Ramulator2_ECC

This plugin adds large-size ECC/EDC emulation to **Ramulator2** to evaluate memory reliability, bandwidth, and latency trade-offs in AI and HPC workloads.

Introduction

Modern AI and high-performance computing (HPC) systems demand extremely high memory bandwidth and capacity, making High Bandwidth Memory (HBM) a critical enabling technology. However, HBM's high cost per gigabyte (GB/$) severely limits the large-scale deployment of advanced AI models. A promising solution is to introduce large-size Error Correction Codes (ECC) into the memory system to achieve a better trade-off between reliability, bandwidth, and cost.

We primarily focus on large-scale AI model workloads, which are particularly well-suited for this optimization approach for the following reasons:

- Memory access patterns are typically coarse-grained and sequential.
- Random bit errors have minimal impact on final model accuracy.
- Different model components exhibit varying sensitivity to errors, allowing differentiated protection strategies.

This project extends Ramulator2, a cycle-accurate DRAM simulator, with an ECC emulation plugin framework that supports large-size ECC and EDC (Error Detection Codes). Key features include:

- Configurable ECC codeword sizes such as 512B, 1KB, and 4KB
- Injection of random bit errors with adjustable error rates
- Simulation of bandwidth and latency overhead caused by redundancy reads
- Modeling of read latency impact introduced by ECC decoding

Our core insight is that increasing ECC codeword size can maintain fault tolerance while reducing redundancy overhead. Since the number of ECC bits typically grows much slower (e.g., logarithmically) than the data block size, the ECC overhead per unit of data decreases, resulting in higher effective bandwidth. In AI-style streaming workloads, pipelined execution can also hide the latency introduced by ECC encoding and decoding. This makes it possible to provide more efficient, reliable, and cost-effective memory fault tolerance for future AI and HPC platforms.

It is important to note that this plugin currently serves mainly as a simulation framework. While it is not yet feature-complete, it is highly extensible and suitable for exploratory research and further development. Users are encouraged to adapt and expand the plugin based on their own needs.

Plugin Integration Guide

Ramulator2 provides a flexible and modular plugin framework that allows users to insert custom processing logic (such as ECC, compression, or encryption) between the arrival of a memory request and the actual write to memory. The frontend (Frontend) and backend (MemorySystem) are designed to be approximately black

boxes, with the goal of minimizing modifications to the core simulation flow. Instead, custom logic can be inserted via the plugin interface within the DRAM controller.

To add a new plugin, follow these steps:

1. Navigate to the plugin interface definition directory:

```
/src/dram_controller
```

2. Use `plugin.h` as a base template, which defines the standard interface and registration mechanism for plugins.

3. Implement your plugin logic as a new C++ file and place it in:

```
/src/dram_controller/impl/plugin/
```

4. Create a class that inherits from both `IControllerPlugin` and `Implementation`.

5. Register the plugin class using the macro:

```
RAMULATOR_REGISTER_IMPLEMENTATION(YourPluginClassName, "PluginName",
"Description")
```

6. Implement the following key methods in your plugin class:

    - `init()`: Initialize internal parameters and register statistics using `register_stat()`. Read configuration values from YAML if needed.
    - `setup()`: Bind the plugin to the DRAM controller context.
    - `update(bool request_found, ReqBuffer::iterator& req_it)`: Define the logic that processes requests during simulation.
    - `finalize()`: Clean up and output final statistics after simulation ends.

7. Add your plugin's `.cpp` file to the build system:

    - Open `src/dram_controller/CMakeLists.txt`
    - Add your new file to the source list

8. Finally, activate the plugin in the YAML configuration by adding it under the `plugins` section of the `Controller`. You can also define any required parameters here.

Once completed, the plugin will be automatically integrated into the simulation pipeline.

ECCPlugin Complete Processing Flow

Plugin Initialization Stage (`init()`)

- Read configuration parameters:
    - Data block size `DATA_BLOCK_SIZE`
    - EDC size `EDC_SIZE`
    - ECC size `ECC_SIZE`
    - Raw bit error rate `bit_error_rate`
    - Maximum allowed failure probability `max_failure_prob`
    - Selected EDC type (e.g., `crc32`)
    - Selected ECC type (e.g., `bch`)
- Register these input parameters and runtime statistics via `register_stat()`, so that they can be automatically reported at the end of the simulation.

Runtime Phase (`update()`)

When a request is received (`update(bool request_found, ReqBuffer::iterator& req_it)`)

Determine the request type:

Regular Write (`Write`)

1. Extract new data from the request (`req_it`), or generate random data if payload is missing.
2. Calculate the EDC checksum for the data block and append it to the end.
3. Calculate the ECC codeword for the data block (including EDC).
4. Store:
    - `[Data + EDC]` in `m_data_storage`
    - `[ECC]` in `m_ecc_storage`
5. Update statistics:
    - `total_ecc_size`
    - `total_edc_size`

Regular Read (`Read`)

1. Find the `[Data + EDC]` block in `m_data_storage`.
    - If not found, generate a fake data block.
2. Separate the `Data` and `EDC`.
3. Verify `Data` using EDC:
    - If EDC passes: return the data directly to the payload.
    - If EDC fails:
        - Read the ECC codeword.
        - Perform ECC decoding (`ECCDecode()`) to attempt correction.
        - If correction succeeds:
            - Update `[Data + EDC]` and `[ECC]` in storage.
            - Return the corrected data.
        - If correction fails:
            - Attempt retry, RAID recovery, or report a Fatal UE (Uncorrectable Error).
4. Update statistics:

- On success or failure, update `edc_success_count`, `edc_failure_count`, `ecc_success_count`, `ecc_failure_count`.

Partial Write (`PartialWrite`)

(**Extended command; the payload must specify `offset` and `length`**)

1. Read the existing `[Data + EDC]` from `m_data_storage`.
2. Verify the old `Data` using EDC:
   - If EDC verification fails, perform full ECC decoding to repair.
3. Extract the partial region:
   - old_chunk = the original `[offset, offset+length)` region
   - new_chunk = new data to write (from the payload)
4. Update the data:
   - Overwrite the `[offset, offset+length)` region in `old_data` with `new_chunk`.
5. Incrementally update the ECC (based on the linearity property):
   - Encode both `old_chunk` and `new_chunk` using RS Encode.
   - Update the original ECC:
     `new_ecc = old_ecc ⊕ Enc(old_chunk) ⊕ Enc(new_chunk)`
6. Recalculate the new EDC for the updated data block.
7. Update `[Data + EDC]` and `[ECC]` in `m_data_storage` and `m_ecc_storage`.

Simulation Finalization (`finalize()`)

- Clear `m_data_storage` and `m_ecc_storage`.
- Output a summary log, e.g., `[ECCPlugin] Storage cleared.`

User-Configurable Variables

To support flexible simulation and formula-based analysis, the ECC plugin allows users to define the following key parameters via the YAML configuration file. These variables directly affect ECC/EDC overhead, latency modeling, and unit cost estimation.

Basic Variables

These parameters define the core characteristics of the error detection and correction mechanisms:

- **ECC Size**: Number of ECC (error correction code) bytes per memory data block (e.g. 16B, 32B, 128B, etc.)
- **EDC Size**: Number of EDC (error detection code) bytes per memory data block (e.g. 4B for CRC32, 8B for CRC64)
- **ECC Type**: Type of correction algorithm used (e.g. Hamming, BCH, Reed-Solomon)
- **EDC Type**: Type of detection algorithm used (e.g. Checksum, CRC32, CRC64)
- **Protected Data Block Size**: Logical size of the raw data block covered by one ECC/EDC unit (e.g. 64B, 128B, 512B, 1KB, etc.)

Environmental Variables

These variables model the bandwidth constraints of the system's external and internal buses and directly impact transmission latency and bandwidth calculations:

- **External Bus Type / Bandwidth**: Represents the bandwidth of off-chip or channel-level buses (e.g. PCIe, NVLink, HBM interface) (unit: GB/s)
- **Memory Bus Type / Bandwidth**: Represents the read/write bandwidth between the memory controller and DRAM modules (unit: GB/s)

Algorithms Used

Error Detection Code (EDC)

- **Checksum**
  - Sums all bytes together.
  - Fast to compute, suitable for simple error detection but has weaker detection capabilities.
- **CRC32**
  - 32-bit Cyclic Redundancy Check (CRC) using the Ethernet standard polynomial.
  - Very effective at detecting random bit errors.
- **CRC64**
  - 64-bit Cyclic Redundancy Check.
  - Provides even stronger error detection capability, suitable for large data blocks.

Error Correction Code (ECC)

- **Hamming Code (Simplified)**
  - Protects against single-bit errors (simplified model here).
- **Reed-Solomon Code (Simplified)**
  - Symbol-based (byte-level) linear error-correcting code.
  - Real RS codes can correct burst errors and*support incremental updates thanks to their linearity.
- **BCH Code (Simplified)**
  - Capable of correcting multiple random bit errors.
  - Based on polynomial operations, but simplified in this implementation.

Dynamic ECC Strength Estimation

- Automatically determines the minimum required ECC strength (i.e., number of correctable errors `t`) based on data block size, raw bit error rate (BER), and the target maximum failure probability.

Random Error Injection

- Random bit-flips are injected into the data at runtime based on the configured `bit_error_rate`, simulating real-world hardware error environments.

ECC Design & Size

This section outlines how ECC size and structure are defined in the simulation, and how they influence memory partitioning and access latency.

Input Variables

These are key user-configurable inputs used to describe the ECC configuration:

- **ECC Size Selection**: Classical ECC sizes such as 256B, 512B, 1KB, 4KB
- **Protected Data Block Size**: Logical block sizes protected by one ECC codeword (e.g., 1KB, 4KB, 8KB, 16KB, 32KB)
- **ECC Scheme Selection**: Supported schemes include Reed-Solomon (RS), Hamming, LDPC, etc.

Controller Logic

ECC configuration affects how memory is partitioned and how error protection is applied at runtime:

- **ECC Partition Size Calculation** Computes how much memory is reserved for ECC data:

  ```
  ECC Region Size = Total Memory Space × (ECC Size) / (ECC Size + EDC Size + Target Protected Data Block Size)
  ```

- **ECC Encoder/Decoder Simulation** Models the latency incurred by ECC encoding and decoding operations.

- **Maximum Correctable Bit Count** Simulates the number of bit errors each ECC scheme can correct; used to define error correction capability and applicable Bit Error Rate (BER) thresholds.

- **Maximum Supported Bit Block** Indicates the largest supported data block per ECC scheme, which influences block granularity and ECC overhead.

- **Storage Latency**

  ```
  = Encoding Computation Latency + ECC Write Latency
  ```

- **Access Latency**

  ```
  = ECC Decoding Latency + DRAM Read Latency
  ```

EDC Design & Size

This section introduces the input parameters for various EDC (Error Detection Code) types and the controller-side logic used to model EDC overhead, latency, and its role in the ECC flow.

Input Variables

- **EDC Type Selection** Different EDC algorithms require different code lengths and support varying maximum data block sizes:
    - Parity Bit
    - CRC-8
    - CRC-16
    - CRC-32
    - Reed-Solomon Detection Code
    - Hash Functions (e.g., SHA, MD5)

Controller Logic

- **EDC Region Size Calculation**

```
EDC Region Size = Total Memory Space × EDC Size / (ECC + EDC + Protected
Data Block Size)
```

- **EDC Check on Read** Reads the data block and its corresponding EDC checksum, recomputes the EDC for the current data, and compares it with the stored EDC:
    - **EDC Pass**: Data is valid and directly returned
    - **EDC Fail**: Data may be corrupted and requires ECC correction
- **Partial Write EDC Check** Reads the data block and EDC checksum to determine if a direct partial write is safe:
    - **EDC Pass**: Directly update the target data block and incrementally update ECC
    - **EDC Fail**: Read the full ECC codeword, perform error correction, then update the data and recompute ECC/EDC
- **Large Block Write EDC Check** Performs a full read-modify-write (RMW) sequence:
    - Read the original data and ECC
    - Decode and correct if needed
    - Write the new data, updated EDC, and ECC

Redundant Read and Error Injection Experiment

This module implements a simulation framework to evaluate ECC (Error Correction Code) behavior under bit error conditions and quantify the bandwidth impact of redundant reads caused by ECC verification failures.

Input Variables

- **Bit Error Injection Rate**: Configurable injection rates ranging from 10^{-3} to 10^{-9}, used to emulate realistic bit error scenarios. This allows stress-testing of ECC response mechanisms across a wide error spectrum.

Controller Logic

- **Additional ECC Reads**: When an ECC verification check fails, the system triggers secondary ECC codeword reads. These redundant reads simulate real-world recovery behavior and contribute to increased bandwidth consumption, providing insight into ECC overhead under error conditions.

Build and Execution Scripts

- **build.bat**
  A script to compile the Ramulator2 project.
  It handles the standard build process using the provided CMake configuration.

- **exec_HBM3.bat**
  A script to run simulations using a predefined HBM3 (High Bandwidth Memory 3) configuration.
  It sets up the simulation environment and launches the corresponding workloads.

- **make_build.bat**
  A script for initial project setup and compilation.
  This is typically used for the first-time build or when setting up a new environment.

- **trace_generator.py**
  A Python script designed to generate synthetic memory access traces for testing and validation purposes.
  It can create customized read/write patterns to feed into the simulation framework.

Statistical Reporting

Ramulator2 automatically collects and reports statistics from all components at the end of simulation. This is handled through the `finalize()` and `print_stats()` methods in both the frontend and memory system.

Workflow Overview

1. When the simulation completes, Ramulator2 calls:

   ```
   frontend->finalize();
   memory_system->finalize();
   ```

2. The `finalize()` method performs:
   - Finalization of all subcomponents
   - Creation of a YAML emitter
   - Invocation of the `print_stats()` method

3. The `print_stats()` method recursively traverses the component tree and reports:
    - Interface name
    - Implementation class name
    - Unique instance ID (if any)
    - All registered statistics for the current object
    - Statistics from all subcomponents
4. The final output is a structured YAML-formatted report printed to `stdout`.

Adding Custom Statistics

To add custom metrics in a component, use:

```
register_stat(variable_name).name("output_name");
```

This will track the variable during simulation and include it in the final statistics report under the specified name.

DRAM Controller Tick Execution Logic

Each simulation cycle (`tick()`), the DRAM controller performs a series of operations to manage memory requests and maintain DRAM timing behavior. The core steps are as follows:

1. **Process completed read requests** Check the pending queue for completed read requests. If their departure time has been reached (`depart_time <= m_clk`), finalize the request and notify the frontend.
2. **Perform DRAM refresh** Issue periodic refresh commands to prevent data loss caused by charge leakage in DRAM cells.
3. **Select an executable request** Search for an executable memory request in the read, write, or priority buffers, based on timing constraints and scheduling policies.
4. **Manage row policy** Update the row buffer state by evaluating whether to close an open row or pre-open a new one, based on the current row management policy (e.g., open-row or close-row).
5. **Update plugins** Allow controller plugins to inject logic into each cycle, such as simulating additional delays or adjusting scheduling behavior.
6. **Issue DRAM command** If a valid request is selected, issue the appropriate DRAM command (e.g., ACTIVATE, READ, WRITE) according to the DRAM protocol.
7. **Handle request completion** If the current command is the final one needed for the request, set the request's departure timestamp and move it to the pending queue for eventual completion.

This tick-based execution framework allows Ramulator2 to accurately simulate DRAM behavior cycle-by-cycle, while supporting extensible logic through plugins and row policies.

Optional ECC/EDC Statistics and Formula Reference

This section provides core formulas and variable definitions related to ECC/EDC memory partitioning and access latency. These formulas are not computed by default, but can be used as optional statistical metrics if registered manually via `register_stat()` in your plugin implementation.

Memory Space Allocation Formula

To estimate ECC/EDC overhead and available memory capacity, use the following:

Formula:

```
ECC Size / EDC Size / Memory Data Block Size
```

ECC Partition Size:

```
ECC Region Size = Total Memory Size × (ECC Size) / (ECC Size + EDC Size + Target
Protected Data Block Size)
```

EDC Partition Size:

```
EDC Region Size = Total Memory Size × (EDC Size) / (ECC Size + EDC Size + Target
Protected Data Block Size)
```

Available Memory:

Assuming HBM uses a 1024-bit bus (128B per clock), the effective usable memory is:

```
Available Memory = Total Memory – ECC Region Size – EDC Region Size
```

Latency Calculation Formula

To estimate the latency introduced by ECC/EDC processing, you can define the following metrics:

Formula:

```
External Bus Type / Memory Bus Type / ECC Size / ECC Type / EDC Size / EDC Type /
Memory Data Block Size
```

Raw Read Latency:

```
T_raw-read = T_transmit + T_mem-read
```

ECC/EDC Effective Read Latency:

- **EDC Passes (No Errors)**

```
T_ECC/EDC-read = T_transmit + T_EDC-read + T_EDC-check + T_mem-read
```

- **EDC Fails, ECC Corrects Successfully**

```
T_ECC/EDC-read = T_transmit + T_EDC-read + T_EDC-check + T_ECC-read + T_ECC-
check + T_mem-read
```

- **EDC Fails, ECC Also Fails (UE)** For unrecoverable errors (UEs), total time includes full reprocessing:

```
T_transmit + T_EDC-read + T_EDC-check + T_ECC-read + T_ECC-check + T_ECC-
read + T_ECC-check + T_mem-read
```

- **EDC fails, ECC fails again**: If both EDC and ECC fail, the controller raises an unrecoverable error (UE) to the processor.

```
T_ECC/EDC-read = ∞   // Data cannot be read, triggers exception handling
```

- **Total ECC/EDC Effective Read Latency** A weighted average across all ECC/EDC outcomes:

```
T_ECC/EDC-read-total =
  P_EDC-pass × T_EDC-pass
+ P_EDC-fail, ECC-pass × T_EDC-fail, ECC-pass
+ P_EDC-fail, ECC-fail1, ECC-pass × T_EDC-fail, ECC-fail1, ECC-pass
+ P_EDC-fail, ECC-fail1, ECC-fail2 × T_EDC-fail, ECC-fail1, ECC-fail2
```

Raw Write Latency

```
T_raw-write = T_transmit + T_mem-write
```

ECC/EDC Effective Write Latency

```
T_ECC/EDC-write = T_transmit + T_EDC-calc + T_EDC-write + T_ECC-calc + T_ECC-write
+ T_mem-write
```

Component Time Calculations

You can use the following equations to derive the timing of various operations:

Bus Transmission Time

```
T_transmit = Data Block Size / BW_bus
```

Memory Read Time

```
T_mem-read = Data Block Size / BW_mem
```

Memory Write Time

```
T_mem-write = Data Block Size / BW_mem
```

EDC Read Time

```
T_EDC-read = EDC Size / BW_mem
```

EDC Write (Storage) Time

```
T_EDC-write = EDC Size / BW_bus
```

EDC Check Time

```
T_EDC-check = α_EDC × log₂(EDC Size)
```

EDC Computation Time

```
T_EDC-calc = γ_EDC × Data Block Size
```

ECC Read Time

```
T_ECC-read = ECC Code Size / BW_bus
```

ECC Write (Storage) Time

```
T_ECC-write = ECC Code Size / BW_bus
```

ECC Check Time

```
T_ECC-check = α_ECC × log₂(ECC Code Size)
```

ECC Computation Time

```
T_ECC-calc = β_ECC × Data Block Size
```

Bandwidth Calculation Formulas

Raw Read Bandwidth

```
BW_raw = Data Amount / (T_transmit + T_mem-read)
```

or equivalently:

```
BW_raw = Total Data Amount / Total Transmission Time
       = HBM Frequency × Number of Channels × Total Bus Width
```

ECC/EDC Effective Read Bandwidth

- **EDC Passes (No Errors)**

  ```
  BW_ECC/EDC-read = D_ECC / (T_transmit + T_EDC-read + T_EDC-check + T_mem-
  read)
  ```

- **EDC Fails, ECC Corrects Successfully**

```
BW_ECC/EDC-read = D_ECC / (T_transmit + T_EDC-read + T_EDC-check + T_ECC-
read +   T_ECC-check + T_mem-read)
```

- **EDC Fails, ECC Also Fails but Secondary Retry Succeeds**

```
BW_ECC/EDC-read = D_ECC / (T_transmit + T_EDC-read + T_EDC-check + T_ECC-
read +   T_ECC-check + T_ECC-read + T_ECC-check + T_mem-read)
```

- **EDC Fails, ECC Also Fails Completely (Uncorrectable Error)**

```
BW_ECC/EDC-read = 0   (Data Invalid, Unable to Read)
```

- **Total ECC/EDC Effective Read Bandwidth (Weighted Average)**

```
BW_total =
    P_EDC-pass                × BW_EDC-pass
  + P_EDC-fail, ECC-pass      × BW_EDC-fail, ECC-pass
  + P_EDC-fail, ECC-fail1     × BW_EDC-fail, ECC-fail1, ECC-pass
```

Raw Write Bandwidth

```
BW_raw-write = Data Amount / (T_transmit + T_mem-write)
```

ECC/EDC Effective Write Bandwidth

```
BW_ECC/EDC-write = Data Written / (T_transmit + T_EDC-calc + T_EDC-write + T_ECC-
calc + T_ECC-write)
```

Error Rate and Correction Effectiveness Formulas

These formulas help evaluate the impact of ECC/EDC schemes on data reliability. Users may optionally compute these values based on error tracking during simulation.

Raw Bit Error Rate (Without ECC)

```
P_err, raw = Number of Erroneous Bits / Total Number of Bits
```

ECC Correction Probability (For t-bit Correctable Codes)

If an ECC code can correct up to `t` errors in a codeword of size `n`, then the correction probability is:

```
P_corr = ∑ (i = 0 to t) [ C(n, i) × (P_err, raw)^i × (1 - P_err, raw)^(n - i) ]
```

Data Error Rate With ECC/EDC

```
P_err, ECC = P_err, raw × (1 - P_corr)
```

Correction Rate

```
R_ECC = Number of Successfully Corrected Data / Total Data Amount
```

ECC Bit Error Rate Improvement Factor

```
Δ_BER = (P_err, no ECC/EDC – P_err, with ECC/EDC) / P_err, no ECC/EDC
```

Unit Cost Calculation Formulas

These formulas help evaluate the cost impact of ECC/EDC overhead on High Bandwidth Memory (HBM). They are particularly useful when analyzing the trade-off between reliability and cost-efficiency.

Raw HBM Cost

Cost per gigabyte of raw HBM memory:

```
C_HBM-raw = Total HBM Procurement Cost / HBM Total Capacity (GB)
```

Additional HBM Storage Cost (Considering ECC & EDC Overhead)

This formula reflects the effective cost increase after ECC/EDC reduces usable memory:

```
C_HBM-extra =
    HBM Total Capacity
  / [HBM Total Capacity × (1 – ECC Overhead – EDC Overhead)]
  × C_HBM-raw
```

Overhead

```
ECC Overhead = ECC size / (ECC + EDC + Data block size)
EDC Overhead = EDC size / (ECC + EDC + Data block size)
```

Trace File

To simulate realistic CPU memory access patterns, Ramulator2 supports reading trace files in a simple, extensible format. The trace parser is implemented in `trace.cpp` under the specific processor backend (`frontend/impl/processor`).

Format

Each line in the trace file follows the format:

```
<bubble_count> <load_addr> [store_addr]
```

- `bubble_count`: The number of pipeline bubbles (NOPs) before the instruction, representing delay in the pipeline.
- `load_addr`: The address accessed by a load instruction.
- `store_addr` *(optional)*: The address to be written if the instruction includes a store operation.

Usage

Create a trace file with one instruction per line using the above format. This allows you to simulate CPU-level memory behavior, including stalls, loads, and stores, and is especially useful for testing ECC behavior under realistic access sequences.

Metadata Design (Conceptual Only)

To further explore intelligent memory management and adaptive ECC strategies, we propose a conceptual metadata tagging system for each data block stored in memory. Although this design is not implemented in the current plugin, it outlines a possible direction for enhancing error protection and memory-level optimization based on data semantics.

Purpose

By attaching lightweight metadata to each data block, the memory controller can potentially make better-informed decisions about:

- ECC strength allocation
- Access scheduling
- Bandwidth prioritization

- Selective redundancy

This concept supports semantic-aware memory: treating data differently depending on its type, size, access pattern, and error sensitivity.

Proposed Metadata Fields

Each data block may carry the following 7-bit metadata (stored in a controller-managed metadata index area):

Field	Size	Categories
Data Type	1 bit	Model Weights (P0), Activations (P1), Index Tables (P0), Temp Buffers (P1), Static Params (P0), Batch Data (P1)
Access Pattern	2 bits	Sequential (P2), Random (P1), Bursty (P0)
Data Size	2 bits	Large+High Freq (P0), Small+High Freq (P1), Large+Low Freq (P2), Small+Low Freq (P3)
Criticality	1 bit	Mission-Critical (P0), Redundant (P1)
Error Sensitivity	1 bit	High-Reliability (P0), Error-Tolerant (P1)

> **Note**: This metadata system is a design concept only. It is not implemented in the current release, but may serve as a reference for future extensions or experimental research.

Dynamic ECC (Conceptual Only)

Dynamic ECC is a conceptual framework designed to allocate error correction strength dynamically based on the importance of each data block. The system determines the criticality and error tolerance requirements of the data through a multi-factor priority scoring mechanism, enabling appropriate ECC level assignment.

Input Variables

- **Maximum Priority Levels**: Defines the total number of priority tiers the system can allocate, controlling the granularity of classification.
- **Selectable ECC Sizes**: The strength (or size) of ECC is dynamically assigned based on the data block's priority level.
- **Tag Weights**: Weighted factors used in the scoring algorithm to balance the influence of each contributing variable.

Memory Controller Logic

- **Weighted Scoring Method**: A composite priority score is generated by integrating several variables. The formula is as follows:

$$\text{Priority Score} = w_1 \cdot \text{Access Frequency} + w_2 \cdot \text{Recency Factor} + w_3 \cdot \text{Criticality} + w_4 \cdot \text{Data Size} + w_5 \cdot \text{Error Sensitivity}$$

- **Priority Stratification**: The scores are normalized and mapped to predefined priority levels, which then guide ECC allocation decisions.

Example Scoring Table

Data Block	Access Count	Most Recent Access Time	Criticality	Data Size	Error Tolerance	Calculated Priority Score
A	5000	Recent High Frequency	High	Small	Low	0.9
B	20000	Past High Frequency	Medium	Large	Medium	0.8
C	300	Recent Low Frequency	Low	Medium	High	0.4

> **Note**: This model is conceptual and intended for illustration purposes only. It is not an implemented feature.

TODO & Functional Limitations Summary

Incomplete ECC Decoding Logic

The current `decodeECC()` function only implements decoding for Reed-Solomon (RS) codes. Other ECC types such as Hamming and BCH are not supported, and users are expected to integrate external libraries or algorithms. This limits the simulation of diverse ECC strategies.

Simplified Memory Structure Limits Accuracy

Ramulator2 does not simulate real physical memory or data buses; it instead relies on an abstract DRAM model that returns pre-defined latency values. Since instructions only carry addresses and request types (not actual data), ECC/EDC operations must maintain their own copies of data and parity codes. This simplification restricts accurate simulation of real ECC/EDC behavior, and it's recommended to introduce a more complete memory abstraction model to better support data integrity and realistic access pathways.

Missing DRAM Write Latency Modeling

The default DRAM model in Ramulator lacks an `m_write_latency` parameter, meaning write delays are not explicitly modeled. This omission limits the realism of ECC encoding on write operations. It is recommended to extend the DRAM model to support write latency and ensure it is synchronized with the ECC plugin. Full-system delay validation is also advised to maintain result accuracy.

Lack of Realistic ECC Codewords

Because memory read/write operations are effectively "pseudo-accesses" with no real data payloads, ECC/EDC checks rely solely on synthetically generated data and codes. This prevents accurate emulation of scenarios involving codeword corruption, original data reconstruction, and uncorrectable error (UE) signaling. Enhancing realism would require deeper data modeling or coupling with real memory state.

ECC/EDC Latency Not Modeled

Although the plugin defines parameters like `ECC_COMPUTE_PER_BYTE_NS` and `EDC_COMPUTE_PER_BYTE_NS`, these computation delays are not currently integrated into the simulation pipeline or used to affect request timing or queuing. These should be incorporated into the pipeline model, with optional dynamic load modeling to enhance accuracy.

ECC Retry and RAID Recovery Are Placeholder Logic

Currently, ECC correction failure only logs an error, without performing actual retry or RAID-based data recovery. These routines are placeholders and lack real memory access or recovery path simulation. Depending on the simulation scope, they should be either fully implemented or removed for clarity.

HBM Row Policy Compatibility Needs Improvement

Ramulator's current row policy logic provides limited support for HBM configurations. Some policies (e.g., open/closed row, refresh schemes) may lead to parameter mismatches or behavioral conflicts when used with the ECC plugin. Enhancements to the HBM3/HBM4 controller models are recommended to ensure compatibility and stable performance under different memory types.

Incomplete Performance Evaluation Methodology

Due to the number of uncontrolled variables and varying test conditions, it is difficult to make stable comparisons between ECC/EDC strategies in terms of bandwidth, latency, and correction capability. A unified "effective bandwidth" evaluation model should be designed, incorporating error rate, data size, computation delay, and other key metrics, to support consistent benchmarking and comparative studies.

Conclusion

The Ramulator2_ECC plugin provides a powerful and flexible simulation framework for exploring error correction strategies in high-bandwidth memory systems, particularly for AI and HPC workloads where memory reliability and bandwidth are critical. By supporting large-granularity ECC codewords, configurable error injection, and latency-aware modeling, the plugin is designed to help researchers deeply analyze the trade-offs between reliability, bandwidth, and performance—while also highlighting the benefits of large-block ECC designs.

The current version primarily serves as a research prototype and does not yet implement all planned features. It lays a solid foundation for future extensions, including: integration of more realistic memory structures, support for advanced ECC algorithms, more detailed latency modeling, and validation through hardware or RTL-level simulation.

However, Ramulator2's native architecture still has significant limitations when it comes to full ECC support. It cannot fully capture all critical error correction behaviors in real hardware. While this plugin partially compensates through simulation mechanisms, achieving comprehensive and accurate ECC behavior requires

trade-offs between implementation complexity and simulation fidelity. Users are advised to adapt or extend the plugin based on their specific research goals, with awareness of the modeling limitations and engineering overhead involved.