Assessing Rules in Memory Controllers with Hardware Simulator Executing Real Programs

Authors and affiliations omitted for a blind review process

***Abstract*— Electronic memories are subject to failure due to magnetic fields and radiation, causing temporary faults that can be mitigated by error correction codes. Evaluating the efficacy of these mechanisms in hardware can make the design more expensive, take a long prototyping time, and even lose time to market. Hardware simulators or emulators are often generic, requiring specific and specialized development to evaluate memory controllers. This work proposes Absimth to assess the computational system behavior in the presence of memory errors. 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.**

Keywords*— *Hardware Simulator; Memory Controller; Fault Tolerance evaluation.

2022 XII Brazilian Symposium on Computing Systems Engineering (SBESC) | 978-1-6654-7425-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/SBESC56799.2022.9964880

I. Introduction

Hardware reliability is a central requirement for high-performance computing and storage projects [1]. The memory reliability is particularly sensitive to variation in the manufacturing process, environmental conditions, or wear caused by the time of use [2][3]. Many areas are affected by memory errors, like (i) big service providers such as Facebook [4] with 2.5% per month and Google with 8% of memory modules per year [1]; (ii) personal computers turned on for 30 days have a 0.059% chance of failing, according to Microsoft [5]; and (iii) space projects, where it is common to use commercial memories due to their high density and low cost. However, commercial memories are not resilient to the numerous problems that can occur due to radiation, with the loss of cell functionality being the main consequence in intense radiation scenarios [6][7][8][9].

Fast and accurate simulation tools are essential for computer architects to analyze problems and compare the efficacy of each solution. It is even more critical to try a new set of constraints or incorporate an emerging technology with widely different properties than the standard technologies [10]. Computer system simulators are built for general purposes and cannot run native programs in general. Besides, there are few proposals regarding memory controller simulators, like FaultSim [10], whose operation is based on an analytical model.

This work proposes Absimth, an extensible fast memory controller simulator for improving studies of memory controller architectures used in various mobile devices to extreme-scale supercomputers. Absimth enables the behavior evaluation of the traditional applications over intense bitflips with different Error Correction Codes (ECC). To reach the behavior assessment, we simulated a homogeneous multiprocessor architecture composed of RISC-V 32I processors that access memory modules from a DDR4 SDRAM with ECC model MT40A1G16 from Micron [11], applying a synthetic application described in C.

Absimth allows evaluating the ECC flow applied to memories; it enables configuring and creating custom modules for processors, memory devices, and memory controllers. Additionally, the simulator provides error injection models into memory for assessing the system behavior in the face of memory runtime faults with tons of metrics.

II. Related Work

The design of general and specific purpose simulators is a significant research challenge, addressed over decades by several works. This section discusses some recent work and presents the motivation for proposing Absimth.

Nair et al. [10] proposed FaultSim, a configurable simulator to assess the reliability of memories, employing the Monte Carlo algorithm that generates pattern faults equivalent to the real world. The authors implemented and evaluated BCH-1 and Chipkill ECCs in real executions to validate FaultSim; the experiments show a deviation between 0.036% and 8.41%.

Dong et al. [12] presented a simulator to model Non-Volatile Memory (NVM) at the circuit level. The simulator evaluates timing, energy, and area consumption, supporting validating various NVM technologies, such as STT-RAM, PCRAM, ReRAM, and the traditional NAND FLASH.

Balasubramonian et al. [13] added a model to assess memory energy consumption to CACTI [14], allowing defining new interconnection types. The analysis has shown that the design parameters significantly impact energy consumption. A simple topology change can increase performance by 22% and reduce cost by 65%. Additionally, the article presents DDR3 and DDR4 designs that improve performance by 18% and reduce energy consumption by 23%.

Kim, Yang, and Mutlu [15] introduced Ramulator, a generic memory simulator that supports several standards like DDR3/4 LPDDR3/4, GDDR5, WIO1/2 HBM, and other academic proposals like SALP, AL-DRAM, ROW Clone, TL-DRAM, and SARP.

Mittal, Wang, and Vetter [16] presented Destiny, a tool to model, at the semiconductor level, the manufacturing process of 2D and 3D memories, such as SRAM, resistive STT-RAM, eDRAM, SOT-RAM, DWM, and flash. The simulator enables latency and area validation and provides comprehensive design modeling and spatial exploration. Destiny results were validated with several commercial prototypes, including the 2D and 3D designs of SRAMs and eDRAMs. The experiments presented less than 10% of modeling errors for most cases and less than 20% for all cases.

Chatterjee et al. [17] described the USIMM (Utah SIMulated Memory Module), an infrastructure for simulating DRAM memory, focusing on the energy consumption of the request schedulers that are part of the memory controller.

Wang et al. [18] developed MEMRES, a fast simulator to test the main memory reliability. MEMRES enables simulating memory failures, being computationally efficient in obtaining failure probabilities, and optimizing the memory system reliability. The authors performed a case study based on Spin-Transfer Torque Random Access Memory (STT-RAM); the results indicated that in-memory ECC could significantly mitigate the write error rate of STT-RAM.

Binkert et al. [19] proposed the gem5 simulator, a modular platform for researching computer system architecture, ranging from system level to processor microarchitecture. Gem5 is an open software project conceived initially for research into computer architecture in academia. Still, it has been used in academic works and industrial research by companies such as ARM, AMD, Google, HP, and Samsung.

Several works focus on the modeling and simulation of general-purpose computational systems, and some are dedicated to evaluating specific aspects of memories. Nevertheless, to the best of our knowledge, this is the first work on memory controller modeling and simulation focusing on fault tolerance. The Absimth simulator allows evaluation techniques and architectures of memory controllers, enabling the exploration of tradeoffs between error correction efficacy and implementation costs.

III. Absimth Architecture Description

This chapter details the Absimth simulator and the functionalities available to evaluate the behavior of memory controllers in a stressful environment.

A. Absimth Architecture

Fig. 1 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.

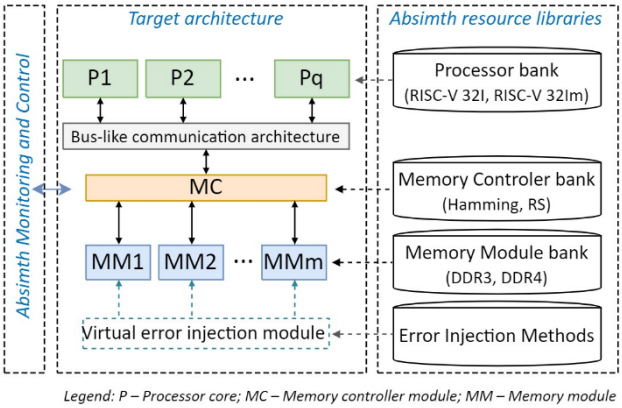

Fig. 1. High-level description of the Absimth platform.

All modules described next have a standardized interface allowing them to be customized or changed.

Processor

Absimth includes the three following RISC-V 32-bit compatible processors [20] and allows executing 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 of RISC-V 32Im but including floating point).

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, and Reed-Solomon.

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.

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 [1][5] and a five-error injection based on error occurrence probability:

- *NoFaultError* – meaning a memory without errors to assess the application execution time and amount of data transferred. It is the basis for comparing with other scenarios, occurring in about 91.78% of server cases [1][5].
- *OneError* – implying a single bitflip to simulate the most common error scenario, occurring in approximately 8% of server cases [1][5]. 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 [1][5].
- *One2MultipleErrors* – one bit corrupted initially followed by multiple bitflips to simulate 0.154% to 0.176% of the multiple-error scenarios [1][5]. 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 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.

Operating System

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 instruction 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.

Reports

At the end of the application execution, Absimth creates 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 read and write 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 (in milliseconds), (ii) the total number of ticks, and (iii) the maximum tick for each core.

B. 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, implements the execution, insertion of errors, and application task monitoring.

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. Therefore, 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.

Task Simulation

Fig. 2 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.

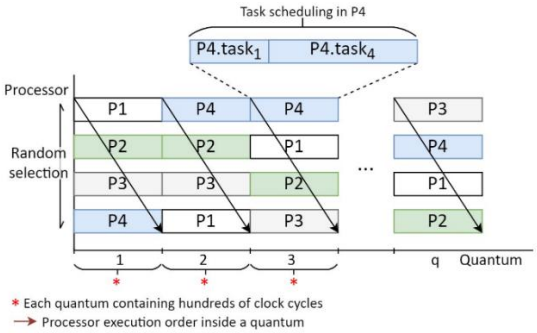

Fig. 2. Example of Task Simulation phase encompassing four processors (P1…P4) execution during q quanta of simulation. This figure emphasizes intra-quantum scheduling of P4, covering $task_1$ and $task_4$.

IV. Memory Bitflip Observation

This section shows the memory bitflip observation processes, exemplifying three synthetic tasks executing in a single processor (Fig. 3). This processor is connected to a DDR4 through a memory controller that performs reading and writing operations using Hamming ECC.

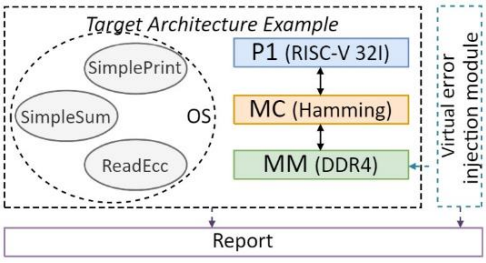

Fig. 3. Target architecture and tasks employed in the simulation example.

A. Synthetic Application and Hardware Description

Fig. 4 describes *SimplePrint*, *ReadEcc*, and *SimpleSum* - three synthetic low-complex tasks developed to observe the behavior of an application operating over a memory RAM with bitflip. Fig. 4(a) displays the small source code of the program *SimplePrint*, which only sends a message to the standard output. Fig. 4(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. Fig. 4(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)

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

(c)

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

(b)

Fig. 4. Source code of three synthetic tasks: (a) *SimplePrint* (b) *ReadEcc*, and (c) *SimpleSum.*

Fig. 5 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 [11] 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
```

Fig. 5. Simulation Setup.

B. Memory Inspector

Absimth allows us to look in-depth at the processor status or memory during each execution or in the final process. Fig. 6 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 0x3E8. 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 1 byte.

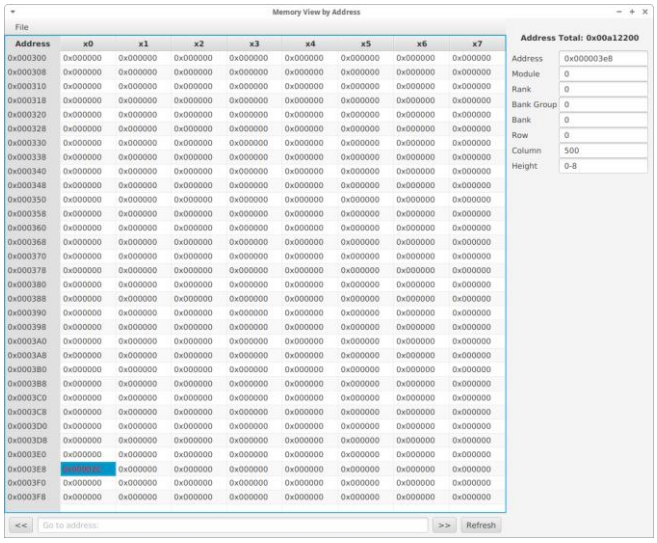

Fig. 6. *Memory Area Inspector* tool, containing an error in address 0x3E8.

Absimth provides the DDR memories default 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. Fig. 7(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, considering that chip 0 of module 0, rank 0 contains errors since it is colored in red in Fig. 7(a), Fig. 7(b) shows the visualization inside this chip, encircling the organization of memory banks inside each bank group.

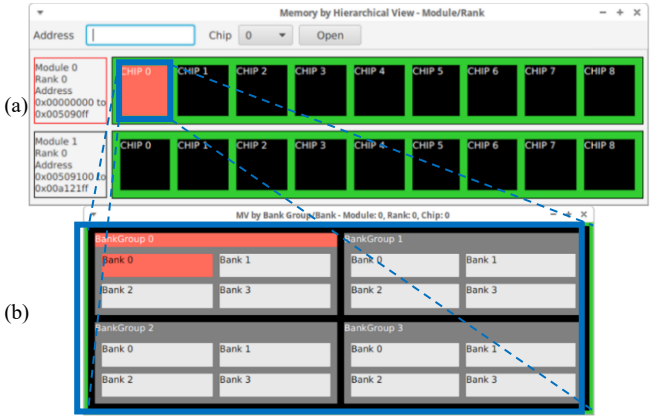

Fig. 7. (a) The highest memory level, including memory modules, rank, and chips, showing that Chip 0, Module 0, and Rank 0, contain at least one bit with error; (b) Memory bank organization inside the bank groups. The Bank 0 of Bank Group 0 contains errors since it is colored in red.

Fig. 8 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. The designer can also see the 3D view of a memory cell (along with height); Fig. 9 exemplifies a 3D view.

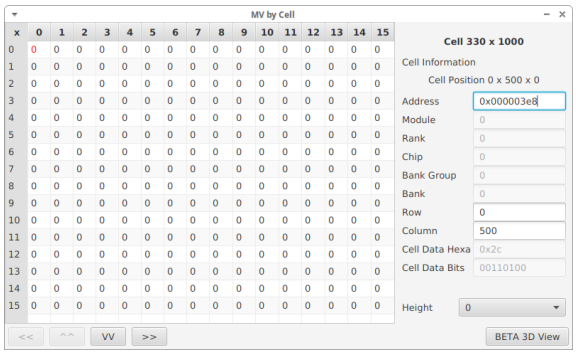

Fig. 8. Memory cell window; this view displays a single error on bit 0 (colored in red) of address 0x3E8.

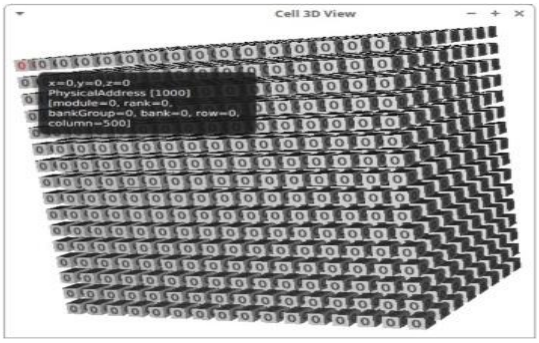

Fig. 9. 3D memory cell preview window.

C. 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.

Fig. 10 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.

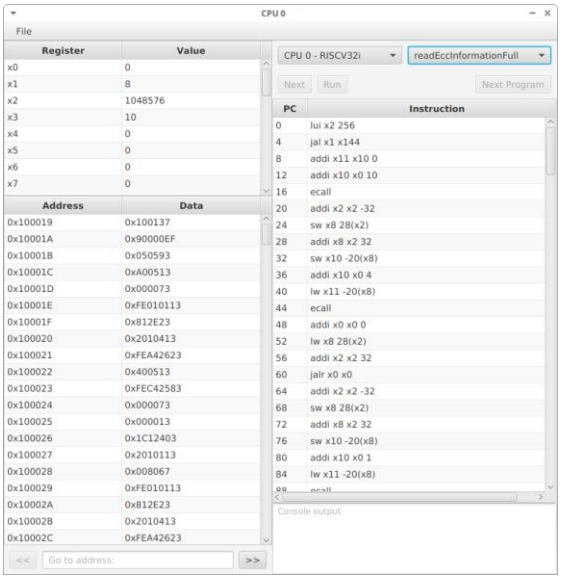

Fig. 10. *Processor Management* tool, covering processor registers, task code objects, and memory data addresses.

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 Fig. 11.

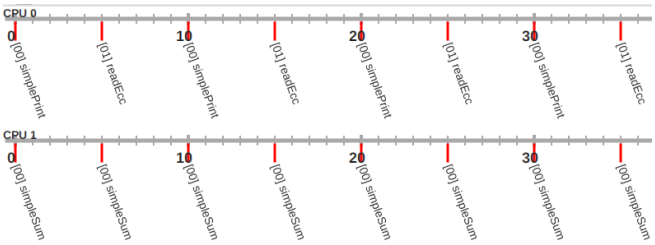

Fig. 11. Window for viewing the timeline of all processors.

D. Execution Report

Absimth finishes the target architecture simulation generating the report of Fig. 12, 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
```

Fig. 12. Simulation Report.

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 that determines if the application finished the execution successfully or was affected by the bitflip occurrence.

V. Benchmark Exploration

Table I displays the results of a benchmark executing in scenarios combining different error rates and computational resource combinations. This benchmark 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.

The **Benchmark Information** contains the (a) last tick, (b) sum of all ticks executed by the processors, (c) total of instruction 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 executed considering the four 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;
5. **RandomArbitratyError** – describes a scenario with a random number of bitflips arbitrarily placed.

The **RandomNearError** and **RandomArbitratyError** characteristics are detailed in Table II, 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 II – Characterization of bitflip scenarios.

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 I produces results consistent with the ones explored on Google and Microsoft servers [1][5], opening opportunities for a vast behavior study of memory devices and memory controllers. Table I 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; the next section evaluates this approach.

VI. Memory Controller Alternative Exploration

As demonstrated in Section V, some applications do not finish the execution due to a bitflip error; one alternative to this issue could be to start reading and writing a more powerful ECC, like RS over HM when the first bitflip occurs.

To reach this exploration, we implement the *Double ECC Memory Controller* (DEMC), using the architecture defined in Section III, having a RISC-V 32 Bits operating with a DDR4. DEMC stops to use HM and starts using RS when it finds any bitflip. The rightest bit of ECC identifies the encoding type used to write/read data (0: HM and 1:RS).

Table I - Simulation Summary.

	Benchmark information						WithoutError			SingleBitflip			One2Many			RandomNearError				RandomArbitraryError			
#	a	b	c	d	e	f	WE	HM	RS	WE	HM	RS	WE	HM	RS	N°BF	WE	HM	RS	N°BF	WE	HM	RS
i	782	782	7038	738	656	208										1	OK	OK		4	OK		
ii	4210	4210	37890	10766	1536	4546										10				9			
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			OK	4			OK
x	3635	3635	32715	18166	1112	4542										9				8	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	OK	OK		4	*NOK*	OK	
xiv	3880	3880	34920	6256	1153	3994										9				9			
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 that the simulation finished without or with error, respectively

We evaluated this scenario using the ReadECC application with the configuration explained in Section III:

- Use the *NoFaultError* to simulate 91.78% of the cases;
- *OneError* to simulate approximately 8%;
- *MultipleErrors*, which represents around 0.055%;
- *One2MultipleErrors*, which represents about 0.16% of the multiple error scenarios, and
- *One2MultipleErrors* with two applications running simultaneously over this environment.

For the scenario without faults or with one bitflip, it was possible to run the program successfully for the three controllers. With one bit corrupted initially, followed by multiple errors, the DEMC and RS controllers were effective. For a scenario with multiple faults, only RS executed the program successfully; DEMC, having started with HM, could not correct the data but changed the ECC to RS, making the next application better protected as it is.

Table III contains the synthesis of the results obtained with the number of reads and writes performed by the memory controller.

TABLE III – ALTERNATIVE MEMORY CONTROLLER EXPLORATION.

Scenario			i	ii.	iii	iv	v
HM	Read		2400133	2400133	* 900094	* 900094	* 1800188
	Write		200180	200180	* 100178	* 100178	* 200356
RS	Read		2400133	2400133	2400133	2400133	4800266
	Write		200180	200180	200180	200180	400360
DEMC	HM	Read	2400133	1369107	1369107	* 900094	** 2268218
	HM	Write	200180	200180	200180	* 100178	** 2268218
	RS	Read	-	1031028	1031028	-	** 1032011
	RS	Write	-	-	-	-	-

Legend: * application did not finish
** only one application did not finish

DEMC mitigates approximately 80% of the cases of multiple bitflips that previously had one corrupted bit, thus guaranteeing an increment of 0.17% more in the total probability of the application to continue executing, resulting in 99.956% of efficacy, as shown in Table IV.

TABLE IV – MEMORY ERROR PROBABILITY.

State	Without ECC	Hamming.	DEMC	Reed Solomon
OK	91.780%	99.780%	99.956%	~99.999%
Fail	8.220%	0.220%	0.044%	~0.001%

DEMC demonstrates another benefit, greater efficiency and efficacy for 91.78% of the cases in which it is unnecessary to have a robust fault tolerance coding, ensuring lower energy consumption and latency, and a powerful and assertive coding for only the addresses with high error incidence.

VII. CONCLUSION

Fault tolerance strategies in memories have a significant impact on providing computational reliability to higher levels. Evaluating techniques in memory controllers is a challenge widely researched with a high time cost. Absimth facilitates the memory controller and ECC research in an era when main memory is undergoing rapid changes.

This paper evaluates a wide range of scenarios with the Absimth tool based mainly on research from Google and Microsoft [1][5] and understands their behavior. With the understanding of these scenarios, it was possible to prototype a *Double ECC Memory Controller* (DEMC) with a slight alteration of the memory controller but with a significant impact on the error correction rate.

The fast memory controller prototyping on Absimth enables much research and ideas evaluation before the hardware prototype and how the application will behave under this memory controller, such as the proposed DEMC.

This work also introduces Absimth, a tool for building and simulating multiprocessor target architectures that access memory modules through memory controllers. The simulator enables some tools for performing injection error patterns and evaluating fault tolerance techniques with support for many ECC standards.

ACKNOWLEDGMENT

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

REFERENCES

[1] B. Schroeder, E. Pinheiro, W. Weber, “DRAM Errors in the Wild: A large-scale field study”, Communications of the ACM, vol. 54, n. 2, pp. 100-107. Feb. 2011.

[2] S. Mittal, “A survey of architectural techniques for managing process variation”, ACM Computing Surveys, vol. 48, n. 4, pp. 1-29, May 2016.

[3] 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, vol. 104, n.7, pp. 1410-1448. Jul. 2016.

[4] 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.

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

[6] H. Quinn, P. Graham, T. Fairbanks, “SEEs Induced by High-Energy Protons and Neutrons in SDRAM”, Proceeding of the IEEE Radiation Effects Data Workshop, pp. 1-5, 2011.

[7] S. Duzellier, D. Falguère, R. Ecoffet, “Heavy ion/proton test results on high integrated memories”, IEEE Radiation Effects Data Workshop, pp. 36-42,1993.

[8] H. Shindou, S. Kuboyama, N. Ikeda, T. Hirao, S. Matsuda, “Bulk damage caused by single protons in SDRAMs”, , IEEE Transactions on Nuclear Science vol. 50, n. 6, pp. 1839-1845, Dec. 2003.

[9] 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, vol. 70, n. 11, Nov. 2021.

[10] P. Nair, D. Roberts, M. Qureshi, “FaultSim: A fast, configurable memory-reliability simulator for conventional and 3D-stacked systems”, ACM Transactions on Architecture and Code Optimization (TACO), vol. 14, n. 4, art. 44, pp.1-24, Jan. 2016.

[11] Micron, “DDR4 SDRAM - MT40A4G4, MT40A2G8, MT40A1G16”, https://datasheet.octopart.com/MT40A2G8JC-062E%3AE-Micron-datasheet-141417503.pdf. Jun. 2022.

[12] X. Dong, C. Xu, Y. Xie, N. Jouppi; “NVSim: A circuit-level performance, energy, and area model for emerging non-volatile memory”. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 31, n.7, pp. 994-1007, Jul. 2012.

[13] R. Balasubramonian, A. Kahng, N. Muralimanohar, A. Shafiee, V. Srinivas, “CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories”, ACM Transactions on Architecture and Code Optimization (TACO), vol. 14, n. 2, pp. 1-25, Jul. 2017.

[14] HP Labs, “CACTI - An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model”, https://www.hpl.hp.com/research/cacti/, Agu. 2022.

[15] Y. Kim, W. Yang, O. Mutlu, “Ramulator: A fast and extensible DRAM simulator”, IEEE Computer Architecture Letters, vol. 15, n. 1, pp. 45-49, Jun. 2016.

[16] S. Mittal, R. Wang, J. Vetter, “DESTINY: A comprehensive tool with 3D and multi-level cell memory modeling capability”, Journal of Low Power Electronics and Applications, vol. 7, n. 23, pp. 1-24, Apr. 2017.

[17] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, Z. Chishti. “USIMM: the Utah SImulated Memory Module”. University of Utah, Technical Report UUCS-12-002, 24p., Feb. 2012.

[18] S. Wang, H. Hu, H. Zheng, P. Gupta, “MEMRES: A Fast Memory System Reliability Simulator”, IEEE Transactions on Reliability, vol. 65, n. 4, pp. 1783-1797, Dec. 2016.

[19] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, D. Wood. “The gem5 simulator”. ACM SIGARCH Computer Architecture News, vol. 39, n. 2, pp. 1-7. May 2011.

[20] RISC-V, “Specifications”, https://riscv.org/technical/specifications/, Agu. 2022.

[21] A. Waterman, K. Asanovic´, “The RISC-V Instruction Set Manual - Volume II: Privileged Architecture - Document Version 20190608-Priv-MSU-Ratified”, Section 2.4, Load and Store Instructions, https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf, Agu. 2022.