Figure 34(a) shows the MLC chip and the ECC mapping table indexed by Physical Block Address (PBA). Each entry in the ECC mapping table records the Physical Page Address (PPA) of the associated ECC sector, which is located on the first page of the corresponding ECC area for the mapped MLC block. The block mapping table and the ECC mapping table are maintained in RAM. The ECC area is composed of continuous SLC pages. The scheme groups a fixed number of MLC pages as a mapping unit, and the ECC for each page in the mapping unit is collected in the associated ECC area due to the space efficiency concern. The ECC sector maintains the management information of the ECC for each page in the mapping unit.

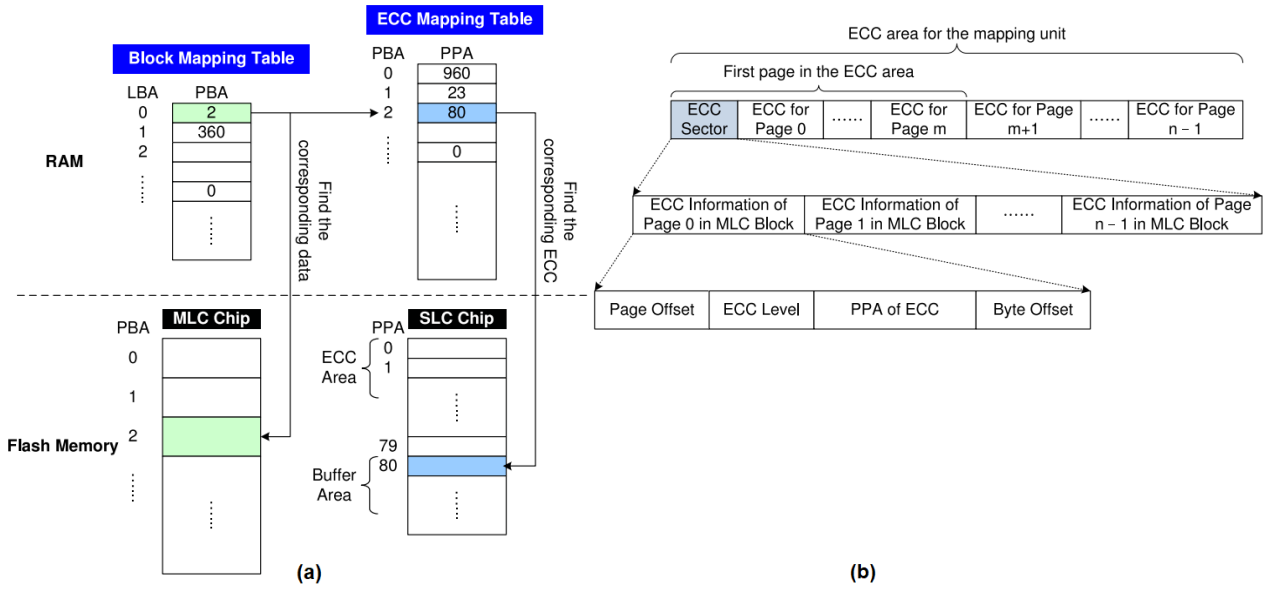


Figure 34. (a) Block and ECC mapping tables and (b) ECC sector organization [91][92].

Figure 34(b) shows the layout of an ECC sector. The management information of ECC for each page contains four fields: (i) The page offset, which ranges from 0 to n-1,

records the offset of the target page in the requested MLC physical block, where *n* is the total number of pages in the management unit. (ii) The ECC level records the current error correction capability maintained for the target MLC page, which ranges from 0 to 3 for 9-bit, 14-bit, 19-bit, or 24-bit error correction capability. (iii) PPA records the associated ECC physical page address in SLC. This field can avoid a linear search in finding the ECC for the MLC page. (iv) The byte offset records the starting location of the associated ECC within the page. Since each MLC page in the mapping unit might differ in ECC level, the byte offset helps locate the required ECC directly.

The ECC and block mapping tables require about 1.2 MB of memory space for a 128 GB MLC SSD with 8 GB SLC. The authors conducted a series of trace-driven simulations to evaluate the extended lifetime of SSDs, and the experiment results show that a 32 GB MLC-based SSD managed by our scheme with 1 GB SLC could extend its lifetime up to 318% with a write amplification of 1.06 for writes to the SLC, while the static wear leveling for SLC can be achieved.

3.1.11 Using Low Cost Erasure and Error Correction Schemes to Improve Reliability of Commodity DRAM Systems [93]

Chen et al. [93] describe that most server-grade systems provide Chipkill-correct error protection at the expense of power and performance. Therefore, the authors evaluated and proposed five Erasure and Error Correction Codes (E-ECC) that provide at least the same Chipkill protection with low power and performance overhead.

Three schemes were proposed for ×4 DRAM systems: (i) a rotational (144, 128) code [58] as a representative Chipkill code; (ii) a scheme that changes the rotational code to process in four-part using Reed Solomon (RS) (36, 32) over the Galois Field (GF) (2^8); and (iii) a scheme that uses sixteen parts of RS (9, 8) to get the 144 necessary to give strong reliability in 64 bits. Besides, they proposed the use of the RS (36, 32) code for ×8 DRAM systems and a scheme based on the RS (20,16) code over GF(2^8) for ×16 DRAM systems.

The three schemes on the ×4 DRAM increase 12.5% the storage overhead but differ in the number of ranks activated (one or two). The schemes that activate two ranks per memory access have lower timing, power, and energy performance but higher reliability than those that activate only one rank. The ×8 DRAM system has the lowest power dissipation and highest energy efficiency among all five schemes; the ×16 DRAM system has a storage overhead of 25% and has the highest timing performance among all the schemes. The authors also compare the five schemes to other ECC schemes, demonstrating that the E-ECC schemes use more logic die area but have the lowest storage and infrastructure

overhead compared to the existing schemes.

3.1.12 Lifetime Adaptive ECC in NAND Flash Page Management [94]

Wang et al. [94] proposed the Lifetime Adaptive ECC in NAND Flash Page Management (LAE-FTL). The authors explain that the storage reliability of NAND flash memory decreases as the density or program/erase cycle increases.

Based on the above observation, the LAE-FTL system was divided into two stages: (i) responsible for selecting the maximum error correction capability in the segment, and (ii) adaptively adjusting ECC redundancies as needed to compensate for the increasing Raw Bit Error Rate (RBER). When the program/erase cycle is relatively small and the Out-Of-Band (OOB) area is large enough, it uses the traditional ECC scheme or BCH with different sizes. When a better ECC scheme is necessary, the data user is squeezed, and the ECC uses the user area. The squeezing data space makes user data of 4KB distributed in two different units, even on two different pages. Figure 35 illustrates the 4k flash page layout with the adaptative size of the OOB area.

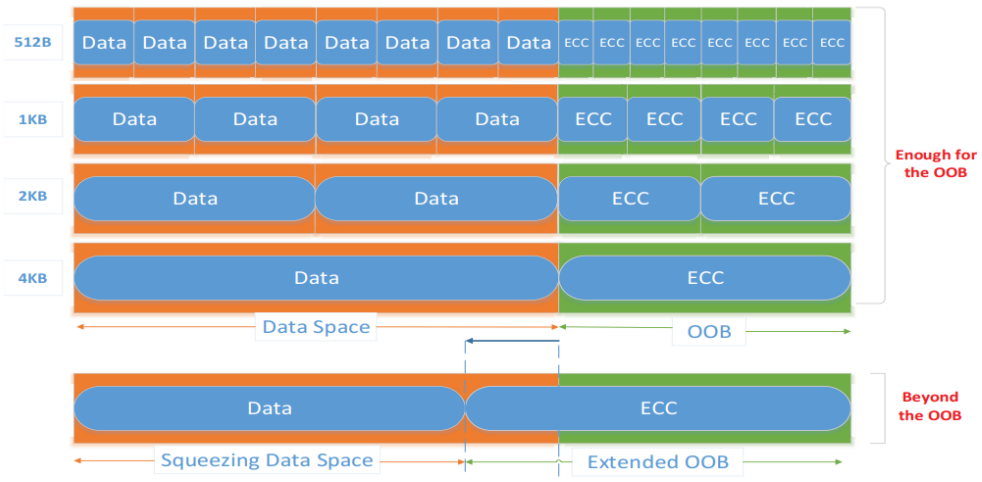


Figure 35. 4KB flash page layout with adaptive OOB sizes [94].

The results demonstrate that LAE-FTL squeezes the data space to store the enhanced ECC redundancy when it becomes too large to fit in the OOB area. Consequently, LAE-FTL improves the lifetime reliability with low accessing cost and the read performance by up to 63,42% at the early stage compared to the worst case.

3.2 Dynamic Error Protection Schemes for Memory Controllers

This section covers memory controllers that employ a dynamic scheme to provide reliability at the cache level, main memory, or the entire memory hierarchy.

3.2.1 Reliability-Driven ECC Allocation for Multiple Bit Error Resilience in Processor Cache [82]

Paul et al. [82] present the Reliability-Driven ECC (RD-ECC) to protect the cache from runtime failures induced by soft errors, voltage or thermal noise, and aging effects. The authors implemented a 2 MB L2 cache memory with 256 memory blocks of 8 KB each, accessed by an associative cache with 8-way - each way has 64 bits, totaling 512 bits per access. RD-ECC uses the shortened BCH cyclic code with zero padding, providing a high random error correction capability with a modest amount of check bits; for tolerate (t) 2 or 3 bitflips in a 64-bit word, BCH needs 14- or 21-bits, respectively. The traditional extra 8 bits used for Hamming are stored separately from the associate cache. Considering the 8-way associative cache, the BCH with t=3 needs more 1-way or 2-way than t=2, reducing the data from 512- to 448- or 384-bits. Figure 36(b) shows the proposed architecture.

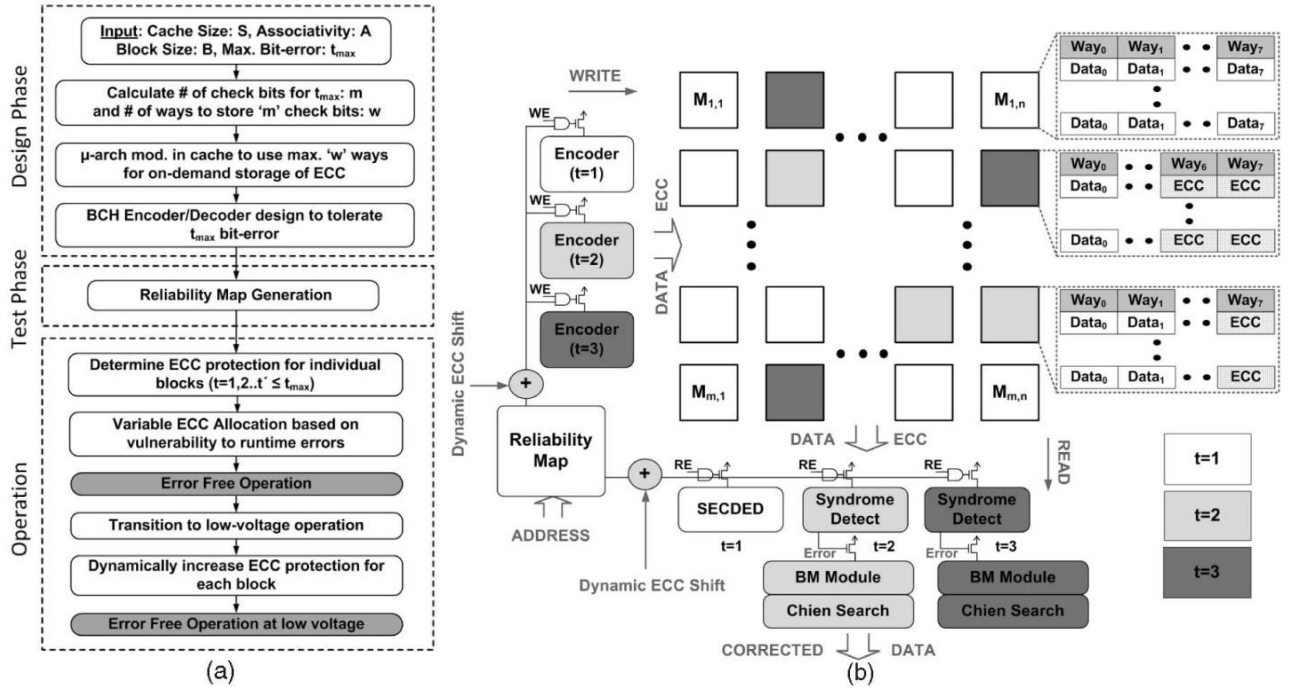


Figure 36. (a) Major steps in variable ECC allocation. (b) Architecture for post-fabrication variable ECC allocation based on the process corner of the individual memory blocks [82].

Two approaches were proposed for mapping the memory blocks: (i) mapping during the design phase and storing the reliability map with a ROM or (ii) mapping during the design phase and storing the reliability map with non-volatile memory to change at runtime based on the maximum bitflip tolerated by the ECC encoding. The ECC encoding is changed by evaluating if the number of errors is more than the ECC can tolerate; due to the fast change on the L2 cache, do not execute recoding. The dynamic adaptation was used to save power by combining voltage scaling for the memory cells with higher ECC protection, illustrated in Figure 36(a).

This article analyzed many error rates with different combinations of techniques, like bit interleaving and voltage scaling. In all cases, demonstrate efficient circuit/architecture-level optimizations of the ECC encoding/decoding logic to minimize the impact on the area, performance, and energy. Simulation results for SPEC2000 benchmarks show that such a variable ECC scheme tolerates high error rates with negligible performance (4%) and area (0.2%) penalty.

3.2.2 MAGE: Adaptive Granularity and ECC for Resilient and Power Efficient Memory Systems [95]

Li et al. [95] mention that resiliency is one of the toughest challenges in HPC, and memory accounts for a significant fraction of errors. Providing strong error tolerance in memory usually requires a wide memory channel that incurs a large access granularity. They proposed a Memory system with Adaptive Granularity and ECC (MAGE) to achieve high performance, power efficiency, and resiliency. MAGE can adapt memory access granularities and ECC schemes (with straightforward SECDEC and Chipkill codes) to applications with different memory behaviors.

MAGE is a hardware-software collaborative solution consisting of: (i) a smart memory controller that manages data layout, memory scheduling, and memory channel integration; (ii) a combination of modified decoupled sector cache and pool-of-subsectors cache that manages data with mixed access granularities in the cache hierarchy; and (iii) software support that identifies per-page or per-application access granularity and manages multi-granularity physical memory. The modifications to the system stack are minor and can be easily implemented in current mainstream systems.

Figure 37 depicts an overview of the MAGE solution applied to a multicore processor, where all cores and caches share multi-channel memory controllers via an on-chip network. Each physical memory channel has a standard 72-bit wide bus, with 64-bit data and 8-bit ECC. MAGE architecture supports three memory modes for the memory systems constructed with x4, x8, and x16 DRAM devices. The OS can choose three granularity modes: (i) fine-grained - each physical memory channel is used as a 64-bit wide logical channel, enabling 64B memory access; (ii) medium-grained - two physical memory channels are connected in lock-step to construct a 128-bit wide logic channel providing stronger memory protection than the fine-grained mode; and (iii) coarse-grained - four physical memory channels are used as a 256-bit wide logical channel, enabling 256B memory access while providing stronger memory protection than the medium-grained mode.

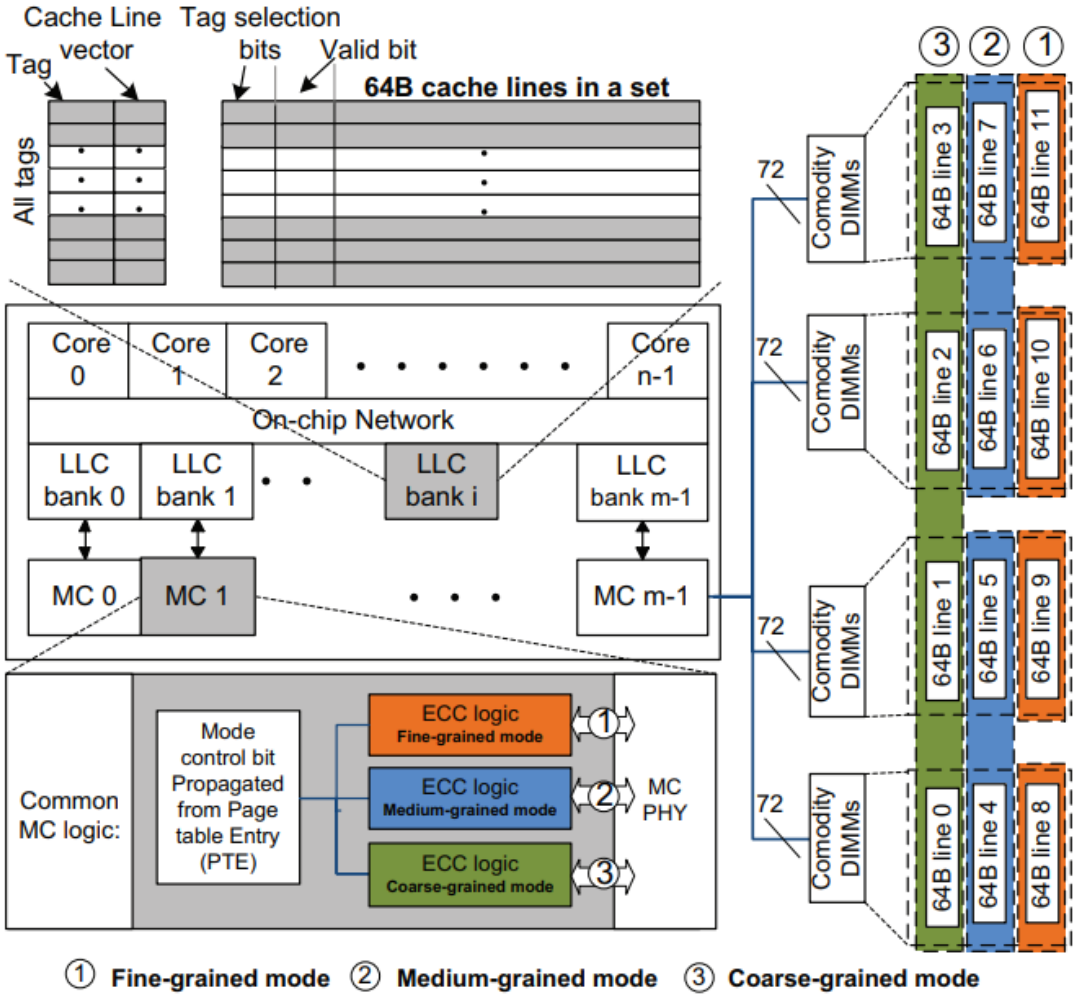


Figure 37. MAGE architecture overview. LLCs can be shared or private, and memory controllers can be attached through either LLCs or directly through the NoC [95].

MAGE manages memory pages and segments and translates memory addresses adaptively. The virtual memory manager operates the mode information for each physical memory page and propagates this information through the memory hierarchy to the memory controller, allowing the optimum access granularity and ECC scheme for each memory access. MAGE allows the adaptation for each memory access; however, it incurs a significant storage overhead for managing per-block mode information.

The experimental results demonstrate that MAGE enables an adaptive selection of appropriate modes that combine access granularities and ECC schemes for applications with different memory behaviors. MAGE concurrently satisfies three key requirements: improved performance, power efficiency, and resiliency for large-scale computing systems.

3.2.3 Reconfigurable ECC for Adaptive Protection of Memory [96]

Basak et al. [96] expand the evaluation of Paul et al. [82] in the Reconfigurable ECC for Adaptive Protection of Memory (RAPM) to change the ECC by adding spatial and temporal variation. The authors explain that spatial evaluation is necessary due to inter and

intra-die variations; as demonstrated in [82], there are 7-10x more variations in reliability across memory blocks. The proposal evaluates the spatial condition to generate the block reliability map during the manufacturing test. The work of Basak et al. demonstrated an improvement in the reliability-aware in diverse applications with the pre-define map based on the spatial condition.

3.2.4 Adaptive Reliability Chipkill Correct (ARCC) [97]

Jian and Kumar [97] optimize the straightforward Chipkill solutions with the Adaptive Reliability Chipkill Correct (ARCC), which maintains similar reliability as a stronger Chipkill correct solution but consumes less energy. ARCC is based on the observation that, on average, only a tiny fraction of memory experiences any fault during the typical operational lifespan of a server. Therefore, ARCC proposes applying weaker but more energy-efficient ECC for regions in the main memory that are fault free and dynamically increasing the ECC strength of a region in the main memory after detecting faults in the memory region.

ARCC operates a *relaxed* page that consists of two check symbols per codeword in 64 bits per line and an upgraded page with four check symbols per codeword in 128 bits per line, as demonstrated in Figure 38. When an error is detected during memory scrubbing, ARCC increases the protection strength of the page with an error by increasing the number of check symbols per codeword from 2 to 4, combining two adjacent 64-bit lines.

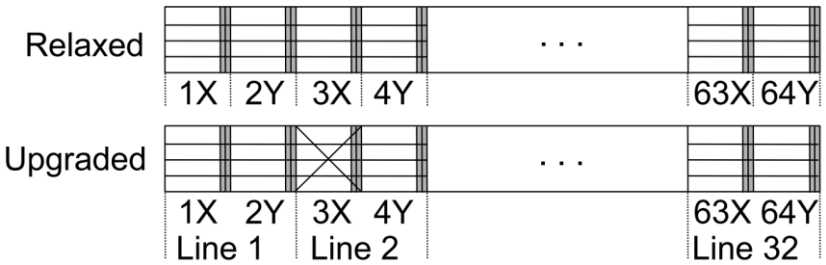


Figure 38. ARCC page operations using two (relaxed) or four (upgraded) codewords [97].

ARCC upgrades the Chipkill correction strength after faults are detected in a page during memory scrubbing, which executes the following steps:

i. Read a line and store its value aside.

ii. Write all 0s to the line location in memory and then read the location in memory to see if only 0s are returned. If true, go to step iii. If false, a stuck-at-1 fault may be present; go to step iv and upgrade the page afterward.

iii. Write all 1s to the line and then read the line to see if only 1s are returned. If true, go to step iv. If false, a stuck-at-0 fault may be present; go to step iv and upgrade the page afterward.

iv. Correct any errors in the original line content and write the line back to memory.

This implementation reduces memory power dissipation by 36% and improves performance by 5.9%, on average, when applied to commercial Chipkill solutions with negligible reliability degradation.

3.2.5 VL-ECC: Variable Data-Length Error Correction Code for Embedded Memory in DSP Applications [98]

J. Park, J. Park, and S. Bhunia [98] explain that the vulnerability of SRAM cells to failures becomes more complex when errors occur within the Most Significant Bits (MSB) in Digital Signal Processing (DSP) applications. Errors on MSB give rise to much more extensive data quality degradation than the failures in the least Significant Bits (LSB).

To avoid this problem, the authors present the Variable data-Length ECC (VL-ECC) approach, where codeword length is dynamically reconfigured to protect MSB preferentially using BCH. When the number of SRAM failures in a code word exceeds the error correction capability in low voltage operation, the input data length of VL-ECC is reduced to focus on the more MSB, as illustrated in Figure 39.

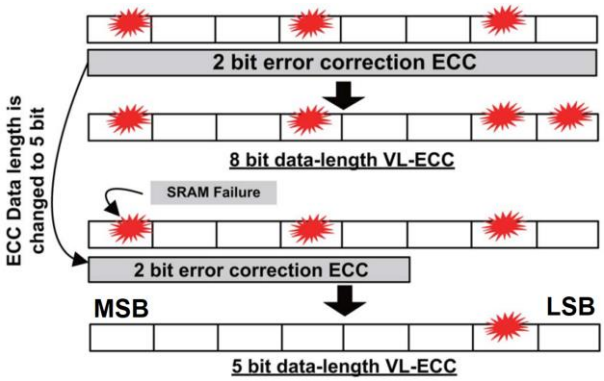


Figure 39. Overview of the proposed VL-ECC approach [98].

The proposed VL-ECC scheme shows significantly better system output quality than the conventional ECC approach in the H.264 and Fast Fourier Transform cases. The optimal data length of the VL-ECC under different supply voltages to minimize the output quality degradations due to memory failures. Experimental results with 65nm CMOS technology showed that the proposed VL-ECC scheme could tolerate a high failure rate at low power with graceful degradation in output quality.

3.2.6 An Adaptive ECC Scheme for Dynamic Protection of NAND Flash Memories [99]

Yuan et al. [99] describe that basic ECC methods fail to consider the variable reliability types, resulting in high waste of computations. To achieve better implementation, they proposed the Adaptive ECC Scheme for Dynamic Protection (AES-DP) of NAND Flash

Memories using Hamming and BCH codes.

AES-DP evaluates Program/Erase (P/E) cycle count and retention time to have a long-term accumulation influence on reliability [101]. The effects of these two factors are additive; each increase can result in a more significant error rate [102]; both factors are evaluated separately to discover the appropriate ECC and avoid this problem.

The threshold method illustrated in Figure 40(a) is divided into three levels of retention time and P/E cycle count: level 1 uses Hamming; level 2 uses BCH (4122, 4096, 5), and level 3 uses BCH (4148, 4096, 9). The initial ECC level is 1; in the first evaluation, if the P/E cycle count is greater than the threshold, the ECC level adds 1; afterward, if the prospective retention time is greater than the threshold, the ECC level adds another 1. After the two comparisons, this ECC selection is the output of the adaptive ECC encoder illustrated in Figure 40(b). The adaptative ECC encoder determines which level of ECC should be chosen; apply Hamming encoder if the level is 1; otherwise, the adaptive BCH encoder is invoked.

The adaptive ECC decoder uses the number of ECC check bits to determine which ECC is selected; if it is 24, the Hamming decoder is selected; If the number of check bits is 26, use 2-bit BCH; otherwise, the 4-bit BCH is selected.

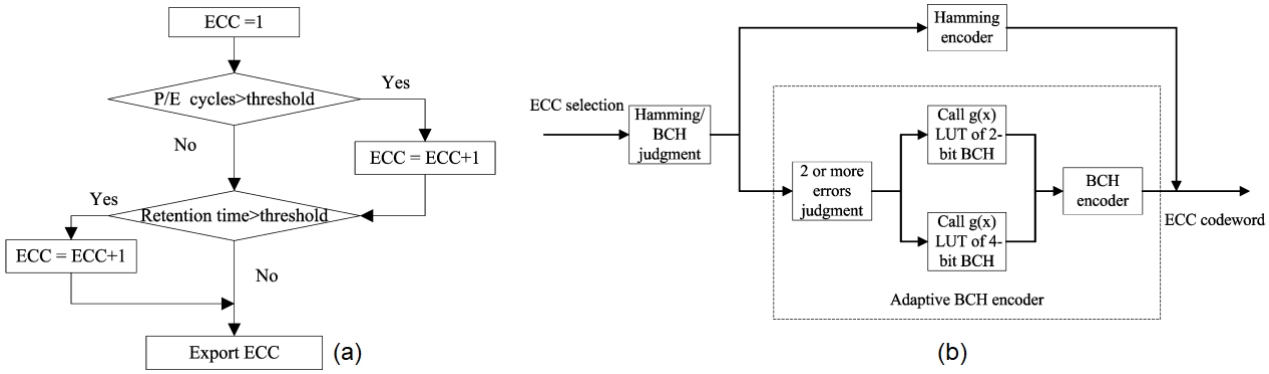


Figure 40. (a) Major steps to determine the ECC and (b) in the adaptive ECC encoder [99].

The authors demonstrate good predictability with the retention time and the P/E cycles. Several error models evaluated the feasibility of the proposed to simulate typical stages of running conditions and different lifetimes of P/E cycles, demonstrating better results than regular ECC methods in most situations. Through the proposed ECC scheme, it was possible to avoid the waste of error-correcting capability and reduce the coding time to ensure the throughput fits real-time requirements.

3.2.7 Adaptive ECC for Tailored Protection of Nanoscale Memory [100]

Shin et al. [100] improve the works [82][96][98] by proposing Variable Capability ECC (VC-ECC). Their work focuses on DSP and evaluates static, spatial, and temporal

variations in manufacturing time, as exposed in Figure 41(a). Besides, Figure 41(b) shows they evaluate the number of bitflips prioritizing the ECC MSB in runtime.

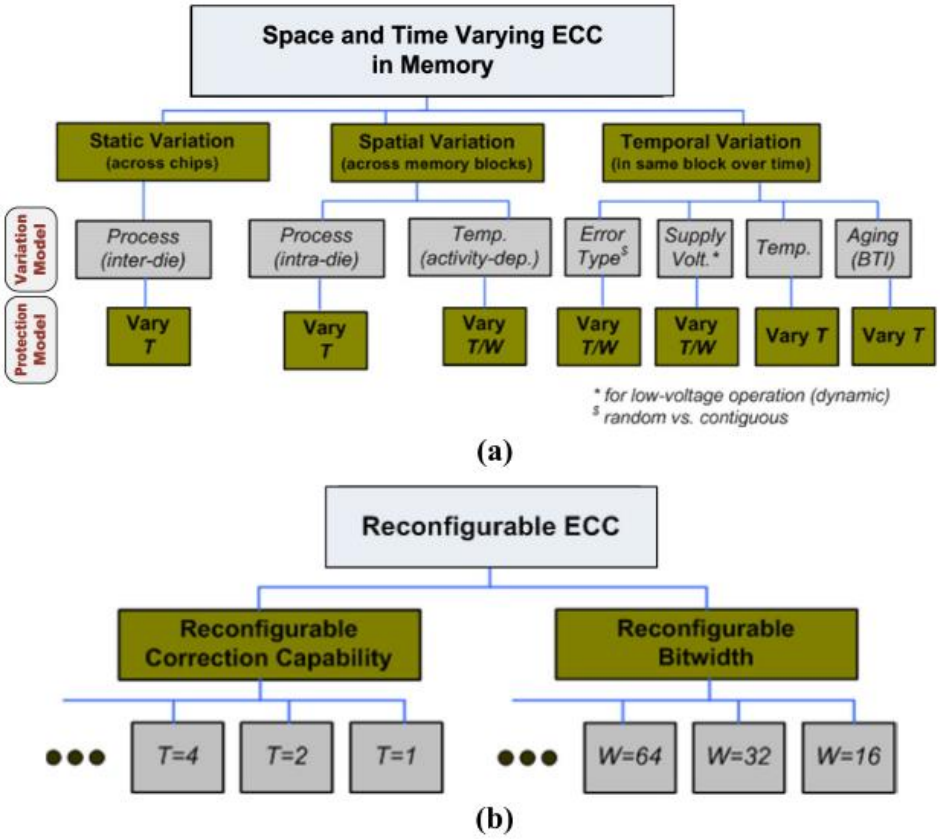


Figure 41. (a) Scheme for the proposed variable error correction. The correction capability changes over space and time. *T* and *W* indicate the number of correct bits and the codeword width, respectively. (b) Two types of configurability for dynamic error correction in the memory array [100].

The *Static Variation* is the memory block reliability difference across the chips; the reliability map is generated during the manufacturing test. The *Spatial Variation* is evaluated (i) in the manufacturing test, to generate a custom reliability map due to intra-die differences and (ii) in runtime, where depending on the activity, some memory blocks can be affected by the temperature more than others. The *Temporal Variation* is evaluated by a *mode_selection* responsible for monitoring the type of error in the same block over time - random or contiguous, voltage or temperature noise, and the aging induced by bias temperature instability. The *mode_selection* can be evaluated by a fixed time or adaptative by the number of errors.

The runtime evaluation demonstrated in [98] and applied in [100] introduces another important design, an extra control unit for dynamically changing the input data length between MSB and LSB, as depicted in Figure 41(b). When the number of memory errors in a codeword exceeds the maximum number of correctable bits, the length of the ECC input data is reduced to focus on the most important MSB parts. As a result, error corrections in MSBs can be assured, and the overall system quality degradation can be minimized, as

uncorrected LSB errors have a much less prominent system effect.

The VC-ECC approach provides the right amount of error-correction capability to the individual memory blocks depending on their relative vulnerabilities to runtime errors without incurring large hardware and power overhead. The proposed time-varying ECC was tested in a 2MB L2 cache at 65nm. The case study reduces the increasing number of errors in the L2 cache due to the gradual voltage scaling scheme together with the adaptive time duration control, proving a good adaptive ECC with a high level of reliability for the cache while maintaining its low-power advantage.

3.2.8 Proposal of an Adaptive Fault Tolerance Mechanism to Tolerate Intermittent Faults in RAM [103]

Baraza-Calvo et al. [103] present an Adaptive Fault Tolerance (AFT) mechanism based on ECC, able to modify its behavior when the error conditions change without increasing the redundancy. AFT mechanism can detect and identify intermittent errors in RAM and swap from the Hsiao-based SECDED ECC to EPB3932, a specific ECC capable of tolerating one intermittent error.

EPB3932 is an ECC encompassing 32-bit data and 7-bit redundancy; it applies asymmetric error control for tolerating intermittent faults in one bit, marked as an Error-Prone Bit (EPB). EPB3932 was designed to correct single errors, double errors containing the EPB, and triple-adjacent errors containing the EPB; it also can detect double-adjacent errors not containing the EPB. EPB3932 has the same redundancy as Hsiao-based ECC but has unbalanced correction capabilities to overprotect the corresponding EPB. As EPB3932 applies special error control to the EPB, unlike Hsiao-based ECC, this code is not uniform for all 39 codeword bits. Instead, each bit has a specific EPB3932 with its associated ECC encoder and decoder modules.

AFT starts reading and writing with the simplest Hsiao-based SECDED ECC. As the memory degrades because of the upset increase, the *Intermittent Error Detector* increases the *BITi counter*, which is used as a threshold to evaluate if AFT must change to EPB3932. When the threshold is reached, the adaptive memory controller sends a *busy memory* signal to stop the processor operation; the entire RAM is scrubbed by reading all memory addresses sequentially using the Hsiao-based ECC decoder and reencoding them with the EPB3932 encoder. The microprocessor is resumed unsetting the *memory busy*, continuing its operation from the same point. Each memory module can have a different ECC managed by the reconfiguration manager and stored in the configuration ROM, as shown in Figure 42.

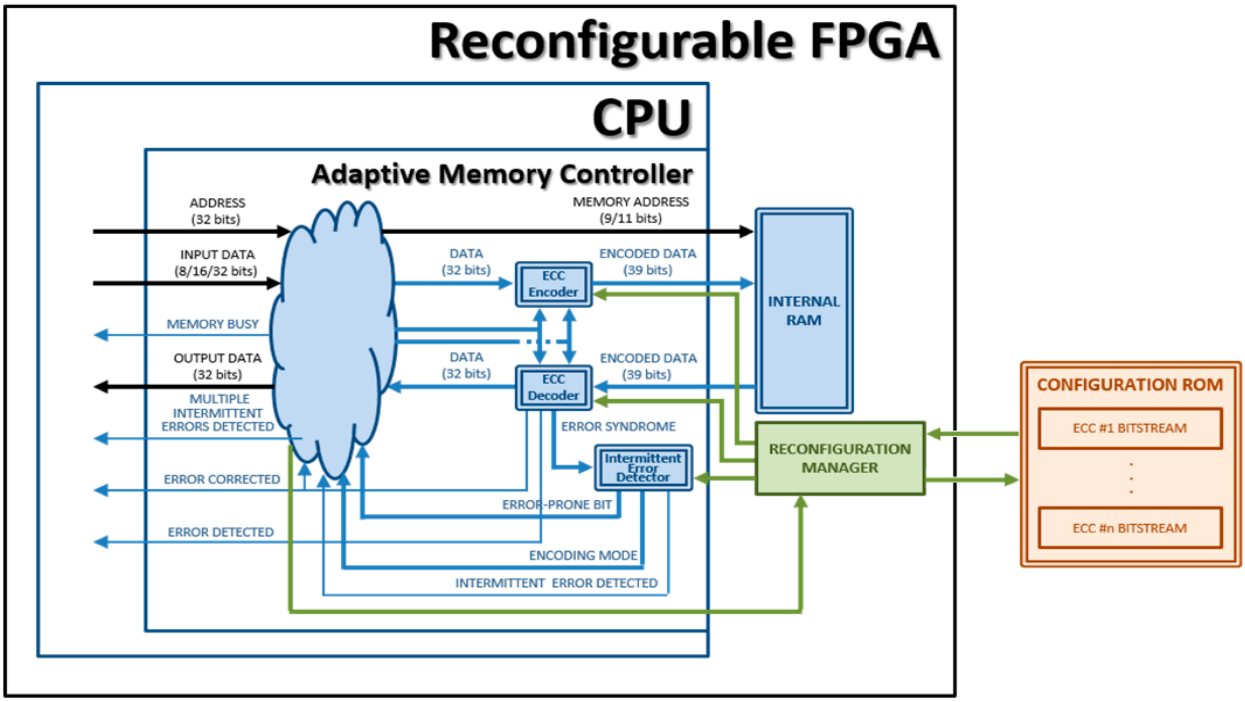


Figure 42. Block diagram of the AFT mechanism (synthesized in a reconfigurable FPGA) [103].

The authors have designed this AFT mechanism in VHDL and integrated it into the memory controller of a 32-bit RISC microprocessor. They stressed the system injecting single/multiple, random/adjacent, transient/intermittent, and combinations of intermittent and transient faults. The experiments demonstrated that the proposed AFT mechanism could detect, switch the encode and correct transient and intermittent faults. The authors also analyzed that the overhead introduced by the AFT mechanism proposed was affordable in terms of hardware, latency, and energy consumption for a reconfigurable FPGA.

3.2.9 CARE: Coordinated Augmentation for Elastic Resilience on DRAM Errors in Data Centers [104]

Chen et al. [104] propose CARE, a novel error-tolerant framework for effective and elastic resilience on DRAM. Their work introduces a cache-like structure in the memory controller for dynamic error tracking and proactive resilience enhancement. Figure 43 displays that CARE is compounded by an ECC cache, BCH encoder/decoder, error counting module, and the set of registers for page retirement, which are integrated into the memory controller and no modification in other parts of the system is required.

On the write memory transaction, the ECC Cache looked up to the DRAM address (1). If it is a miss, no additional action is needed; otherwise, the data to be written is encoded with BCH (2), and the corresponding BCH code, along with the address, is updated in the ECC Cache (3). On the read path of the memory controller, for each 64-byte data block returned from the main memory, the ECC Cache is looked up for the potential BCH code

regardless of whether the decoder detects an error for the data block (4).

If a SECDED decoder detects no error and there is no ECC Cache hit for the data block, the returned data proceed to the LLC without further delay or performance penalty. However, if there is an ECC Cache hit for the data block, the corresponding BCH code is returned from the ECC Cache (5). This procedure is necessary because the number of errors in the data block might exceed the error detection capability of SECDED. The SECDED and BCH data decoded are compared to derive the error counts, which are further sent to the ECC Cache to update the error counters and block states (6). Thus, the data returned from memory must be held in the buffer to have the same BCH Decoding latency. Finally, if there is no ECC Cache hit, but the SECDED decoder detects data block errors, the data block is sent to the BCH Encoder, and the derived BCH code is stored in the ECC Cache (3) for enhanced ECC protection in the future. Conversely, the data block does not need to wait until the encoding is finished and can proceed to the LLC without delay. The ECC Cache controller is also responsible for the coordinated page retirement. Each time a page retirement condition is triggered, the ECC Cache controller sets up the page retirement and sends an interrupt signal to the OS (7).

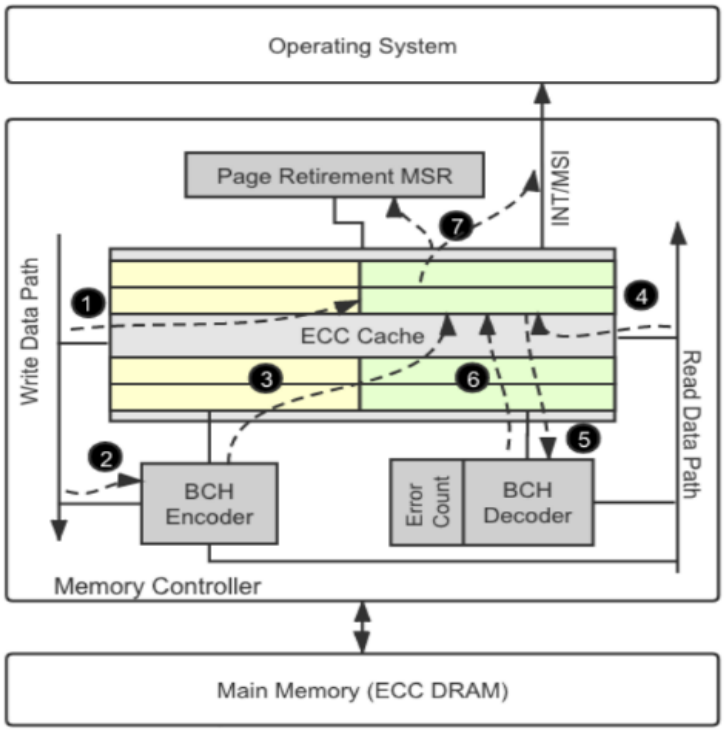


Figure 43. CARE framework and its operation details [104].

Experiment results show that, with around 58KB area overhead in the memory controller, CARE achieves near Chipkill reliability without any memory capacity penalty and incurs negligible performance overhead compared to the baseline SECDED system.

3.2.10 Stealth ECC: A Data-Width Aware Adaptive ECC Scheme for DRAM Error Resilience [105]

Lee et al. [105] exploit the use of MSB part as an additional area to store ECC. According to [108], 32-bit narrow-width values account for an average of 85.4% in the data cache when running the SPEC CPU 2017 benchmark suite [107] in 64-bit architectures. The authors also investigate the proportion of 32-bit narrow-width values in DRAM for various workloads from several benchmark suites (SPEC CPU 2017 [107], PARSEC [87][88][87], and GAP [106]). Figure 44 shows that, on average, the proportion of 32-bit narrow-width values in DRAM accounts for 47.4%, enabling the exploitation of a considerable portion of zero parts in DRAM to store more parity bits for meaningful data, improving DRAM reliability without storage overhead.

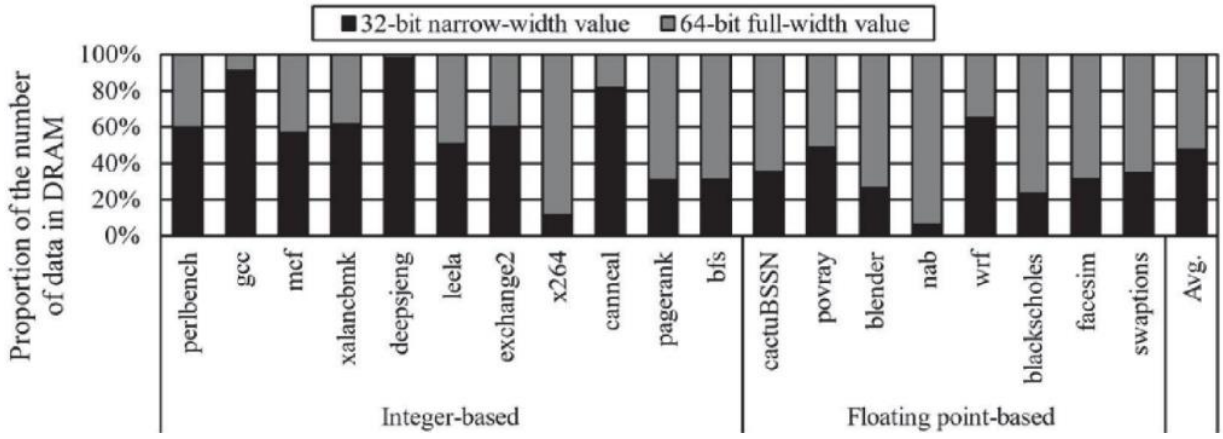


Figure 44. The proportion of 32-bit narrow-width values in DRAM [105].

Stealth ECC (Figure 45) mitigates system failure probability by exploiting a more robust BCH code for narrow-width values, storing more parity bits on the MSB side instead of zeros. While for full-width values, Stealth ECC provides the identical correction and detection capacity as a SECDED code.

When storing a 64-bit data word in the memory, the data-width aware BCH encoder classifies the data word as either 32-bit narrow-width or 64-bit full-width. When the data word is narrow-width, the 32 data bits are encoded by the multi-bit BCH code. Otherwise, the data word is encoded by the SECDED code. Since data words are encoded by different BCH codes depending on the data width, a flag is coded together. To store the additional flag code, Stealth takes advantage of the bursts to encode two words at the same time. In comparison, the SECDED code for each of two data words requires 16-bit parity (8+8 bits), and the SECDED code for two data words (128-bit) requires only 9-bit parity; this additional free space is used to store the flag code.

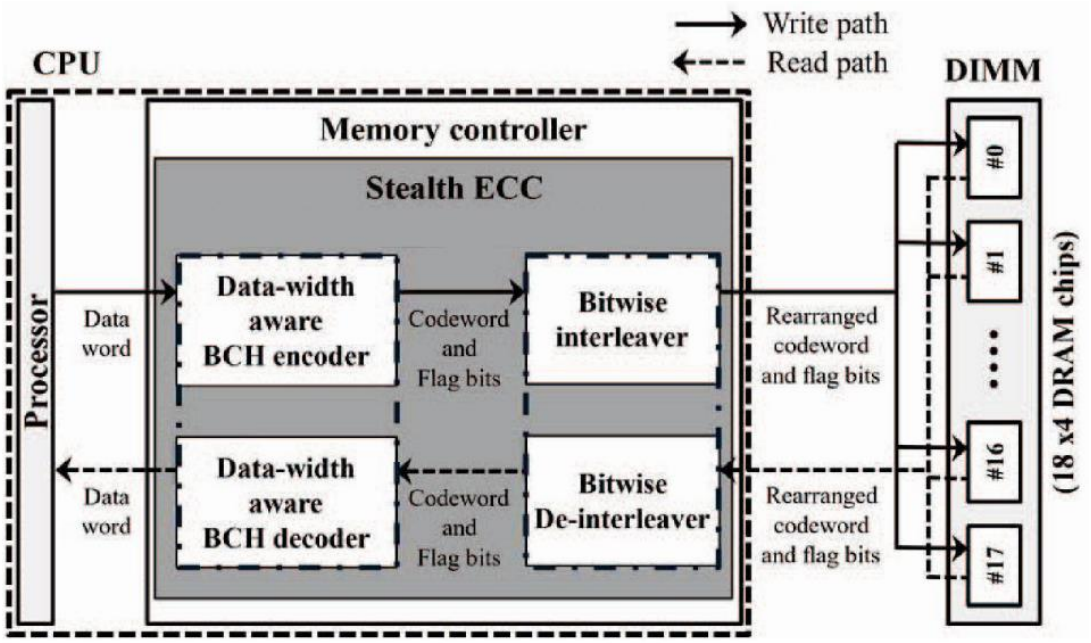


Figure 45. Overview of the scheme proposed by Lee et al. [105].

When the Stealth ECC is used, data reliability increases to 3-bit correctable BCH code by storing 12-bit additional parity into the zero part of a narrow-width value over the traditional SECDED. Stealth ECC enhances overall DRAM reliability while incurring negligible performance reduction and no storage overhead. The simulation results show that Stealth ECC reduces the system failure probability caused by DRAM errors, by 47.9%, on average, with only 0.9% performance overhead compared to a conventional SECDED code.

3.3 Conclusions

Many works explore the memory reliability increase in different levels and locations. However, only a few works explore memory controller approaches, confirming the statement by Lin et al. [83] on the scarcity of research on RAM controllers. Also, many reliable works have the same authors or academy, for example, Kumar in [80][81][89][97] and Bhunia in [82][96][98]. This theme has a high interest in the industry, evidenced by some partnerships between authors from the academy and companies like AMD [95][89], Intel [56][96][100], Samsung [100], HP [64], and Oracle [97].

To the best of our knowledge, our work is the first one that proposes a dynamic approach for managing the ECC on the memory controller considering the dynamic behavior of memory error rate reaching, at the same time, memory reliability with efficient energy consumption. Other works with dynamic approaches operating in the L2 cache and/or after increasing the ECC, keeping the high encode without considering a temporary failure Table 4 compares all the researched works.

Table 4. Comparative research work, considering static and dynamic ECC approaches [Author].

	Name	Ref.	Year	Local	Correction procedure	ECC
Static	PERC	[84]	2006	L1/L2 cache	Evaluates parity and only execute the single error correction when an error is founded	Parity, Single error Correction
	MM-ECC	[61]	2009	LLC	Operate with two levels of protection – T1EC with a local and T2EC with a block protection	Parity, Hamming, and DECTED
	Hi-ECC	[56]	2010	LLC	Disable areas and only executes ECC if an error occurs; try a low cost ECC; if it is not possible to fix, execute the most powerful ECC	Parity, BCH, 5EC6ED
	VF-ECC	[63]	2011	LLC, Main memory (DRAM)	Operate in two levels as MM-ECC; Integrate OS with hardware to guarantee reliability; and add protection for Non-ECC memories	Chipkill
	LOT-ECC	[64]	2012	Main memory (DRAM)	Operate with local error detection layer and global error correction layer	Parity and checksum
	ML-ECC	[89]	2013	Main memory (DRAM)	Chipkill enhanced with erasure code to increase error correction and adding rows with checksum	Chipkill with erasure code
	ECC-P	[80]	2014	Main memory (DRAM)	Resilience technique for a multi-channel memory. Use parity; when found an error apply an XOR stored in another channel	Parity, XOR
	CPT	[81]	2015	L1 cache	Executes tests on L1; if it does not have an error, deliver data; if it has an error, try to perform a weak ECC, else execute a strong ECC	Parity, BCH
	Bamboo	[90]	2015	Main memory (DRAM)	Apply ECC schemes for grouping data as symbols enabling several types of pin-error corrections	Bamboo
	MLC	[91]	2015	Main memory (SSD)	Use one area for data and other for ECC. ECC power changes depending on the error rate	Parity, BCH
	E-ECC	[93]	2016	Main memory (DRAM)	Evaluates three encoding schemes to improve the basic Chipkill code	Chipkill, RS, GF
	LAE-FTL	[94]	2017	Main memory (NAND flash)	Use different BCHs depending on the error rate; when the error rate is high, LAE-FTL squeezes the data area to store the extra redundancy bits	Parity, BCH
Dynamic	RD-ECC	[82]	2011	L2 cache	Block mapping occurs in runtime or in the design phase. Evaluates the number of errors to select the appropriated ECC	Hamming, BCH
	MAGE	[95]	2012	Main memory (DRAM)	Hardware-software collaborative solution - OS changes the ECC in runtime	SECDED, Chipkill
	RAPM	[96]	2013	L2 cache	Employ ECC considering the reliability variations across memory blocks; reliability assessment is performed during the manufacturing test	Hamming, BCH
	ARCC	[97]	2013	Main memory (DRAM)	Apply adaptively weaker Chipkill for the fault-free region and a strong one in regions with fault	Chipkill
	VL-ECC	[98]	2014	L2 cache	Select in runtime or in the design phase the ECC according to the number of errors. Explore MSB of DSP applications	BCH
	AES-DP	[99]	2015	Main memory (NAND flash)	ECC changes regarding the number of errors or lifetime (P/E cycle count and retention time)	Hamming, BCH
	VC-ECC	[100]	2017	L2 cache	Chose ECC evaluating static, spatial and temporal variations in manufacturing time. Explore MSB of DSP applications	BCH
	AFT	[103]	2020	Main memory (DRAM)	Explore a threshold of errors to change the ECC dynamically	Hsiao-SECDED, EPB3932
	CARE	[104]	2021	LLC	Dynamically explore SECDED and BCH ECCs according to the error correction capacity	SECDED, BCH
	Stealth	[105]	2022	Main memory (DRAM)	Dynamically explore SECDED and BCH ECCs according to the error correction capacity. Use MSB to store ECC	SECDED, BCH
	DFMC	**[127]**	**2023**	**Main memory (DRAM)**	**Evaluates in runtime the number of errors in a slice of time and select the most efficacious and efficiency ECC**	**Parity, Hamming, and LPC**

4 PROPOSED MEMORY CONTROLLER ARCHITECTURE

This chapter describes the proposal of the Dynamic Fault Tolerant Memory Controller (DFMC) implemented through the fault-tolerant methodology introduced in Section 1.2 into the architectural abstraction of the memory controller described in Section 2.6. Figure 46 illustrates the union of the proposed method with the basic architecture of the memory controller, implying the insertion of the Dynamic Fault Tolerance Module (DFTM) that interfaces the frontend with the backend of the memory controller.

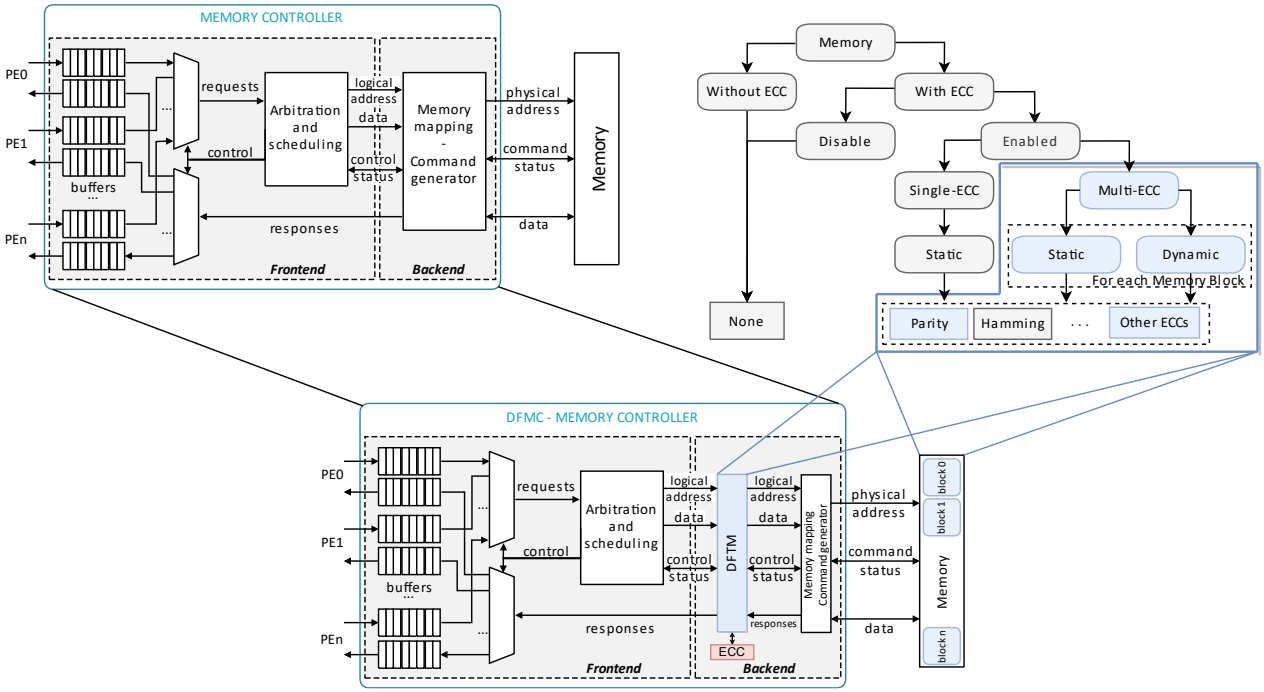


Figure 46. DFMC encompassing memory controller circuits and DFTM, which implements the proposed fault-tolerance methodology [Author].

4.1 Dynamic Fault Tolerance Memory Controller (DFMC) Description

This work proposes an innovative method that combines the ***Multi-ECC*** technique and **Dynamic** method for selecting ECC during the execution time in memory blocks according to the fault scenario.

The ***Multi-ECC*** technique divides the memory into memory blocks, as illustrated in Figure 47. A finer granularity enables us to select the ECC according to user criteria for a given memory block. For example, one block can have a high ECC for OS operation and another block Without ECC (WE) for data like bitmap images; consequently, the approach achieves the efficacy required by the user without compromising the memory efficiency. The **multi-ECC** can be configured in **Static** or **Dynamic** approaches. When the **Static** approach is set, the memory block does not change the defined ECC during all operation time. the ECC can change according to a configurable rule when setting the **Dynamic** approach.

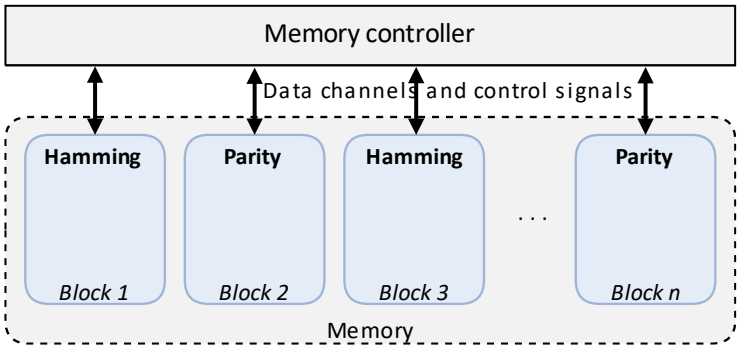


Figure 47. Example of a memory encompassing *n* blocks codified with Parity or Hamming [Author].

The **Dynamic** method employs a threshold decision that enables the memory controller to change the ECC of a memory block dynamically; thresholds are used to define the scenarios that a memory block must change from a high- to low-power ECC and vice versa. The threshold can be defined as *fixed* or *self-programmable*. The *fixed threshold* has a pre-defined number of faults; when this number is reached, the **dynamic** mechanism can select another ECC; e.g., when the number of bitflips is more than 12 faults, switch to a higher efficacy ECC. The *self-programmable threshold* changes the value according to external conditions, e.g., a sensor can catch the temperature of each memory block and associate the number of bitflips with the ECC switching.

Although our experiments use only a *fixed threshold*, we illustrate Figure 48 with a level (5), expanding the original fault-tolerant memory organization shown in Figure 3.

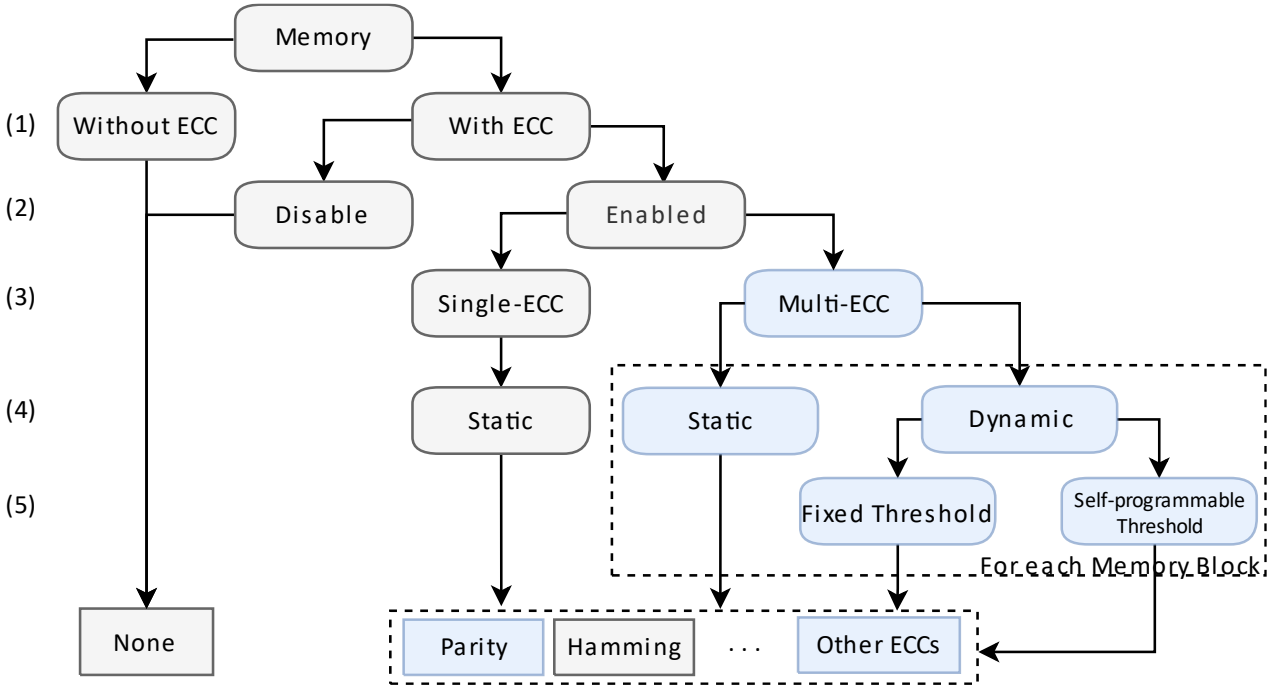


Figure 48. Fault-tolerant memory organization example encompassing the commercial (in gray) and proposed (in blue) approaches with threshold level [Author].

DFTM is the principal module of DFMC, whose primary function is to manage which ECC will be used in each memory block and switch the ECC depending on memory status.

Figure 49 shows that DFTM implements this functionality enclosing (i) **Internal memory**, (ii) **ECC module**, split into (ii.a) **ECC encoder** and (ii.b) **ECC decoder**, and four processing modules: (iii) **RAM process**, (iv) **Threshold**, (v) **Recoding**, and (vi) **Configuration process**. Besides, the **RAM process** contains (iii.a) **R+W manager**, (iii.b) **Block finder**, and (iii.c) **ECC evaluator** modules.

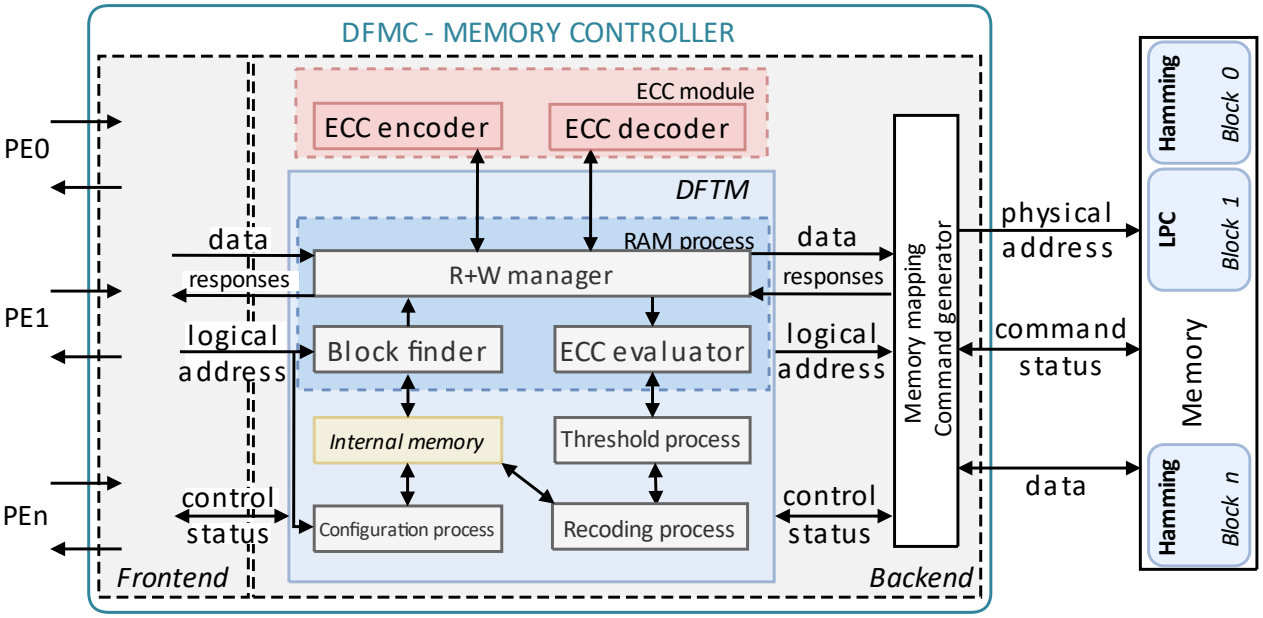


Figure 49. DFMC architecture; Frontend detailing was omitted to highlight the aspects explored in this work [Author].

i. **Internal memory** contains the DFTM operation settings. This memory should be lowly susceptible to errors, such as Radiation Hardening [109], to mitigate failures in the DFMC operation;

ii. **ECC module** in this work implements Parity, Hamming, and LPC and contains two modules:

iii. **ECC encoder** is responsible for encoding the data to store in memory according to the ECC configured;

iv. **ECC decoder** decodes data from memory according to the configured ECC and informs the **R+W manager**, which afterward notifies the **ECC evaluator** if some errors occur during the decoding;

v. **RAM process** oversees managing all the reads and writes, applying the ECC according to the configuration; this process implements:

vi. **R+W manager** encodes/decodes data according to the memory block ECC – e.g., Hamming or LPC. The manager also evaluates if it has at least one memory block that uses two words to store data, turning the second RAM on or off. Note that the dynamic approach handles the encodings of the memory

blocks independently. The second memory is only activated when the encoding requires optimizing power dissipation;

vii. **Block finder** calculates the memory block location from the logical address provided by the memory controller *Frontend* as well as accesses the DFTM **Internal memory** to get the block configuration;

viii. **ECC evaluator** validates and updates the number of errors that happen from the **ECC decoder** results and forwards the number of errors back to the client (*Frontend*) in case of reading requests;

ix. **Threshold process** evaluates the number of errors that are occurring in each memory block and, according to a predefined threshold and application requirements, requests the memory block recoding;

x. **Recoding process** converts all data of a memory block from one encoding to another, e.g., Hamming to LPC;

xi. **Configuration process** allows setting dynamic or static ECC managing approach and the initial ECC encode for each block.

4.2 Double RAM Manager

Traditional memory controllers use the second RAM to provide additional address capacity. In cases where the system employs a strong ECC, such as LPC or Chipkill, the second RAM is used to complement the word bit-size - i.e., 144 bits over the traditional 72 bits. In the proposed system, the highest efficiency ECCs need double of bits to write the ECC, requiring a second RAM module - e.g., Hamming when compared with LPC, Reed Solomon, or Chipkill. Due to this characteristic, managing when the second RAM can be turned off is crucial to reducing energy consumption. DFMC accomplishes this management by implementing three modules: (i) the **RAM manager** inside the **Configuration process** and the (ii) **read** and (iii) **write** managers in the **R+W manager**, as shown in Figure 50:

i. **RAM manager** enables the second RAM when at least one ECC needs a double-memory requirement in any block. This logic is based on the *double memory counter*; the second RAM is enabled when the counter is higher than zero; otherwise, the second RAM is disabled;

ii. **Write manager** controls whether the current ECC needs to be written in one memory or parallelly in two memories; if the ECC needs only one RAM and the double memory is enabled, only write in the first RAM; the second keeps waiting for a memory block which needs the two RAM. The Write Manager utilizes the Write RAM, which is responsible for requesting the written data and

sending this data to the *Memory Mapping and Command generator (MMCG)* that interfaces with the RAM receiving back the written status;

iii. **Read manager** handles whether the current ECC needs to be read in one or in two memories; if the ECC needs only one RAM and the double memory requirement is enabled, it only reads the first RAM; the second keeps waiting. The Read Manager uses the Read RAM for requesting data from a specific address to MMCG that interfaces with the RAM, delivering back the read data.

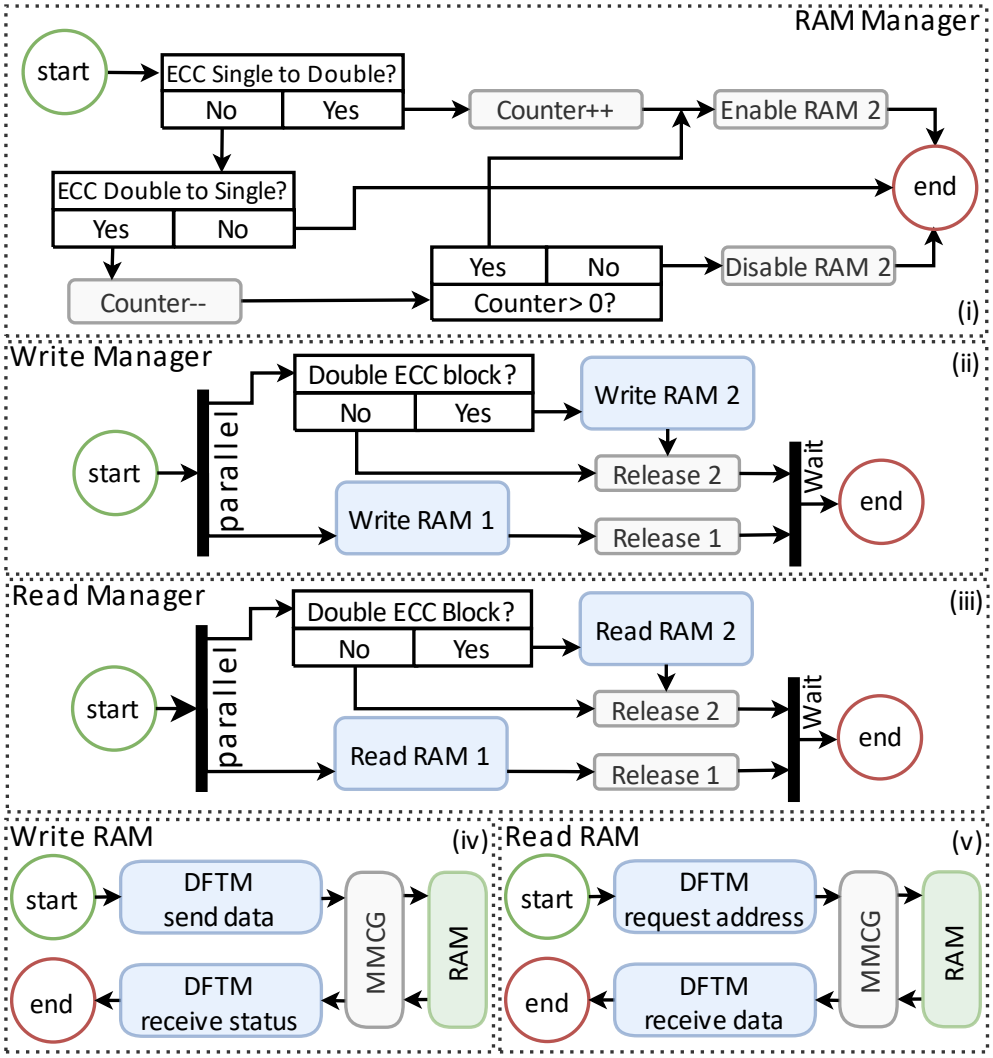


Figure 50. Modules necessary to manage the double memory used by DFTM: (i) RAM manager, (ii) Write manager, (iii) Read manager, (iv) Write RAM and (v) Read RAM [Author].

4.3 RAM Process

The **RAM process** encompasses the **Block finder**, **R+W manager**, and the **ECC evaluator** introduced in Section 4.1, which executes the write and read steps.

In the write step, upon receiving the write request from the memory controller *Frontend*, the **Block finder** calculates the memory block from the address, and the data is directed to the **R+W manager** (read and write manager) to be encoded by **ECC encoder**

according to the ECC configuration of the memory block. Subsequently, the data is directed to the *Write manager* to write in one or two memory modules; finally, the data is sent to the *Memory Mapping and Command generator (MMCG)* of the DFMC Backend.

The first steps of the read request are similar to the first steps of the write request; i.e., the *Frontend* forwards the request to the **Block finder**, which calculates the memory block from the address, passing the logical memory address to the *MMGC* and activates the **R+W manager**. Subsequently, the memory returns the read data decoded by the **ECC decoder**. **ECC evaluator** receives the decoded data and decoding status; if the decode contains an error, this error is counted and stored in internal memory; the data is forwarded to the *Frontend*, which returns the information to the PE that requests the reading. Figure 51 presents the flowchart of read and write in the RAM Process; note that components outside the DFTM were omitted, like *Frontend*, MMCG, and RAM.

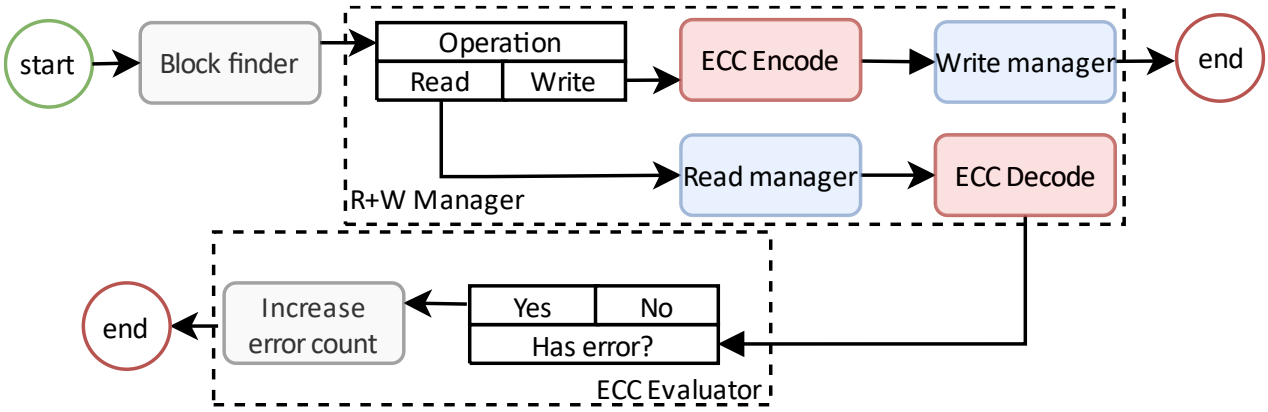


Figure 51. Flowchart of reading and writing data in memory [Author].

The **Block finder** is responsible for discovering the memory block location from the address to get the configuration from the DFTM **Internal memory**. The configuration contains the operation mode that controls each memory block independently and can be changed in runtime according to the number of errors and the system operation mode. The **ECC evaluator** receives the number of errors from the **ECC decoder** and sums it with the number of errors in the memory block configuration. The static operation mode maintains the pre-programmed ECC, although the dynamic approach changes the ECC depending on the rule to be reached. Thus, the memory controller can manage efficiency and efficacy per application requirements and memory reliability.

Let $\mathbf{blockSize}$ be configured and stored at BIOS, and $\mathbf{numberOfBits}$ be the number of bits necessary to configure each memory block; then, the first-bit block location containing the encoding information can be computed by Equation 47. Using $\mathbf{blockSize} = 2000$ and $\mathbf{numberOfBits} = 8$, and considering $\mathbf{address} = 5000$ as an example, $Location = \left\lfloor \frac{5000}{2000} \right\rfloor \times 8 = \lfloor 2.5 \rfloor \times 8 = 16$; therefore, the corresponding 8-bit read block location is 16.

Equation 47. $Location = \left\lfloor \frac{address}{blockSize} \right\rfloor \times numberOfBits$

4.4 Configuration Process

When the DFTM is in the **Configuration process**, the same memory controller signals are used to read or write the configuration; the data and address correspond to the configuration value and block position, respectively. The **RAM manager** is executed after the **Write configuration** to check whether the configuration has at least one ECC that needs a second RAM module, as illustrated in Figure 52.

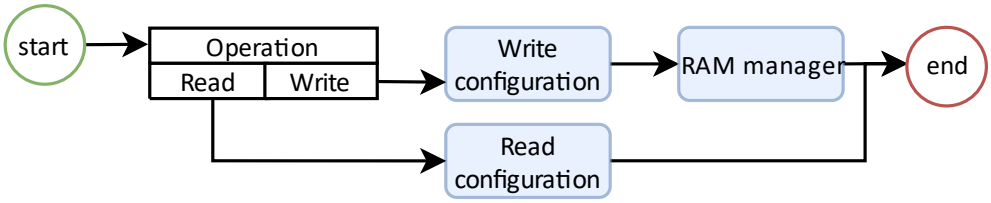


Figure 52. Flowchart of reading/writing configuration data in internal memory [Author].

Note that the configuration process can be customized to meet different operational requirements when managed by the operating system (OS). However, it is important to emphasize that this process was specifically designed to be executed solely during hardware initialization, as OS-related activities are outside the Thesis scope.

4.5 Threshold Process

One essential part of this work is choosing when to change the ECC, executed by the **Threshold process** and presented in Figure 53.

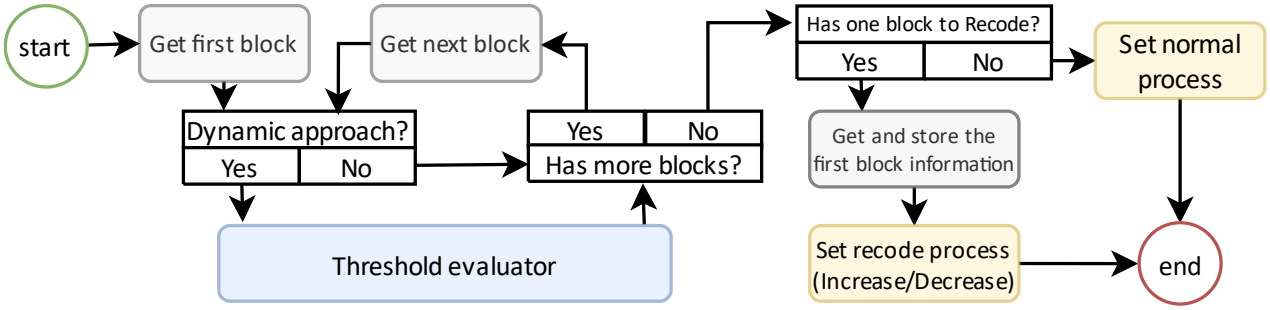


Figure 53. Flowchart for the Threshold process data [Author].

The **Threshold process** executes a configurable time unit named *Threshold Cycle Size (TCS)* for all memory blocks at every new cycle. When starting a new cycle, the **Threshold evaluator** checks the parameters programmed in the memory controller and evaluates the error rate with a dynamic approach, defining whether the threshold has been reached and which threshold policy to apply. If the threshold is reached by at least one memory block, the **Threshold process** triggers a command for the **Recoding process** to

change the first memory block found by the **Threshold evaluator** that defines which ECC should be used to encode all the addresses inside the memory block. This work employs a **Fixed threshold**, where the rule does not change in runtime, while the **Self-programmed threshold** changes the rule used at runtime depending on an external factor.

4.5.1 Threshold Evaluator - Fixed Threshold

The **Threshold evaluator** is a mechanism responsible for choosing the ECC for each memory block based on a dynamic error rate throughout the entire block operation and the error intervals that define using a given ECC, called ***threshold***. The dynamic error rate considers the sum of all errors in a given block reduced by a parametrizable number of errors in each TCS, enabling exploring the variability of error occurrence in operation windows. The **Threshold evaluator** implements this dynamic error rate for each block with the Number of Accumulated Errors (NAE) incremented whenever a data error is detected. At each cycle, NAE is decremented by the Number of Errors to Reduce (NER) – a parameterizable value defined for all memory blocks; using NER avoids accumulating errors indefinitely and enables changing to a less powerful ECC in case of error rate reduction.

The **Fixed Threshold** proposed and demonstrated in Figure 54 receives (i) NER and (ii) a list of ECC thresholds, such that each ECC threshold is a tuple(*min*, *max*) containing the error interval with the minimum and the maximum number of errors to keep in a given ECC. If NAE exceeds *max*, the **Threshold evaluator** selects a more powerful ECC, and if NAE becomes lesser than *min*, the **Threshold evaluator** programs a less powerful ECC for a given block. If it is required to change the ECC, the mechanism signals to recode the first memory block marked to change the ECC.

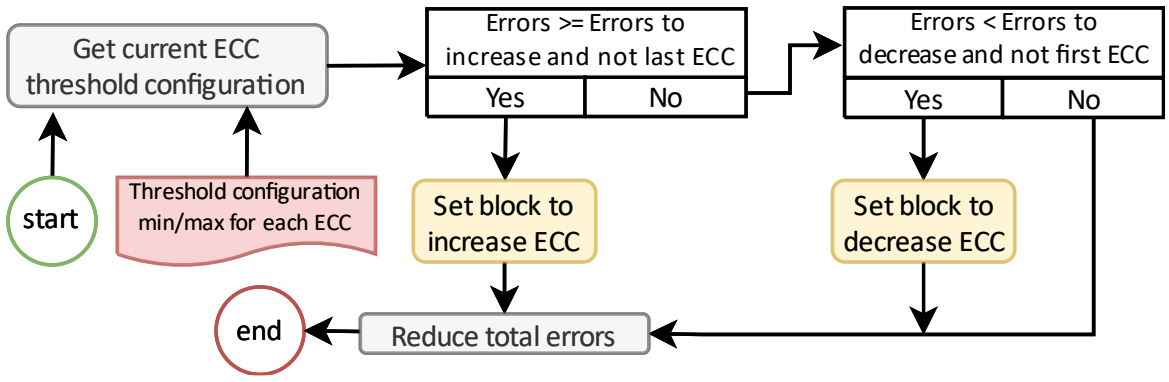


Figure 54. Threshold evaluation workflow [Author].

For instance, considering the following threshold configurations for Parity [0,1], Hamming [2,3], and LPC [4,+∞); if in the moment of the **Threshold evaluator**, it is in Hamming and counts more than three errors, the **Threshold evaluator** signals to the **Threshold process** that commands the **Recoding process** to change the ECC to LPC;

conversely, if the error count is less than one, the **Recoding process** changes the ECC to Parity. Note that the **Threshold evaluator** can receive more than one block to recode, but it recodes just one per cycle because of the considerable time spent in recoding.

One safe approach is immediately switching to the most powerful ECC when a bitflip is detected, allowing the Threshold process only to evaluate when changing to a less powerful ECC is necessary. However, this approach can lead to system blockage for an extended period, especially in extreme cases where one block might need to be recoded after another. Another drawback is the frequent switching between ECCs; a block can change to a powerful ECC and revert in the subsequent Threshold evaluation. We have decided to change the ECC only after a designated time slice, as configured in TCS, to avoid this. This approach aims to strike a balance between system stability and the need for dynamic ECC adjustments, allowing for the best setup to be configured in the TCS, NAE, and NER for each scenario.

4.6 Recoding Process

The **Recoding process** is complex and blocking, starting with a message to the OS informing the memory range that will be switched. Recoding implies reading all the data stored within that range and rewriting it using a new encoding, which starts by configuring the first address of the block in variable N. While N is less than *blockSize*, the process reads and decodes the data at address N with the current ECC and writes the data with the new ECC at the same address N. Subsequently, the **Recoding process** updates the block with the new ECC and checks whether it needs to turn on or off the second RAM (RAM manager). Once the **Recoding process** is activated, all requests for access to the module are queued and only released after the operation is completed, implying a latency of several clock cycles. Therefore, the application requirements must justify the operating costs of the **Recoding process**. In the end, OS is notified again; this information allows OS to apply another level of fault tolerance – this work does not cover the OS functionality. Figure 55 presents the **Recoding process** flowchart.

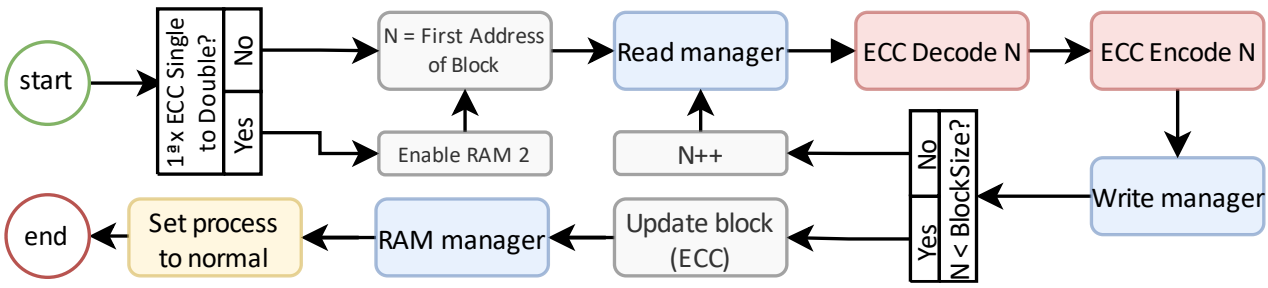


Figure 55. Flowchart for the Recoding process data [Author].

4.7 Dynamic Fault Tolerance Module (DFTM) Workflow

The final version of DFTM also contains the (i) **Workflow process** to manage when each process has to be executed; (ii) **Debug** that sends data to the first slot of the memory; (iii) **Configuration process** to read and write each DFTM configuration block; (iv) **RAM process** to manage the data to read and write at RAM; (v) **Threshold process** to assess the threshold for all blocks; and (vi) **Recoding process** for changing the ECC for all addresses in the block. To manage when each process is executed, a workflow process was included. Figure 56 demonstrates the unified DFTM workflow.

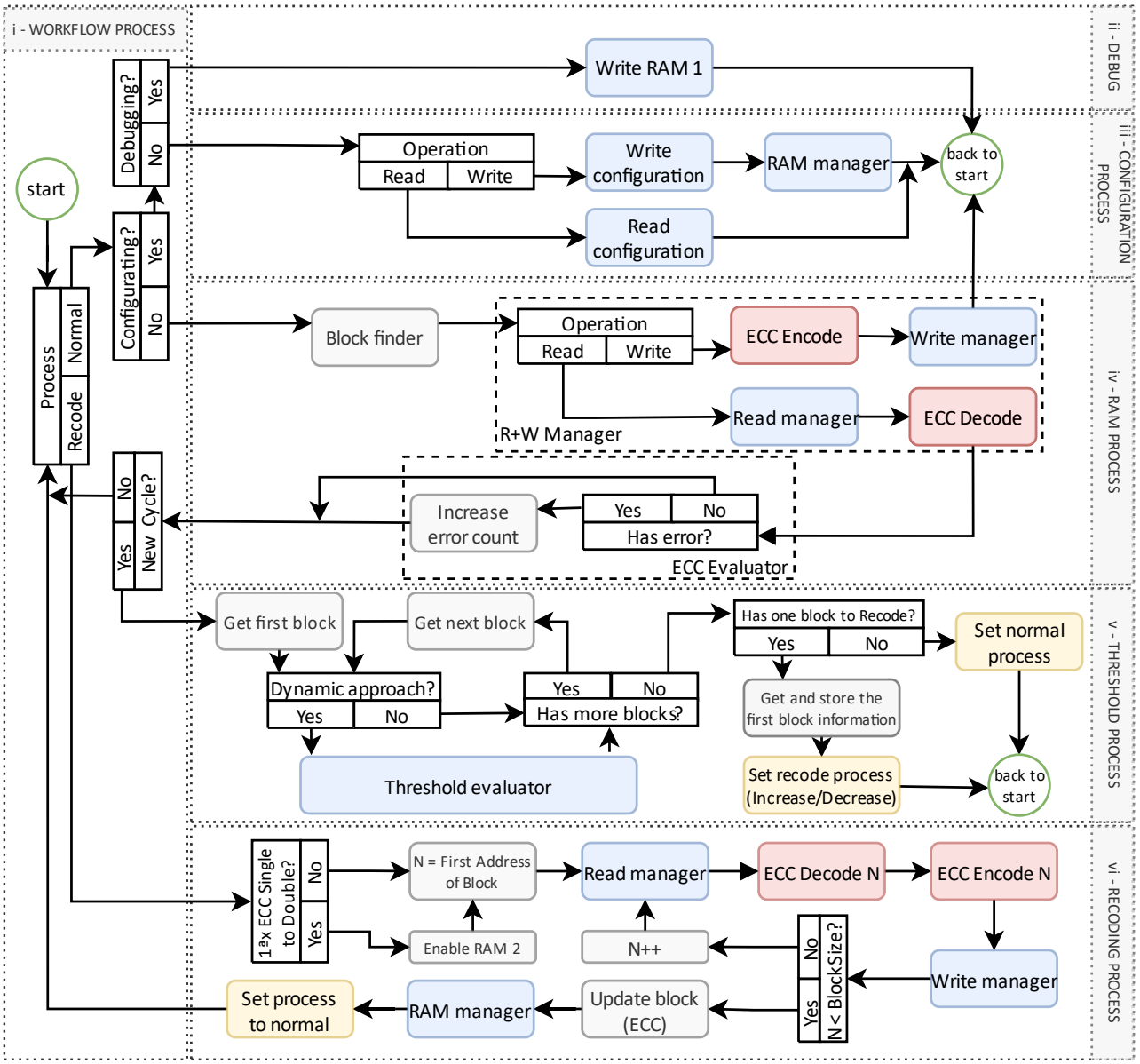


Figure 56. DFTM workflow [Author].

4.8 DFTM Theoretical Operation

Figure 57 illustrates four memory controllers with their respective modules abstracting the front end: (i) DFMC, (ii) Static LPC (SL), (iii) Static Hamming (SH), and (iv)

Without ECC (WE). Including the SL, SH, and WE memory controllers is meant to provide theoretical explanations of their operation and highlight the potential of DFMC in terms of optimized energy consumption.

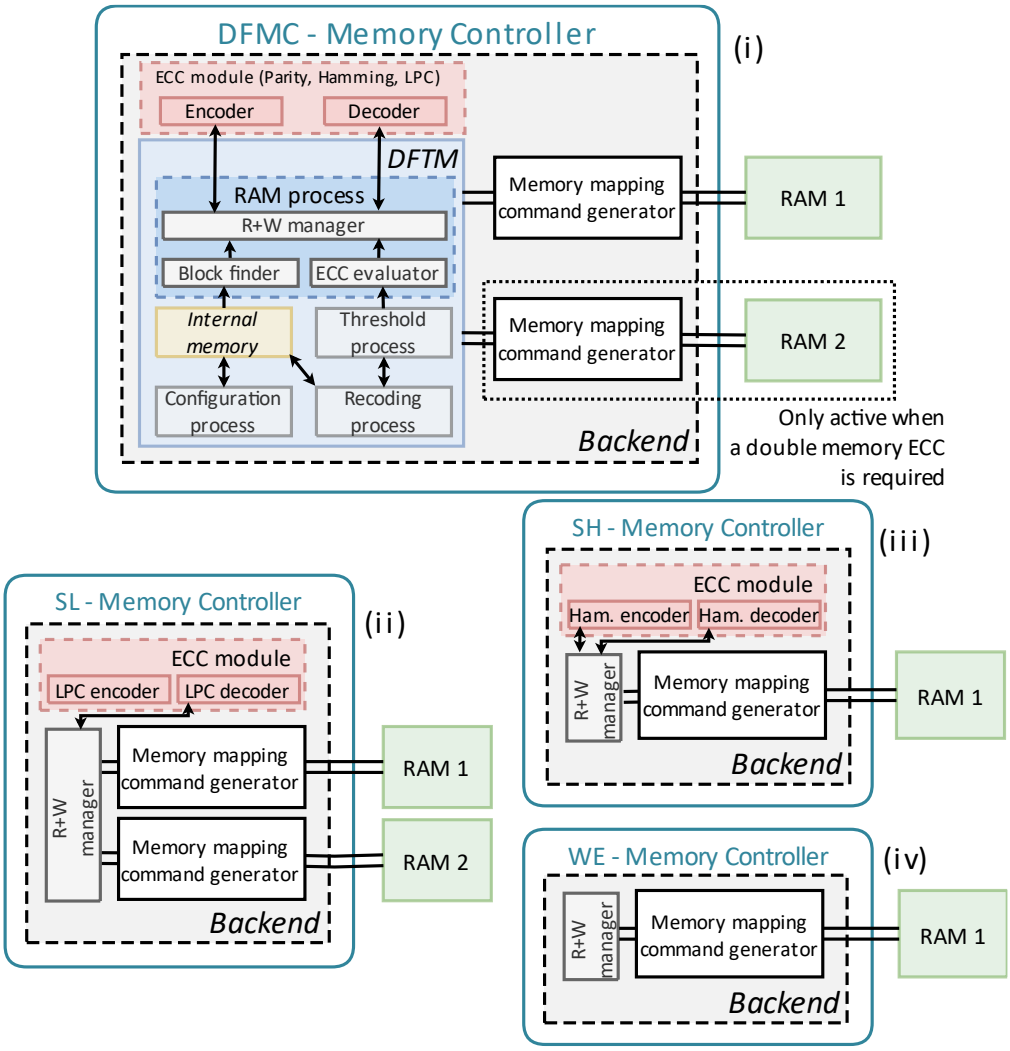


Figure 57. (i) DFMC, (ii) SL, (iii) SH, and (iv) WE memory controllers [Author].

As explained in Section 4.1, ***DFMC*** has (i) an ECC module encompassing Parity, Hamming, and LPC; (ii) the RAM process including an R+W manager, block finder, and ECC evaluator; an internal memory; (iii) Threshold, Recoding and Configuration processes; and a pair of (iv) MMCG connected with 4GB RAMs. ***SL*** encompasses an LPC ECC module, an R+W manager, and two MMCG connected with two RAM modules. ***SH*** includes a Hamming module, an R+W manager, and an MMCG connected with one memory module. Finally, ***WE*** contains an R+W manager and an MMCG controlling a single RAM module.

The R+W manager module is the initial reference for the time spent on the critical path and energy consumption across all processes in all memory controllers. The WE controller assumes a constant alpha (α) value for this explanation. Additionally, gamma (γ) and epsilon (ε) represent the SH and SL controllers' average power dissipation over time.

DFMC exhibits varying energy consumption across its modules. Here is a breakdown of the energy consumption characteristics for each module:

i. The Workflow process is executed in all operational steps. However, due to its low complexity, its impact on the critical path latency and power dissipation is negligible and, therefore, not considered in this analysis;
ii. The Debug module is only implemented in the evaluation version and is not included in this example, as mentioned;
iii. The Configuration process occurs only once during startup to load pre-defined configurations from the BIOS and set up the DFMC configuration. This process has low complexity and is expected to have minimal energy consumption and a short duration compared to the R+W manager. It operates with a beta (β) power dissipation. It is worth noting that the time spent in this process may vary depending on the number of configurations that need to be loaded, with each configuration being written into the internal memory.
iv. The RAM process includes a Block finder, R+W manager, and ECC evaluator. The block finder and ECC evaluator are expected to have a minor impact on the overall system, contributing omega (ω) to the power dissipation of the R+W manager, regarding their respective ECC modules (γ+ω for SH / ε+ω for SL);
v. The Threshold process occurs in each TCS and performs a low-complexity verification to determine if it is necessary to switch the current ECC. This process incurs a delta (δ) power dissipation for a brief period;
vi. The most resource-intensive process is the Recoding process due to the number of interactions required to read, decode, encode in a new ECC, and write all the addresses within the block. This process consumes significant resources and has a notable impact on energy consumption.

In the theoretical operations of DFMC, the configuration was set to use Hamming initially and, in the event of an error, migrate to LPC. The bitflip occurs at 2.5 times (t It is important to note that the Recode process requires reading and writing, doubling the time required for this process.

Figure 58 illustrates each memory controller's average power dissipation over time, focusing on a single block. The WE controller maintains a constant power dissipation over time, similar to the SH and SL controllers. However, due to the configuration process, DFMC starts with low power dissipation. The Threshold process evaluation occurs at regular t-time intervals, resulting in power dissipation peaks. When a bitflip occurs, the subsequent Threshold process triggers the Recode process, leading to a high power dissipation due to

the intensive read and write operations in Hamming and LPC, respectively, for all addresses within the affected block. It is worth emphasizing that when a bitflip occurs, a power dissipation peak occurs when the corresponding address is read, and the ECC decoder fixes the data content. When the RAM power dissipation is considered, the power spicules of the Threshold process and bitflip correction become negligible once the average power dissipation of RAM significantly exceeds that of the memory controller.

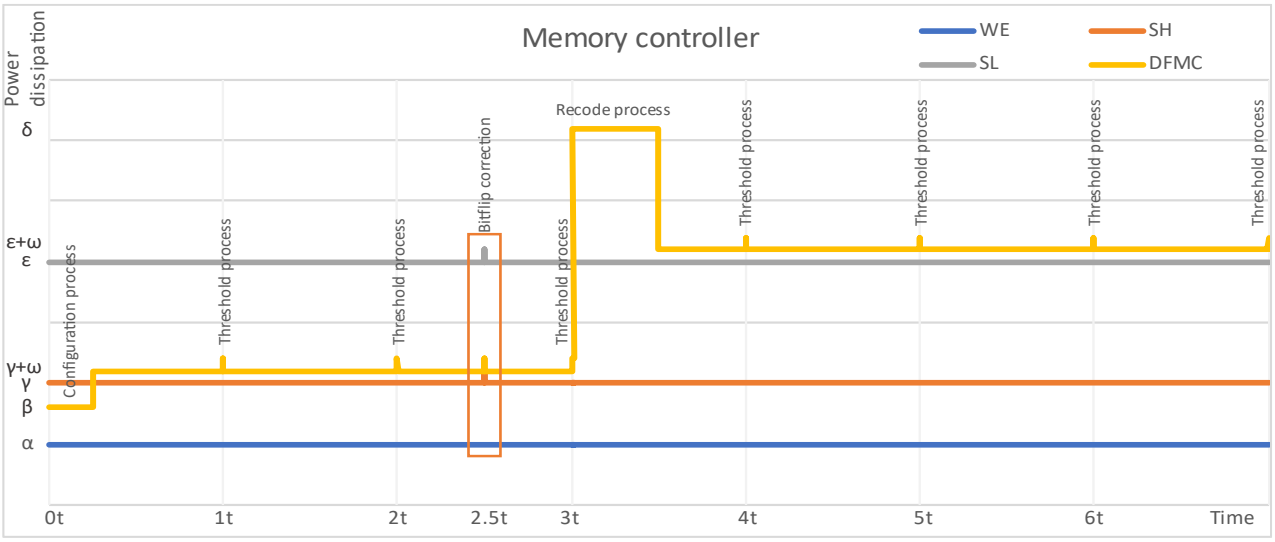


Figure 58. Power dissipation over time for the four memory controllers [Author].

Figure 59 illustrates the memory controllers with RAM, considering multiple blocks. During the Recode process, the power dissipation becomes slight, while DFMC becomes more power-efficient compared to SL and closest to the SH efficiency. The power dissipation efficiency of DFMC depends on the number of blocks and can be virtually the same as SH or slightly higher than SL.

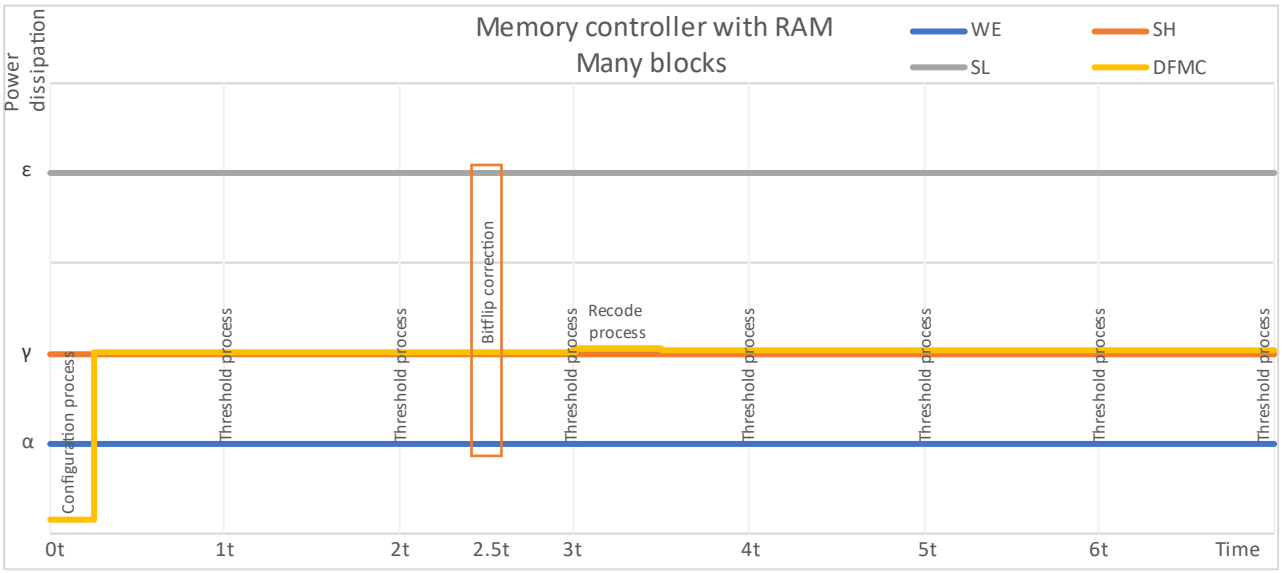


Figure 59. Power dissipation over time for the four memory controllers considering the RAM having many blocks [Author].

The energy savings achieved by DFMC becomes even more evident as the execution time increases, as illustrated by Figure 60 (a) and (b).

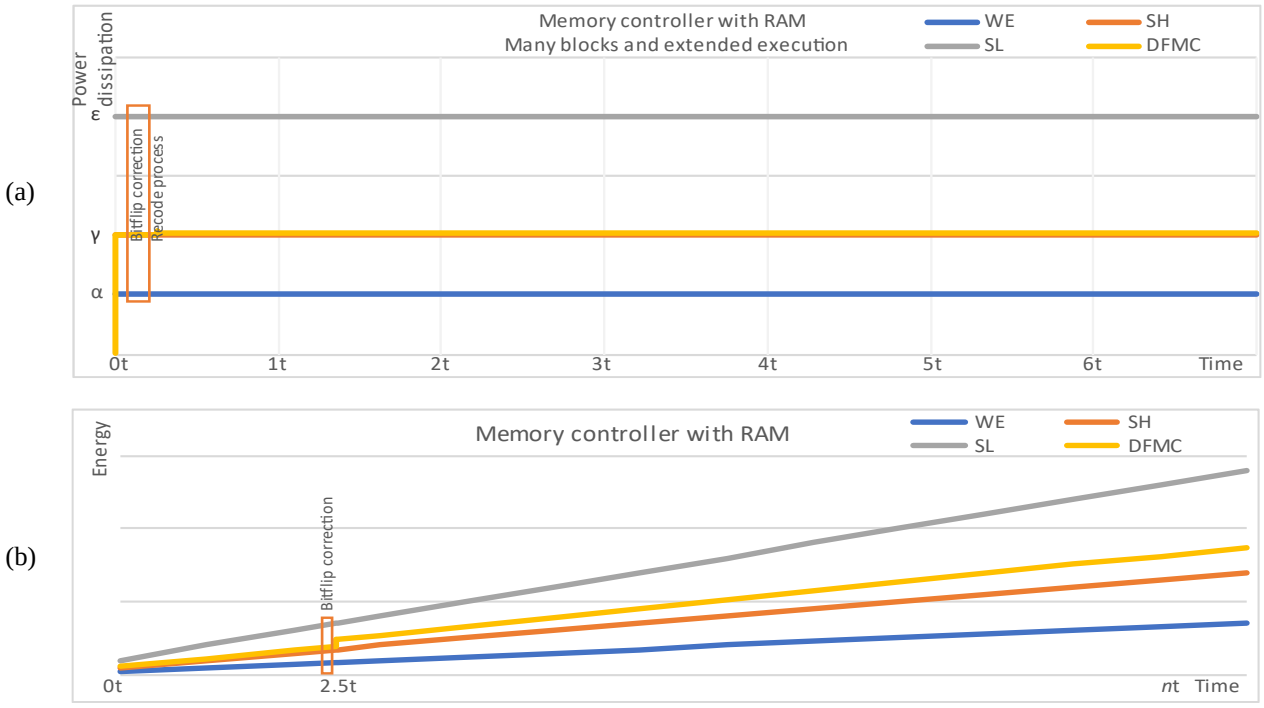


Figure 60. (a) Power dissipation and (b) energy consumption over the extended time execution for the four memory controllers considering the RAM has many blocks [Author].

Figure 61 displays the average power dissipation when the memory block is configured with a substantial number of addresses. As the number of addresses increases, the time spent on the Recoding process also increases proportionally. During the Recode process, the memory controller signals OS to put the current process in a HALT state. Subsequently, the decoding and encoding of addresses are initiated. Note that during the Recode process, no other processes are executed until it is completed. This may take longer than the interval between each Threshold process. Once all the addresses have been recoded, the memory controller signals OS to resume the execution of the current program.

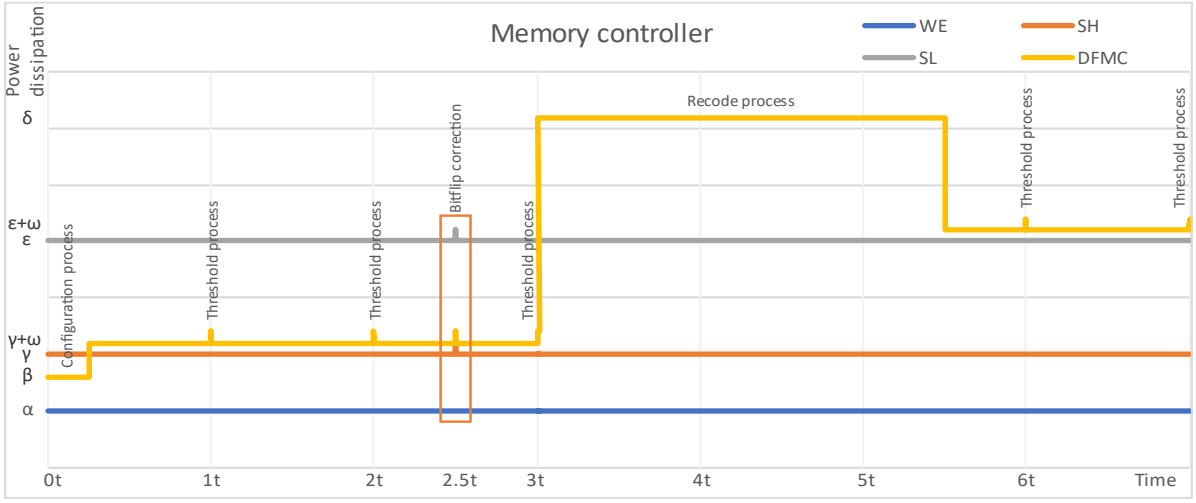


Figure 61. Power dissipation over time with many addresses manage by block [Author].

This representation highlights the impact of a larger number of addresses on the duration of the Recode process and the subsequent HALT state of the current program, ultimately influencing the overall power dissipation.

4.9 DFMC Operation and Hardware Design

DFMC enables working with a memory divided into blocks, each with its operation mode and encode. The operation mode can be either static, where the encode remains unchanged, or dynamic, where the ECC dynamically changes depending on the case. The ECC configuration includes three types: (i) Parity is the most straightforward data check, (ii) Hamming is a well-known SECDED ECC, and (iii) LPC is a high-efficacy ECC - the LPC-adapted on this version does not implement the parity bit. With these ECCs, DFMC employs 8 bits to configure each block: (i) one bit informs the memory block operating mode - (0) static or (1) dynamic; (ii) two bits contain the data encoding - (00) without encoding, (01) Parity code, (10) Hamming code, and (11) LPC; and (iii) five bits to store the error counting. Servers experiencing more than five bitflips a year are rare [3]. Therefore, having a count of 31 errors provides a significant margin in this environment. Table 5 presents the bit configuration for the encoding and operating mode combinations.

Table 5. ECC configuration of the memory blocks [Author].

Configuration		Operation mode	Encoding mode	
		Bit 0	Bit 1	Bit 2
Without ECC	Static	0	0	0
Parity code		0	0	1
Hamming code		0	1	0
LPC		0	1	1
Without ECC	Dynamic	1	0	0
Parity code		1	0	1
Hamming code		1	1	0
LPC		1	1	1

It is necessary to highlight that some recent memory technologies, like MRAM [110] and LPDDR5 [111], include on-chip ECC to mitigate errors due to scaling issues [112][113]. The on-chip ECC mechanism is built to be invisible to the memory controller, maintaining compatibility with memory standards [113]; however, this work focuses on the ECC being managed by the memory controller.

During project time, the hardware designer can configure the **Internal memory** as necessary. The internal memory size for DFMC is given in Equation 48, where the maximum number of blocks is multiplied by the number of bits needed to store the configuration; since this thesis uses 8 bits, it is necessary 256 bytes to store 256 blocks of the ECC configuration, i.e., $Interal\ memory\ size = 256 \times 8 \text{ bits} = 2048 \text{ bits} = 256 \text{ B}$.

Equation 48. $Internal\ Memory\ Size = \ blockSize \times 8b$

Once the hardware is prototyped, the manageable memory configuration is given by the number addressed by the block. Knowing that 128 GB is the maximum per memory module for a DDR5 [118] in a 64 bits architecture and considering 256 blocks, the number of addresses per block necessary to handle 128 GB is 62.5 Million, as Equation 49 describes.

Equation 49. $Number\ of\ addresses\ per\ block = \frac{\frac{memorySize}{NumberOfBits}}{blockSize} = \frac{\frac{128\ GB}{64\ b}}{256} = \frac{16\ \mathrm{G}}{256} = 62.5\ \mathrm{M}$

5 ABSIMTH HARDWARE SIMULATOR

Absimth is a hardware simulator focusing on memory controller data flow, allowing the creation and configuration of custom modules. The simulator aims to optimize the design of next-generation memory controller architectures, meeting fault tolerance requirements with fast validation before the hardware implementation phases, thus, enabling us to assess the computational system behavior in the presence of memory errors. This chapter presents the Absimth architecture, execution flow, memory bitflip observation, and benchmarks exploration used to design and validate the DFMC architecture.

5.1 Absimth Architecture

Figure 62 displays a high-level description of Absimth for creating and configuring custom modules of processors, memory controllers, and a memory device split into memory modules. Absimth also disposes of a low-complex Operating System (OS) for task management and a virtual module for simulating memory error injection.

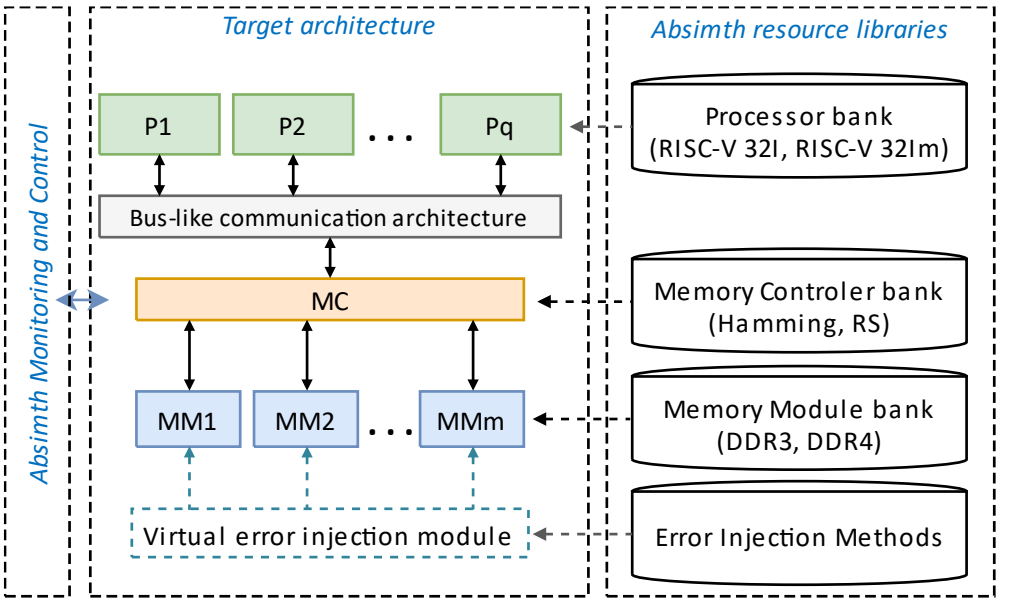


Figure 62. High-level description of the Absimth platform [126].

All modules described next can be customized or changed, respecting the standard interfaces and protocols.

5.1.1 Processor Library

Absimth includes the following RISC-V 32-bit compatible processors [119], enabling us to execute several programs on single or multiple heterogeneous processors.

- *RISC-V 32I* - 32 bits and ISA composed by integer instructions;

- *RISC-V 32Im* - including multiplication and division instructions at the RISC-V32I, and;
- *RISC-V 32f* - the same features as RISC-V 32Im but including floating point.

5.1.2 Memory Controller

Absimth connects the processor and memory modules through customizable memory controllers and includes many types of memory controllers available, including or not ECCs; the available ECCs are Parity, Hamming, LPC, and Reed-Solomon.

5.1.3 Memory Device

Absimth helps to create memory according to the user configuration, and OS accesses the memory devices using the memory controller address. The designer can navigate the memory hierarchy by examining any bit, byte, or word value of modules, rank, bank group, or each bank regarding any column, row, or height.

5.1.4 Error Injection Module

Employing a virtual module for error injection, the designer can create several error scenarios based on the predefined templates, such as creating bitflip at random memory addresses in a given execution cycle. Absimth encompasses four error injection models based on [3][15] and a five-error injection based on error occurrence probability:

- *NoFaultError* – Error-free memory, allowing us to assess application execution time and amount of data transferred compared with other scenarios; it occurs in about 91.78% of server cases [3][15];
- *OneError* – implying a single bitflip to simulate the most common error scenario, occurring in approximately 8% of server cases [3][15]. This model makes the simulator generate an error at a specific address and bit position;
- *MultipleErrors* – describing a memory with multiple corrupted bits to simulate scenarios representing from 0.044% to 0.066% of server cases [3][15];
- *One2MultipleErrors* – one bit corrupted initially followed by multiple bitflips to simulate 0.154% to 0.176% of the multiple-error scenarios [3][15]. The simulator executes the OneError configuration; next, it performs the MultipleErrors format to evaluate the system behavior from one to multiple errors, one common memory scenario;
- *BitFlipProbability* – enables to configure of the following bitflip modes and probability rates: (a) probability rate of a bitflip occurring in every tick; (b) the

maximum number of bitflips can occur in one cycle; (c) random or specific error address; (d) module memory subject to error occurrence; (e) the address distance between bitflip occurrence; (f) range allowed to bitflip inside the address (bit position range); (g) probability rate to generate a bitflip out of the address range; and (h) a seed for generating random errors.

5.1.5 Operating System (OS)

Absimth implements a simple distributed OS that uses specific memory space; all data used by OS is neither passed nor allocated in the memory defined by the configuration. OS loads instructions from one task into memory per time or executes the instruction in application load mode. OS executes a specific number of user-defined instructions (quantum); once the quantum is completed, OS schedules the next task. Absimth employs a global Round-Robin with random-order task choices within each quantum as default scheduling. When the execution cycles reach the quantum defined by the user, OS saves the processor context, looks for the next task according to the scheduling algorithm, and loads its context into the processor. The current Absimth version does not allow disabling the default OS or implementing a customized OS.

5.1.6 Reports

At the end of the application execution, Absimth creates (i) a trace report of CPU, memory, and instruction; and (ii) a simulation report containing the following information:

- *Programs* – containing a list of programs executed, each one containing the (i) name and identification of the program, (ii) initial memory address, (iii) instruction length, (iv) initial data address, (v) stack size, (vii) a total of allocated memory used, (vii) last allocated address, (viii) first data address used, (ix) total of data address used, (x) last data address used, (xi) processor identifier (processor + core), (xii) processor number, (xiii) core number, (xiv) processor type, (xv) the total of cycles executed by the application, (xvi) task identification and (xvii) information if the program was executed with success;
- *Memory* – comprising the number of instructions, data, and total reading and writing operations;
- *Memory* faults – encompassing a list of physical memory addresses with error and the associated error type. An error injected is classified as INVERTED when the program execution does not access the error address; if the program accesses the error address, the error is classified as FIXED or UNFIXED,

depending on the success of the error correction algorithm;

- *Memory controller* – containing information about the numbers of memory reads and writes performed by the memory controller module;
- *Processor list* – containing a list of processors with the (i) identification, (ii) type, (iii) core number, and (iv) number of the last tick executed;
- *General information* – including (i) the entire simulation execution time, (ii) the total number of ticks, and (iii) the maximum tick for each core.

5.2 Absimth Execution Flow

The Absimth execution flow goes through two macro phases. The first phase, called Task Initialization, defines the simulator initialization activities; the second phase, called Task Simulation, performs the execution, insertion of errors, and application task monitoring.

5.2.1 Task Initialization

Absimth initializes loading the application settings defined by the designer; in this step, the simulator allocates the memory areas of each task and maps these tasks to the processors of the target architecture.

The simulator checks the *PeripheralAddressSize* parameter, which contains the memory size allocated to each peripheral, allowing mapping peripherals into the memory addresses. Thus, if the developer creates a specific module, the simulator allocates part of the memory address for this module operation. Afterward, Absimth loads the application tasks, forwarding the following items to OS: (i) the chosen processor identification and target architecture; (ii) task identification; (iii) total memory used according to the stack size configured at program compilation time, and (iv) a reference for application loading.

5.2.2 Task Simulation

Figure 63 displays that Absimth randomly selects the processor order execution inside a quantum - i.e., a predefined number of clock cycles, simulating random concurrency among processors. Although the order of processors is random, all processors must execute a quantum before starting a new random sequence of processor execution. After choosing the processor, OS schedules the task that must be executed in each processor. For each task in each processor, Absimth executes an instruction and waits for the next cycle.

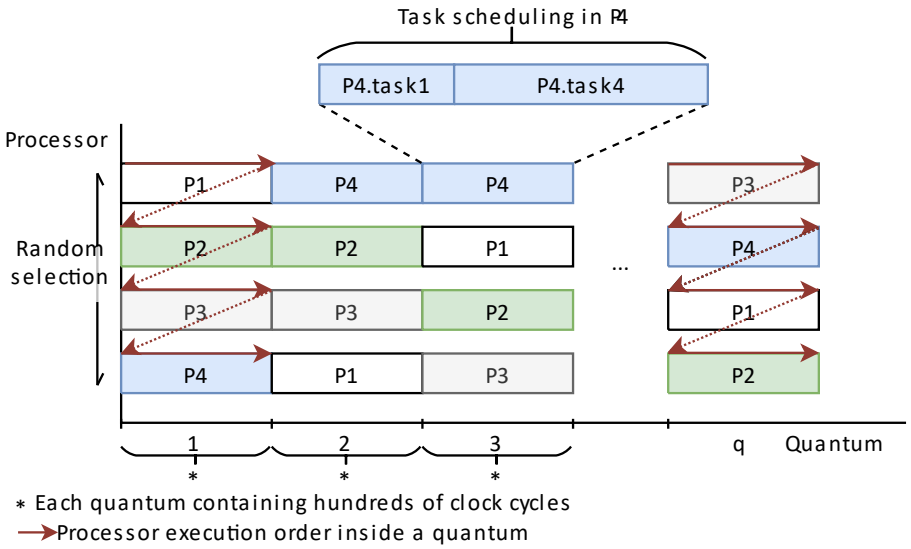


Figure 63. Example of Task Simulation phase encompassing four processors (P1…P4) execution during q quanta of simulation. This figure emphasizes intra-quantum scheduling of P4, covering task1 and task4 [126].

Absimth implements data consistency to evaluate whether an application is only accessing data from the process reserved area and control sequencing verification to mitigate code execution errors. The simulator aborts the application execution if any of these issues are detected.

5.3 Memory Bitflip Observation

Figure 64 shows the memory bitflip observation processes, exemplifying three synthetic tasks executing in a single processor connected to a DDR4 through a memory controller that performs reading and writing operations using Hamming ECC.

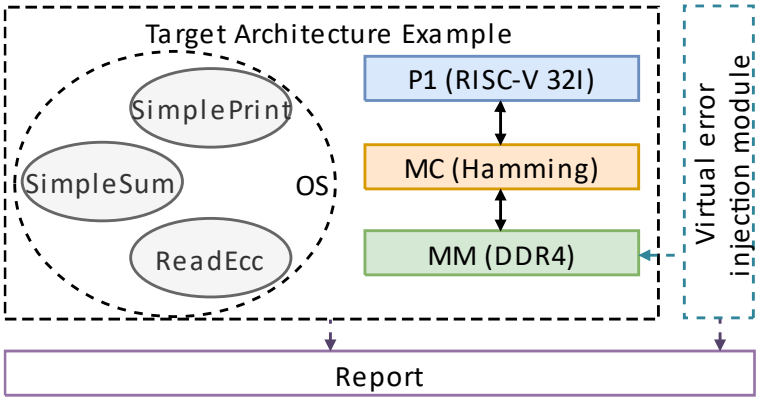


Figure 64. Target architecture and tasks employed in the simulation example [126].

5.3.1 Synthetic Application and Hardware Description

Figure 65 describes *SimplePrint*, *ReadEcc*, and *SimpleSum* - three synthetic low-complex tasks developed to observe the behavior of an application operating over a RAM with bitflip. Figure 65(a) displays the small source code of the program *SimplePrint*, which

only sends a message to the standard output. Figure 65(b) shows the *ReadEcc* source code. The program starts by executing a loop to simulate reading and writing in the final position of the memory allocated for this task; since OS allocates the initial memory area for the program code and the final memory area for the program data. Subsequently, the program reads the information provided by the memory controller, informing which address has an error. This error information enables the beginning of reading the affected page, making the memory controller change the application encoding and OS transparently. Figure 65(c) exhibits the *SimpleSum* source code, a simple program that sums and returns two values.

```
#include "../library/absimth.hpp"
void main() {
    print_str("HELLO");
}
```

(a) *SimplePrint*

```
int main() {
    int a = 1, int b = 499999;
    return a+b;
}
```

(c) *SimpleSum*

```
#include "../library/absimth.hpp"
int main() {
    int *eccLocation = (int *)0x4;
    int *zeroAdd = (int *)0x0;
    int appIniAdd = read_initial_add();
    int eccAddContError = *(eccLocation - appIniAdd);
    int len = 50000;
    int arr[len];
    for(int k = 0; k < len; k++)
        arr[k] = k;
    int errorAdd = *(zeroAdd - appIniAdd + eccAddContError);
    for(int k = 0; k < len; k++)
        arr[k] = *(errorAdd + k);
    return 0;
}
```

(b) *ReadEcc*

Figure 65. Source code of three synthetic tasks [126].

Figure 66 shows the Absimth configuration, encompassing (i) memory size reserved for hardware modules; (ii) processors used, in this case, two RISC-V 32i; (iii) memory configured with a DDR4 based on MT40A2G8 model [120] but with a smaller size, since for these applications require few kilobytes.

```
----- HARDWARE ------
Reserved Peripheral Address size for
Modules:0x00000008
PROCESSOR
Processors Model=RISCV32i
Number of Processors=2
Frequency=1000Mhz

MEMORY
Name=smallDDR4
Frequency=1000Mhz
Maximum memory bandwidth=18000000000 Bytes/s-
(18.0GB/s)
Latency for memory access=20.0 nanoseconds
Total of address=0x00a12200
Channel mode=SINGLE_CHANNEL
Word size=72
Lines per clock=2
Column Address Strobe (CAS) latency=10
Module amount=2
Rank amount=1
Chip amount=9
Bank Group amount=4
Bank amount=4
Cell=330, 1000

------ CUSTOM MODULES LOADED ------
MEMORY
Controller type=HammingMemoryController
Fault injection type=OneError; error position=0x3E8

------ PROGRAMS LOADED ------
Operational System
Cycles by Program=5

Task mapping
at CPU_0, program=simplePrint
at CPU_0, program=readEcc
at CPU_1, program=simpleSum
```

Figure 66. Simulation Setup [126].

Absimth currently is not compatible with the C Standard Library; it implements a

proprietary library with (i) math functions such as *log*, *pow*, and *multiplication*; (ii) standard definitions like *bool* and *itoa*; (iii) string and memory functions as *strchar*, *memcmp*, and *memset*; and (iv) functions to interact with the user interface, printing the current value for debugging or discovering the initial process address.

5.3.2 Memory Inspector

Absimth allows us to look in-depth at the processor status or memory during each execution or in the final process. Figure 67 presents the *Memory Area Inspector*, exemplifying a memory data region. The memory information is grouped in 32 bits, with eight columns per line, for better screen use and to demonstrate a more significant amount of simultaneous data. Data containing errors are colored in red, in this case, at address 0×3E8. When selecting an address, the rightmost column shows its physical breakdown. Consequently, in addition to the *Address* field, it shows the following fields: (i) *Module*, (ii) *Rank*, (iii) *Bank Group*, (iv) *Bank*, (v) *Row*, (vi) *Column*, and (vii) *Height range*. The simulator does not provide information on the chip, as each chip receives one byte.

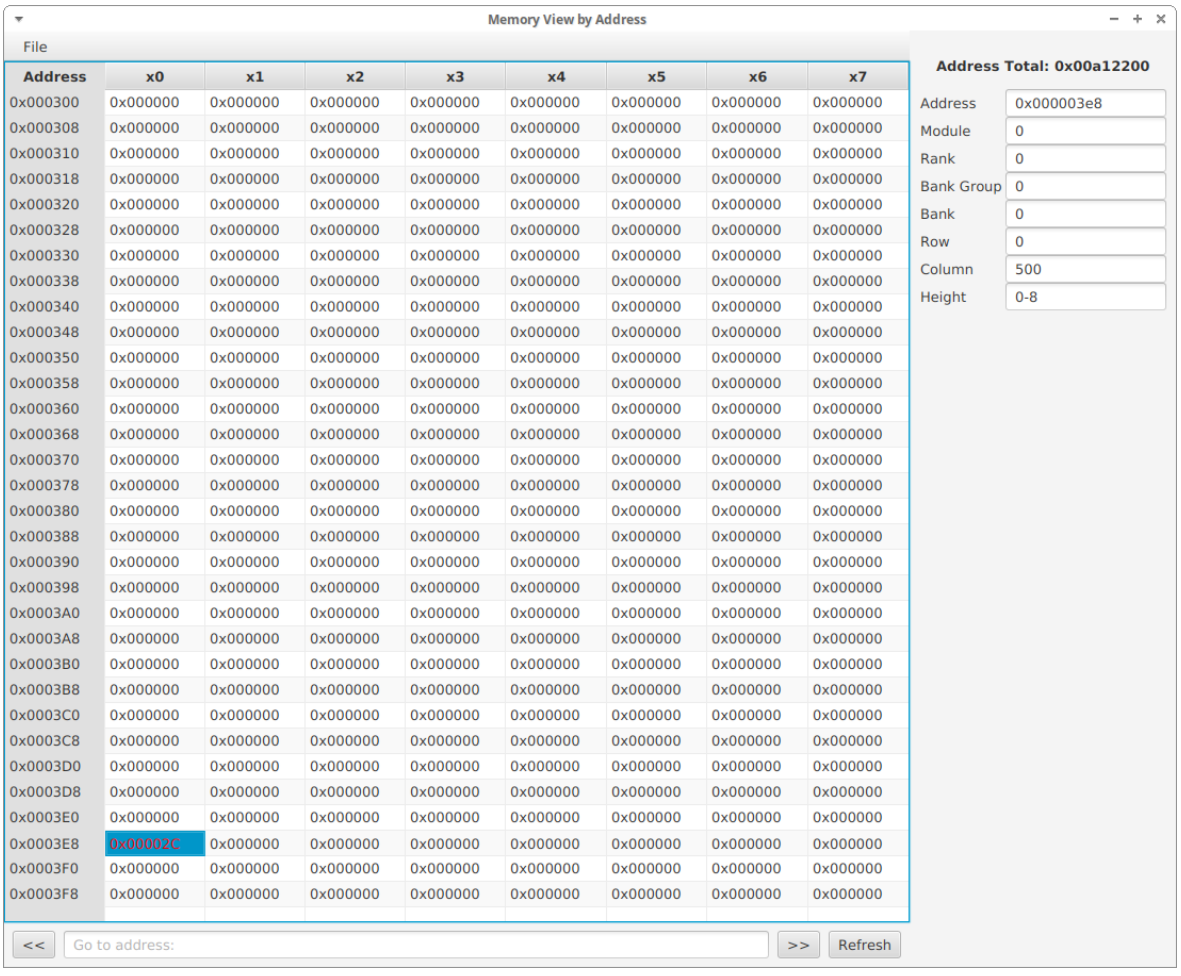


Figure 67. Memory Area Inspector tool, containing an error in address 0×3E8 [126].

Absimth provides the default DDR memories hierarchical model, enabling the

designer to navigate throughout all the memory addresses, giving the spatial location. Besides, the designer can provide a customized memory structure by adjusting the memory template of Absimth.

The spatial location is crucial to evaluate and simulate scenarios where memory has a location more susceptible to bitflips than other locations – e.g., the rightmost location of the memory is close to a heated area because of the processor place. Figure 68(a) presents the highest memory level for navigating the structure of memory modules, ranks, and chips. The leftmost column reports the module number, rank, and addressing range. In case of an error in any data, the units are colored in red. It is possible to go into any chips to see the internal memory structure, exploring the error neighborhood in a three-dimensional (3D) format. The hierarchical view continues selecting a given module, rank, and chip. For instance, since Chip 0 of Module 0 and Rank 0 contains errors colored in red in Figure 68(a), Figure 68(b) shows this chip visualization, encircling the organization of memory banks inside each bank group.

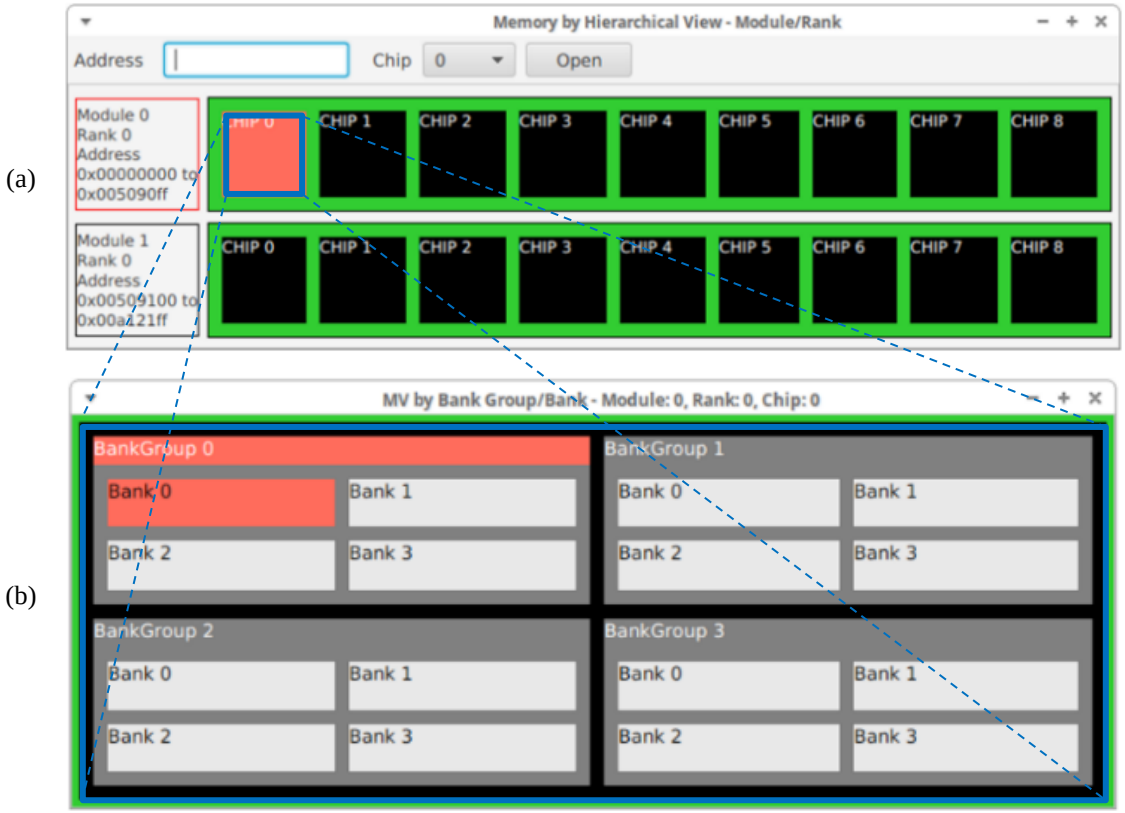


Figure 68. (a) The highest memory level, including memory modules, rank, and chips, shows that Chip 0, Module 0, and Rank 0 contain at least one bit with error; (b) Memory bank organization inside the bank groups. The Bank 0 of Bank Group 0 contains errors since it is colored in red [126].

Figure 69 displays the Absimth window when selecting the memory bank 0, which enables browsing the bank by memory address and physical cell placement - i.e., row, column, and height.

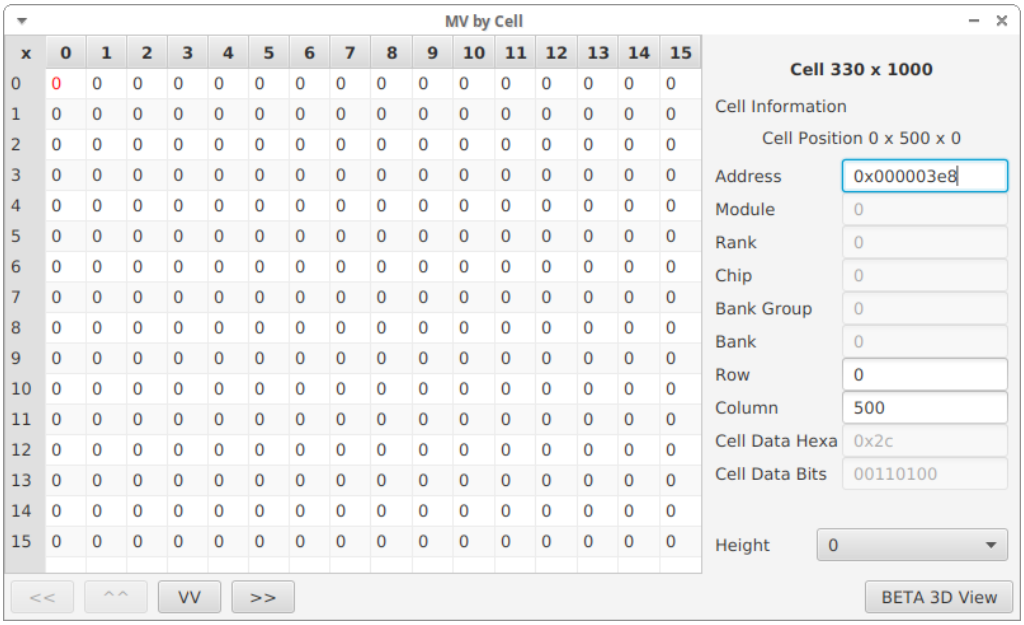


Figure 69. Memory cell window; this view displays a single error on bit 0 (colored in red) of address 0x3E8 [126].

The designer can also see the 3D view of a memory cell (along with height); Figure 70 exemplifies a 3D view.

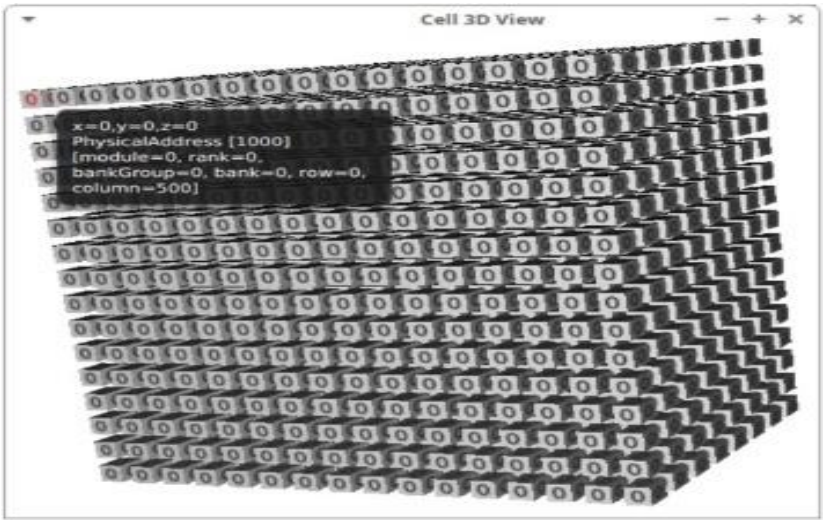


Figure 70. 3D memory cell preview window [126].

5.3.3 Execution Investigation

The execution exemplification employs a memory controller with Hamming ECC and a virtual module generating a bitflip at address 0×3E8 to simulate a stuck bit. The *Processor Management* tool helps to watch the exact moment the instruction or data is corrupted and enables selecting a given processor and task. Each instruction of the assigned task can be executed step by step on the designated processor - the OS is bypassed in this execution.

Figure 71 shows the Processor Management tool running ReadEcc on CPU_0, covering the object code of the task, the status of the processor registers, and the memory address accessed at the instruction moment.

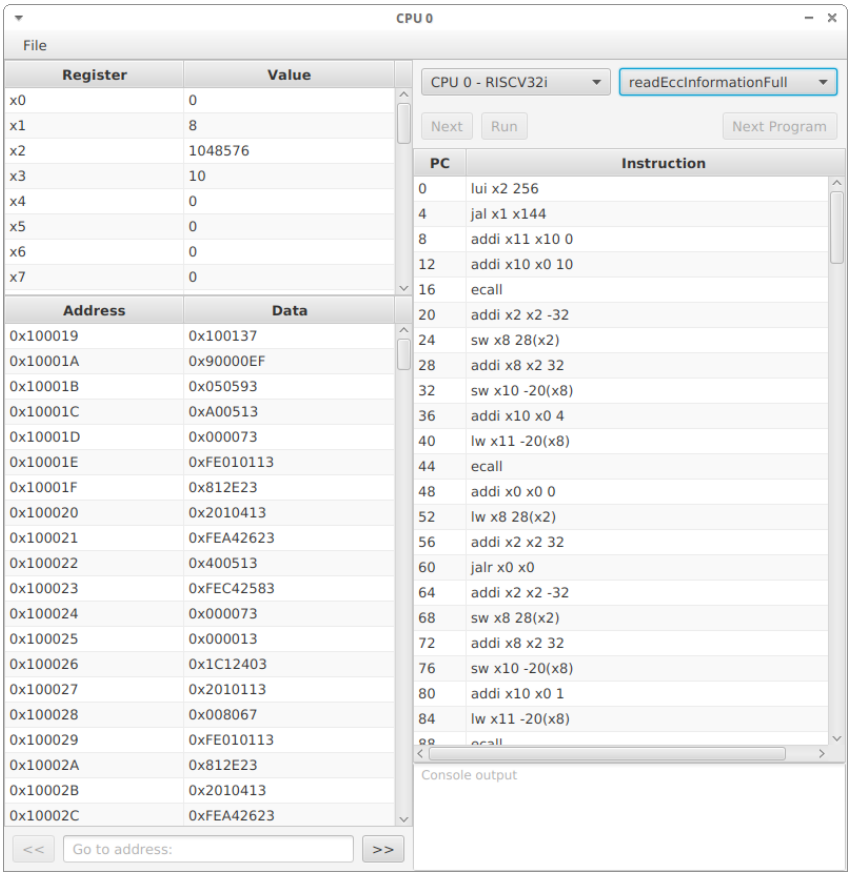


Figure 71. Processor Management tool covering processor registers, task code objects, and memory addresses [126].

At the end of the simulation, Absimth provides a timeline window of the target architecture processors displaying the execution of each processor task, as demonstrated in Figure 72.

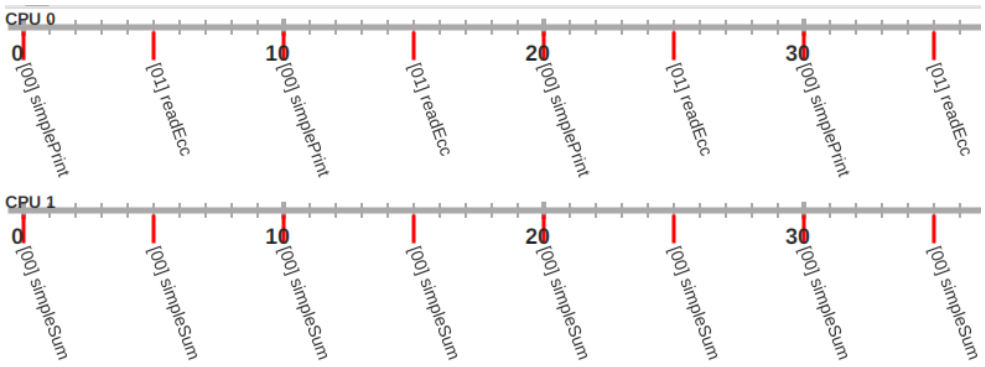


Figure 72. Window for viewing the timeline of all processors [126].

5.3.4 Execution Report

Absimth finishes the target architecture simulation generating the report of Figure 73, containing statistics on (i) tasks performed; (ii) data traffic among processors; (iii) execution status (with error or success); (iv) number of data reads/written from/in memory; (v) memory positions with errors; and (vi) data traffic and instructions on the memory controller with each ECC used.

```
[SIMULATION]
Application bytes: 3145803

[PROGRAMS]
simpleSum
  programId=2
 MEMORY
  initialAddress=0x000800cf
  instructionLength=0x00000014
  initialDynDataAddress=0x000800e3
  stackSize=0x00040000
  totalOfMemory=0x00040014
  lastAddress=0x000c00e3
 DYNAMIC MEMORY USED
initialMemoryAddressUsed=0x000c00c9
totalMemoryAddressUsed=0x00000005
lastMemoryAddressUsed=0x000c00ce
 CPU
  cpu=0
  core=1
  cpuId=1
  cpuType=RISCV32i
  totalOfTicks=40
 OTHERS
  task=0
  successful=true
simplePrint
  programId=0
 MEMORY
  initialAddress=0x00000002
  instructionLength=0x00000032
  initialDynDataAddress=0x00000034
  stackSize=0x00040000
  totalOfMemory=0x00040032
  lastAddress=0x00040034
 DYNAMIC MEMORY USED
initialMemoryAddressUsed=0x0003fff9
totalMemoryAddressUsed=0x00000008
lastMemoryAddressUsed=0x00040001
```

```
CPU
  cpu=0
  core=0
  cpuId=0
  cpuType=RISCV32i
  totalOfTicks=131
  OTHERS
  task=0
  successful=true
readEcc
  programId=1
 MEMORY
  initialAddress=0x00040034
  instructionLength=0x0000009b
  initialDynDataAddress=0x000400cf
  stackSize=0x00040000
  totalOfMemory=0x0004009b
  lastAddress=0x000800cf
 DYNAMIC MEMORY USED

InitialMemoryAddressUsed=0x1fffffff
totalMemoryAddressUsed=0xe0040035
lastMemoryAddressUsed=0x00040034
  CPU
  cpu=0
  core=0
  cpuId=0
  cpuType=RISCV32i
  totalOfTicks=156
  OTHERS
  task=1
  successful=true

[MEMORY]
Instructions
Number of instructions read: 408
Number of instructions written: 900
Number of instructions r+w: 1308
#Bytes of instructions read: 918
#Bytes of instructions written: 2025
#Bytes of instruction r+w: 2943
```

```
Data
Number of data read:  52
Number of data written:  28
Number of data r+w:  80
#Bytes of data read: 117
#Bytes of data written: 63
#Bytes of data r+w: 180
Total
Number of totals read: 460
Number of totals written: 928
Number of totals r+w: 1388
#Bytes of totals read: 1035
#Bytes of total written: 2088
#Bytes of total r+w: 3123

[MEMORY ECC STATUS]
address=0x000003e8, type=INVERTED,
position=[0]

[MEMORY CONTROLLER]
READ HAMMING_SECDEC: 115
WRITTEN HAMMING_SECDEC: 232

[Processor List]
CPU 0
Cpu Type = RISCV32i
Core 0
CpuId 0
  Last Tick at: 287
Core 1
CpuId 1
  Last Tick at: 40

[General information]
Simulation took 2472 milliseconds
Total of Ticks:327
Last Ticks At:287
```

Figure 73. Simulation Report [126].

The report demonstrates the exact point of bitflip in Section *Memory ECC Status*, allowing us to explore the memory area and final status of the memory. Section *Programs* - item *Successful* is another relevant point determining if the application finished the execution successfully or was affected by the bitflip occurrence.

5.3.5 Trace Report

Absimth traces all accesses for evaluating a program's data, memory, instruction, and flow. This configuration is normally disabled due to performance reasons. Figure 74 demonstrates a small part of an execution containing CPU status and identifier, program counter, and the register values in decimal; RAM execution, indicating if it is reading (R) or writing (W), address, and the data; and the instruction executed.

```
[CPU00] pc=503, reg=[0, 2220, 1048384, 12, 0, 0, 0, 0, 1048416, 0, 1, 67, 67, 67, 67,
67, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[MEM] R - 0x00000203 - 0xfef40523
[INS] sb x15 -22(x8)
[MEM] W - 0x0003ffde - 0x43434300
```

Figure 74. Trace report of CPU, memory, and instruction executed [126].

5.4 Benchmark Exploration

Table 6 displays the results of a benchmark executing in scenarios combining different error rates and computational resource combinations. The benchmark applications were selected from PARSEC [87][88], BOTS [121], and HARDINFO [122] to evaluate a large specter of application behavior, and encompasses 15 applications: (i) Binary Sort, (ii) Black Scholes, (iii) Blowfish, (iv) Bubble Sort, (v) CRC8, (vi) Factorial, (vii) Fibonacci, (viii) Frequent Pattern Growth, (ix) Greatest Common Divisor, (x) Hanoi, (xi) Huffman Encoding, (xii) Insertion Sort, (xiii) JKiss32, (xiv) Matrix Multiplication, (xv) Prime Number.

We explored four combinations of programs and computation scenarios: (1) single-task, single-processor [i, xv] - all the benchmark programs running separately in a single processor (CPU 0); (2) multi-task, single-processor [xvi] - Bubble Sort and CRC8 executing in the same processor; (3) single-task, multiprocessor [xvii] - Bubble Sort and CRC8 executing on processors 0 and 1, respectively; (4) multi-task, multiprocessor [xviii] - Bubble Sort and CRC8 executing on processor 0, and two Bubble Sort tasks executing on processors 1 and 2.

Each tick took around 0.075ms without tracing or 4ms with log access trace on a CPU i7-9750h with 64 GB DDR4.

Absimth does not implement cache hierarchy; all instructions access the memory controller. A prefetch can access more addresses to anticipate the accesses of the application, increasing the probability of loading an address with corrupted data. However, bitflips can also occur after a prefetch. The likelihood of bitflip occurrence is small for both cases, being disregarded.

Table 6. Simulation summary [Author].

	Benchmark information						WithoutError			SingleBitflip			One2Many			RandomNearError				RandomArbitraryError			
#	a	b	c	d	e	f	WE	HM	RS	WE	HM	RS	WE	HM	RS	NºBF	WE	HM	RS	NºBF	WE	HM	RS
i	782	782	7038	738	656	208										1				4			
ii	4210	4210	37890	10766	1536	4546										10	OK	OK		9	OK		
iii	225329	225329	2027961	1167094	43718	316652										334	NOK	NOK		341	NOK		
iv	4183	4183	37647	9785	993	4936										10	OK	OK		9	OK	OK	
v	100949	100949	908541	219267	13897	127079										150	NOK	NOK		153	NOK		
vi	678	678	6102	693	571	184										1	OK	OK		4	OK		
vii	1310	1310	11790	5114	347	1766										0				4			
viii	30622	30622	275598	137994	8206	43493										14	NOK	NOK		50	NOK	NOK	
ix	691	691	6219	792	581	178	OK	OK	OK	NOK	OK	OK	NOK	OK	OK	1				4			
x	3635	3635	32715	18166	1112	4542										9			OK	8	OK		OK
xi	34349	34349	309141	114747	8416	44570										52				76			
xii	2721	2721	24489	9351	859	2901										7				8			
xiii	1352	1352	12168	2456	635	1193										3				4	NOK		
xiv	3880	3880	34920	6256	1153	3994										9	OK	OK		9		OK	
xv	1641	1641	14769	7092	371	2058										3				4			
xvi	8363	8363	75267	15525	2307	9022										207				205	OK		
xvii	100949	105132	946188	228892	14890	131995										207				205			
xviii	8363	16729	150561	34855	4293	18884										210				219			

Legend: "OK" and "NOK" means the simulation finished without or with error, respectively.

The Benchmark Information contains the (a) last tick, (b) sum of all ticks executed by the processors, (c) total of instructions read and written in bytes, (d) total of data read

and written in bytes, and the number of (e) writes and (f) reads to/from memory.

All simulations were performed considering the five following error scenarios: (1) *WithoutError* – a straightforward scenario without bitflip occurrence; (2) *SingleBitflip* – a minimum error occurrence scenario; (3) *One2Many* – states a scenario with minimal occurrence of errors, and after a period, multiple errors occur; (4) *RandomNearError* – a scenario with a random number of bitflips placed in a nearby neighborhood; and (5) *RandomArbitratyError* – a scenario with a random number of bitflips arbitrarily placed.

Table 7 details the *RandomNearError* and *RandomArbitratyError* characteristics, according to the following: (α) probability of bitflips occurring during each clock; (β) interval containing the minimum and maximum numbers of bitflips that can occur during each clock period; (γ) range containing the minimum and maximum distances of the address of the next bitflip from the occurrence of the previous bitflip – the objective is to explore bitflips inside or outside the same word; (δ) range containing the minimum and maximum distances of the bitflip in the same memory address – the objective is to explore bitflips within a memory module (note that the same word is physically placed in more than one memory module); (ω) probability of the next bitflips occurring outside the intervals defined in (γ) and (δ).

Table 7. Characterization of bitflip scenarios [Author].

Scenario	α	β	γ	δ	ω
Bitflips in random places	0.2%	[0, 2]	-	-	100%
Bitflips in a nearby neighborhood	0.2%	[0, 2]	[0, 3]	[1, 16]	0.2%

Additionally, for each one of the five error scenarios, we verified situations Without ECC (WE) and with Hamming (HM) and Reed Solomon (RS) codes.

Table 7 produces results consistent with the ones explored on Google and Microsoft servers [3][15], opening opportunities for a vast behavior study of memory devices and memory controllers. Table 7 highlights two results: (i) the HM encoding increases the correct execution probability slightly, but it is not acceptable for critical applications, and (ii) the RS encoding can handle all the scenarios evaluated but with high energy consumption and memory area costs. Considering the above scenarios, one of the possibilities to increase the execution quality is to employ ECCs with different correction potentials according to the occurrence of errors in the memory module.

6 EXPERIMENTAL RESULTS

We conducted several experiments assessing the tradeoff between reliability and power dissipation to validate the dynamic ECC memory controller proposal. Figure 75 shows that the experimental results encompass four groups of experiments, described next.

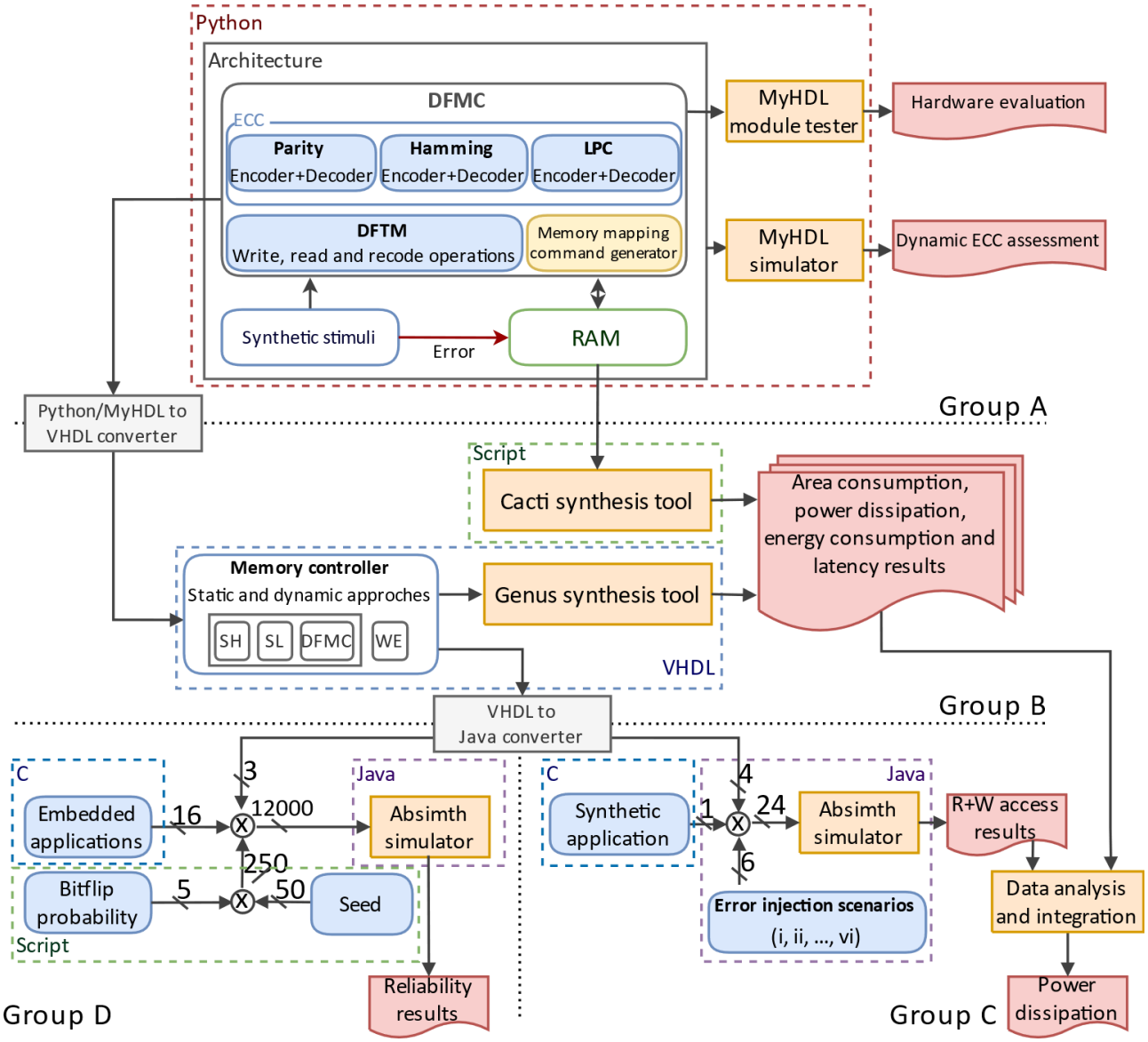


Figure 75. Experiments performed to validate the memory controller proposal [Author].

Section 6.1 contains the experiments of Group A, including the main tools and languages used to describe and validate the basic operations of memory controllers derived from DFMC. Besides, the ECC dynamic operation is explored using a stimulus tool emulating a processor reading/written data together with random error rate insertions.

Section 6.2 describes the experiments of Group B, related to evaluations of area consumption, power dissipation, and latency of all memory controller modules and types for static and dynamic ECC approaches.

The experiments of Group C are described on Section 6.3.2, encompassing synthetic applications executing in a homogeneous multiprocessor architecture, considering the static and dynamic approaches of fault tolerance and some error injection scenarios.

Finally, Section 6.3.3 contains the experiments of Group D, which uses the same target architecture of the experiments of Group C but evaluating a benchmark execution containing several embedded programs.

6.1 Hardware Implementation and Validation

This work uses MyHDL resources [114] to implement and test the target architecture used in the experiments. MyHDL is a framework with a free and open-source package for using Python as a hardware description and verification language. In addition to having all the Python ecosystem, MyHDL ensures their hardware designs' reliability and functionality through comprehensive unit and integration testing (uncommon in hardware design). Moreover, MyHDL also supports waveform viewing by tracing signal changes in a Value Change Dump (VCD) file, enhancing hardware verification.

Figure 76 describes the main activities carried out with MyHDL support. All the DFMC functionality, highlighting its main modules and interactions, were described in Python, along with a synthetic set of stimuli representing sequences of writings and readings, with random bitflip occurrences, in a synthetic memory. The stimulus set was used to evaluate the behavior of the DFMC and verify if the module description followed the specification. Likewise, the ECC modules were verified and validated at this same abstraction level. Once the Python DFMC description was validated, MyHDL generated a VHDL DFMC version correct by construction.

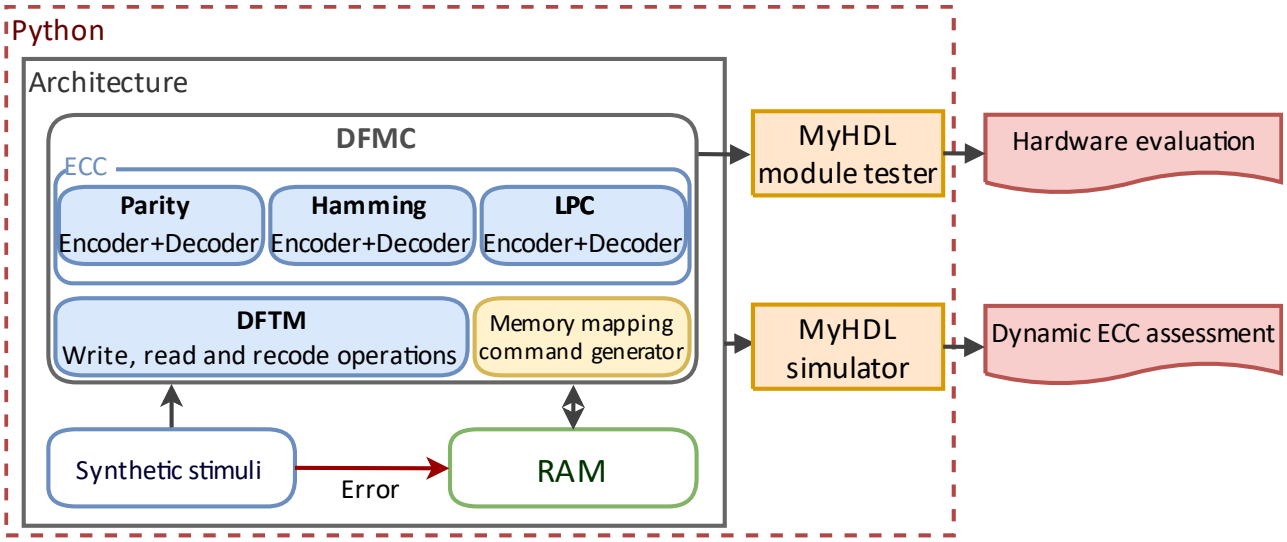


Figure 76. MyHDL framework using synthetic stimulus sequences for simulating the DFMC behavior and the dynamic ECC approach – Group A of experiments shown in Figure 75 [Author].

6.1.1 DFMC and RAM Implementation and Basic Assessments

We implemented DFMC containing the following modules, which were assessed using MyHDL module tester:

i. A reliable *Internal memory*, consisting of 2048 bits, allows configuring 256 blocks, each one managing 2 million addresses connected with 4 GB RAM. The internal memory is called reliable because we envision implementing it with a rad-hard memory to minimize the error occurrence. Therefore, the simulated model does subject this memory to error injection;

ii. The *RAM process*, responsible for managing the operation and controlling which memory receives the information;

iii. The *Threshold process* to evaluate if the memory block needs to change the ECC according to the rate of errors;

iv. The *Configuration process* which allows configuring ECC and the operation mode;

v. The *Recoding process* responsible for changing the ECC from an entire memory block;

vi. The *ECC module* including Parity, Hamming, and an adapted LPC;

vii. Two Memory Mapping/Command Generators (MMCGs), one for each RAM, containing the minimum instruction set [115][116]. MMCG is responsible for communicating with 4 GB RAM, receiving the logical address and converting it to the physical address, waiting for power-on initialization, precharge SDRAM banks, and making the memory self-refresh.

Figure 77 illustrates the macro view of the hardware architecture developed and the environment used for testing[8], which encompasses Stimulus Module, DFMC, and RAM. The frontend content queue messages were not developed.

MyHDL provides the MyHDL Test Module describe in Figure 76, an integrated test unit for hardware development, where we assessed the DFMC modules and RAM.

RAM was evaluated employing all the commands and steps necessary to read and write data - i.e., *nop, active, delay*, *post-edge* signal, and the sequences of flags like *address_input, read_input*, and *write_input*.

DFMC includes a basic MMCG to perform memory commands and mapping, enabling reading/writing data in a commercial RAM. Although the MMCG module was pretested by XESS Corporation [115][116], we performed a sequence of read-and-write

[8] The complete architecture and test environment description is at the link described in [117].

commands to evaluate if the memory mapping and command generator is converting and working correctly.

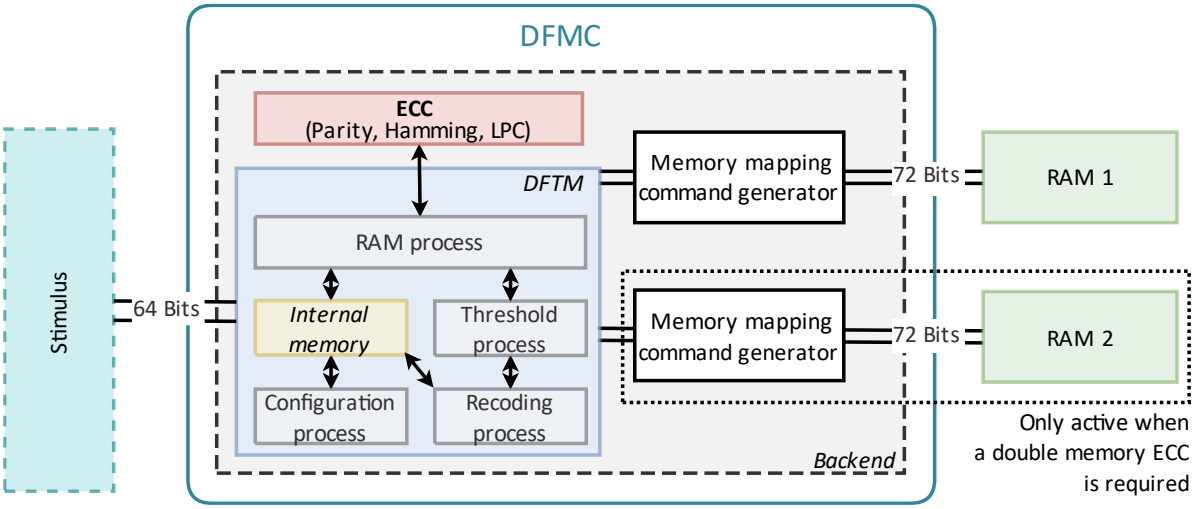


Figure 77. Macro view of the developed architecture [Author].

All encoders and decoders of Parity, Hamming, and LPC, encompassing the ECC module of DFMC, were tested exhaustively (all 64-bit positions) considering their error correction capacity.

Finally, DFTM was assessed with some working scenarios described next:

i. A simple read-write operation using DFTM without ECC or bitflip – to guarantee the most straightforward case;

ii. A single write followed by multiples read on the same address – to evaluate the synchronism between all modules;

iii. A simple read-write configuration of DFTM with Parity and without bitflip – to evaluate if DFTM and Parity are working correctly;

iv. With one bitflip and configurated with Parity – to evaluate the Parity when having one bitflip;

v. Stating with Parity, reading the address with a bitflip, and read-and-writing again but now in Hamming – to evaluate if DFTM changes the ECC;

vi. Starting with Parity, reading the address with bitflip, reading and writing the address again but now with Hamming, receiving another bitflip and reading again to evaluate if it was changed to LPC – to test changing the ECC multiples times;

vii. Starting with Parity, reading and writing an address receiving bitflips - to evaluate if DFTM changes to LPC and maintains with this ECC.

6.1.2 Stimulus Module Description

Figure 78 presents the basic behavior of the Stimulus Module, which produces sequences of address and data ensembling a CPU operation, thus, producing scenarios that enable assessing the DFMC implementation.

Initially, the module programs how DFMC will operate, for example, configuring the static or dynamic way to handle ECC, the ECC threshold, and the number of memory blocks. Once programmed DFMC, Stimulus executes a pre-defined number of ticks (TicksSimulation); while this number is not reached, Stimulus emulates a PE sending and receiving random data for a random address to/from the memory controller. A bitflip can occur in each simulation tick, depending on the error injection probability – a programmed value. The error injection reads the random data and alters a programmable number of bits at random positions (the validation process used from 1 to 3 bitflips). The random data with or without bitflips is written in memory using the produced random address. Subsequently, the memory controller reads the same address returning the data read. This data is then evaluated and eventually corrected by DFMC.

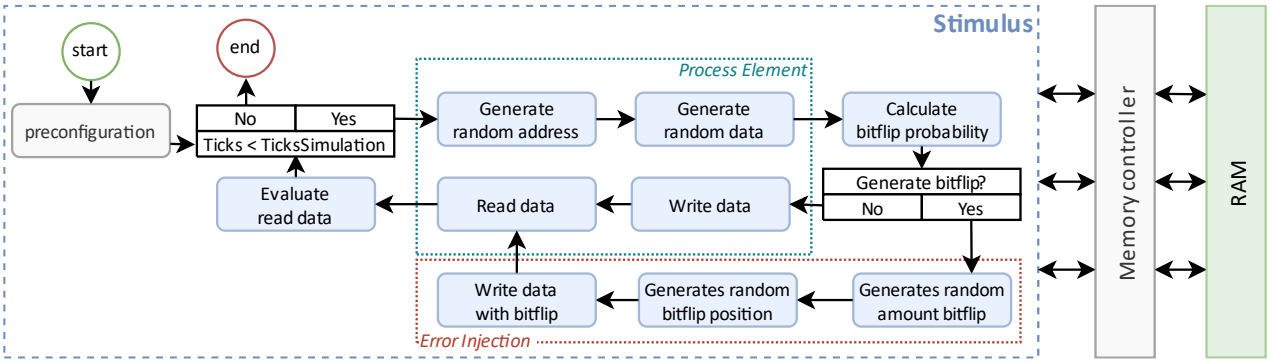


Figure 78. Stimulus module for synthetic data production and error injection [Author].

6.1.3 DFMC Adaptability Assessment

The primary objective of this set of experiments is to evaluate the ECC impact on DFMC for specific error scenarios, assessing the DFMC adaptability under some error rate variations. Depending on the NAE of a block during the system execution, DFMC results in either a low error-correction efficacy but high performance or a high error-correction efficacy but low performance.

The experiment was conducted using software simulation configured to execute for 20 million ticks, enabling conducting an acceptable timeframe while still capturing meaningful data. The **Threshold process** is evaluated at every TCS, decreasing NAE during the system execution by the programmed NER. To ensure efficiency and minimize the number of evaluations, we defined *TCS* as 1 million ticks, resulting in 20 executions of

the Threshold process during the evaluation timeframe.

The Stimulus module takes 10 ticks to read and 16 to write, totaling 26 ticks for a single write-and-read operation. This approach allows each threshold cycle to include up to 38,461 read/write operations, $Maximum\ RAM\ operation = \frac{TCS}{Total\ of\ Ticks\ for\ a\ w+r\ operation} = \frac{1000000}{26} = 38461.53$. The error injection occurs sequentially after a write operation on the same address, adding 34 ticks, totaling 60 ticks when a bitflip occurs. In case of have a bitflip in all operations the minimum read/write operation was 16666,67, $Minimum\ RAM\ operation = \frac{TCS}{Total\ of\ Ticks\ for\ a\ w+r\ operation\ with\ bitflip} = \frac{1000000}{60} = 16666.67$.

Table 8 displays the Stimulus configuration for interacting with a 64-bit DFMC combined with two modules of 4 GB RAM, each memory module divided into 256 blocks, giving 2 million addresses for each block. The random address is in the range [0;8000000) for concentrating the test area in four blocks and minimizing 64 times the time spent on the simulation with extra memory allocation - i.e., 8 million 64-bit addresses (8 bytes), resulting in 64 MB, consequently, 64 MB of memory allocation against the total 4 GB RAM. The data was generated in any random value for 64 bits - i.e., a value in the range [0; 2^{64}).

Table 8. Stimulus test configuration [Author].

Description	Value
Address range	[0, 8000000)
Data range	[0, 2^{64})
Error probability	(i) 0.004%, (ii) 0.008% (iii) 0.012%, (iv) 0.016%
Number of address per block	2 M
Number of blocks	256
Number of blocks used	4
Threshold Cycle Size (TCS)	1 M
Timeframe size (in cycles)	20

DFMC was evaluated with a minimal yet comprehensive error coverage that encompasses scenarios with four bitflip rates: (i) 0.004%, (ii) 0.008%, (iii) 0.012%, and (iv) 0.016%, enabling us to explore some error severity levels.

This experiment execution also requires programming DFMC with basic operating parameters. Table 9 presents the main parameters used in this experiment: Ecout, NER, and the ECC thresholds used in the simulation.

Table 9. DFMC test configuration [Author].

Description	Value
Errors counted for each error detection of the ECC decoder (Ecount)	1
Number of Errors to Reduce (NER)	1
Parity threshold	[0, 1]
Hamming threshold	[2, 3]
LPC threshold	[4, ∞)

It is crucial to mention that decoders can often detect an error's occurrence but cannot extract the precise number of errors that occurred in a given word. For example, a Parity code detects an odd number of errors but cannot identify this odd number. Other codes, such as LPC, allow accounting for a certain number of errors; however, numbers of errors higher than the detection capacity can be captured with values different from those that occurred. DFMC mitigates this issue by programming a fixed value defined in the Ecount register each time the decoder reports an error or accepts the number of errors returned by the encoder. This experiment sets Ecount = 1; thus, DFMC counts a bitflip for each error occurrence detected by the decoder. Besides, this experiment uses NER = 1 for assessing the DFMC behavior regarding a fine error grain variation. Finally, we defined the ECC thresholds considering a very conservative interval that would allow a quick ECC change according to the dynamicity of the error occurrence computed by the NEA variable. Thus, we define the intervals [0,1] for Parity and [2,3] for Hamming, and the block starts to have LPC coding for more than four errors.

All blocks start programmed with Parity. On the one hand, in the case of NAE > 1, DFMC automatically switches the block to operate with Hamming ECC at the end of the Threshold process. Similarly, if NAE > 3 inside a block protected by Hamming, DFMC further switches to LPC at the end of the Threshold process. On the other hand, at the end of the Threshold process, if NAE < 4 inside a block protected by LPC, DFMC switches to LPC; if NAE < 2 inside a block protected by Hamming, DFMC switches to Parity.

Figure 79 to Figure 82 depict the four-error probability described in Table 8, showing the number of bitflips when ECC changes over the Threshold process, which occurs in a given $cycle = \frac{Current\ Tick}{TCS}$, based on the configuration, for the four blocks tested.

Figure 79 shows the simulation configurated with 0.004% probability, having bitflips in blocks 0 and 2. In Block 0, the issues encountered were insufficient to change the ECC, maintaining Parity throughout the entire simulation. Block 0 demonstrates the most usual scenario, where only a few bitflips happen.

The behavior of DFMC works as expected, maintaining the lowest ECC due to the low error rates. Nevertheless, in Block 2, the ECC changes from Parity to Hamming during cycle 8, when the number of errors exceeded the threshold configurated for Parity. Subsequently, in the following cycle evaluation, the ECC reverts to Parity. Note that the costs of changing the encoding of a block are high, such as energy consumption and latency, not to mention the computational effects for the application using the block. To mitigate this problem, the mechanisms proposed in this work use the ECC threshold range, TCS and

NER, which can be dynamically adjusted.

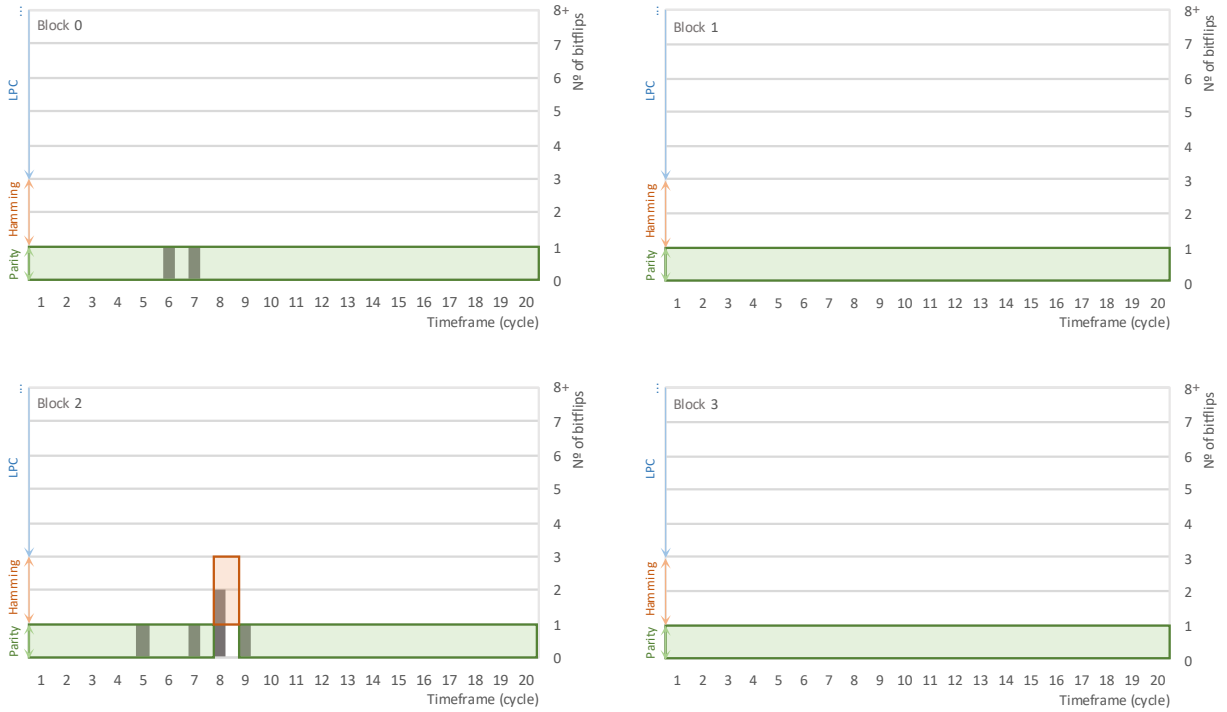


Figure 79. (i) Experiment to evaluate the threshold for ECC change according to NAE in each cycle. Four blocks with 0.004% bitflip probability were evaluated during a 2M tick-timeframe [Author].

Figure 80 shows the operation scenario configurated with 0.008% error probability. This scenario highlights periods of an aggressive number of errors in a small timeframe, making DFTM briefly activate the second memory module to use LPC and guarantee data quality, as demonstrated in Block 0 – cycle 7 and Block 2 – cycle 8.

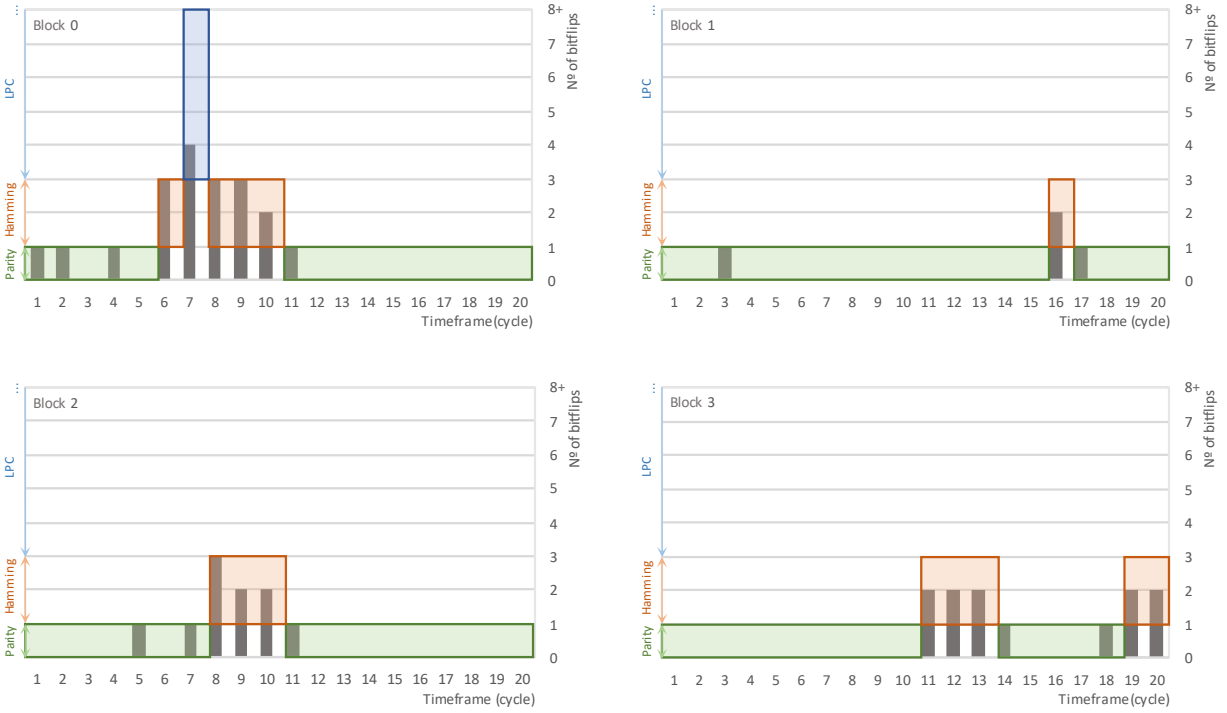


Figure 80. (ii) Experiment to evaluate the threshold for ECC change according to NAE in each cycle. Four blocks with 0.008% bitflip probability were evaluated during a 2M tick-timeframe [Author].

Figure 81 and Figure 82 demonstrate the most aggressive error scenarios, having 0.012% and 0.016% bitflip probability, respectively. The high number of bitflips rapidly requires the most effective ECC, enabling the employment of the second RAM.

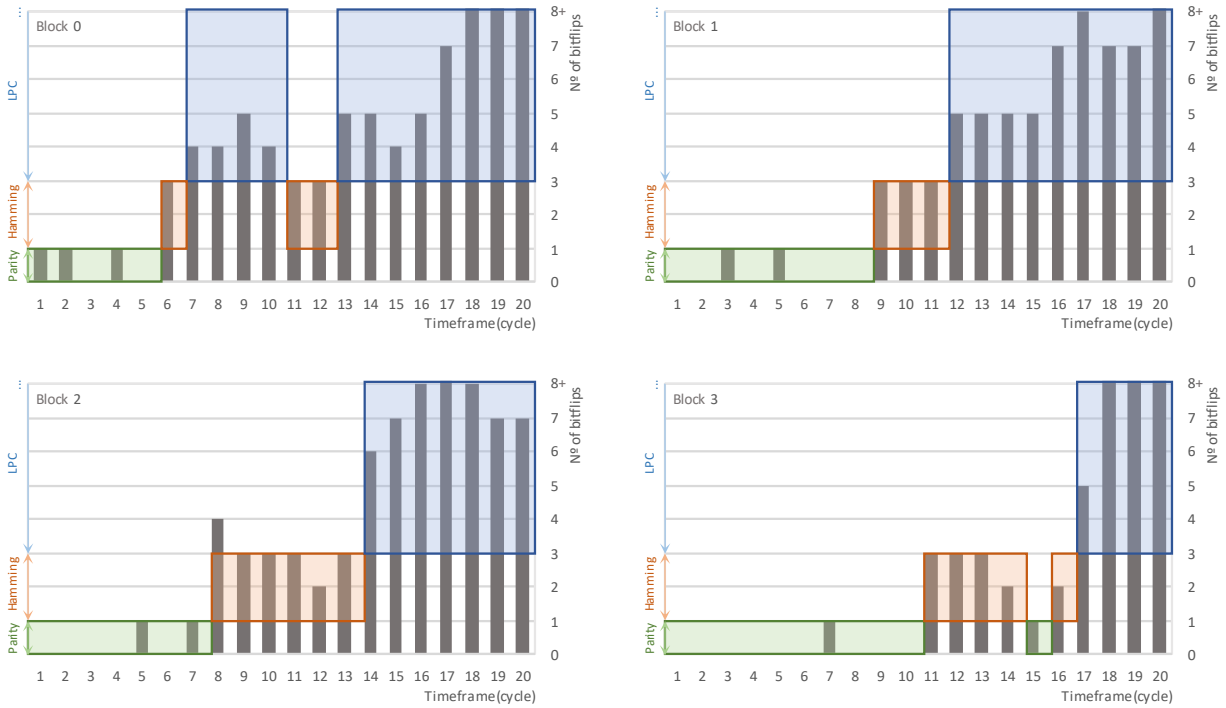


Figure 81. (iii) Experiment to evaluate the threshold for ECC change according to NAE in each cycle. Four blocks with 0.012% bitflip probability were evaluated during a 2M tick-timeframe [Author].

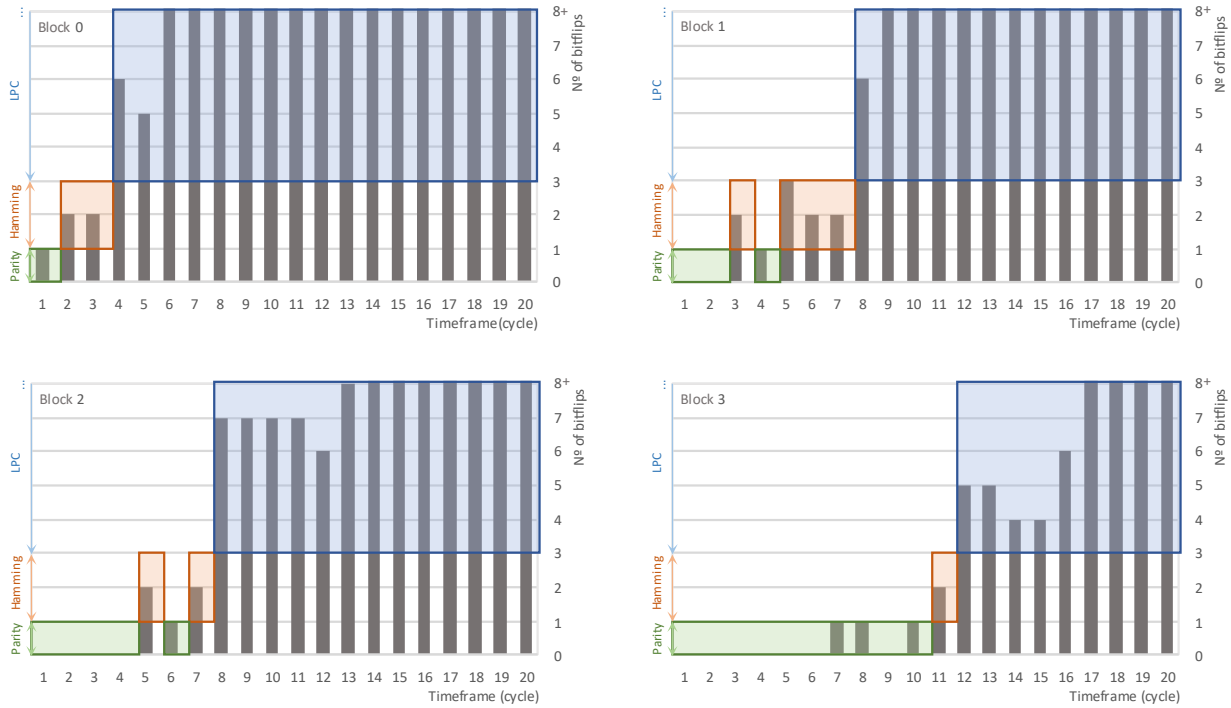


Figure 82. (iv) Experiment to evaluate the threshold for ECC change according to NAE in each cycle. Four blocks with 0.016% bitflip probability were evaluated during a 2M tick-timeframe [Author].

We emphasize that the aggressiveness of Scenario iv in Block 0 implies a high error rate, requiring the utilization of the most robust ECC and its continuous maintenance. The

scenario starts with Parity; during the first cycle, NAE reaches the upper limit of the Parity threshold, requiring upgrading to the Hamming ECC, which remains in effect for two cycles until seven bitflips are detected. The ECC is further upgraded to the most robust option, LPC, to ensure effective error correction, becoming capable of fixing all the next errors.

Note that, due to the high tax of errors, the probability of the Hamming being unable to handle this amount of errors is very high, showing the high ability of DFMC to increase the data delivery over static Hamming.

This experiment demonstrates the DFMC adaptation for all proposed scenarios: (i) keeping a fast ECC without the capacity to fix bitflip under the best environment – i.e., without or a few bitflips; (ii) selecting the best ECC encoding for the current situation; and (iii-iv) under a highly stressed environment changing to a high and secure ECC rapidly and keeping. Table 10 shows the seven threshold possibility cases, where only scenarios i - Parity to Parity and iii - Parity to Hamming can recognize errors but not fix them.

Table 10. DFTM threshold possibility cases [Author].

Case	Current ECC	Next ECC	# of errors to change	Examples in Figure 80		Error correction capacity of the current ECC
				Block	Cycle	
i	Parity	Parity	1	0	1-2	False
ii	Hamming	Hamming	1	0	11-12	True
iii	Parity	Hamming	> 1	0	5-6	False
iv	Hamming	LPC	> 1	0	6-7	True
v	LPC	LPC	> 0	0	7-8	True
vi	LPC	Hamming	0	0	10-11	True
vii	Hamming	Parity	0	3	14-15	True

6.2 Memory Controllers and RAM Synthesis

MyHDL can convert a design to Verilog or VHDL, allowing rapid circuit/application development and experimentation. Group B of Figure 75, detailed in Figure 83, displays that the proposed memory control module described in MyHDL was converted to VHDL, and subsequently, the VHDL code was synthesized using the Genus synthesis tool [123]. Additionally, the other memory controllers (SL, SH, and WE) are implemented based on the DFMC description – more details are in Section 4.8.

The Genus was configured to synthesize hardware employing 28nm CMOS technology with a 2 GHz clock, operating under 1V and 25ºC conditions to estimate the area consumption, power dissipation, latency, and energy consumption for some timing aspects. Memory was synthesized to a 4GB RAM with CACTI [124], operating under the same hardware conditions used in Genus synthesis. The internal DFMC memory also uses CACTI to drive the general idea of consumption and space of one memory with tolerant characteristics.

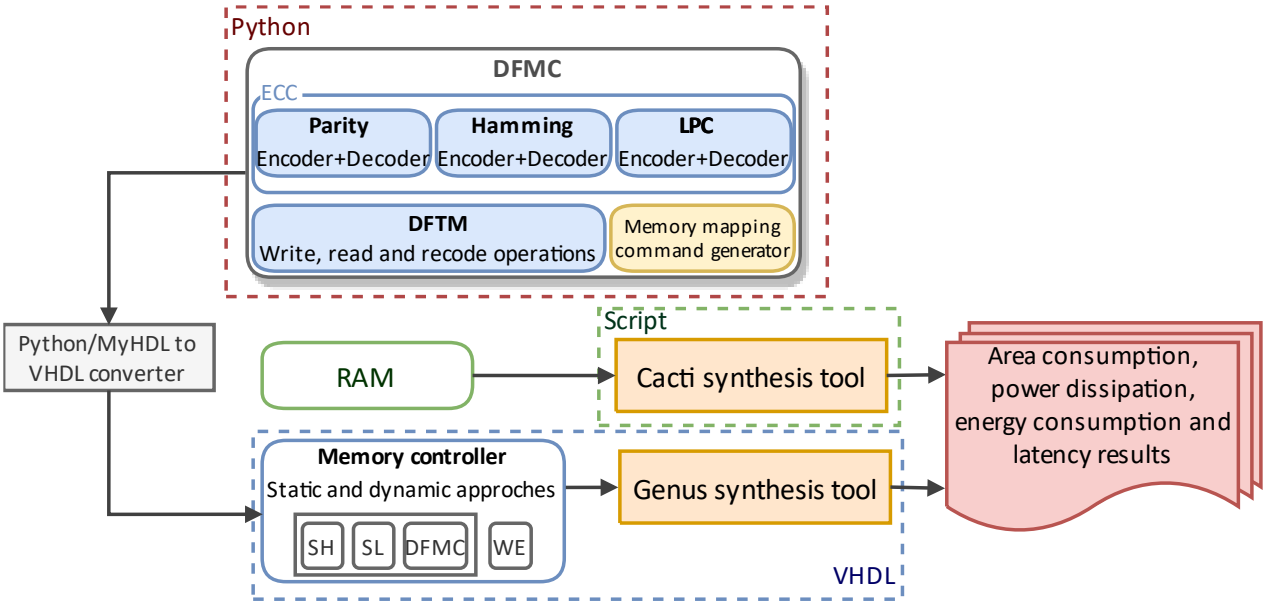


Figure 83. MyHDL framework using Python to VHDL converter to generate four memory controller architectures, which are synthesized with Genus tool to get area usage, power dissipation, energy consumption and latency– Group B of experiments shown in Figure 75 [Author].

Each module was computed separately on Genus, allowing every part of the process to be evaluated. Table 11 to Table 14 show the DFMC, SL, SH and WE area consumptions, respectively. DFMC consumes 8205.67 nm² and SL 5124.51 nm²; i.e., 60.1% more area than SL. This difference comes from the memory controller dynamicity requiring to implement more two ECCs – Parity and Hamming that totalize 1167.03nm² (~22.78%), and DFTM, responsible for the ECC dynamic management with 1914.13nm² (~37.32%)[9]. Figure 84 illustrates the area used for all memory controllers.

Table 11. DFMC and RAM area used in nm² [Author].

		Description	Sequential	Logic	Network	Total	%
DFMC	ECC	Parity	229.79	71.97	87.24	389.00	10.77%
		Hamming	245.45	334.23	198.36	778.04	21.55%
		LPC	736.35	1075.59	632.19	2444.13	67.68%
		Total	**1211.59**	**1481.79**	**917.79**	**3611.17**	**44.01%**
	DFTM	Internal Memory – 256B	3.90	264.43	124.40	392.73	18.47%
		Recoding Process	216.89	140.84	99.05	456.78	21.48%
		Threshold Process	120.12	65.28	46.79	232.19	10.92%
		Configuration Process	431.66	114.24	180.19	726.09	34.14%
		*RAM Process	100.27	117.31	101.45	319.03	15.00%
		Total	**872.84**	**702.10**	**551.88**	**2126.82**	**25.92%**
	2x MMCG		1361.74	521.26	584.69	2467.69	30.07%
	Total		**3446.17**	**2705.15**	**2054.36**	**8205.68**	**0.06%**
RAM	2x 4 GB DDR4		147740.00	10015340.00	4711550.00	14874630.00	99.94%
Total			**151186.17**	**10018045.15**	**4713604.36**	**14882835.68**	**100.00%**

Legend: *RAM Process compose by block finder, ECC evaluator, and R+W process.

[9] RAM Process contains the R+W Manager, which should be subtracted, $Additional\ Area = DFTM\ Area - R + W\ Manager = 2126.81 - 212.68 = 1914.13nm^2$.

Table 12. SL - Memory controller and RAM area used in nm² [Author].

	Description	Sequential	Logic	Network	Total	%
SL	LPC	736.35	1075.59	632.19	2444.13	47.69%
	R+W Manager	66.85	78.21	67.63	212.69	4.15%
	2x MMCG	1361.74	521.26	584.69	2467.69	48.15%
	Total	**2164.94**	**1675.06**	**1284.51**	**5124.51**	**0.03%**
RAM	2x 4 GB DDR4	147740.00	10015340.00	4711550.00	14874630.00	99.97%
Total		**149904.94**	**10017015.06**	**4712834.51**	**14879754.51**	**100.00%**

Table 13. SH - Memory controller and RAM area used in nm² [Author].

	Description	Sequential	Logic	Network	Total	%
SH	Hamming	245.45	334.23	198.36	**778.04**	34.97%
	R+W Manager	66.85	78.21	67.63	**212.69**	9.56%
	1x MMCG	680.87	260.63	292.35	**1233.85**	55.46%
	Total	**993.17**	**673.07**	**558.34**	**2224.58**	**0.03%**
RAM	1x 4 GB DDR4	73870.00	5007670.00	2355780.00	**7437320.00**	99.97%
Total		**74863.17**	**5008343.07**	**2356338.34**	**7439544.58**	**100.00%**

Table 14. WE - Memory controller and RAM area used in nm² [Author].

	Description	Sequential	Logic	Network	Total	%
WE	R+W Manager	66.85	78.21	67.63	**212.69**	14.70%
	1x MMCG	680.87	260.63	292.35	**1233.85**	85.30%
	Total	**747.72**	**338.84**	**359.98**	**1446.54**	**0.02%**
RAM	1x 4 GB DDR4	73870.00	5007670.00	2355780.00	**7437320.00**	99.98%
Total		**74617.72**	**5008008.84**	**2356139.98**	**7438766.54**	**100.00%**

Figure 84 shows that our SH implementation increases the area compared to a memory controller without ECC by 53.8%, the SL by 254.3% and DFMC by 467.3%.

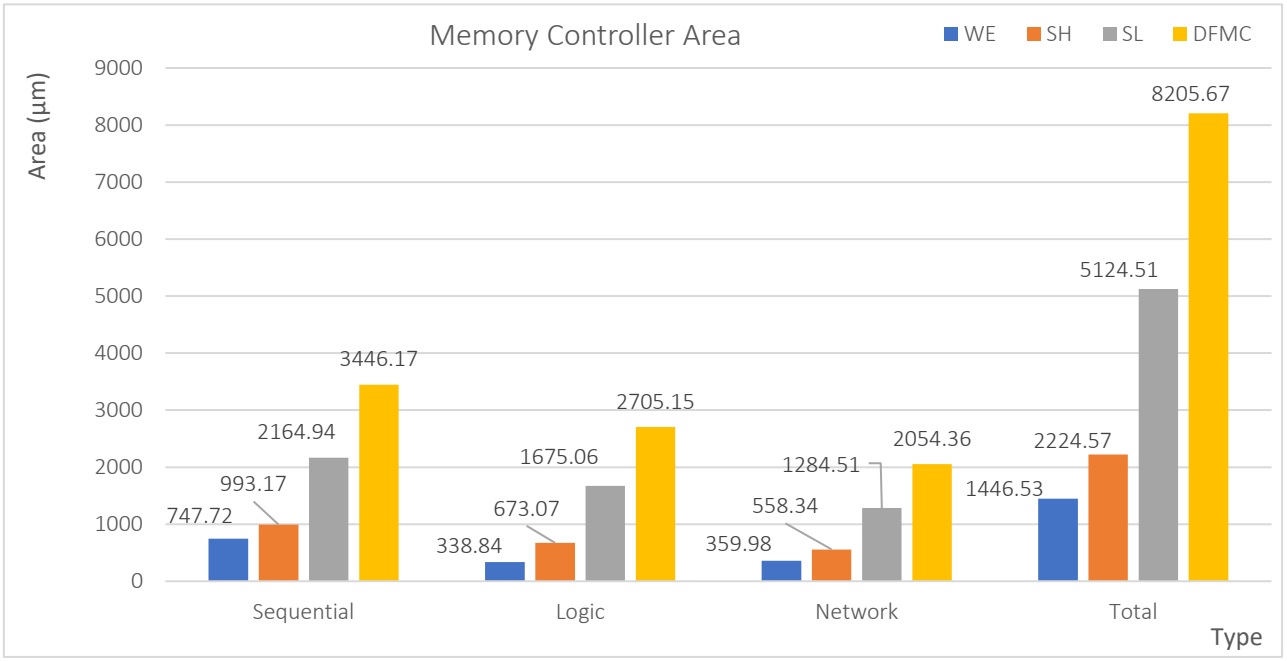


Figure 84. The used area for all synthesized memory controllers by type and total [Author].

Note that the SL, SH and WE memory controllers are simplified versions of DFMC, and for this reason they result in lower synthesis costs (such as area) and, in general,

operating costs (such as energy consumption and power dissipation). On the other hand, DFMC provides a customized ECC operation according to the application needs in the face of error scenarios, which allows optimizing reliability and energy consumption tradeoff.

Comparisons of memory controllers used in sections 6.3.2 and 6.3.3 are performed by executing synthetic and embedded applications in the Absimth simulator. These comparisons require estimating power dissipation, energy consumption and latency costs obtained with specific operations, such as reading and writing with memory access, data encoding and decoding for each ECC used and a memory block recoding; these costs are presented from Table 15 to Table 18, as well as in Figure 85. Additionally, we decided to program DFMC to operate only with Hamming and LPC, to make a fair comparison with static controllers that use only Hamming (SH) or only LPC (SL); consequently, the subsequent tables do not present DFMC costs with parity. The DFMC programmed to operate only with Hamming and LPC is called DFMC-DHL or just DHL.

Table 15 presents the power dissipation, timing of the critical path, and energy consumption values for ECC, each memory controller module and RAM. Table 15 also shows these values divided by reading and writing operations when executing *a single memory address* - note that Threshold Process, Recoding Process and MMCG have the same critical path and energy consumption for both reading and writing operations.

Table 15. Power dissipation, critical path and energy consumption for the ECC encoder/decoder, DFMC and the 4GB DDR4 RAM [Author].

	Submodule			Power (µW)	Critical path (ns)	Energy (fJ)
DFMC	ECC	Encoder	WE	0.000	0.000	0.000
			Parity	12.096	0.186	2.245
			Hamming	26.082	0.666	17.370
			LPC	40.315	0.819	33.026
		Decoder	WE	0.000	0.000	0.000
			Parity	10.789	0.546	5.891
			Hamming	32.283	1.707	55.108
			LPC	46.426	2.287	106.194
	DFTM	Internal Memory	Read	1817.773	0.050	91.424
			Write	924.579	0.129	119.043
		R+W Manager	Read	1.803	0.340	0.766
			Write	0.756	0.046	0.043
		Block Finder	Read	0.180	0.034	0.077
			Write	0.076	0.005	0.004
		ECC Evaluator	Read	0.270	0.051	0.117
			Write	0.113	0.007	0.007
		1x Threshold Process Base (R|W)		4.368	1.270	5.548
		1x Recoding Process Base (R|W)		11.018	1.615	17.794
	1x MMCG (R|W)			13.970	1.657	23.148
RAM	1x 4GB DDR4		Read	168967.170	2.650	447812.000
			Write	67602.419	6.785	458665.000

Table 16 demonstrates the energy consumption for every single read/write operation of the WE, SH and SL memory controllers with and without DDR4. The energy consumption values are the sum of the ECC, R+W manager and MMCG; as these modules operate in parallel, the table provides the worst-case timing among them. Note that, on the LPC, MMCG is called twice and operates parallelly; thus, the time is the same, but the energy consumption is doubled.

Table 16. Energy consumption and critical path for both reading and writing of WE, SH and SL memory controllers [Author].

	Memory controller	ECC + MC		ECC + MC + DDR4	
		Critical path (ns)	Energy (fJ)	Critical path (ns)	Energy (pJ)
Read	WE	1.657	0.024	2.650	447.836
	SH	1.707	0.079	2.650	447.891
	*SL	2.287	0.153	2.650	895.777
Write	WE	1.657	0.023	6.785	458.688
	SH	1.657	0.041	6.785	458.706
	*SL	1.657	0.079	6.785	917.409

Legend: * Use a second memory module and MMCG.

Table 17 shows the energy consumption for every single read/write operation for each DFMC module to calculate the entire circuit in the next table. DFTM was divided into three submodules: (i) the RAM process is compounded by one read of the internal memory, R+W manager, block finder and ECC Evaluator; (ii) the Threshold process is combined by one read-and-write internal memory and the Threshold process base; and (iii) the Recode process contains one read-and-write internal memory and the Recode process. Only for this case, it was considered these modules operate sequentially, summing the energy consumption and critical path.

Table 17. Energy consumption for every single read/write DFTM module, considering ECC and DDR on the critical path [Author].

Modules		Critical path (ns)		Energy (pJ)	
		Read	Write	Read	Write
DFTM	1x RAM Process	0.475	0.107	0.092	0.091
	1x Threshold Process		1.449		0.216
	1x Recode Process		1.794		0.228

Table 18 presents the energy consumption and the critical path for reading and writing on DFMC. The entire recoding is formed by one Hamming read, six Recode processes due to the implementation logic and one LPC write; the critical path is the major of them, while the energy consumption is computing summing all individual circuits.

Table 18. Energy consumption and the critical path to read, write and recode on DFMC [Author].

Action	ECC	ECC + Memory Controller		ECC + Memory Controller + DDR4	
		Critical path (ns)	Energy (pJ)	Critical path (ns)	Energy (pJ)
Read	Hamming	1.707	0.171	2.650	447.983
	LPC	2.287	0.245	2.650	895.869
Write	Hamming	1.657	0.132	6.785	458.797
	*LPC	1.657	0.171	6.785	917.501
Recode	*Hamming to LPC	10.764	1.882	10.764	1367.024

Legend: * Second memory module use.

These values are used in the subsequent section to compare the energy consumption of the experiments with dynamic encoding approaches. Figure 85 compares critical path and energy consumption for each reading and writing operation of the memory controllers. The comparison is based on the values presented in Table 16 and Table 18. To improve the comparison between the controllers and allow evaluating the coding effect, Figure 85 displays the energy consumption and latency costs considering the DFMC operating with Hamming (DFMC-Hamming) and with LPC (DFMC-LPC).

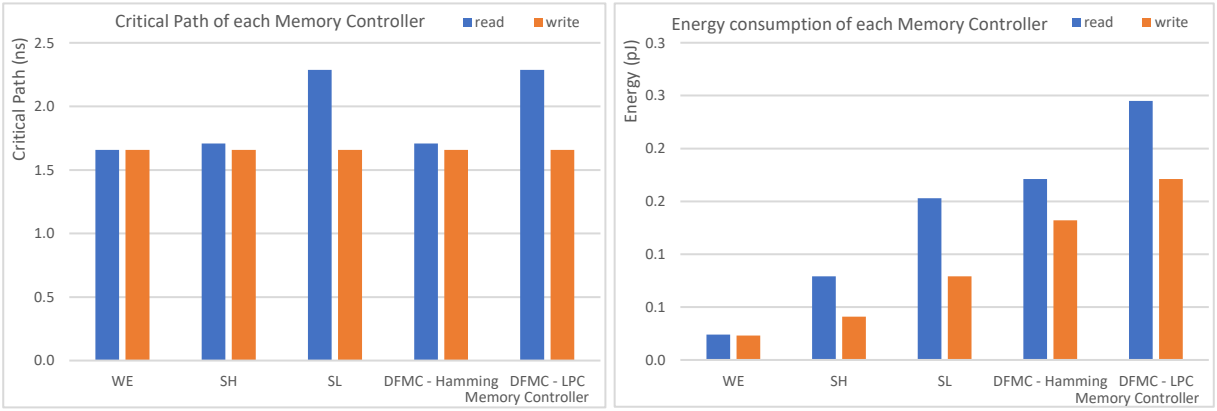


Figure 85. Comparative of the (a) critical path and (b) energy consumption of reading and writing operations of each memory controller [Author].

As observed, the inclusion of the extra module in the DFTM does not impact the critical path. However, the energy consumption of the read/write operation increases by 221% when comparing the SH with DFMC-Hamming and 116.45% when comparing the SL with DFMC-LPC.

6.3 Evaluating the Efficacy and Efficiency of Memory Controllers running Synthetic and Embedded Applications

This section compares the memory controllers – WE, SH, SL, and DFMC-DHL, to evaluate their error correction efficacy and energy consumption efficiency when executing synthetic and embedded applications on a homogeneous multiprocessor architecture, regarding several error scenarios.

6.3.1 Multiprocessor Architecture Implementation

Chapter 5 presented Absimth [125][126], an extensible fast hardware simulator used to evaluate the proposed memory controller and understand the application behavior in the server environment related in [3][15]. Absimth enables us to explore scenarios with several bitflip probabilities and facilitates the assessment of applications' behavior under intensive bitflips while using different ECCs. It also enables configuring and creating custom modules for processors, memory devices, and memory controllers. The simulator provides error injection models into memory to assess the system's behavior in the presence of memory runtime errors, providing a wide range of metrics for analysis.

To reach the behavior assessment, we built in Absimth a homogeneous multiprocessor architecture composed of RISC-V 32F [119] processors that access memory modules from a DDR4 SDRAM, specifically model MT40A1G16 from Micron [120], applying synthetic applications described in C. The multiprocessor architecture was implemented with only one memory controller for all processors, accessed through a shared bus-like communication system, to avoid data synchronization problems in case of simultaneous access by more than one processor to the same memory location. Additionally, the second DDR4 memory module is only used in the LPC operation. Figure 86 illustrates that the WE, SH, SL and DFMC-DHL memory controllers were assessed in this hardware and stressed by a virtual error injection to simulate error scenarios.

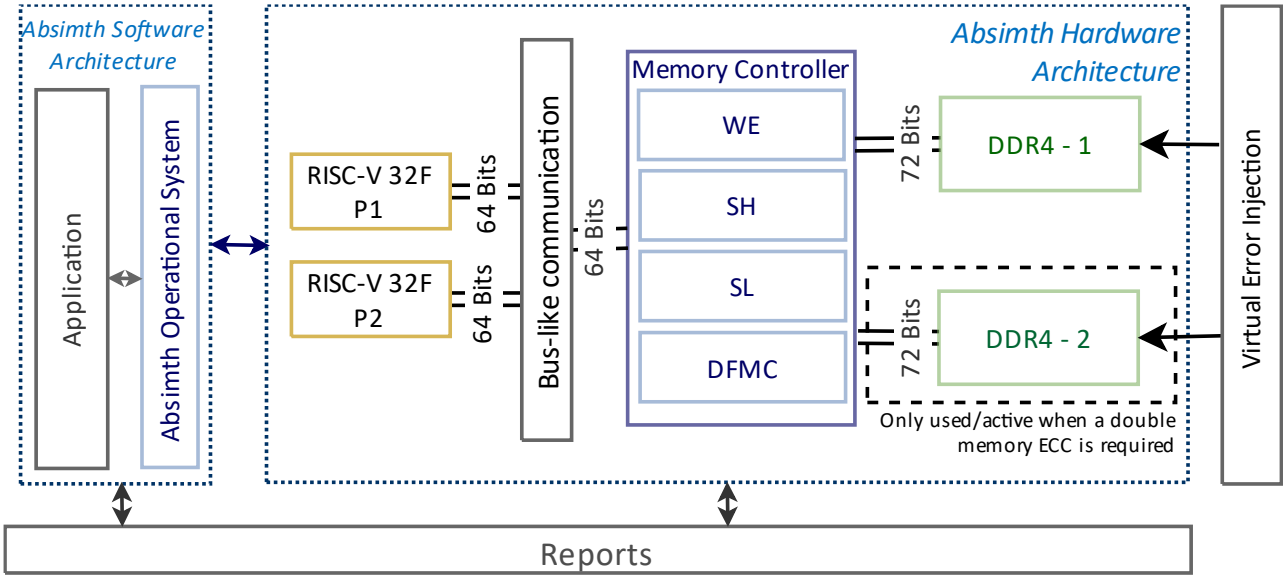


Figure 86. Multiprocessor architecture used to evaluate the memory controllers under evaluation scenarios - implemented in Absimth [Author].

The target architecture implemented in Absimth comprises four sets: (i) the software part that executes an operating system kernel of Absimth to manage applications and interface with memory controllers; (ii) the hardware part that implements the multiprocessor

with RISC-V processors, memory modules and includes a memory controller for evaluation; (iii) an error injection module that produces bitflips on the memory blocks; and (iv) a set of resources that report on the system operation.

This Thesis evaluates the dynamic memory controller proposal dividing the assessment into two moments: (i) Section 6.3.2 describes a comprehensive assessment of the scenario presented by [3][15], using a controlled application to evaluate WE, SH, SL and DFMC-DHL; (ii) and Section 6.3.3 presents a *benchmark* to evaluate multiple applications executing on error scenarios to determine the reliability of the SL, SH and DFMC-DHL memory controllers.

6.3.2 DFMC Power Dissipation and Reliability Assessments using Synthetic Applications

According to [3][15], in a period of eighteen months, approximately 91.78% of memories never fail, 8% have one bitflip, 0.154% to 0.176% have one bitflip with multiple bitflips after, and only 0.044% to 0.066% is affected by multiples bitflip at the same time. The works of Schroeder, Pinheiro and Weber [3] and Nightingale, Douceur and Orgovan [15] describe that RAM faults have spatial locality with almost 80% chance of failing again at the same physical address. Based on this scenario, where it is not possible to anticipate where failures will occur, but it is known that they will be spatially close and that the percentage of memory blocks that failed is small, a dynamic fault-tolerant approach is crucial to enhance server reliability, ensure optimal performance, and save energy.

To evaluate this dynamic approach, we chose to program DFMC with Hamming and LPC only, with Hamming being the initialization ECC. This approach, called DFMC-DHL or just DHL, allows correcting any single error and detecting double errors in the simplest configuration, meeting the reliability requirements described in [3][15] with optimized energy consumption. In DHL mode, all memory blocks start with Hamming and only memory blocks that have their error rate increased beyond the programmed threshold have their ECC modified to LPC, increasing the error correction capacity. In the experiments of this section, we defined a rigorous threshold, where upon encountering one error, the recoding submodule transitions to LPC *without reverting to Hamming*.

Figure 87 describes the experiments conducted to evaluate the performance of SL, SH, and DHL memory controllers in synthetic applications. The simulations performed six error injection scenarios, exploring different levels of aggressiveness. This evaluation allowed for assessing the power dissipation and reliability of WE, SH, SL, and DHL memory controllers.

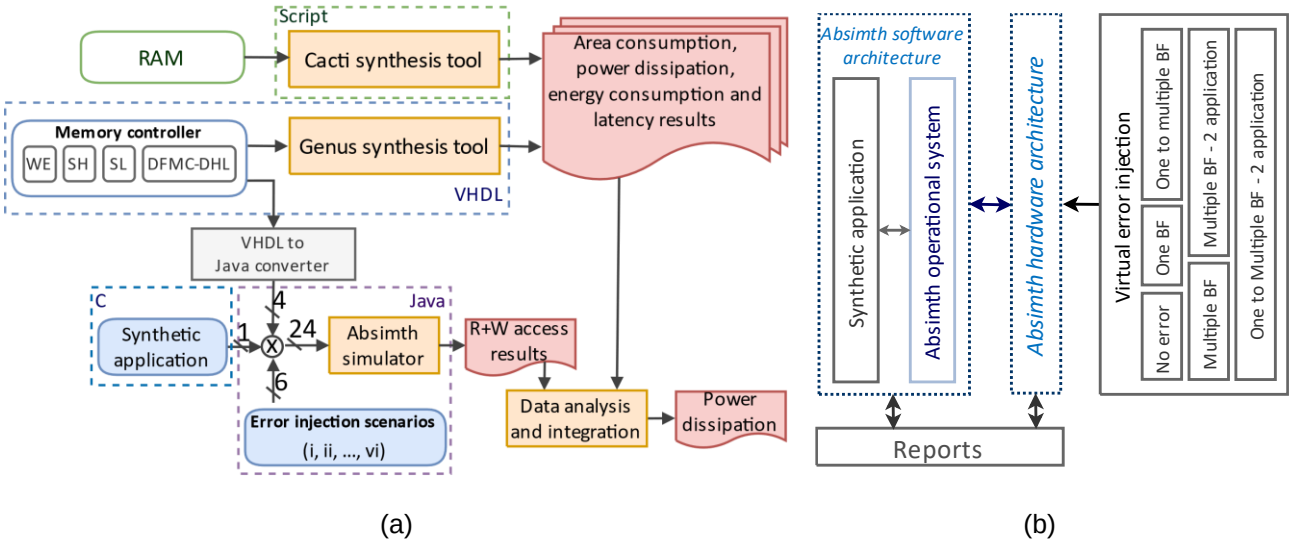


Figure 87. Details about the experiment conducted in Group C (Figure 75), including the (a) experimental setup and tools used and (b) the software/hardware interactions performed in the Absimth simulator [Author].

The DFMC-DHL configuration ensures a robust error detection and correction mechanism, allowing for more reliable data integrity. The memory block size was configured with 32000 addresses, totaling 15625 blocks managed by the DFMC – i.e., $\left\lceil \frac{Memory\ Size}{Address\ per\ Block \times 64} \right\rceil = \left\lceil \frac{4\ GB}{32000 \times 64} \right\rceil = \lceil 15625 \rceil = 15625$. This block size allows DFMC accurately handle the data area in proportion to the 4GB, giving a good tradeoff between the area size to be recoded and the number of blocks to be evaluated in the Threshold process.

A specific synthetic application was developed, enabling the evaluating of particular characteristics, such as read and write access in bursts and/or alternately; exploration of some access rates for addresses on the same or different blocks; and exploration of several read and write flows in regions with and without errors. The synthetic programs used in this experiment start executing a loop to simulate reading and writing data memory allocated for this task. Each read memory access enables evaluating whether the read address has an error; according to the number of errors, the dynamic approach implemented into DFTM changes the encoding model.

Six error scenarios were employed to explore error patterns with some aggressiveness degrees, the errors are injected in a parameterized spatiotemporal way but only in data area. The six scenarios are performed according to [3][15]:

i. Without memory error - to evaluate the execution time, amount of data transferred, and application execution; additionally, it serves as a basis for comparison with other scenarios, occurring in approximately 91.78% of cases;

ii. With one bitflip - to simulate the most common error scenario, occurring in approximately 8% of cases;

iii. With one bitflip initially and then multiple bitflips - to simulate 0.154% to 0.176% of the multiple error scenarios;

iv. With multiple bitflips - to simulate pending scenarios with multiple corrupted bits representing from 0.044% to 0.066%;

v. One bitflip and after multiples bitflips with two applications sharing the same address – to evaluate the most common error scenario running one application and then a second application in the same area; and, finally,

vi. Multiple bitflips with two applications sharing the same address – to evaluate an area constantly receiving multiple errors, executing an application followed by a second application in the same memory area.

Table 19 summarizes the results obtained in this experiment.

Table 19. Status of execution - scenario versus memory controllers [Author].

Coding approach Read/Write operation	Scenario							
	i	ii	iii	iv	v		vi	
					1º Prog.	2º Prog.	1º Prog.	2º Prog.
WE	OK	ERR	ERR	ERR	ERR	ERR	ERR	ERR
SH	OK	OK*	ERR	ERR	ERR	ERR	ERR	ERR
SL	OK	OK*	OK*	OK*	OK*	OK*	OK*	OK*
DHL	OK	OK*	OK*	ERR	OK*	OK*	ERR	OK*

Legend: OK - programs finish without error;
OK* - error occurrence but without compromising the execution;
ERR - error occurrence affecting the execution.

As expected, the four approaches execute the program successfully without error in **Scenario i**. In **Scenario ii**, all three ECCs executed the program successfully. Only the SL and DHL approaches concluded the application execution in **Scenario iii**, which starts with a bitflip, followed by injecting several bitflips. For **Scenario iv**, which is the worst case, only the SL memory controller could run the program successfully for a system with multiple bitflips simultaneously. In **Scenario v,** SL and DHL could run all the programs; SH fixed the first bitflip, but when they started receiving multiples bitflip could not fix the errors anymore. In **Scenario vi** – several bitflips, DHL started with Hamming, could not correct the data but changed the ECC to LPC, making the next instruction and application better protected; the SL was unique to fix both applications. We emphasize that more than one error within the same word was generated in multiple error scenarios, preventing the Hamming correction.

According to the article [3], a study on Google's servers for two years and a half exposed that more than 8% of memories are affected by errors yearly, and 0.22% have multiple bitflips. Memory errors are highly correlated; their analyses found that the probability

of a corrected or nearby address having another error is in the range of 13× to 228× compared to an address far from a previous error address. Also, 70% to 80% of multiple bitflip cases had one bitflip in the same address. Based on this information, one of the benefits of this work is to mitigate approximately 80% of cases of multiple bitflips that previously had one bitflip. DHL ensures an increase of 0.170% more in the total probability of the application to continue executing and only 0.043% less than SL, resulting in an efficacy probability of 99.956%, as shown in Table 20.

Table 20. Application-fail probability according to the fault-tolerant approach in Google's servers [Author].

Execution status	WE	SH	DHL	SL
Ok	91.780%	99.780%	99.956%	~99.999%
Fail	8.220%	0.220%	0.044%	~0.001%

Note that the approach adopted in the experiments only allows identifying the occurrence of errors, and consequent correction, when the memory location with error is accessed. This approach allows parts of memory to be receiving cumulative errors generated by multiple events over time. Thus, only an ECC with high robustness can mitigate the problem, enabling very spaced memory accesses. If the access rate to memory locations is accelerated, even less robust ECCs can achieve acceptable efficiency. However, there is also a tradeoff here, since scrubbing procedures [10][20] penalize power dissipation, energy consumption and execution time.

Table 21 demonstrates the application's total read and write operations executed in scenarios i to vi obtained with the Absimth.

Table 21. Number of data reading and written, according to the scenario and coding approach [Author].

Read/Write operations for each Memory controller approach			Scenario					
			i	ii	iii	iv	v	vi
WE	Read		7200083	7200083	7200083	7200083	14400166	14400166
	Write		800177	800177	800177	800177	1600354	1600354
SH	Read		7200083	7200083	7200083	7200083	14400166	14400166
	Write		800177	800177	800177	800177	1600354	1600354
SL	Read		7200083	7200083	7200083	7200083	14400166	14400166
	Write		800177	800177	800177	800177	1600354	1600354
DHL	Hamming	Read	7200083	6094167	6094167	6094167	11081960	11081960
		Write	800177	494074	494074	494074	681960	681960
	LPC	Read	0	1105916	1105916	1105916	3318206	3318206
		Write	0	306103	306103	306103	918394	918394
	Recode	Hamming to LPC	0	32000	32000	32000	32000	32000

Table 22 presents the number of threshold executions and the total energy consumed in each scenario. The energy consumed in one Threshold process is 5.548 fJ, giving the energy consumption of 86.69 pJ per RAM, where $threshold\ evaluation =$

$number\ of\ blocks \times\ threshold\ energy = 15625 \times 5.548\ fJ = 86687.5\ fJ.$

Table 22. Energy consumption by the Threshold process in DFMC operating with Hamming or LPC, based on the number of cycles of Scenario ii [Author].

Scenario	Number of executions						Energy (pJ)					
ECC	i	ii	iii	iv	v	vi	i	ii	iii	iv	v	vi
DFMC-Hamming	800	659	659	659	659	659	69350	57127.06	57127.06	57127.06	57127.06	57111.81
DFMC-LPC	-	141	141	141	941	941	-	12222.94	12222.94	12222.94	81572.94	81572.94

Table 23 to Table 25 compare the memory controllers' energy consumption, critical path, and power dissipation. The data was calculated separately for the Hamming, LPC, and Recode processes to consider DFMC's execution with different ECCs and processes. This approach enables a more comprehensive analysis of the memory controller's performance and energy consumption across six error scenarios. The Effective tags described in the last lines of Table 23 to Table 25 represent the cumulative energy spent, the total time spent on the critical path, and the average power dissipation for DFMC-DHL.

Table 23. Total energy consumption according to scenarios i to vi and WE, SH, SL, and DHL memory controllers, with and without the RAM energy consumption [Author].

Coding approach		Energy Consumption (µJ)											
		Scenario - Only Memory controller						Scenario - Memory controller and RAM					
		i	ii	iii	iv	v	vi	i	ii	iii	iv	v	vi
WE		0.19	0.19	0.19	0.19	0.38	0.38	3591.49	3591.49	3591.49	3591.49	7182.97	7182.97
SH		0.60	0.60	0.60	0.60	1.20	1.20	3591.90	3591.90	3591.90	3591.90	7183.80	7183.80
SL		1.17	1.17	1.17	1.17	2.33	2.33	7183.76	7183.76	7183.76	7183.76	14367.52	14367.52
DHL	*Hamming*	*1.40*	*1.16*	*1.16*	*1.16*	*2.04*	*2.04*	*3592.70*	*2956.82*	*2956.82*	*2956.82*	*5277.46*	*5277.46*
	LPC	-	*0.34*	*0.34*	*0.34*	*1.05*	*1.05*	-	*1271.62*	*1271.62*	*1271.62*	*3815.39*	*3815.39*
	Recode	-	*0.06*	*0.06*	*0.06*	*0.06*	*0.06*	-	*43.74*	*43.74*	*43.74*	*43.74*	*43.74*
	Effective	1.40	1.56	1.56	1.56	3.15	3.15	3592.70	4272.18	4272.18	4272.18	9136.60	9136.60

Table 24. Execution time according to scenarios i to vi and WE, SH, SL, and DHL memory controllers, with and without the RAM latency [Author].

Coding approach		Critical path (µS)											
		Scenario - Only Memory controller						Scenario - Memory controller and RAM					
		i	ii	iii	iv	v	vi	i	ii	iii	iv	v	vi
WE		13256.43	13256.43	13256.43	13256.43	26512.86	26512.86	24509.42	24509.42	24509.42	24509.42	49018.84	49018.84
SH		13616.43	13616.43	13616.43	13616.43	27232.87	27232.87	24509.42	24509.42	24509.42	24509.42	49018.84	49018.84
SL		17792.48	17792.48	17792.48	17792.48	35584.97	35584.97	24509.42	24509.42	24509.42	24509.42	49018.84	49018.84
DHL	*Hamming*	*13616.43*	*11221.42*	*11221.42*	*11221.42*	*20046.91*	*20046.91*	*24509.42*	*19501.83*	*19501.83*	*19501.83*	*33994.29*	*33994.29*
	LPC	-	*3036.44*	*3036.44*	*3036.44*	*9110.52*	*9110.52*	-	*5007.59*	*5007.59*	*5007.59*	*15024.55*	*15024.55*
	Recode	-	*344.45*	*344.45*	*344.45*	*344.45*	*344.45*	-	*344.45*	*344.45*	*344.45*	*344.45*	*344.45*
	Effective	13616.43	14602.31	14602.31	14602.31	29501.88	29501.88	24509.42	24853.87	24853.87	24853.87	49363.29	49363.29

Table 23 displays the total energy consumptions in scenarios i to vi for the memory controller with and without RAM. These energy consumptions are attained with the values of Table 15, Table 16 and Table 18 multiplied by the read and write numbers described in Table 21, considering the Recode and Threshold process. The same method was applied to calculate the execution time using only the critical path in Table 24. The power dissipation in Table 25 was calculated from Table 23 and Table 24 results. The threshold limits are evaluated each 10 thousand cycles, having been calculated 800 times; note that the

Threshold process can be optimized, stopping it when the memory controller programming does not allow rolling back to a less-power correction ECC; i.e., the programming used in DHL. Table 25 shows that the dynamic approach makes DHL more energy intensive than the other static controllers. However, this consumption is practically neglected when added the RAM consumption. Also, when considering the memory controller and RAM, even the SH approach has an energy consumption close to DHL, which is much more energy efficient than the SL memory controller.

Table 25. Total power dissipation according to scenarios i to vi and WE, SH, SL, and DHL memory controllers, with and without the RAM power dissipation [Author].

Coding approach		Power (µW)											
		Scenario - Only memory controller						Scenario - Memory controller and RAM					
		i	ii	iii	iv	v	vi	i	ii	iii	iv	v	vi
WE		14.39	14.39	14.39	14.39	14.39	14.39	146534.98	146534.98	146534.98	146534.98	146534.98	146534.98
SH		44.17	44.17	44.17	44.17	44.17	44.17	146551.74	146551.74	146551.74	146551.74	146551.74	146551.74
SL		65.59	65.59	65.59	65.59	65.59	65.59	293102.01	293102.01	293102.01	293102.01	293102.01	293102.01
DHL	*Hamming*	*103.08*	*103.57*	*103.57*	*103.57*	*101.67*	*101.67*	*146584.46*	*151617.42*	*151617.42*	*151617.42*	*155245.59*	*155245.59*
	LPC	-	*110.44*	*110.44*	*110.44*	*115.36*	*115.36*	-	*253938.25*	*253938.25*	*253938.25*	*253943.48*	*253943.48*
	Recode	-	*174.82*	*174.82*	*174.82*	*174.82*	*174.82*	-	*126999.61*	*126999.61*	*126999.61*	*126999.61*	*126999.61*
	Effective	103.08	106.68	106.68	106.68	106.75	106.75	146584.46	171891.96	171891.96	171891.96	185088.86	185088.86

The energy consumption of memory controllers with RAM described in Table 25 provides the following analyses according to error scenarios:

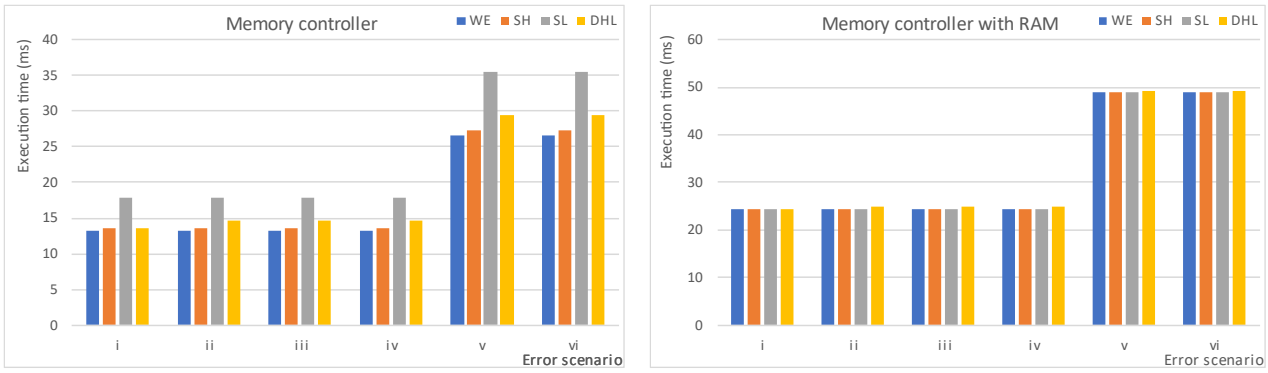
i. **Scenario i** – DMFC-DHL consumes 133% and 57% more energy than the SH and SL controllers, respectively. The energy consumption of the SH and DHL controllers are practically the same (DHL consumes only 0.02% more) when evaluating the combined effect of RAM and 49.99% less than the SL memory controller;

ii. **Scenarios ii to iv** - the DHL energy consumption increases by 17.29% compared to the SH energy consumption due to the ECC Recoding cost and the use of LPC in part of the application execution, but it is still 41.35% lower than the SL controller consumption;

iii. **Scenario v** – DMFC-DHL completes the application execution by saving 36.85% of energy compared to SL memory controller, while the SH controller fails to complete the application execution. **Scenario iv** was not compared because DHL and SH failed to execute the application.

It is important to emphasize that server systems are composed of several memory modules, and the dynamic selective approach allows switching to a higher power encoding only those modules with a higher error rate. Besides, the proposed approach uses a threshold that dynamically enables varying to a code with less energy consumption in a situation with a lesser error rate.

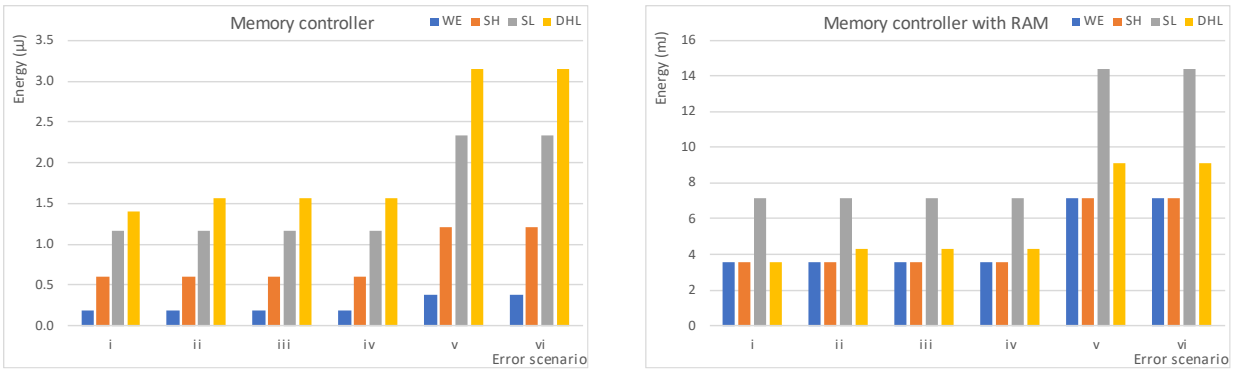
Figure 88, Figure 89 and Figure 91 compare execution time, total energy consumption and power dissipation for all tested scenarios with and without the RAM effect.



(a) (b)

Figure 88. Execution time of the four memory controller architectures (a) without and (b) with the RAM latency [Author].

Figure 88 and Table 24 show that the execution time for all memory controllers is similar when considering the RAM effect. In scenarios where a bitflip occurs (except Scenario i), DFMC-DHL exhibits a slight increase in execution time due to the Recode process. However, this increase is almost imperceptible due to the small number of addresses within the block. It is worth noting that in Scenario i, where there is no bitflip, the additional DFTM module performs practically the same as the SH controller.



(a) (b)

Figure 89. Energy consumption of the four memory controller architectures (a) without and (b) with the RAM energy consumption [Author].

Figure 89 and Table 23 reveal that when considering a single block and focusing solely on the memory controller, DFMC consumes more energy than SL, representing the worst-case scenario for DFMC. However, as DFMC handles multiple blocks with different ECCs, its energy consumption tends to decrease proportionally and consistently surpasses both the SH and SL controllers. Furthermore, Figure 90 shows that when considering the RAM's energy consumption, the findings align with the theoretical expectations outlined for memory controllers' energy consumption in Section 4.8 and Figure 60.

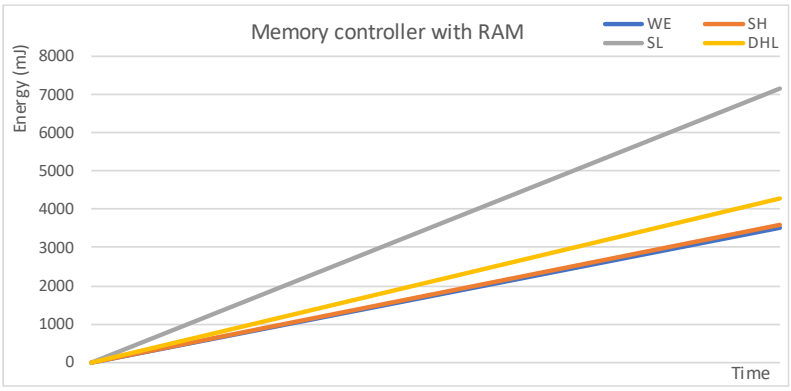
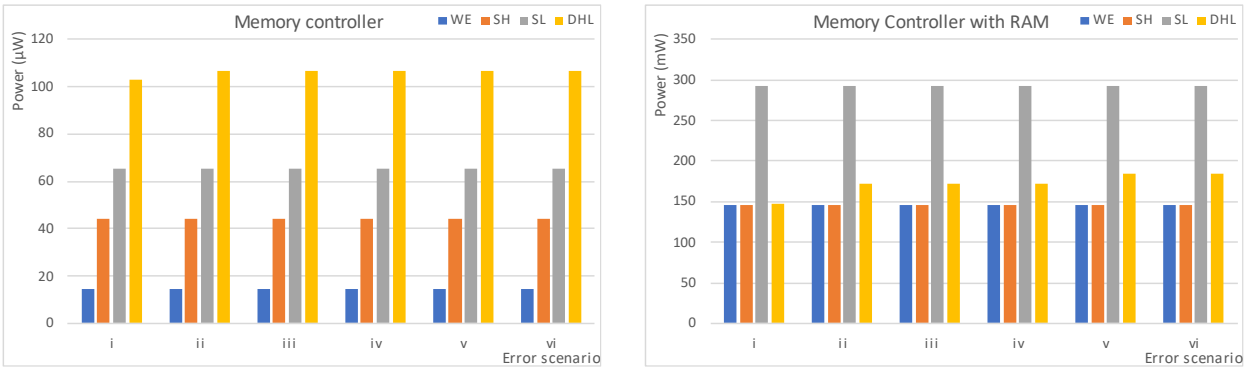


Figure 90. Comparative of energy consumption of the four memory controller architectures with the RAM energy consumption [Author].



(a) (b)

Figure 91. Power dissipation of the four memory controller architectures (a) without and (b) with the RAM power dissipation [Author].

Figure 91 and Table 25 illustrate that the DFMC power dissipation remains relatively unchanged when considering the RAM effect in Scenario i (without bitflip). However, in scenarios ii, iii, and iv, the power dissipation increases due to the Recode process. In scenarios v and vi, the power dissipation is further amplified by both the Recode process and the second application execution.

Figure 92 illustrates the average energy consumption over time with DDR4 in **Scenario ii** based on the worst operating frequency; the recode stop all the other process and is calculated separately. Note that the threshold compared to the RAM energy spent is so small that it can be despised.

It is worth noting that article [3] highlights that 91.78% of servers do not experience bitflips, while more than 8% of memories are affected by errors annually, as depicted in **Scenario ii**. In our evaluation, **Scenario i** does not provide a representative assessment as it lacks a bitflip occurrence. Considering that bitflips are crucial factors in evaluating memory controllers, it is important to ensure that the analyzed scenarios incorporate instances where bitflips occur. **Scenario ii** represents more than 8% of cases with one bitflip, providing a more accurate evaluation of the memory controllers' performance and their ability to handle

errors. Conducting an extensive analysis of all other cases was deemed unnecessary due to their low representation (less than 1%), high complexity in obtaining the values, and their tendency to be similar to **Scenario ii**. However, it is worth mentioning that this topic was extensively discussed in Section 4.8.

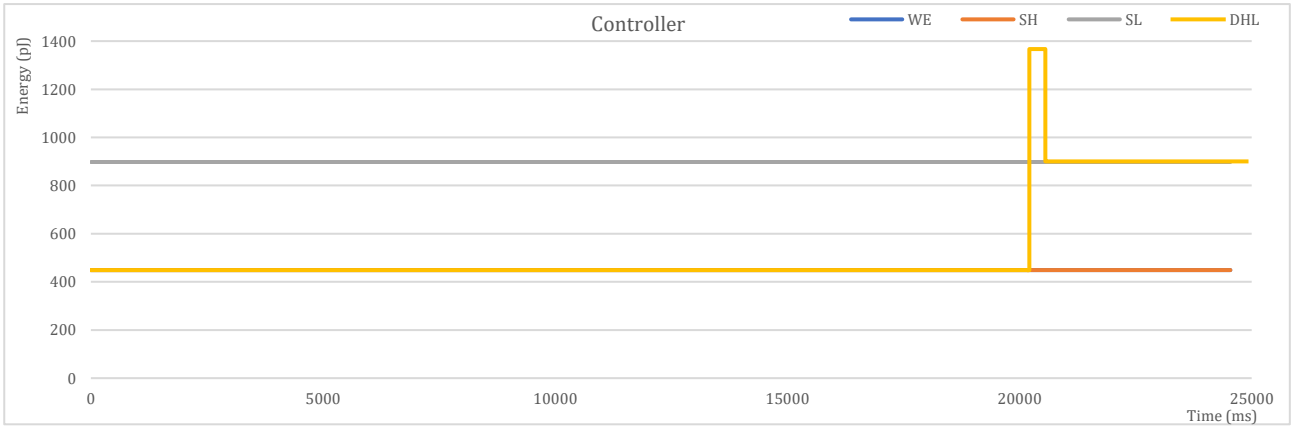


Figure 92. Energy consumption of a single memory block over time for all memory controllers in Scenario ii [Author].

The energy spent demonstrated was for only a memory block; the DHL energy consumption efficiency becomes even more evident when considering all 15625 blocks. Regarding the usage of 30% of the memory – i.e., 4687 memory blocks, being one affected by bitflips as demonstrated in scenario ii and disregarding the leakage memory, we have the power dissipated for WE, SH and DMFC controllers almost the same, as illustrated in Table 26. In contrast, the SL memory controller is double. This significant energy consumption difference between LPC and the other ECCs is even more expressive when considering the energy spent in a year, which is the usual scenario of large server farms [3][15].

Table 26. Power dissipation considering 30% of memory usage with Scenario ii [Author].

Execution status	WE	SH	SL	DHL
Power dissipation (W)	686.81	686.89	1373.77	687.08
Norm*	49.99%	50.00%	100.00%	50.01%

Legend: Norm* - Percentage values are normalized regarding the power dissipation of the SL memory controller.

This section showed that the dynamic fault-tolerant approach, represented by DFMC-DHL, is crucial for enhancing server reliability, optimizing performance, and saving energy. DFMC-DHL demonstrated reaching the best tradeoff between error correction efficacy and energy consumption efficiency by dynamically adjusting the ECC mechanism from Hamming to LPC based on the error rate. As a result, the DFMC-DHL controller achieved reliable data integrity while maintaining optimal energy consumption.

6.3.3 DFMC Reliability Assessments using and Embedded Benchmark

Figure 93 describes a comprehensive benchmark conducted to evaluate the performance of SL, SH, and DHL memory controllers in multiple applications under different bitflip probabilities. The simulations inserted bitflips in the instruction area and nearby regions for various programs, exploring different levels of aggressiveness based on the number and proximity of bitflips. This evaluation allowed for assessing the error correction power of SH, SL, and DHL in a benchmark of programs executing on DDR4 memory with 64-bit data words plus ECC.

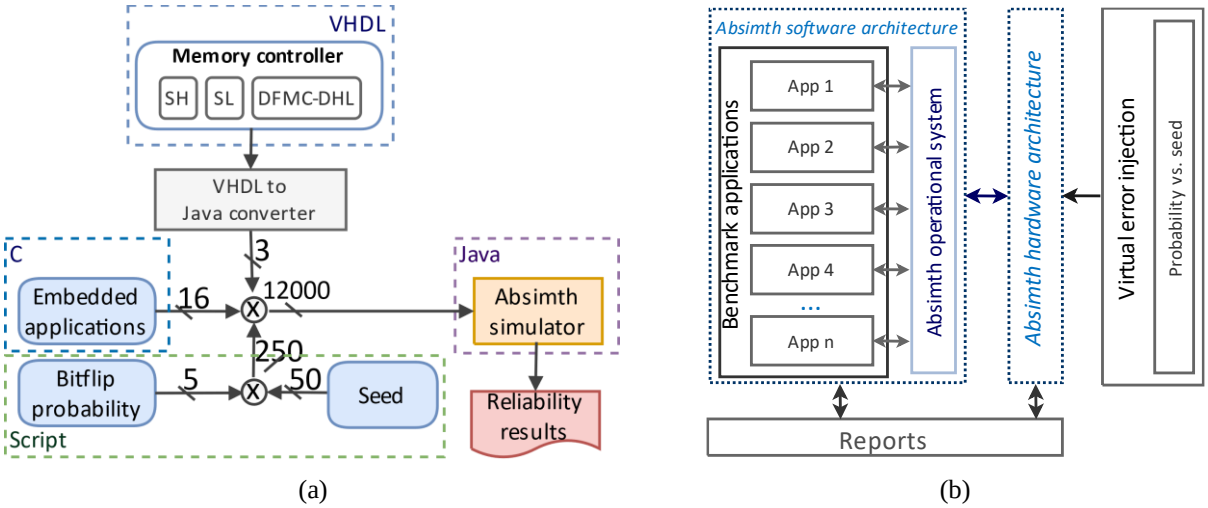


Figure 93. Details about the experiment conducted in Group D (Figure 75), including the (a) experimental setup and tools used and (b) the software/hardware interactions performed in the Absimth simulator [Author].

The benchmark applications were selected from PARSEC [87][88], BOTS [121], and HARDINFO [122] to capture a wide range of application behaviors. It comprises 13 applications, including (**i**) Binary Sort, (**ii**) Black Scholes, (**iii**) Bubble Sort, (**iv**) CRC8, (**v**) Factorial, (**vi**) Fibonacci, (**vii**) Frequent Pattern Growth, (**viii**) Greatest Common Divisor, (**ix**) Hanoi, (**x**) Insertion Sort, (**xi**) JKiss32, (**xii**) Matrix Multiplication, (**xiii**) Prime Number. The experiment also explored the Bubble Sort and CRC8 programs executing (**xiv**) on the same processor and (**xv**) on different processors, and (**xvi**) three instances of Bubble Sort and one instance of CRC8 executing on three processors. Roman numerals are used to represent each application in the Table 27 (a) to (e).

The benchmark evaluated various probabilities of bitflips occurring, initially identifying the optimal probability and doubling it four times to examine the behavior of DHL under extreme scenarios. The probabilities of bitflips occurring during each clock period were found to be 0.1%, 0.05%, 0.01%, 0.001%, and 0.005%. Table 27 (a) to (e) presents the total number of successful and unsuccessful executions for each application with ten different seeds at different bitflip probabilities, ranging from 0.001% to 0.1%.

Table 27. Number of applications executed (OK) or not executed (NOK) with bitflip rates of 0.001%, 0.005%, 0.01%, 0.05%, and 0.1%. ALL is a table that consolidates all bitflip rates [Author].

0.001%	NOK			OK		
#	SH	DHL	SL	SH	DHL	SL
i				50	50	50
ii				50	50	50
iii				50	50	50
iv	1	1		49	49	50
v				50	50	50
vi				50	50	50
vii	4	2		46	48	50
viii				50	50	50
ix				50	50	50
x				50	50	50
xi				50	50	50
xii				50	50	50
xiii				50	50	50
xiv				50	50	50
xv				50	50	50
xvi				50	50	50
Total	**5**	**3**	**0**	**795**	**797**	**800**

0.005%	NOK			OK		
#	SH	DHL	SL	SH	DHL	SL
i				50	50	50
ii				50	50	50
iii				50	50	50
iv	7	2		43	48	50
v				50	50	50
vi				50	50	50
vii	16	13		34	37	50
viii				50	50	50
ix				50	50	50
x				50	50	50
xi				50	50	50
xii				50	50	50
xv				50	50	50
xvi				50	50	50
xv				50	50	50
xvi	1			49	50	50
Total	**24**	**15**	**0**	**776**	**785**	**800**

0.01%	NOK			OK		
#	SH	DHL	SL	SH	DHL	SL
i				50	50	50
ii				50	50	50
iii				50	50	50
iv	14	5		36	45	50
v				50	50	50
vi				50	50	50
vii	27	20		23	30	50
viii				50	50	50
ix				50	50	50
x				50	50	50
xi				50	50	50
xii				50	50	50
xv				50	50	50
xvi	1			49	50	50
xv	1			49	50	50
xvi	1	1		49	49	50
Total	**44**	**26**	**0**	**756**	**774**	**800**

0.05%	NOK			OK		
#	SH	DHL	SL	SH	DHL	SL
i				50	50	50
ii	4	2		46	48	50
iii	3	2		47	48	50
iv	24	4		26	46	50
v				50	50	50
vi				50	50	50
vii	44	36		6	14	50
viii				50	50	50
ix	3			47	50	50
x	1			49	50	50
xi				50	50	50
xii	2	2		48	48	50
xv				50	50	50
xvi	2	1		48	49	50
xv	2	1		48	49	50
xvi	7	3		43	47	50
Total	**92**	**51**	**0**	**708**	**749**	**800**

0.1%	NOK			OK		
#	SH	DHL	SL	SH	DHL	SL
i				50	50	50
ii	11	4		39	46	50
iii	3	1		47	49	50
iv	25	4		25	46	50
v				50	50	50
vi				50	50	50
vii	48	42		2	8	50
viii				50	50	50
ix	4	1		46	49	50
x	1	1		49	49	50
xi				50	50	50
xii	3	3		47	47	50
xv				50	50	50
xvi	5	4		45	46	50
xv	5	4		45	46	50
xvi	8	4		42	46	50
Total	**113**	**68**	**0**	**687**	**732**	**800**

ALL	NOK			OK		
#	SH	DHL	SL	SH	DHL	SL
i				250	250	250
ii	15	6		235	244	250
iii	6	3		244	247	250
iv	71	16		179	234	250
v				250	250	250
vi				250	250	250
vii	139	113		111	137	250
viii				250	250	250
ix	7	1		243	249	250
x	2	1		248	249	250
xi				250	250	250
xii	5	5		245	245	250
xv				250	250	250
xvi	8	5		242	245	250
xv	8	5		242	245	250
xvi	17	8		233	242	250
Total	**278**	**163**	**0**	**3722**	**3837**	**4000**

The configuration used in the experiment was based on [3][15], considering the spatial locality of faults in RAM. The number of bitflips that could occur during each clock period was set to one, and the distances between consecutive bitflips and within the same memory address were defined within specific ranges. The probability of bitflips occurring outside the defined intervals was also set to 0.2%. To increase the likelihood of a double bitflip, the ECC did not fix the affected area, allowing it to remain “dirty” after a successful correction.

The experiment evaluated the worst-case scenarios for DHL, where the same application constantly experienced bitflips. The best-case scenario for DHL represents the most common scenario in servers, where bitflips occur with reasonable intervals between them. The results demonstrated that the DHL approach was more effective than the SH

approach, except in cases where two errors coincided at the same memory address in a single clock cycle, rendering them unfixable by a SECDED ECC. However, it is essential to note that the obtained results were influenced by the choice of ECCs employed. The proposed dynamic approach aims to protect memory modules by utilizing codes that offer the best tradeoff between error correction efficacy and operational cost.

Figure 94 (a) to (e) graphically represents the total number of successful and unsuccessful executions. An application that completed its execution was considered successful, while an unsuccessful application either did not finish execution or encountered unfixable data, such as two bitflips in the Hamming code. Table 27 (f) and Figure 94 consolidate the previous results.

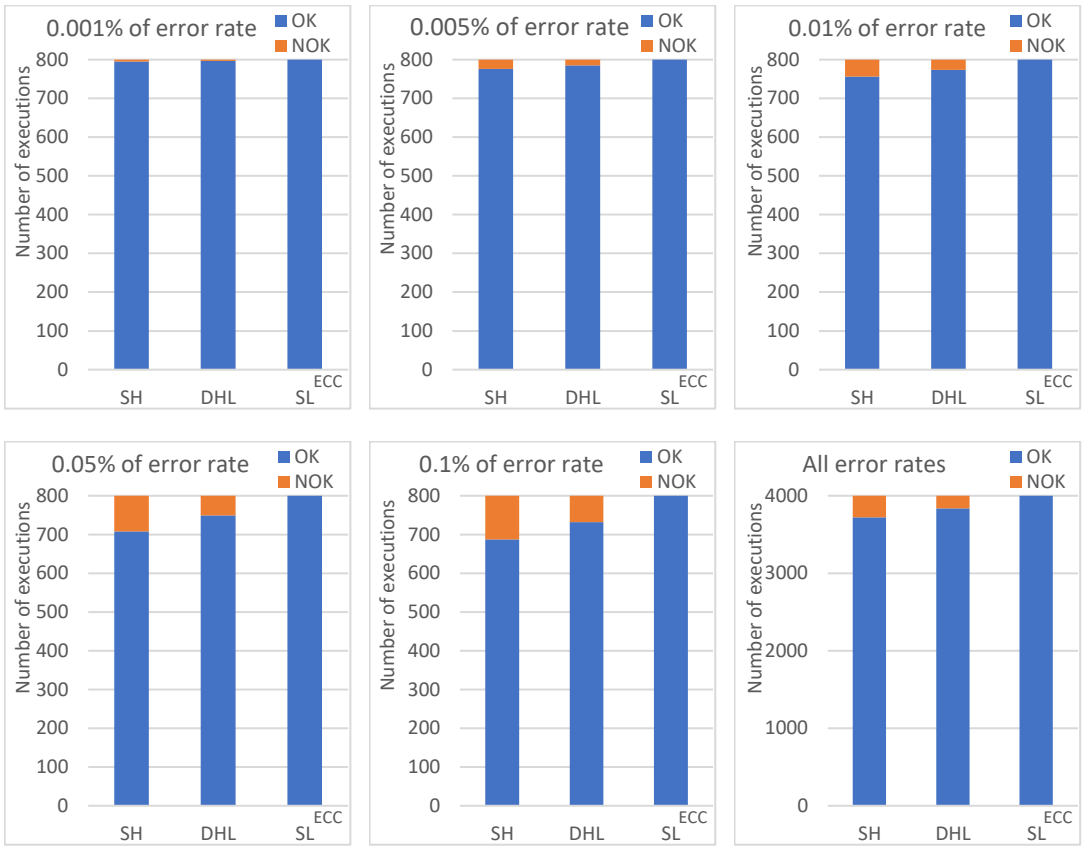


Figure 94. Total of executions with success (OK) or not (NOK) with 0.001%, 0.005%, 0.01%, 0.05% and 0.1% of bitflip probability. ALL is a table that consolidates all bitflip probabilities [Author].

Figure 95 provides a clear comparison of the reliability of the evaluated ECCs, illustrating the number of failed cases concerning the bitflip probability. It demonstrates the superiority of DHL over SH and showcases the increased system reliability across all tested bitflip rates. In the worst-case scenario, DHL improved reliability from about 60.17% to 80.4%. The green area represents the probability of improving the reliability range between the worst and best cases. The system effectively selected the optimal tradeoff between energy efficiency and reliability for other cases.

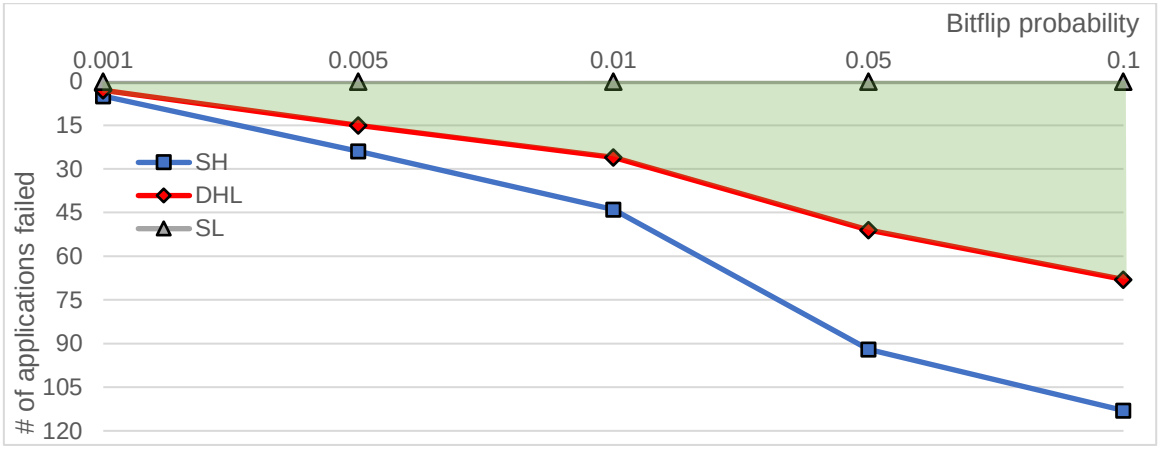


Figure 95. The number of failed applications versus bitflip probability by ECC.

This section further confirms the superior efficiency and efficacy of DFMC-DHL compared to SH and SL. The efficacy of the DFMC-DHL approach was demonstrated in most scenarios, except for instances where a simultaneous bitflip occurred at the same memory address during a single read using Hamming. These simultaneous bitflips posed a challenge as they were unfixable by a SECDED ECC.

6.4 Conclusion

This chapter presents the benefits of this Thesis through three sets of experiments. The first set used MyHDL to implement and validate all processes involved in the DFMC functionality and ECC modules. MyHDL facilitated comprehensive unit and integration testing. The hardware architecture, consisting of the DFMC, RAM, and related modules, underwent thorough assessment using synthetic stimuli and diverse scenarios to evaluate its behavior and adaptability. The experiments showcased the efficacy of the DFMC in dynamically adjusting the ECC based on the number of errors, ensuring reliable data delivery even under highly stressed environments.

In the second set, the memory controller developed was converted to VHDL, and the Genus synthesis tool provided valuable insights into its performance and efficiency. The synthesis results showcased the area consumption of the WE, SL, SH, and DFMC memory controllers, with DFMC exhibiting the highest area usage due to its dynamic ECC capabilities. However, it also offered the advantage of customizing the ECC operation based on specific error scenarios, optimizing reliability.

Additionally, power dissipation, energy consumption, and critical path were evaluated for the memory controllers, specifically during read and write operations. It was observed that including DTMF in the DFMC did not impact the critical path when considering only the memory controllers. However, the energy consumption of the read/write operation increased by 221% when comparing SH with DFMC-Hamming and by 116.45% when

comparing SL with DFMC-LPC.

This study highlights the significance of a dynamic fault-tolerant approach in memory controllers, particularly the DFMC-DHL controller, in ensuring server reliability, energy efficiency, and robust error detection and correction mechanisms. These findings contribute to advancing memory controller designs and their practical implementation in server systems, mitigating the impact of memory errors and improving overall system performance.

7 CONCLUSIONS AND FUTURE WORK

ECCs to tolerate memory failures have been widely researched due to the increased error rate in the latest technologies [22][36][37]. Large servers demand even more of these requirements as programs have become increasingly dependent on large amounts of data [77]. This work brings innovative technology to achieve high reliability and low energy consumption impact. We proposed the Dynamic Fault-Tolerant Memory Controller (DFMC), a fault-tolerant technology applied to the memory controllers that brings out new possibilities, providing high reliability due to the strong correlation of memory errors with low energy consumption and performance for memory areas not affected by errors.

This work presented Absimth [125][126], an extensibility simulation tool for describing a multiprocessor target architecture that accesses memory modules through a memory controller. The simulator enables some tools for performing injection error patterns and evaluating fault tolerance techniques with support for many ECC standards. The fast memory controller prototyping on Absimth enables lots of research and ideas evaluation before the hardware prototype and how the application will behave.

To assess this Thesis, DFMC was prototyped and evaluated with MyHDL [114]; posteriorly, the VHDL code was synthesized using the Genus synthesis tool [123] to estimate power dissipation, area, and energy consumption for some timing aspects.

The first assessment - Section 6.1 - considers the behavior flexibility of DFMC. DFMC demonstrated an excellent adaptation for all proposed scenarios: (i) keeping a fast ECC without the capacity to fix bitflip under the best environment – i.e., without or a few bitflips; (ii) selecting the best ECC encoding for the current situation; and (iii-iv) under a highly stressed environment changing to a high and secure ECC rapidly and keeping.

The second assessment - Section 6.2 - shows that the area increased by 53.8%, 254.3%, and 467.3% when comparing the SH, SL, and DFMC-DHL to a memory controller without ECC; the critical path does not have a rise with the extra module included by DFTM; in contrast, the energy in the instance of read/write increases by 221% when comparing the SH with DHL using Hamming and 116.45% when comparing the SL with DHL using LPC.

In Section 6.3.2, this Thesis evaluates a wide range of scenarios with the Absimth tool based mainly on research from Google [3] to understand their behavior. With the understanding of these scenarios, it was possible to emulate and evaluate DFMC-DHL, demonstrating a considerable improvement in error correction probability.

Considering an environment without bitflips, DHL consumes 133% and 57% more when compared to the SH and SL, respectively. Adding the RAM, the energy consumption

of the SH and DHL approaches are practically the same, only 0.02% more, and significantly better than the SL, consuming 49.99% less. In an environment with one or one and after multiple bitflips, the DHL energy consumption increases by only 17.29% compared to the SH due to the ECC Recoding cost and the use of LPC in part of the application execution. However, it is still 41.35% lower than the SL approach. Lastly, in an environment with one or one and after multiples bitflips, and with two application execution in the same area - one and after the other, shows that the DHL approach completes in all cases the second application execution and saves 36.85% of energy compared to the SL approach. In contrast, the SH approach fails the second application execution in all cases.

When we look based only on the article [3], the proposed approach keeps approximately 92% of the total memory with a simple encoding – consuming little energy; for 0.22% of the memories that need a robust ECC, the proposed approach improves efficiency by at least 70%. It is a crucial approach for servers with thousands of memory modules, such as servers from Google [3] and Facebook [6]. Additionally, the experimental results show that the proposed methodology achieves a probability of 99.956% success in executing applications with multiple bitflips.

Finally, Section 6.3.3 evaluates a benchmark test of the SH, SL, and DHL memory controllers, where it was explored a series of applications with different probabilities of bitflips under one of the worse scenarios for DHL - receiving bitflips constantly in the instruction area, which may receive more than one bitflip. The evaluation demonstrates that DHL increases reliability in all bitflip rate scenarios tested. The reliability increases according to the bitflip probability rise, improving between 60% to 80.4% depending on the probability.

As demonstrated, DFMC performs better than a unique ECC method in most situations with a low energy cost and higher flexibility. We can avoid wasting time and energy on a high error correction rate and reduce the encoding time for most of the memory through the proposed dynamism. The proposed approach is even more evident for large servers that require high reliability and consume enormous energy.

7.1 Discussion and Future Work

The methodology for the dynamic memory controller brings to light possibilities of works that were evaluated throughout the doctorate and could be part of the Thesis; the idealized possibilities are described below:

- **Non-Volatile Memory**: When the hardware’s energy is lost, it is essential to keep the previous configuration when the hardware restarts, recommending to use non-volatile memory.

- **Scrubbing**: As discussed in Section 2.3.2, the scrubbing technique can increase the hardware's reliability even more. The threshold evaluation could be executed at the same of the scrubbing.
- **ECC for Common Memories**: Personal computers are unprotected because they do not use ECC memories. A change on the controller can enable hardware error correction, even if the memory does not have the additional chip for ECC. The final addresses can be used as ECC areas or after each page has a block for ECC, reducing the total memory disposed to data. This technique increases reliability but burdens data access with one more cycle for writing the ECC. If the computer operates a dual-channel memory, this operation can be performed in one cycle by writing the ECC in the opposite memory, which writes the data or can take advantage of the burst read in a single channel.
- **Optimized Technique for Recoding the Page:** The critical aspect of this work lies in modifying the page coding. It is essential to employ an optimized technique for data recoding without blocking the page by implementing a two-level approach; the first level manages the page, while the second manages its addresses. When a specific area needs to undergo migration, the first level marks it as "in migration", while the second references all the addresses that require migration. The recoding process is then executed in parallel, addressing each address individually. Consequently, if a read or write operation is requested within an area marked as "in migration", the second level is consulted to determine the current ECC. Otherwise, only the first level is accessed. Once all the addresses have been successfully recoded in the second level, the first level is updated with the new ECC information, ensuring a seamless transition and uninterrupted access to the page during the recoding process.
- **Disabling Addresses:** After a certain number of errors, even a more robust coding may be unable to correct the information. Error addresses can be turned off to avoid it or to avoid using robustness encoding on the page, optimizing power consumption and latency.
- **OS Integration:** The OS integration enables awareness of the characteristics of running programs and the acceptable errors within that specific context. For instance, it allows for protecting OS or critical applications by employing a robust encoding scheme while employing a less stringent encoding scheme

for non-critical areas such as video storage. The system can optimize performance and reliability by tailoring the encoding technique based on the program's requirements, ensuring that resources are allocated efficiently, and error protection is focused where it matters the most.

- **Memory Switch**: In our demonstration, the data always utilizes the first RAM, with the second RAM serving as a backup for ECC purposes. However, in scenarios where the first RAM experiences more bitflips than the second RAM, it is possible to switch the roles. The second RAM can become the primary RAM, while the first RAM is used only when necessary, effectively avoiding areas prone to bitflips. This approach ensures optimal utilization of memory resources and enhances the system's reliability by minimizing the impact of bitflip.

8 REFERENCES

[1] A. Teman, G. Karakonstantis, R. Giterman, P. Meinerzhagen, A. Burg, “Energy versus Data Integrity Trade-Offs in Embedded High-Density Logic Compatible Dynamic Memories”, Proceedings of the Design Automation & Test in Europe Conference (DATE), pp. 489-494, 2015.

[2] N. Jouppi, “DRAM Errors in the Wild: Technical Perspective”, Communications of the ACM, v. 54, n. 2, pp. 99, Feb. 2011.

[3] B. Schroeder, E. Pinheiro, W. Weber, “DRAM Errors in the Wild: A Large-Scale Field Study”, Communications of the ACM, v. 54, n. 2, pp. 100-107, Feb. 2011.

[4] S. Mittal, “A Survey of Architectural Techniques for Managing Process Variation”, ACM Computing Surveys, v. 48, n. 4, art. 54, pp. 1-29, May 2016.

[5] A. Rahimi, L. Benini, R. Gupta, “Variability Mitigation in Nanometer CMOS Integrated Systems: A Survey of Techniques from Circuits to Software”, Proceedings of the IEEE, v. 104, n. 7, pp. 1410-1448, Jul. 2016.

[6] J. Meza, Q. Wu, S. Kumar, O. Mutlu, “Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field”, Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 415-426, 2015.

[7] S. Borkar, A. Chien, “The future of microprocessors”, Communications of the ACM, v. 54, n. 5, pp. 67-77, May 2011.

[8] S. Mittal, M. Inukonda, “A survey of techniques for improving error-resilience of DRAM”, Journal of Systems Architecture, v.91, pp. 11-40, Nov. 2018.

[9] T. Mittelholzer, M. Stanisavljevic, N. Papandreou, H. Pozidis, “High-Throughput ECC with Integrated Chipkill Protection for Nonvolatile Memory Arrays”, Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1-5, 2021.

[10] G. He, S. Zheng, N. Jing, “A Hierarchical Scrubbing Technique for SEU Mitigation on SRAM-Based FPGAs”, IEEE Transactions on Very Large-Scale Integration (VLSI) Systems, v. 28, n.10, pp. 2134-2145, Oct. 2020.

[11] D. Sorin, “Fault-tolerant computer architecture”, Morgan & Claypool Publishers, May 2009, 104p.

[12] S. Govindavajhala, A. Appel, “Using memory errors to attack a virtual machine”, Proceedings of the Symposium on Security and Privacy (S&P), pp. 154-165, 2003.

[13] B. Giridhar, M. Cieslak, D. Duggal, R. Dreslinski, H. M. Chen, R. Patti, B. Hold, C. Chakrabarti, T. Mudge, D. Blaauw, “Exploring DRAM organizations for energy-efficient and resilient exascale memories”, Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), pp. 1-12, 2013.

[14] A. Hwang, I. Stefanovici, B. Schroeder, “Cosmic rays don’t strike twice: understanding the nature of DRAM errors and the implications for system design”, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 111-122, 2012.

[15] E. Nightingale, J. Douceur, V. Orgovan, “Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs”, Proceedings of the Conference on Computer systems (EuroSys), pp. 343-356, 2011.

[16] PhysOrg, “Samsung First to Mass-produce 1Gb DDR2 Memory with 80nm Process Technology (2006, August 29)”, available at https://phys.org/news/2006-08-samsung-massproduce-1gb-ddr2-memory.html, 2023.

[17] Samsung, “Samsung Starts Mass Production of Most Advanced 14nm EUV DDR5 DRAM (2021, October 12)”, available at https://news.samsung.com/global/samsung-starts-mass-production-of-most-advanced-14nm-euv-ddr5-dram, 2023.

[18] A. Avizienis, J. Laprie, B. Randell, C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing”, IEEE Transactions on Dependable and Secure Computing, v. 1, n. 1, pp. 11-33, Jan.-Mar. 2004.

[19] S. Rehman, M. Shafique, J. Henkel, “Reliable software for unreliable hardware: A cross layer perspective”, Springer, Apr. 2016, 237p.

[20] A. Chabot, I. Alouani, R. Nouacer, S. Niar, “A Memory Reliability Enhancement Technique for Multi Bit Upsets”, Journal of Signal Processing Systems, v. 93, pp. 439-459, Apr. 2021.

[21] M.-C. Hsueh, T. Tsai, R. Iyer, “Fault injection techniques and tools”, Computer, v. 30, n. 4, pp. 75-82, Apr. 1997.

[22] R. Baumann, “Soft errors in advanced computer systems”, IEEE Design & Test of Computers, v. 22, n. 3, pp. 258-266, May-Jun. 2005.

[23] R. Velazco, P. Fouillat, R, Reis. “Radiation effects on embedded systems”, Springer, Apr. 2007, 259p.

[24] M. Nicolaidis, “Soft Errors in Modern Electronic Systems”, Springer Science, v. 41, 2001.

[25] R. Liu, D. Mahalanabis, H. Barnaby, S. Yu, “Investigation of Single-Bit and Multiple-Bit Upsets in Oxide RRAM-Based 1T1R and Crossbar Memory Arrays”, IEEE Transactions on Nuclear Science, v. 62, n. 5, pp. 2294-2301, Oct. 2015.

[26] A. Pérez-Celis, M. Wirthlin, “Statistical Method to Extract Radiation-Induced Multiple-Cell Upsets in SRAM-Based FPGAs”, IEEE Transactions on Nuclear Science, v. 67, n. 1, pp. 50-56, Jan. 2020.

[27] W. Wei, K. Namba, Y. Kim, F. Lombardi, “A Novel Scheme for Tolerating Single Event/Multiple Bit Upsets (SEU/MBU) in Non-Volatile Memories”, IEEE Transactions on Computers, v. 65, n. 3, pp. 781-790, 1 Mar. 2016.

[28] J. Chen, J. Yu, P. Yu, B. Liang, Y. Chi, “Characterization of the Effect of Pulse Quenching on Single-Event Transients in 65-nm Twin-Well, Triple-Well CMOS Technologies”, IEEE Transactions on Device, Materials Reliability, v. 18, n. 1, pp. 12-17, Mar. 2018.

[29] L. Artola, S. Ducret, F. Advent, G. Hubert, J. Mekki, “SEFI Modeling in Readout Integrated Circuit Induced by Heavy Ions at Cryogenic Temperatures”, IEEE Transactions on Nuclear Science, v. 66, n. 1, pp. 452-457, Jan. 2019.

[30] P. Wang, A. Sternberg, B. Sierawski, E. Zhang, K. Warren, A. Tonigan, R. Brewer, N. Dodds, G. Vizkelethy, S. Jordan, D. Fleetwood, R. Reed, R. Schrimpf, “Sensitive-Volume Model of Single-Event Latchup for a 180-nm SRAM Test Structure”, IEEE Transactions on Nuclear Science, v. 67, n. 9, pp. 2015-2020, Sep. 2020.

[31] V. Sridharan, D. Liberty, “A study of DRAM failures in the field”, Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), pp. 1-11, 2012.

[32] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, S. Gurumurthi, “Feng shui of supercomputer memory: Positional effects in DRAM and SRAM faults”, Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), pp. 1-11, 2013.

[33] V. Sridharan, N. DeBardeleben, S. Blanchard, K. Ferreira, J. Stearley, J. Shalf, S. Gurumurthi, “Memory errors in modern systems: The good, the bad, and the ugly”, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 297-310, 2015.

[34] G. Moore, “Cramming more components onto integrated circuits”, Reprinted from Electronics (1965), IEEE Solid-State Circuits Society Newsletter, v. 11, n. 3, pp. 33-35, Sep. 2006.

[35] P. Nair, D.-H. Kim, M. Qureshi, “ArchShield: architectural framework for assisting DRAM scaling by tolerating high error rates”, Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA), pp. 72-83, 2013.

[36] A. Dixit, A. Wood, “The impact of new technology on soft error rates”, Proceedings of the International Reliability Physics Symposium (IRPS), pp. 5B.4.1-5B.4.7, 2011.

[37] J. Gracia-Morán, L. Saiz-Adalid, D. Gil-Tomás, P. Gil-Vicente. “Improving Error Correction Codes for Memory-Cell Upsets in Space Applications”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems. v. 26, n. 10, pp. 2132-2142. Oct. 2018.

[38] M. Kooli,G. Di Natale, “A survey on simulation-based fault injection tools for complex systems”, Proceedings of the IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS), pp. 1-6, 2014.

[39] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, J. Hoe, “Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding”, Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 197-209, 2007.

[40] S. Mukherjee, J. Emer, T. Fossum, S. Reinhardt, “Cache scrubbing in microprocessors: myth or necessity?”, Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), pp. 37-42, 2004.

[41] M. Marinescu, “Simple and efficient algorithms for functional RAM testing”, Proceedings of the International Test Conference (ITC), pp. 1-4, 1982.

[42] P. Joseph, P. Antony, “VLSI design and Comparative Analysis of Memory BIST controllers”, Proceedings of the International Conference on Computational Systems and Communications (ICCSC), pp. 272-276, 2014.

[43] EInfochips, “Memory Testing: MBIST, BIRA & BISR | An Insight into Algorithms and Self Repair Mechanism”, available at https://www.einfochips.com/blog/memory-testing-an-insight-into-algorithms-and-self-repair-mechanism/, Oct. 2022.

[44] C. Stroud, “A designer’s guide to built-in self-test”, Springer, 2002, 320p.

[45] V. Gupta, G. Singh, A. Asati, “BIST Architecture for combinational circuit”, International Journal of Electrical, Electronics and Data Communication (IJEEDC), v. 7, n. 5, pp. 1-8, Jul. 2019.

[46] V. Sridhar, M. Prasad, “Built-in self-repair (BISR) technique widely used to repair embedded random-access memories (RAMs)”, International Journal of Computer Science Engineering, v. 1. n.1, pp. 42-60, Sep. 2014.

[47] S. Ozdemir, D. Sinha, G. Memik, J. Adams, H. Zhou, “Yield-Aware Cache Architectures”, Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 15-25, 2006.

[48] R. Zaragoza, “The Art of Error Correcting Coding”, Ed. Wiley, 2nd ed., pp. 170-201, 2006.

[49] D. Freitas, C. Marcon, J. Silveira, L. Naviner, J. Mota, “A survey on two-dimensional Error Correction Codes applied to fault-tolerant systems”, Microelectronics Reliability, v. 139, n. 114826, pp.1-16, Dec. 2022.

[50] T. Moon, “Error Correction Coding: Mathematical Methods and Algorithms”, Wiley, 2020, 992p.

[51] I. Alouani, S. Niar, F. Kurdahi, M. Abid, “Parity-based mono-Copy Cache for low power consumption and high reliability”, Proceedings of the IEEE International Symposium on Rapid System Prototyping (RSP), pp. 44-48, 2012.

[52] M. Qureshi, Z. Chishti, “Operating SECDED-based caches at ultra-low voltage with FLAIR”, Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 1-11, 2013.

[53] L.-J. Saiz-Adalid, P. Reviriego, P. Gil, S. Pontarelli, J. Maestro, “MCU Tolerance in SRAMs Through Low-Redundancy Triple Adjacent Error Correction”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, v. 23, n. 10, pp. 2332-2336, Oct. 2015.

[54] C. Shen, H. Li, G. Sahin, H. -A. Choi, Y. Shah, “Golay Code Based Bit Mismatch Mitigation for Wireless Channel Impulse Response Based Secrecy Generation”, IEEE Access, v. 7, pp. 2999-3007, Jan. 2019.

[55] I. Reed, G. Solomon, “Polynomial codes over certain finite fields”, Journal of the society for industrial and applied mathematics, v. 8, n. 2, pp. 300-304, Jun. 1960.

[56] C. Wilkerson, A. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, S. Lu, “Reducing Cache Power with Low-cost, Multi-bit Error-Correcting Codes” ACM SIGARCH Computer Architecture News, v. 38, n. 3, pp. 83-93, Jun. 2010.

[57] B. Day, “The Mathematics of Chipkill ECC”, available at https://www.keepandshare.com/doc6/18669/the-mathematics-of-chipkill-ecc-txt-9k, Oct. 2022.

[58] T. Rao, E. Fujiwara, “Error-Control Coding for Computer Systems”, Longman Higher Education, Jan. 1989, 524p.

[59] C. Benvenuto, “Galois Field in Cryptography”, May 2012, available at https://sites.math.washington.edu/~morrow/336_12/papers/juan.pdf, Oct. 2022.

[60] J. Zhang, Y. Ma, T. Endoh, “Efficient BCH Code Encoding and Decoding Algorithm with Divisor-Distance-Based Polynomial Division for STT-MRAM”, IEEE Transactions on Magnetics, Early access, v. 1, pp. 1-8, Jan. 2022.

[61] D. Yoon, M. Erez, “Memory mapped ECC: low-cost error protection for last level caches”, Proceedings of the International Symposium on Computer Architecture (ISCA), pp.116-127, Jun. 2009.

[62] D. Yoon, M. Erez, “Virtualized and flexible ECC for main memory”, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 397-408, 2010.

[63] D. Yoon, M. Erez, “Virtualized ECC: Flexible Reliability in Main Memory”, IEEE Micro, v. 31, n. 1, pp. 11-19, Jan.-Feb. 2011.

[64] A. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, N. Jouppi, “LOT-ECC: LOcalized and Tiered Reliability Mechanisms for Commodity Memory Systems”, Proceedings of the International Symposium on Computer Architecture (ISCA), pp. 285-296, 2012.

[65] D. Freitas, D. Mota, C. Marcon, J. Silveira, J. Mota, “LPC: An Error Correction Code for Mitigating Faults in 3D Memories”, IEEE Transactions on Computers, v. 70, n. 11, pp. 2001-2012, Nov. 2021.

[66] C. Argyrides, P. Reviriego, D. Pradhan, J. Maestro, “Matrix-based codes for adjacent error correction”, IEEE Transactions on Nuclear Science, v. 57, n. 4, pp. 2106-2111, Aug. 2010.

[67] J. Guo, L. Xiao, Z. Mao, Q. Zhao, “Enhanced memory reliability against multiple cell upsets using decimal matrix code”, IEEE Transaction on Very Large-Scale Integration (VLSI) Systems, v. 22, n. 1, pp. 127-135, Jan. 2014.

[68] S. Rahman, M. Sadi, S. Ahammed, J. Jurjens, “Soft error tolerance using Horizontal-Vertical-Double-Bit diagonal parity method”, Proceeding of the International Conference on Electrical Engineering and Information and Communication Technology (ICEEICT), pp. 21-23, 2015.

[69] D. Freitas, D. Mota, R. Goerl, C. Marcon, F. Vargas, J. Silveira, J. Mota, “PCoSA: A product error correction code for use in memory devices targeting space applications”, Integration, the VLSI Journal, v. 74, pp 71-80, Sep. 2020.

[70] F. Silva, W. Freitas, J. Silveira, C. Marcon, F. Vargas, “Extended Matrix Region Selection Code: An ECC for adjacent Multiple Cell Upset in memory arrays”, Microelectronics Reliability, v. 106, pp. 113582:1-9, Mar. 2020.

[71] C. Argyrides, D. Pradhan, T. Kocak, “Matrix codes for reliable and cost-efficient memory chips”, IEEE Transactions on Very Large-Scale Integration (VLSI) Systems, v. 19, n. 3, pp. 420-428, Mar. 2011.

[72] A. Dutta, N. Touba, “Multiple bits upset tolerant memory using a selective cycle avoidance-based SEC-DED-DAEC code”, Proceedings of the IEEE VLSI Test Symposium (VTS), pp. 349-354, 2007.

[73] D. Freitas, C. Marcon, J. Silveira, L. Naviner, J. Mota, “New decoding techniques for modified product code used in critical applications”, Microelectronics Reliability, v. 128, n. 114444, pp. 1-14, Jan. 2022.

[74] JEDEC Global Standards for the Microelectronics Industry - at https://www.jedec.org/, Dec. 2022.

[75] B. Jacob, S. Ng, D. Wang, “Memory Systems: Cache, DRAM, Disk”, Morgan Kaufmann Publishers, Sep. 2007, 982p.

[76] C. Ababei, “COEN-4730 Computer Architecture Lecture 5 Main Memory”, available at http://www.dejazzer.com/coen4730/doc/lecture05_dram.pdf, Apr. 2022.

[77] C. Slayman, M. Ma, S. Lindley, “Impact of Error Correction Code and Dynamic Memory Reconfiguration on High-Reliability/Low-Cost Server Memory”,

Proceedings of the IEEE International Integrated Reliability Workshop (IIRW), pp. 190-193, 2006.

[78] M. Bach, “ECC and REG ECC Memory Performance”, available at https://www.pugetsystems.com/labs/articles/ECC-and-REG-ECC-Memory-Performance-560/, Jun. 2022.

[79] S. Goossens, K. Chandrasekar, B. Akesson, K. Goossens, “Memory Controllers for Mixed-Time-Criticality Systems - Architectures, Methodologies and Trade-offs”, Springer International Publishing, 2016, 225p.

[80] X. Jian, R. Kumar, “ECC Parity: A Technique for Efficient Memory Error Resilience for Multi-Channel Memory Systems”, Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), pp. 1035-1046, 2014.

[81] H. Duwe, X. Jian, R. Kumar, “Correction prediction: Reducing error correction latency for on-chip memories”, Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 463-475, 2015.

[82] S. Paul, F. Cai, X. Zhang, S. Bhunia, “Reliability-Driven ECC Allocation for Multiple Bit Error Resilience in Processor Cache”, IEEE Transactions on Computers, v. 60, n. 1, pp. 20-34, Jan. 2011.

[83] T. Lin, Y. Li, M. Pedram, L. Chen, “Design Space Exploration of Memory Controller Placement in Throughput Processors with Deep Learning”, IEEE Computer Architecture Letters, v. 18, n. 1, pp. 51-54, Jun. 2019.

[84] N. Sadler, D. Sorin, “Choosing an Error Protection Scheme for a Microprocessor’s L1 Data Cache”, Proceedings of the International Conference on Computer Design (ICCD), pp. 499-505, 2006.

[85] S. Kim, “Area-Efficient Error Protection for Caches”, Proceedings of the Design Automation & Test in Europe Conference (DATE), pp. 1-6, 2006.

[86] Standard Performance Evaluation Corporation, "SPEC CPU 2006", available at https://www.spec.org/cpu2006/, Mar. 2023.

[87] C. Bienia, S. Kumar, J. Singh, K. Li, “The PARSEC benchmark suite: Characterization and architectural implications”, Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 72-81, 2008.

[88] PARSEC, “Overview”, available at https://parsec.cs.princeton.edu/overview.htm, Mar. 2023.

[89] X. Jian, H. Duwe, J. Sartori, V. Sridharan, R. Kumar, “Low-power, low-storage-overhead Chipkill correct via multi-line error correction”, Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), pp. 1-12, 2013.

[90] J. Kim, M. Sullivan, M. Erez, “Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory”, Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 101-112, 2015.

[91] J. Hsieh, C. Chen, H. Lin, “Adaptive ECC Scheme for Hybrid SSD’s”, IEEE Transactions on Computers, v. 64, n. 12, pp. 3348-3361, Dec. 2015.

[92] T. Chen, Y. Hsiao, Y. Hsing, C. Wu, “An adaptive-rate error correction scheme for NAND flash memory”, Proceedings of the IEEE VLSI Test Symposium (VTS), pp. 53-58, 2009.

[93] H.-M. Chen, S. Jeloka, A. Arunkumar, D. Blaauw, C.-J. Wu, T. Mudge, C. Chakrabarti, “Using Low Cost Erasure and Error Correction Schemes to Improve Reliability of Commodity DRAM Systems”, IEEE Transactions on Computers, v. 65, n. 12, pp. 3766-3779, Dec. 2016.

[94] S. Wang, F. Wu, Z. Lu, Y. Zhou, Q. Xiong, M. Zhang, C. Xie, “Lifetime adaptive ECC in NAND flash page management”, Proceedings of the Design Automation & Test in Europe Conference (DATE), pp. 1253-1556, 2017.

[95] S. Li, D. Yoon, K. Chen, J. Zhao, J. Ahn, J. Brockman, Y. Xie, N. Jouppi, “MAGE: Adaptive Granularity and ECC for resilient and power efficient memory systems”, Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), pp. 1-11, 2012.

[96] A. Basak, S. Paul, J. Park, J. Park, S. Bhunia, “Reconfigurable ECC for adaptive protection of memory”, Proceedings of the IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), pp. 1085-1088, 2013.

[97] X. Jian, R. Kumar, “Adaptive Reliability Chipkill Correct (ARCC)”, Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 270-281, 2013.

[98] J. Park, J. Park, S. Bhunia, “VL-ECC: Variable Data-Length Error Correction Code for Embedded Memory in DSP Applications”, IEEE Transactions on Circuits and Systems II: Express Briefs, v. 61, n. 2, pp. 120-124, Feb. 2014.

[99] L. Yuan, H. Liu, P. Jia, Y. Yang, “An adaptive ECC scheme for dynamic protection of NAND Flash memories”, Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 1052-1055, 2015.

[100] D. Shin, J. Park, J. Park, S. Paul, S. Bhunia, “Adaptive ECC for Tailored Protection of Nanoscale Memory”, IEEE Design & Test, v. 34, n. 6, pp. 84-93, Dec. 2017.

[101] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, L. Nevill, “Bit error rate in NAND flash memories”, Proceedings of the IEEE International Reliability Physics Symposium (IRPS), pp. 9-19, 2008.

[102] Y. Cai, E. Haratsch, O. Mutlu, K. Mai, “Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis”, Proceedings of the Design Automation & Test in Europe Conference (DATE), pp. 521-526, 2012.

[103] J.-C. Baraza-Calvo, J. Gracia-Morán, L.-J. Saiz-Adalid, D. Gil-Tomás, P.-J. Gil-Vicente, “Proposal of an Adaptive Fault Tolerance Mechanism to Tolerate intermittent Faults in RAM”, Electronics, v. 9, n. 12, pp. 2074.1-2074.30, Dec. 2020.

[104] J. Chen, X. Jiang, Y. Zhang, L. Liu, H. Xu and Q. Liu, “CARE: Coordinated Augmentation for Elastic Resilience on DRAM Errors in Data Centers”, Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 533-544, 2021.

[105] Y. Lee, G. Koo, Y.-H. Gong, S. Chung, “Stealth ECC: A Data-Width Aware Adaptive ECC Scheme for DRAM Error Resilience”, Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 382-387, 2022.

[106] S. Beamer, K. Asanovic, D. Patterson, “The GAP benchmark suite”, arXiv:1508.03619 [cs.DC], 2015, available at https://arxiv.org/abs/1508.03619, Mar. 2023.

[107] J. Bucek, K.-D. Lange, J. Kistowski, “SPEC CPU2017: Next generation compute benchmark”, Proceedings of the ACM/SPEC International Conference on Performance Engineering (CPE), pp. 41-42, 2018.

[108] N. Rohbani, T. Maiti, D. Navarro, M. Miura-Mattausch, H. Mattausch, H. Takatsuka, “NVDL-cache: Narrow-width value aware variable delay low-power data cache”, Proceedings of the IEEE International Conference on Computer Design (ICCD), pp. 264-272, 2019.

[109] M. Andjelkovic, A. Simevski, J. Chen, Ol. Schrape, Z. Stamenkovic, M. Krstic, S. Ilić, G. Ristić, A. Jaksic, N. Vasovic, R. Duane, A. Palma, A. Lallena, M. Rodríguez, “A Design Concept for Radiation Hardened RADFET Readout System for Space Applications”, Microprocessors and Microsystems, v. 90, pp. 104486:1-18, Apr. 2022.

[110] Everspin Technologies, “16Mb MRAM - Parallel Interface”, available at https://www.everspin.com/16mb-mram-parallel-interface, Dec. 2022.

[111] JEDEC Global Standards for the Microelectronics Industry, “Low Power Double Data Rate 5 (LPDDR5)”, available at https://www.jedec.org/standards-documents/docs/jesd209-5b, Dec. 2022.

[112] M. Patel, G. de Oliveira, O. Mutlu, “HARP: Practically and Effectively Identifying Uncorrectable Errors in Memory Chips that Use On-Die Error-Correcting Codes”, Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 623-640, 2021.

[113] P. Nair, V. Sridharan, M. Qureshi, “XED: Exposing On-Die Error Detection Information for Strong Memory Reliability”, Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA), pp. 341-353, 2016.

[114] J. Decaluwe, “MyHDL - Design hardware with Python”, available at https://myhdl.org/, Jun. 2022.

[115] XESS, “XESS Corporation”, available at https://xess.com/, Jun. 2022.

[116] XESS Libraries, “SDRAM controller and dual-port interface”, available at https://github.com/xesscorp/VHDL_Lib/blob/master/SdramCntl.vhd, Jun. 2022.

[117] M. Stefani, “Dynamic Fault Tolerance Module for Memory Controller”, available at https://github.com/marcops/dftm_module/, Dec. 2022.

[118] C. Stolze, “DDR5 RAM: Preparing for the Next Generation of Memory”, Corsair, available at https://www.corsair.com/uk/en/blog/ddr5-primer. Dec. 2022.

[119] RISC-V, “Specifications”, available at https://riscv.org/technical/specifications/, Dec. 2022.

[120] Micron, “DDR4 SDRAM - MT40A4G4, MT40A2G8, MT40A1G16”, available at https://datasheet.octopart.com/MT40A2G8JC-062E%3AE-Micron-datasheet-141417503.pdf, Jun. 2022.

[121] A. Duran, X. Teruel, R. Ferrer, X. Martorell, E. Ayguade, “Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP”, Proceedings of the International Conference on Parallel Processing (ICPP), pp. 124-131, 2009.

[122] Hardinfo, “Benchmark for Linux”, available at https://github.com/lpereira/hardinfo, Sep. 2009.

[123] Cadence, “Genus Synthesis Solution”, available at https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html, Dec. 2022.

[124] N. Jouppi, A. Kahng, N. Muralimanohar, V. Srinivas, “CACTI-IO Technical Report”, https://www.hpl.hp.com/techreports/2013/HPL-2013-79.pdf, pp. 1-37, Sep. 2013.

[125] M. Stefani, “Absimth: lA Hardware simulator written in Java”, available at https://github.com/marcops/Absimth, Apr. 2023.

[126] M. Stefani, F. Silva, C. Marcon, J. Silveira “Assessing Rules in Memory Controllers with Hardware Simulator Executing Real Programs”, Proceedings of the Brazilian Symposium on Computing Systems Engineering (SBESC), pp. 1-8, 2022.

[127] M. Stefani, F. Silva, C. Marcon, J. Silveira “Memory Controller with Adaptive ECC for Reliable System Operation”, Proceedings of the 36th Symposium on Integrated Circuits and Systems Design (SBCCI), pp. 1-6, 2023.

[128] O. Mutlu, “Memory scaling: A systems architecture perspective”, Proceedings of the 5th IEEE International Memory Workshop (IMW), pp. 21-25, 2013.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br