POLITECNICO DI TORINO

Master's Degree in Ingegneria Informatica

Master's Degree Thesis

Efficient Compaction and Compression Algorithms for Diagnostic Data from Tests of Memories Embedded in Automotive System-on-Chips

Supervisors

Prof. Paolo BERNARDI

Prof. Annachiara RUOSPO

Dr. Giorgio INSINGA

Candidate

Brendon MENDICINO

Accademic Year 2024/2025

Summary

The modern automotive industry makes great use of System-on-Chips (SoCs) in their products. One very relevant component present inside the SoCs is the integrated random access memory. These memories are characterized by a probability of failure graph, which can be expressed using the Bathtub Curve, this curve is subdivided into three sections, which represent the failure rate of the memories during its life cycle: the beginning of the curve is denominated "Infant Mortality", during which the probability for a memory to fail is very high, this probability will decrease over time settling to a constant rate of failure, the time span in which this value is constant is called of "Useful Life". At the end of this interval the failure rate will rise, the so-called "End of Life" of a memory, caused by the wear-out.

After the "packaging" of the SoC is carried out a process called "burn-in" is performed. This procedure consists of a certain number of reading and writing cycles on the memory, these cycles are repeated for a sufficient amount of time to bring the memories out of the "Infant Mortality" region. At the end of this process the SoCs that present some failures inside their memories, can either be repaired or either be discarded. During these procedures, having the ability to keep track of failures within the memory is very important. This operation must be performed entirely on the SoC, mainly driven by a timing advantage, because acting on the system externally will consume a significant amount of time. The stored faults will be used for chip repair strategies or for later data analysis.

The Sub-Line Aggregation and Coloring (SLAC) is an algorithm that is capable of performing on-chip computation, storing the data using a compression algorithm. At the end of the entire test-flow the data can be extracted, allowing further analysis. The algorithm stores the faults inside the memory using shape-like objects, these shapes are called slices. Every slice is an oriented-memory-interval which represent a set of cells where the memory failed. During the execution of a test the memory will be scanned in search of faults, on each new fault a previously created slice can be enlarged, or if none of them would fit the fault a new slice will be created instead. The memories will usually have failures along bit-lines or word-lines, due to physical defects, creating a compact distribution of faults.

The main topic of this thesis will be an extension of the SLAC algorithm called

Delta-SLAC algorithm. The aim is to reduce the memory occupation of the SLAC. The concept behind the idea of the algorithm follows an empirical observation about memory failures, some failures arise due to physical defects inside the design, those bits remain stuck in their state which can be a logical $'0'$ or $'1'$, these are called stuck-at faults. During a test-flow the memory can be written and read multiple times, during each read operation the memory-state will be saved. If stuck-at faults are present, their information will be repeated for every executed test. The Delta-SLAC algorithm aims to keep track of the differences in the memory-state from test to test, only storing the symmetric-difference between two different subsequent tests, allowing repeated information along the test-flow to be omitted from the final storage.

The first test that will be run, will be the base for the Delta-SLAC computations. For every subsequently executed test a comparison will be performed between the faults at the n^{th}-test and the base, the algorithm will store a set of faults called difference, these faults are the result of a XOR operation (base $\oplus$ faults$_{n-test}$ = difference). To extract again the original information for the n^{th}-test an off-line computation has to be conducted, a XOR operation has to be performed between the difference and the base (derived from the XOR properties: $a \oplus b = c \iff c \oplus b = a$).

In order for the algorithm to run on an embedded system, some data structures need to be put in place. The Delta-SLAC uses a similar concept to that of the SLAC slices, these structures are called spooky-slices. These like-slices only live into the RAM and will not be stored at the end of the test, they are only used to perform the XOR computation. To understand which fault needs to be considered as a difference, the Delta-SLAC uses another key-concept, the Next Expected Fault (NEF). For each fault read from the memory, a comparison will be performed against the NEF, if they are not equal the current fault will be marked as difference, if they are equal both the NEF and the faults will be advanced to the next in their sequence. The NEF sequence represents the order of arrival of the base faults.

The NEF sequence is extracted from the spooky-slices, recreating the order of arrival of the base faults. To speed up the search process the concepts of an Augmented-interval Tree have been used. Every spooky-slice has been mapped to its respective y-range: [word-line$_{start}$, word-line$_{end}$] to create the intervals to be inserted into the tree. When performing a search on the data structure the worst time possible will be bound by $\log(n)$, where n is the number of spooky-slices. The x-ranges can be considered as well to create the intervals, the algorithm can be fine-tuned depending on a tendency for the faults to appear more along word-lines or bit-lines, this is specific to each memory technology and fault model.

A memory is usually tested by writing specific patterns, different patterns will trigger different kinds of faults. Some of the most common ones are: writing the whole memory with logical $'0'$s (Solid Zeros), writing the memory with logical $'1'$s (Solid Ones), writing the memory cells with alternate logical $'0'$s and $'1'$s

(Checkerboard), ...

The Delta-SLAC achieves the maximum memory-efficiency by also keeping track of the different patterns that the memory has been written with. If these distinctions where not mode, more memory than necessary would be used. In fact some faults only appear when a certain pattern is written, while others don't. In order to obviate this problem, the first two patterns will be a "Solid Zeros" and a "Solid Ones", called the set-up patterns, these will result in two bases to compute the following deltas. The algorithm will also have separate structures to keep track of the faulty-zeros and faulty-ones, those two structures will be independent of each other. Upon the reading of a specific written pattern, the NEF sequence will be modified accordingly, filtering out the faults that should not be present in the current pattern.

Some statistical analysis has been performed to verify the efficiency of the algorithm, the used data comes from a real production environment. The algorithm was tested on real SoCs, where the faults, coming from previously registered test-flows, where used to perform the analysis and the validation of the algorithm. A set of two subsequent tests where used, where the memory was set with Solid Zeros pattern, an injection of the faults was performed, and the resulting memory occupations where computed both for SLAC and Delta-SLAC. The results clearly show a clear advantage over the usage of a plain SLAC, with peak memory saving greater than 95 %.

Acknowledgements

Desidero esprimere la mia più profonda gratitudine ai miei relatori, Annachiara, Giorgio e Paolo, per la loro guida inestimabile, il supporto e i preziosi consigli forniti durante tutta questa ricerca. La loro esperienza e il loro incoraggiamento sono stati fondamentali per la realizzazione di questo lavoro.

Estendo il mio apprezzamento ai miei colleghi in Infineon, le cui discussioni e la condivisione delle conoscenze hanno arricchito enormemente la mia comprensione dell'argomento.

Sono inoltre grato ai miei amici e alla mia famiglia per il loro costante supporto e la pazienza dimostrata durante questo percorso. Il loro incoraggiamento è stato una fonte continua di motivazione.

In particolore ringrazio Flavia per essermi rimasta accanto lungo tutto questo percorso, senza mai aver dubitato di me, e sempre pronta a supportarmi. Grazie.

$$\iiint_{\Omega} \nabla \cdot \mathbf{F} \, dV = \oiint_{\partial\Omega} \mathbf{F} \cdot \hat{\mathbf{n}} \, dS$$

Table of Contents

List of Tables

List of Figures

Acronyms

SoC	System on Chip
SLAC	Sub-Line Aggregation and Coloring
FoC	Fist of Chain
FoCE	First of Chain Element
FoCL	Fist of Chain List
FoCEL	First of Chain Element List
Δ-SLAC	Delta-SLAC
NEF	Next Expected Fault
RRAM	Resistive Random-Access Memory
eMemory	Embedded Memory

Chapter 1

Introduction

Embedded memories are a relevant sector of interest for the automotive industry. Modern cars contain numerous System-on-Chips (SoCs), where memories are present for a very large part of the die area, assessing the reliability of such memories for a SoC is critical, failures could result in a device being discarded or repaired in the best-case scenario, or could lead to malfunction when inside a vehicle.

To properly assess the reliability of the memories, the automotive SoC manufacturers have put in place a series of actions that must be performed sequentially on a device, called test-flow. There are many kinds of test-flows, each designed to cover a specific range of life of the SoC. A memory, like every other production component, is affected by a failure rate throughout its lifetime. This failure rate can be roughly expressed using the "Bathtub Curve", shown in Figure 1.1. This curve is made up of three basic components: the beginning of the curve is driven by "Infant Mortality" failures, where a newly made memory has a very high probability to fail. The failure rate decreases over time settling to a constant value. The central section of the curve is called the "Useful Life" of a memory, where the failure rate is driven by randomness, this represents the time span where the memory is usable. The end section of the curve is called the "End of Life" of a memory, in this region the failure rate starts to rise, the curve will increase exponentially the more time it passes, the failures in this region are caused by the wear-out of the component.

Before an SoC can be used, manufacturers are required to perform thorough testing to evaluate the reliability of their products, especially in safety-critical settings such as automotive applications. Among the many test-flows applied to the memories, "Front-end testing", "Burn-in" and "Qualification" are some of the most crucial.

After silicon printing, the wafer will be tested for physical defects in the memory and basic operational functionality of the die, excluding dies that are either non-repairable or malfunctioning. This phase is called "Front-end testing". All dies that passed "Front-end testing" will undergo its last steps of factory fabrication,

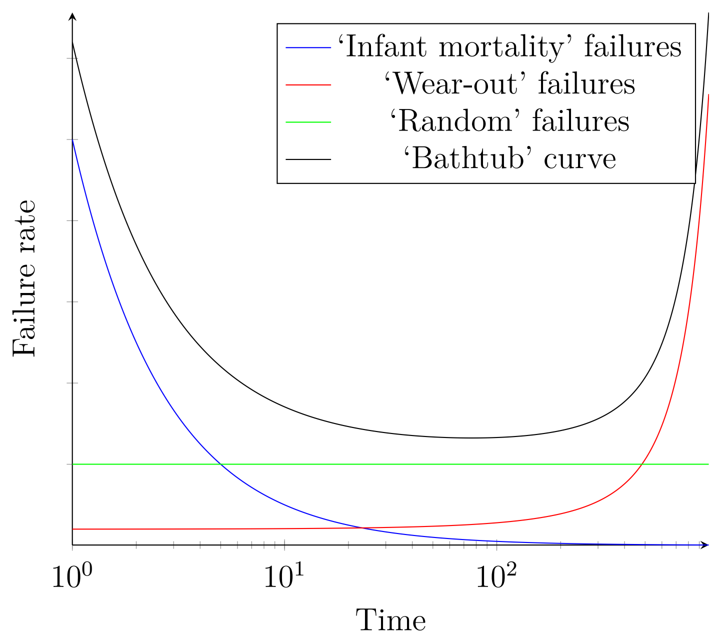

Figure 1.1: 'Bathtub' curve

the "packaging" is the encapsulation of the silicon die within a protective shield; after this procedure, a chip will be usable inside a board, whether for testing or employment. The early stage of every SoC is affected by the "Infant Mortality" failures, at this point an SoC is not deployable because the risk of failure is not tolerable inside safety-critical systems. Every chip will endure the burn-in phase, consisting of a series of reading and writing cycles on the memory. The writing and reading cycles are alternated, also specific patterns are used when writing the memory, some of the most common ones are: writing the whole memory with logical $'0'$s (Solid Zeros), writing the memory with logical $'1'$s (Solid Ones), writing the memory cells with alternate logical $'0'$s and $'1'$s (Checkerboard). The utilization of varied patterns is attributed to the phenomenon in which certain failures manifest exclusively when the physical memory cells are in designated states. The "cycling" is carried out until the SoC life span is out the "Infant Mortality" region. All devices that present some failures will be discarded or repaired.

After the "Burn-in" is performed, a small subset of devices is selected. A series of cycling operations and common actions performed by end-users are all tested in varying conditions, like temperature, sense amplifier margins, drive voltage, ... If the SoC passes those tests without any failure, the time range of the "Useful Life" is validated, this process is called "Qualification".

During memory tests, having the ability to keep track of failures within the memory is very important. Thanks to the memory state history, it's possible to decide if a failed memory can be repairable and which strategy a repair step can be performed with, enables later data analysis for statistical studies, which permits a further understanding of the technology and eventual design problems.

1.1 Thesis work

It's clear that some strategy for data storage of the memory failure state needs to be established. All data storage and computation must be performed on-line inside the SoC test-program while the test-flow is being executed. The test-program is an embedded software that will be injected inside the SoC during the test operations, which will be responsible for the execution of the test-flow. Utilizing a communication interface or information exchange mechanism between the test-program and external software would have an excessively detrimental impact on timing performance.

This thesis will focus on such fault encoding strategy, giving a novel and effective approach to tackle this challenging task. All the operation will be performed on a streaming data of faults, with on-line capabilities, keeping the timing and storage memory as low as possible.

Chapter 2

Background

2.1 Embedded Memories

Non-volatile memories (NVMs) play a crucial role in modern computing systems, providing persistent data storage with high reliability. Among various NVM technologies, Resistive Random-Access Memory (RRAM) has emerged as a promising candidate due to its low power consumption, high scalability, and fast erasure characteristics. However, as memory densities increase, fault management and efficient storage strategies become essential to ensure a competitive advantage.

Fault storing techniques, such as the Sub-Line Aggregation and Coloring (SLAC) algorithm, provide a systematic approach to identifying and classifying faults within memory arrays. While effective, these methods can be resource-intensive, requiring significant storage to capture fault patterns over long-running test-flow operations. To address these limitations, an enhanced approach called Delta-SLAC has been introduced, optimizing memory utilization and improving fault storage encoding efficiency.

2.2 SLAC Algorithm

2.2.1 Overview

The Sub-Line Aggregation and Coloring (SLAC) [1] algorithm is designed to address the issues of memory-agnostic environments. The SLAC algorithm, part of an embedded firmware-level code, is able to compact large quantities of data by performing on-line computations while the memory is scanned for faults. All the intermediate steps of the computations will be stored in a volatile memory, thus enabling a faster access to the data and to avoid time loss by accessing the non-volatile memory. At the end of a test-flow the data stored in volatile memory can be dumped to a non-volatile memory, present in the SoC, retrieved for external sources and subsequently stored for further analysis. If the test is performed during “Front-end testing” or during “Burn-in”, a repair-step can replace broken bit-lines or word-lines of the memory array, the repair algorithm will exploit the information stored by the SLAC, without the need of external programs to intervene, allowing computations to be performed entirely on the SoC.

The SLAC algorithm represents the faults inside a memory using the concept of slices. A slice is a spatially orient shape, defining a region inside the memory where a set of faults is contained. A slice can be one of four different shapes, each shape is defined using colors:

- BLACK: represent a single failure;
- ORANGE: an orange slice can be a horizontally or vertically oriented shape, the interval range defined by the slice is a compact line of failures, without any gaps;
- RED: a red slice can be a horizontally or vertically oriented shape, the interval range defined by the slice is a scattered line of failures, every fault will alternate with a non-fault;
- BLUE: a blue slice can be horizontally or vertically oriented shape, the interval range defined by the slice contains only two faults, indicated by the start and the end of the range;

A bitmap can model the failures that took place inside a memory, where each cell in the matrix will be mapped to each cell in the memory. Every row of a matrix corresponds to a bit-line, and every column corresponds to a word-line, this identifies the physical layout of a cell-array, more accurate models will also take care of the physical disposition of the hierarchical division of the memory: banks, sectors, . . .

In Figure 2.1 a representation of memory failures is shown as a bitmap, where each black cell marks a failure. Given the layout of the memory failures, it’s possible to extrapolate the possible encoding using the slices, an example is show in Figure 2.2, where it’s possible to see how the failures are aggregated into compact shapes with an oriented interval range.

The idea behind the algorithm compaction methodology comes from the physical disposition of the memory, considering each bit-line or word-line as interconnected it is easy to see that if a bit-line (or word-line) breaks, then all the cells of that bit-line (or word-line) will be marked as a failure. Those failures can be represented as a single slice, allowing the storage of many faults inside a single interval range, enabling a massive reduction in memory consumption.

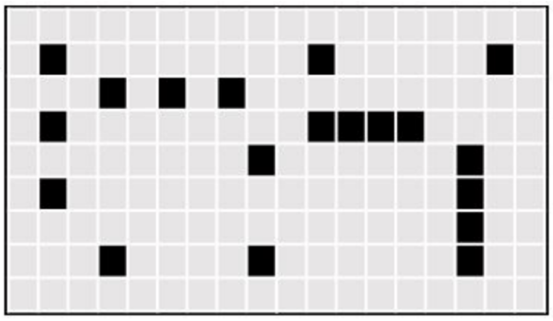

Figure 2.1: Bitmap where each black cell is a fault in the memory.

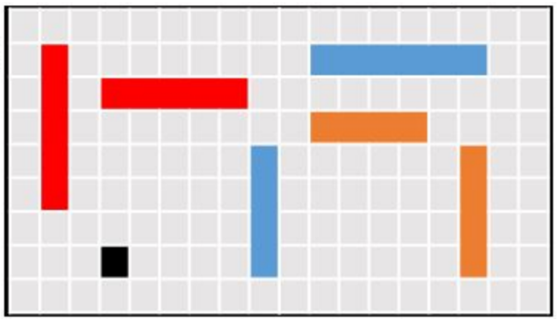

Figure 2.2: SLAC slices color encoded representation of Figure 2.1.

2.2.2 Data structures

Each slice is defined by a specific interval range, shared axis, and color. In order to implement a slice a C structure representation will be used, it shall be said that not all the structures will be identical to the real implementation, the structures shown in the Listing 2.1 are just a blueprint for how the data can to be structured, specific eMemories designs will require different changes in the size of fields, this depends on how many banks, sectors, bit-lines and word-lines are present in the eMemory.

Listing 2.1: C representation of a temporary slice (only present in RAM memory).

```c
enum SliceColor {
    COLOR_BLACK = 0x0,
    COLOR_BLUE = 0x1,
    COLOR_RED = 0x2,
    COLOR_ORANGE 0x3,
};

struct TempSlice {
    /* Linkt to the next slice. */
    struct TempSlice *next;
    enum SliceColor color;
    /* Start of the interval along bit-line/word-line. */
    int start_fail;
    /* End of the interval along bit-line/word-line. */
    int end_fail;
};
```

All the created slices will be kept in the RAM memory for temporary storage. The slices themselves do not contain any information on their location in the memory; in fact, the combination of bit-line and word-line only specify relatives offsets from the beginning of a sector, and not the entire memory, slices also have no information on the current direction they are facing; this information needs to be stored somewhere else, keeping it inside the slices would cause too much redundancy; in fact, each sector or shared axis might contain hundreds or thousands of slices. The structures that will group the slices of the same category are called First of Chain Element (FoCE). The FoCE, shown in the Listing 2.2, will contain information about: the current test ID (the reason is that slices can be kept in memory between different tests), the direction of the slices that the FoCE groups, the current bank, the current sector, a tag to retrieve the original value of the shared axis of a slice.

Listing 2.2: C representation of a First of Chain Element (only present in RAM memory).

```
enum SliceDirection {
    DIRECTION_HORIZONTAL = 0x0,
    DIRECTION_VERTICAL = 0x1,
};

struct FirstOfChainElement {
    /* Link to the next first of chain elment. */
    struct FirstOfChainElement *next;
    enum SliceDirection direction;
    int bank;
    int sector;
    int test_id;
    int tag;
    /* Link to the linked list of slices. */
    struct TempSlice *slice_head;
};
```

To enable a fast search among all the slices hash-map-like structures have been put in place. There are three levels of indirection, creating hierarchical access:

- Fist of Chain List (FoCL): a list of pointers to a linked list of First of Chain Element;

- First of Chain Element List (FoCEL): a list of First of Chain Element, where each element contains a pointer to a linked-list of slices;
- Slice Buffer (SB): a list of all the slices present in memory;

To access these structures a hashing-function is needed. In order to reduce at a minimum the usage of RAM memory, the first level of the hierarchy is designed as a matrix, to access this structure a tag and a set are needed (for the matrix rows and columns respectively). In order to computed set and tag, a parameter α is used as the input for the hash-function, where α is the shared axis of a slice, which can be a bit-line, for vertical slices, or a word-line, for horizontal slices.

Another important parameter is the number of sets, defined as β (a positive integer), used for the computation of the values of the tag and the set. This parameter can be fine-tuned to find the right amount of sets that will cause the least amount of conflicts.

$$\alpha = \begin{cases} \text{bit-line}, & \text{direction}_{\text{slice}} = \text{vertical} \\ \text{word-line}, & \text{direction}_{\text{slice}} = \text{horizontal} \end{cases}$$
$$\beta \in \mathbb{N}^+$$

$$\text{TAG} = \left\lfloor \frac{\alpha}{\beta} \right\rfloor, \quad \text{SET} \equiv \alpha \mod \beta \tag{2.1}$$

Once the set and tag are computed, it is possible to follow the chain links to find a slice. The structures that define the hierarchy are shown in the Listing 2.3.

Listing 2.3: C representation of the heirarchy lavels (only present in RAM memory).

```c
#define N_SETS    (16)
#define N_FOC     (1000)
#define N_FOCE    (2000)
#define N_SLICES  (2000)

/**
 * Matrix of pointer to the a `FirstOfChainElement`.
 * The matrix is accessed using the `tag` and `set`.
 */
struct FirstOfChainElement *first_of_chain[N_SETS][N_FOC];

/**
 * List of `FirstOfChainElement`s, this list act as an
 * "Object-pool" for the objects, where each object is
 * inside a linked list.
 */
struct FirstOfChainElemnt first_of_chain_elements[N_FOCE];

/**
 * List of `TempSlice`s, this list act as an
 * "Object-pool" for the objects, where each object is
 * inside a linked list.
 */
struct TempSlice slices[N_SETS][N_SLICES];
```

A graphical representation of how the structures are interconnected between each other is shown in Figure 2.3. Here it's possible to see how to the FoC List will contain a set of pointers, each pointing to the head of linked list of FoCE. The first level of hierarchy acts as a hash-table to quickly search an element; though, conflicts may arise due to the hash-function nature, causing two different elements with different keys to have the same value of the hash-function, ending up in the same cell, to solve this problem a linked list of elements is used, appending a new element to the tail of the list every time there is a new conflict. In order to find the correct element a simple comparison between the current section of the memory (bank and sector), and the current test ID is executed, once the comparison succeeds the element is found. With the desired element found, by iterating its linked slices, it will be possible to find any slice.

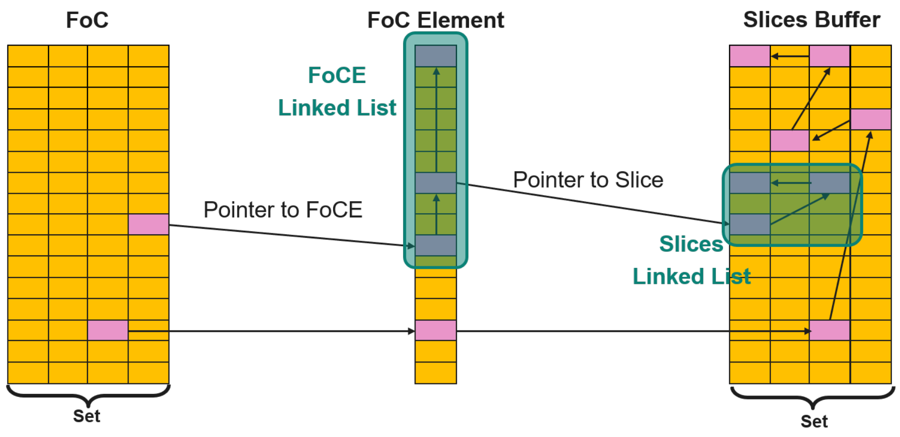

Figure 2.3: Graphical representation of the hierarchical structure of the SLAC data structures.

By using the Equation 2.1 to compute the set and tag, for every FoCE, all the linked slices will be on the same word-line (or bit-line depending on the direction). This is crucial in the workings of the algorithm when a new fault needs to be inserted inside the slices.

Once a test is terminated, slices can be dumped to some physical storage for later retrieval. The representation used for a RAM memory is not suitable for this purpose. The temporary representation contains information that are not relevant to the interpretation of the slices, like the hierarchical structure and pointer between the structures. All the overlaying structures containing the FoCE and the slices are similar to object-pools, if the whole structures would be dumped, then also the unallocated space will be present, thus wasting precious resources. To enable a greater reduction in the storage requirements a conversion needs to take place, only replacing all the temporary slices into a more compact representation.

Every slice can be represented as a sum-type of two different kinds of slices, a horizontal slice or a vertical slice. The difference relies on the bitmap representation, where each cell will have a coordinate (*bit-line*, *word-line*). To represent an orientated slice, three fields are needed: the shared axis, the start of the interval, and the end of the interval. It is clear that if the shared axis is a bit-line, the interval will range on the word-lines.

Listing 2.4: C representation of the slices to be stored in non-volatile memory.

```c
struct HorizonalSlice {
    int start_bitline;
    int end_bitline;
    int shared_wordline;
};

struct VerticalSlice {
    int start_wordline;
    int end_wordline;
    int shared_bitline;
};

/**
 * This is an expanded representation of a slice,
 * in reality both the `direction` and the `color`,
 * will be merged as a bit field inside the
 */
struct BasicSlice {
    /**
     * Discriminates whether to use the horizontal or
     * veritcal slice of the union.
     */
    enum SliceDirection direction;
    enum SliceColor color;
    union {
        struct HorizonalSlice horizonal;
        struct VeritcalSlice vertical;
    };
};
```

Like the temporary representation of slice (Listing 2.1), some information is missing; in order to fill this void a structure similar to the FoCE is used (Listing 2.2), the “Control Slice”. Every control slice contains the missing information (bank, sector, test ID), and an offset to the next control slice. A control slice structure can be shown in the Listing 2.5.

Listing 2.5: C representation of a control slice to be stored in non-volatile memory.

```c
struct ControlSlice {
    int bank;
    int sector;
    int test_id;
    int offset_to_next_ctrl_slice;
};
```

In the non-volatile memory, a contiguous array stores both basic slices and control slices. This arrangement is feasible due to the structure of the control slices, which contain an offset pointing to the next control slice in the array. This offset is based on the array’s indexes. As a result, all slices preceding a control slice in the array are guaranteed to be basic slices. Furthermore, all basic slices that come before the next control slice share the same characteristics as the control slice. This organization allows for efficient storage and management of slices within the non-volatile memory.

Listing 2.6: C representation of the layout of the slices in the non-volatile memory.

```c
#define MAX_SLICES (1000)

/**
 * A slice can be thought of a sum-type between a basic
 * slice and a control slice.
 */
union Slice {
    struct ControlSlices ctrl;
    struct BasicSlice basic;
};

/**
 * This array will be stored in the non-volatile memory,
 * each slice in the array can be a basic or a control
 * slice, to differentiate one from the other, the first
 * slice will always be a control slice, then the control
 * slice contains an offset to the next control slice,
 * thus guaranteeing all the slices before the next
 * control to be basic slices.
 */
union Slice dumped_slices[MAX_SLICES];
```

The implementation of the array of slice in the non-volatile memory is shown in the Listing 2.6. In order to guarantee a consistency between the two structures living in the same array, the first slice will always be a control slice, thus allowing the determination of all the other slices.

2.2.3 Coloring

During the execution of a test there are two main phases, the writing of the memory, and the reading of memory. When reading back the content of the eMemory the expected value coincides with the previously written pattern, when data deviates from the expectation, that cell will be marked as a fault. Faults are caused by a multitude of factors, the most common source are physical defects in an eMemory design.

During the execution of a test an eMemory is read with an alternate pattern, the process is completed when all the single cells have been iterated. By looking at the eMemory cell array as a matrix composed of bit-lines and word-lines (as columns and rows respectively), two zones are defined inside this matrix, which are separately the white and black cells of a checkerboard; these two different zones will be defined as “zone-A”, for the white cells, and the “zone-B” for the black cells; for convention the first cell in uppermost left corner will be a white cell, which is part of the zone-A. During the reading operation, an eMemory will be read by iterating over the banks first, and for each bank it will be read by iterating over the sectors of a bank. Each sector will be divided into zone-A and zone-B. The reading process of a sector begins with reading zone-A, and once that is complete, zone-B is then read. This double reading operation, in practice, is translated as a sequential reading of the word-lines; staring from the beginning of sector, for each word-line the even cells are read first, and then the odd ones are read, alternating this operation on every word-line for the zone-A. Once zone-A has been completed by reaching the last word-line of a sector, the reading is rewound from the first word-line back again, at this point the odd cells are read first, and then the even ones are read, alternating from the word-line to word-line, reading the cells belonging to the zone-B region. Once all the sectors have been read with the zone-A and zone-B patterns, and once all the banks have been read, the reading operation is terminated.

All faults found will be passed to the SLAC, which will take care of the compression with on-line computations. By looking at the way eMemories are read, and understanding how physical defects can cause failures in aligned way, it is now more clear the design choice of the red and orange slices shape, in many cases some set of faults will lay very close to each other in vertical or horizontal orientation. The orange slices will address this issue very well, by being able to compress a lot of faults when they are near each other, that lay contiguously on a straight line.

On the other hand red slices are designed to compensate the limitation of how a memory is read, in fact, if red slices where not used, it would have not been possible to aggregate data while reading zone-A first and then zone-B, e.g. imagine the multiplexer for a word-line broke down, causing all the cells for the first word-line to appear as faulty during the reading operation, the zone-A will be read first and then zone-B, but when reading zone-A an alternate series of faults will be seen, without a red slice, these faults can only be represented as a black slice or a blue slice, it will only be possible to create an orange slice that will cover the entire word-line when the iteration reaches the zone-B, in fact after the zone-B is read the first word-line will appear as a contiguous line of faults, thus being able to be represented as an orange slice. The issue with this approach is that by the time the sector iteration reaches the zone-B, a very large quantity of slices can accumulate in the meantime, in some cases the available storage may not be enough to store slices before reaching the end of the zone-A, causing a loss of information, if on the other hand, the red slices are used, the first word-line during the reading of the zone-A can still be represented as a unique slice, and later compacted to an orange slice when the zone-B will be read.

When the SLAC receives the streaming of faults, it has to act on them with on-line computation in order to store the faults inside the slices. Upon each new fault three actions can be performed: expand a previous previously created slice, split a previously created slice, create a new slice if none of the others can fit the fault.

As shown in the Algorithm 1 it's possible to see a high-level representation of the algorithm to store each faults into a set of slices. This is process in the end is very fast, by finding an appropriate number for the number of sets that the structures will be divided into, will create equally divisions among the data, reducing the linear iteration time for each of current linked list being iterated (Slices Buffer or First of Chain Element List).

Algorithm 1 High level algorithm for processing the faults and creating the set of slices.

```
1: procedure ProcessFaults($Faults, Test_i$)
2:     ▷ Iterate over all the faults where $Fault_k \in WordLine_k$
3:     for $Fault_k, |WordLine_k| \leftarrow Faults$ do
4:         $Direction \leftarrow \begin{cases} Horizontal, & |WordLine_k| > 10 \\ Vertical, & |WordLine_k| \leq 10 \end{cases}$
5:         $Set, Tag \leftarrow H(Fault_k)$      ▷ Get the $Set, Tag$ from the current $Fault_k$.
6:         $Bank, Sector \leftarrow Position(Fault_k)$
7:         $FoCEList \leftarrow FoCList[Set][Tag]$
8:         $ElementParams \leftarrow (Set, Tag, Bank, Sector, Direction)$
9:         ▷ Get the current FoCE.
10:        if $containsElement(FoCEList, ElementParams)$ then
11:            $FoCE \leftarrow getElement(FoCEList, ElementParams)$
12:        else
13:            $FoCE \leftarrow newElement(FoCEList, ElementParams)$
14:        end if
15:        $Slices \leftarrow FoCE$
16:        if $splitable(Slices, Fault_k)$ then
17:            $split(Slices, Fault_k)$
18:        end if
19:        ▷ Try and extend one of the previous slices or create a new one.
20:        if $extendable(Slices, Fault_k)$ then
21:            $extend(Slices, Fault_k)$
22:        else
23:            $Slice \leftarrow blackSlice(Fault_k)$
24:            $newSlice(Slices, Slices)$
25:        end if
26:    end for
27: end procedure
```

Chapter 3

Delta-SLAC Algorithm

The Delta-SLAC (Δ-SLAC) is an algorithm which aims to reduce the memory occupation of the SLAC algorithm. The Delta-SLAC can be considered as an extension of the SLAC algorithm, that can be enabled or disabled at any time. This reduction in memory usage is accomplished by exploiting the amount of reading actions performed during the operation of a test-flow, the algorithm will only store the difference between the set of faults read; as a consequence all the stuck-at faults will be omitted in the final representation of the various bitmap, this reduction in memory usage starts to become really impactful when the number of reading actions inside a test-flow is of a considerable amount.

3.1 Overview

The Delta-SLAC algorithm is an extension of the SLAC algorithm. The Delta-SLAC is designed to reduce the memory foot-print of the SLAC, over a span of multiple reading operations inside a test-flow. The Delta-SLAC is able to perform this task by computing only the set of difference between two different tests; this is difference is equal to the bit-wise XOR operation between two matrices. Each of the two matrix will represent the bitmap of failures of the eMemory, and the bit-wise XOR will be computed cell by cell. The resulting bitmap of this computation, will contain only the set of faults which appeared or disappeared by comparing the two previous matrices.

This difference computation is performed by exploiting a set of helper structures that the Delta-SLAC creates in the RAM memory. These structures will not be dumped to any persistent storage after the end of the operation, they are only created at run-time and then disposed of when the test-flow is completed. This has the benefit of withholding any extra information in the storing process of the SLAC slices, in fact, at the end of a test-flow when the slices from the SLAC will be retrieved by an external test-program, the fault information will remain encoded in the SLAC colored slices, even though the Delta-SLAC was enabled. The final result is not going to be the real bitmap of failures that where recorded during the reading operation, but it will be a set of differences, with the benefit of having the same encoding used for the SLAC.

When the Delta-SLAC is enabled, there are three steps that take place during the execution of a test-flow, during which, every fault will follow the path illustrated in Figure 3.1.

1. **Initialization phase**: the first phase consists of the initialization of the spooky slices for the Delta-SLAC, and the creation of the first colored slices for the SLAC;

2. **Difference computation phase**: the spooky slices are used to check if a fault was already present, if the fault is marked as a difference then it will be sent to the SLAC, otherwise it will be blocked;

3. **SLAC colored slices retrieval phase**: at the end of a test-flow (or between each individual test) there is the ability to retrieve the colored slices of the SLAC;

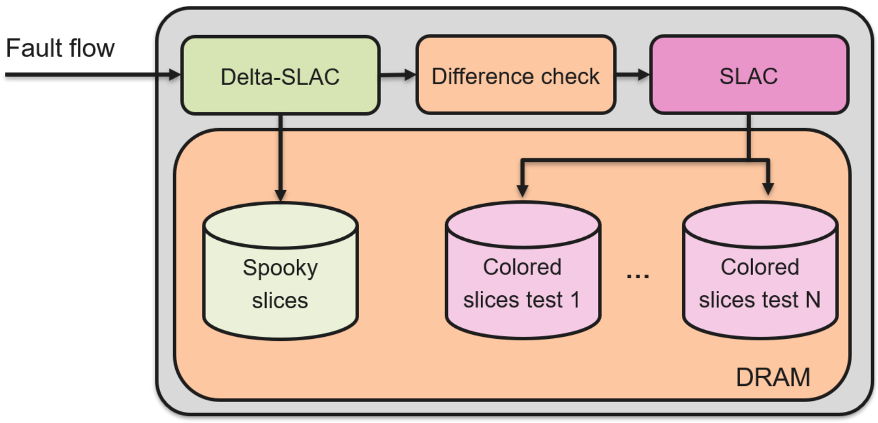

Figure 3.1: SLAC and Delta-SLAC structures in RAM memory and flow of the fault encoding through the algorithms.

During the initialization phase both the SLAC and Delta-SLAC will run in parallel, creating the spooky slices and the colored slices for the first test respectively. The colored slices and the spooky slices have a similar encoding, and they are both designed to store effectively faults on contiguous lines, for the first test, if decoded, their relative fault bitmaps will be equivalent. These relative fault bitmap of the first test will be called “base faults”, the base faults will be the ones where the differences are computed upon. This will have as a consequence that the very first test performed will always have the same output, whether the Delta-SLAC is enabled or not. Only from the second test onwards the differences will be computed.

After the initialization phase, the difference checking phase starts. In this phase the already created spooky slices, will be exploited using fault checking mechanism in order to distinguish which of new incoming faults need to be marked as difference, and which don’t. The Delta-SLAC simply acts as a wall between the incoming faults and the normal SLAC, by deciding which faults can pass through; the faults that will reach the SLAC and that will be compacted into the colored slices will be called the “difference faults” (“Δ faults”), and these encoded faults will not match the real condition of the memory, but they are just a byproduct of the algorithm. In order to retrieve the original faults, some off-line post-processing needs to be performed.

After the test-flow has been fully completed, it will be possible to retrieve the slices, dumped into a local storage of the SoC, thanks to an external test-program.

3.2 Spooky slices

The spooky slices are the fault encoding mechanism that the Delta-SLAC uses to store information. This information will only be kept in the RAM memory and their only usage is for the on-line difference computation.

The spooky slices have a similar structure to the basic slices of the SLAC, in fact both those structures contain information about the interval range of the faults, and information about the current, color, shared axis and interval range. The main difference in their blueprint choices is that the spooky slices are designed to keep track of the order of arrival of the faults, while the basic slices are designed to keep track of the physical layout of the faults.

Listing 3.1: Spooky slices structre in C code.

```
#include <stdint.h>

typdef struct {
    /**
     * 15:    is_creating_slice
     * 14:    unused
     * 13-0:  corrdinates
     */
    uint16_t first_fail;

    /**
     * 15-13: slice_color
     * 13-0:  coordinates
     */
    uint16_t last_fail;

    /**
     * 15:    slice_direction
     * 14:    unused
     * 13-0: corrdinates
     */
    uint16_t shared_coordinate;

    /* Used for the fast search of slices. */
    uint16_t max_high;
} SpookySlice;
```

The spooky slices have two representations in the C code implementation, the compressed one shown in the Listing 3.1 is the one stored in the data structures, while the one shown in the Listing 3.2 is used when manipulating the structure for better ease of use. One of the main limitations of working in an embedded environment is the limited space available in memory that is usable for storing data, by using the compressed representation the structure will only occupy 8 B of space. By using the direction field as a discriminant it will be possible to distinguish if the shared coordinate will be a bit-line or a word-line.

Listing 3.2: Extended version of a spooky slice only used for the data manipultaion

```
#include <stdint.h>

typdef enum {
    SPOOKY_BLACK = 0x0,
    SPOOKY_BLUE = 0x1,
    SPOOKY_RED = 0x2,
    SPOOKY_ORANGE = 0x4,
} SpookySliceColor;

typedef enum {
    SPOOKY_HORIZONTAL = 0x0,
    SPOOKY_VERTICAL = 0x1,
} SpookySliceDirection;

typedef struct {
    bool is_creating;
    SpookySliceColor color;
    SpookySliceDirection direction;
    uint16_t first_fail;
    uint16_t last_fail;
    uint16_t shared_coordinate;
    uint16_t max_high;
} SpookySliceExtended;
```

The spooky slices are created during the initialization phase of the Delta-SLAC, where are all the faults will be read and then processed. To be able to understand the physical distribution of the eMemory failures, the faults are buffered in a three word-lines window. The algorithm will analyze the window and deciding based on some parameters the direction in which the slices will be created.

This buffering mechanism is need, if it where not present the direction in which the slices are initially created would be probably wrong. In order to take better guesses on the general distribution of the faults in eMemory, the buffer will be scanned by columns and then by rows (bit-lines and word-lines respectively), for each fault a decision will be taken based on the number of subsequent vertical or horizontal faults appearing, after the current one. To take this decisions two thresholds are defined, all the faults matching the first threshold will be removed from the buffer and passed down as oriented faults to the insertion function. This mechanism can be better shown in Listing 3.3.

Listing 3.3: C code implementation of the word-line window buffere is iterated in order to create orient spooky slices.

```c
/**
 * @brief Dumps all the faults present in the word-lines buffer
 * to the spooky slices.
 *
 * @param buffer
 */
static void dump_word_lines_buffer(WordlinesBuffer *buffer,
    SpookySlices *spooky_slices)
{
  uint16 row = 0;
  uint16 col = 0;

  for (row = 0; row < SPOOKY_WL_BUFFER_ROWS; row++)
  {
    for (col = 0; col < SPOOKY_WL_BUFFER_COLS; col++)
    {
      if (!bitmap_matrix_is_set(buffer->buffer, row, col))
        continue;

      /**
       * If from the current fault there are at
       * least n consecutive faults that are graeter
       * than the threashold, then we try to
       * create an horizontal slice instead of
       * a vertical one.
       *
       */

      uint32 consecutive_col_faults = consecutive_faults_on_word_line(buffer->buffer, row, col);
      uint32 consecutive_row_faults = consecutive_faults_on_bit_line(buffer->buffer, row, col);

      /* Try to create an horizontal slice */
      if (consecutive_col_faults >= MIN_FAILURES_FOR_H_SLICE)
      {
        wl_buffer_reset_horintal_slice(buffer, spooky_slices, row, col, consecutive_col_faults);
      }
      /* Try to create a vertical slice */
      else if (consecutive_row_faults == SPOOKY_WL_BUFFER_ROWS)
      {
        wl_buffer_reset_vertical_slice(buffer, spooky_slices, row, col, consecutive_row_faults);
      }
      /* Just pass the single fault */
      else
      {
        const Mcal_Shared_Coordinates_t fault = (Coordinates){
          .x = col,
          .y = row + buffer->y_base,
        };

        createOrUpdateSpookySlice(spooky_slices, fault);

        bitmap_matrix_reset(buffer->buffer, row, col);
      }
    }
  }
}
```

The sliding window over the word-lines needs to be kept updated during the iteration of the eMemory faults, this can be performed using a simple check to see if the current fault appears after the window, if that is the case the old buffer is dumped and the window is advanced. The implementation of such logic is shown in the Listing 3.4, here it's possible to see as well how the spooky slices structures are used, but those concepts will be explained in the next sections.

Listing 3.4: C code implementation of the sliding word-line window buffer.

```c
/**
 * @brief Saves a new fault inside the spooky slices.
 *
 * IMPORTANT: it should be remembered that faults need
 * to be inserted with increasing `x`s on the same line
 * and with increasing `y`s otherwise, compared to the
 * last fault saved. If this constraint
 * is not satisfied the the algorithm will not work
 * correctly.
 *
 * @param context delta slac context
 * @param new_fault new fault to save
 * @param bank bank where the fault appeared
 */
void dslac_save_fault(DeltaSlacContext *context, const Coordinates new_fault, const CPLXDRV_ITERATOR_BANK_T bank)
{
    SpookySlicesBuckets *slices;

    if (context->verify_pattern == ZERO_PATTERN_TYPE)
        slices = &context->zero_slices;
    else if (context->verify_pattern == ONE_PATTERN_TYPE)
        slices = &context->one_slices;
    else
        return;

    const uint16 new_y_base = new_fault.y - (new_fault.y % SPOOKY_WL_BUFFER_ROWS);

    /* Check if the current buffer needs to be dumped */
    if (new_y_base != context->word_lines_buffer.y_base || bank != context->word_lines_buffer.bank)
    {
        const CPLXDRV_ITERATOR_BANK_T buffer_bank = context->word_lines_buffer.bank;
        SpookySlices *bucket = &slices->buckets[buffer_bank];
        const uint32 old_size = bucket->size;

        dump_word_lines_buffer(&context->word_lines_buffer, bucket);

        /* Fix the size of the base. */
        slices->base.size += bucket->size - old_size;

        /* The bank are visited in order. */
        if (bank != buffer_bank)
        {
            /* Fix the old capacity, so that there can be no push in the array. */
            /* This should be redudant, but you never know! */
            bucket->capacity = bucket->size;
            SpookySlices *new_bucket = &slices->buckets[bank];

            spooky_slices_init_bucket(new_bucket, &slices->base);
        }

        context->word_lines_buffer.y_base = new_y_base;
        context->word_lines_buffer.bank = bank;
    }

    bitmap_matrix_set(context->word_lines_buffer.buffer, new_fault.y
  % SPOOKY_WL_BUFFER_ROWS, new_fault.x);
}
```

The spooky slices are organized in buckets, each bucket is assigned to a different zone of the eMemory. When the eMemory is iterated, the current bucket has to be taken into consideration, and the dumped faults will end up in the chosen bucket.

3.3 Next Expected Fault

In order to compute the difference between two set of faults a special mechanism is required. The most simple algorithm to produce such result, would be to have the matrix of failures, and perform a bit-wise XOR between the two of them, the output of the operation will be exactly the set of differences between two different reading operations of the eMemory. This is not possible due to the restricted memory environment present in embedded systems, in such conditions it is not possible to store an entire bitmap of the eMemory in the RAM memory. A more refined approach is needed. A solution to this problem would be to have a sequence of faults, which represent the order of arrival in which every fault appeared during a test. Every fault in this sequence will be the Next Expected Fault (NEF). By iterating the sequence of next expected faults, it will be possible to reconstruct the original bitmap of failures. It's possible to define the sequence of next expected faults as N.

$$N = NEF_n \tag{3.1}$$

In order to compare the sequence of next expected faults with a sequence of faults of a following test in the test-flow, a new sequence F is defined as the following.

$$F = Fault_n \tag{3.2}$$

It is possible to define the sets of all the next expected faults and all the faults of different test, by simply taking the set of the sequences.

$$\mathbb{N} = \{NEF_n\} \tag{3.3}$$

$$\mathbb{F} = \{Fault_n\} \tag{3.4}$$

Given the two sets $\mathbb{N}, \mathbb{F}$, it is now possible to define the set of the differences Δ as, each fault p belonging to the union of $\mathbb{N}$ and $\mathbb{F}$, where all the fault inside the intersection of $\mathbb{N}$ and $\mathbb{F}$ need to be filtered out. By using mathematical notation, Δ can be expressed like in the Equation 3.5.

$$\Delta = \Big\{p \in \mathbb{N} \bigcup \mathbb{F} \mid p \notin \mathbb{N} \bigcap \mathbb{F}\Big\} \tag{3.5}$$

Given the set of differences Δ and the set of the next expected faults $\mathbb{N}$, it is also possible to retrieve back the original set of faults. The actions required to perform this operation are identical to ones used to compute the set of differences Δ.

$$\mathbb{F} = \Big\{p \in \mathbb{N} \bigcup \Delta \mid p \notin \mathbb{N} \bigcap \Delta\Big\} \tag{3.6}$$

A formal proof for Equation 3.6 is shown in the following Equation 3.7.

$$\begin{aligned}
&\Big\{p \in \mathbb{N} \bigcup \mathbb{F} \mid p \notin \mathbb{N} \bigcap \mathbb{F}\Big\} \\
&\qquad = \Delta \bigcup \mathbb{N} - \Delta \bigcap \mathbb{N} \\
&\qquad = \Big(\mathbb{N} \bigcup \mathbb{F} - \mathbb{N} \bigcap \mathbb{F}\Big) \bigcup \mathbb{N} - \Big(\mathbb{N} \bigcup \mathbb{F} - \mathbb{N} \bigcap \mathbb{F}\Big) \bigcap \mathbb{N} \\
&\qquad = \Big(\mathbb{N} \bigcup \mathbb{F}\Big) - \Big(\mathbb{N} - \mathbb{N} \bigcap \mathbb{F}\Big) \\
&\qquad = \mathbb{N} \bigcup \mathbb{F} - \mathbb{N} + \mathbb{N} \bigcap \mathbb{F} \\
&\qquad = \mathbb{F} - \mathbb{N} \bigcap \mathbb{F} + \mathbb{N} \bigcap \mathbb{F} \\
&\qquad = \mathbb{F}
\end{aligned} \tag{3.7}$$

It is now clear why the operation of computing a difference is identical to the bit-wise XOR, in fact performing such operations between sets is identical to one of the XOR properties.

$$a \oplus b = c \iff c \oplus b = a \tag{3.8}$$

Now that N and F have been defined, an algorithmic strategy must be implemented to create the sequence of next expected faults N, while the sequence F is trivially obtained when iterating the eMemory, N is more complicated to retrieve.

As discussed previously the faults are encoded into a list of spooky slices, in order to get the sequence N an ordering among the coordinates must be defined. The ordering will be defined by inserting in the sequence N a Next Expected Fault by iterating over the spooky slices, first by bit-lines and then by word-lines, every time a new fault is found it will be inserted in the ordered sequence.

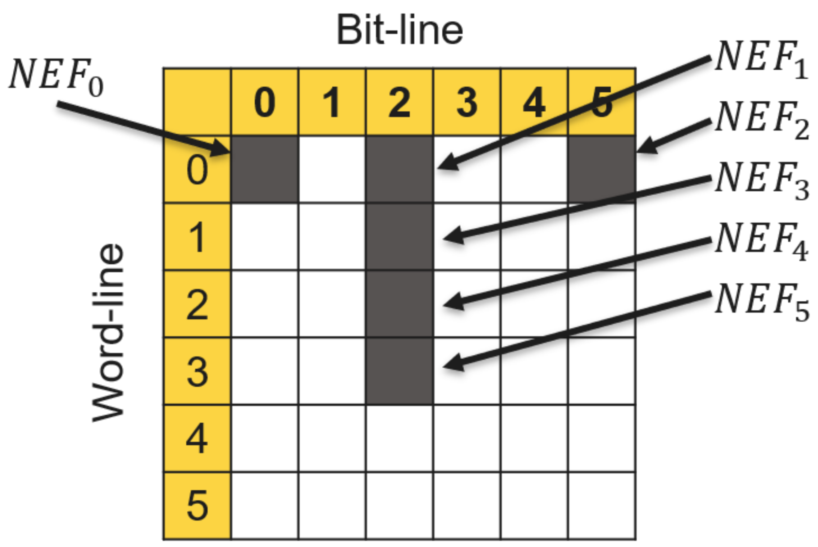

Figure 3.2: Sequence of Next Expected Fault (NEF) inside the spooky slices.

To be more precise it's possible to define the sequence from the spooky slices by defining ordering between each coordinate c. This is shown in the Definition 3.3.1.

Definition 3.3.1 (NEF ordering)**.** A coordinate a is strictly less than a coordinate b, if the word-line of a is less than the word-line of b, or, when they are on the same word-line, the bit-line of a is less than the bit-line of b.

$$a < b \iff (a|_{wl} < b|_{wl}) \text{ or } (a|_{bl} < b|_{bl} \text{ and } a|_{wl} = b|_{wl})$$

Given the fault ordering it is now trivial how, with a set of spooky slices, the sequence N should be built; this is graphically illustrated in Figure 3.2. The code for the creation of such iterator will be discussed later, after the explanation of the spooky slices coordinates system.

Now that it is possible to build the sequence N as well, an algorithm is needed to create the set of differences Δ. A high level description of the actions needed to be taken are shown in the Algorithm 2.

By iterating over each fault of the F sequence, a comparison with the current NEF is performed. If the NEF is less than the fault, the NEF will be advanced until it will be equal or greater than the fault, every time this operation is performed the NEF is encoded as difference. If on the other hand, the fault is less than the current NEF, the fault is encoded as a difference and the current fault will be advanced. When they are equal, no encoding takes place, and both progress forward in their sequence.

The Algorithm 2 is not so far from the reality, in fact the code in the Listing 3.5, is the actual implementation for the difference encoding of the faults. Some

Algorithm 2 High level algorithm for the difference computation given the sequences N and F.

```
procedure ENCODE-DIFFERENCE($N, F$)
    $NEF_k \leftarrow NEF_0 \in N$
    ▷ Iterate over all the faults in $N$.
    for $Fault_n \leftarrow F$ do
        ▷ Encode and advance $NEF_k$ while it is less than the current fault.
        while $NEF_k < Fault_n$ do
            $encode(NEF_k)$                    ▷ Encode the NEF as a difference.
            $NEF_k \leftarrow NEF_{k+1}$                                  ▷ Next NEF.
        end while
        if $NEF_k > Fault_n$ then
            $encode(Fault_n)$                  ▷ Encode the fault as a difference.
        else if $NEF_k = Fault_n$ then
            $NEF_k \leftarrow NEF_{k+1}$                                  ▷ Next NEF.
        end if
    end for
end procedure
```

extra information are required to obtain the comparison, in fact faults and Next Expected Faults are encoded in coordinates, but the coordinates only refer to an offset of specific bank of the eMemory. When comparing the two coordinates need to be "augmented" with more data, the current bank in this case, for this reason all the comparisons functions start with `augmented`. Other checks are performed for the current NEF, in case the iterator is terminated, and no more items can be retrieved from the sequence.

Listing 3.5: C code implementation of the differences encoding mechanism, here the faults come from the caller function which will contain a for loop of the eMemory, and for the nef an iterator is given for the iterating the sequence.

```c
static void encode_difference_single(DeltaSlacContext *context, NefIterator *nef_iter, const SpookySlicesBuckets *slices, const Coordinates linearized_fault, const CPLXDRV_ITERATOR_BANK_T bank)
{
    const AugmentedCoordinates augmented_fault = augmented_coord_init(linearized_fault, bank);

    while (NEF_ITERATOR_ITERATING == nef_iter->state && augmented_nef_compare(nef_iter_augmented_coord(*nef_iter), augmented_fault))
    {
        encode_fault_as_difference(&context->page_buffer, context->encode_faults_to_slac, nef_iter_augmented_coord(*nef_iter));

        nef_iter_next_with_bank(nef_iter, slices);
    }

    if (NEF_ITERATOR_TERMINATED == nef_iter->state || augmented_nef_compare(augmented_fault, nef_iter_augmented_coord(*nef_iter)))
    {
        encode_fault_as_difference(&context->page_buffer, context->encode_faults_to_slac, augmented_fault);
    }

    if (NEF_ITERATOR_ITERATING == nef_iter->state && augmented_coord_eq(augmented_fault, nef_iter_augmented_coord(*nef_iter)))
    {
        nef_iter_next_with_bank(nef_iter, slices);
    }
}
```

A graphical representation of the difference computation starting from some base faults ($\mathbb{N}$) and set of faults from a different test ($\mathbb{F}$), it's possible to compute the difference faults (Δ), this is shown in Figure 3.3.

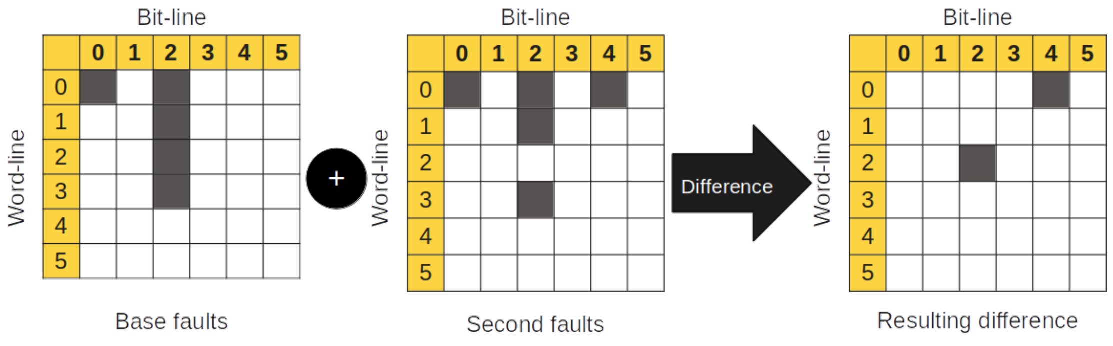

Figure 3.3: Difference computation starting from a bitmap of base faults compared with a set of faults coming from another test, with the result difference bitmap.

3.4 Address linearization

During the fault detection process in eMemory, the data retrieved from the test program on the board does not correspond directly to the physical memory layout. Ideally, an eMemory would be read sequentially, progressing word-line by word-line and bit-line by bit-line. However, this approach is not feasible due to two primary constraints. First, as previously discussed, eMemories are accessed using a checkerboard-style iteration, which partitions each sector of every bank into zone-A and zone-B. Second, memory scrambling must always be considered, as it further influences the data retrieval process.

Memory scrambling in Resistive Random-Access Memory (RRAM) is an essential technique to enhance data security, mitigate write disturbances, and improve endurance. RRAM devices are susceptible to security threats such as data remanence and side-channel attacks, where predictable memory patterns can be exploited. Scrambling disrupts these patterns by applying transformation functions, typically XOR-based operations or permutation algorithms, before writing data to memory cells. This reduces the likelihood of repeated stress on specific memory locations, thereby improving the device's reliability and lifespan. Additionally, scrambling helps in mitigating resistance drift and variability issues, which are critical challenges in RRAM technology. While it does not provide full encryption, memory scrambling significantly enhances data integrity and security in RRAM based storage and computing applications.

To address this issue, an "address linearization" mapping is introduced. The primary objective of this approach is to ensure that faults are received in an ordered manner, conforming to Definition 3.3.1, thereby enabling sequential iteration of the generated spooky slices.

To maintain this structure, the spooky slices in the Delta-SLAC and the basic slices in the SLAC, are designed as distinct structures. While the spooky slices

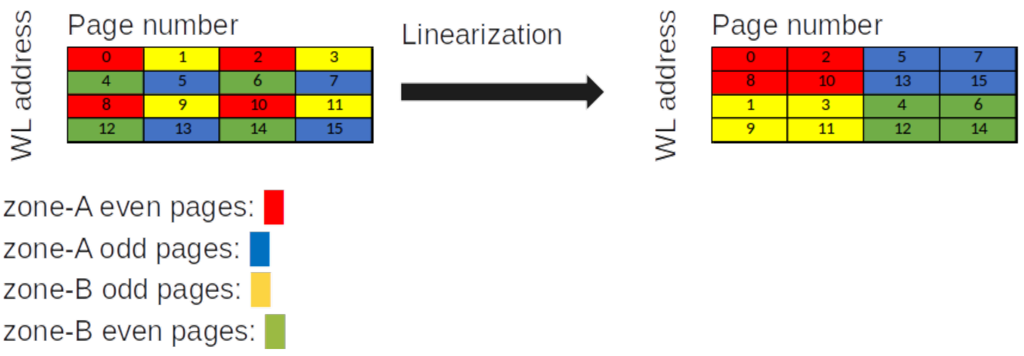

Figure 3.4: Example of the logical address linearization mapping.

must preserve the fault arrival order, this information is lost when transitioning from logical addresses to physical representation. This loss occurs due to the checkerboard-style iteration of zone-A and zone-B, as well as the scrambling mechanism inherent in the physical layout of eMemory.

For this reason the spooky slices of the Delta-SLAC and the basic slices of the SLAC, are designed as two different structures, the spooky slices must keep track of the order of arrival of the faults, but when switching from logical addresses to a physical representation this information is lost due to the zone-A and zone-B iteration, and due to the scrambling taking place in the physical cell layout of the eMemory.

By mapping logical addresses to linearized coordinates, vertical physical faults remain aligned, facilitating higher compaction of spooky slices. Figure 3.4 illustrates an example of this linearization process, where logical addresses are reordered to reflect the sequential arrival of data.

To create the sequence N, the iterator concept has been used, by calling the `nef_iter_next` function it's possible to retrieve the next NEF in the sequence.

The function `nef_iter_next` is responsible for advancing the Next Expected Fault iterator within a given set of spooky slices, ensuring correct traversal across faults while maintaining spatial and directional consistency. This function operates within a fault-mapping structure where faults are stored in non-sequential slices categorized by their orientation and color. It takes as input a pointer to the NEF iterator, which tracks the current position, and a pointer to the spooky slices structure containing fault-mapped data.

The function begins by retrieving the current spooky slice where the NEF is currently located. It determines if the spooky slice is horizontally aligned and the next expected fault matches the last fault, the function updates the expected fault position based on the slice's color classification. For each horizontally matched slice the current Next Expected Fault is simply updated to the next fault inside the spooky slice by following the increasing x-axis coordinates. The case of black colored slices is considered invalid and does not alter state. If the fault remains within the same slice, the function exits early.

If the current slice is vertically aligned, the function then searches for the closest Next Expected Fault using `spooky_slices_walk()` with a `walk_context`, this function will perform a "walk" on all the slices matching only the ones within a certain y-interval range, and passing a filtering operation. Searching into adjacent spooky slices for vertical slices is more expensive than a simple iteration over the last remaining faults, like in the horizontal case, but problem comes from the definition of the NEF ordering, in fact the search interval will be limited between the current Next Expected Fault and the possible next NEF present the in the same vertical slice.

When no immediate next fault is found, the function attempts to locate the next ordered slice using `spooky_slices_next_ordered_slice()`. If a valid next slice exists, the search interval is updated to check for overlapping slices at the same y-coordinate, and another iteration of `spooky_slices_walk()` is performed. Finally, if a valid next NEF is identified, the iterator updates its index and expected fault position. If no valid fault is found, the iterator state is set to "terminated", indicating the completion of traversal.

Listing 3.6: C code implementation of the next Next Expected Fault retreival from the iterator.

```c
static void nef_iter_next(NefIterator *nef_iter, const SpookySlices *
    spooky_slices)
{
    const SpookySliceExtended current_spooky_slice =
    spooky_slices_get(spooky_slices, nef_iter->nef_slice_index);
    const Coordinates last_fault = spooky_slice_end(
    current_spooky_slice);

    // still in the same HORIZONTAL slice, no need to search another
    one
    if (SPOOKY_SLICE_HORIZONTAL == current_spooky_slice.direction &&
    nef_compare(nef_iter->next_expected_fault, last_fault))
    {
        switch (current_spooky_slice.color)
        {
        case SPOOKY_BLUE:
            nef_iter->next_expected_fault.x = last_fault.x;
            break;
        case SPOOKY_RED:
            nef_iter->next_expected_fault.x += 2;
            break;
        case SPOOKY_ORANGE:
            nef_iter->next_expected_fault.x += 1;
            break;
        case SPOOKY_BLACK:
            /* Should not be here! */
            break;
        }

        return;
    }

    sint32 maybe_next_nef_index = -1;
    Coordinates maybe_next_nef = coordinates_init(0xffff, 0xffff);

    /* Try to initialize the NEF with next fault in the current vertical slice. */
    if (current_spooky_slice.direction == SPOOKY_SLICE_VERTICAL && !coordinates_eq(nef_iter->next_expected_fault, spooky_slice_end(current_spooky_slice)))
    {
        switch (current_spooky_slice.color)
        {
        case SPOOKY_ORANGE:
            maybe_next_nef = coordinates_init(nef_iter->next_expected_fault.x, nef_iter->next_expected_fault.y + 1);
            maybe_next_nef_index = nef_iter->nef_slice_index;
            break;
        case SPOOKY_RED:
            maybe_next_nef = coordinates_init(nef_iter->next_expected_fault.x, nef_iter->next_expected_fault.y + 2);
            maybe_next_nef_index = nef_iter->nef_slice_index;
            break;
        case SPOOKY_BLUE:
            maybe_next_nef = spooky_slice_end(current_spooky_slice);
            maybe_next_nef_index = nef_iter->nef_slice_index;
            break;
        case SPOOKY_BLACK:
            break;
        }
    }

    /* Ceck if there is a fault on the current y-interval of the nef */
    WalkContext walk_context = (WalkContext){
        .curr_nef = nef_iter->next_expected_fault,
        .maybe_nef = maybe_next_nef,
        .maybe_index = maybe_next_nef_index,
    };

    WalkSpookySlices walk = (WalkSpookySlices){
        .walk_op = find_closest_nef,
        .walk_context = &walk_context,
    };

    const Interval nef_interval = (Interval){
        .low = nef_iter->next_expected_fault.y,
        .high = nef_iter->next_expected_fault.y + 1,
    };

    /**
     * This loop finds the fault in the spooky slices that is the closest
     * to the current NEF.
     *
     */
    spooky_slices_walk(spooky_slices, nef_interval, walk);

    /* Check if we found a possible nef after the first walk. */
    if (walk_context.maybe_index == -1)
    {
        const sint32 maybe_next_index = spooky_slices_next_ordered_slice(spooky_slices, nef_iter->nef_slice_index, nef_iter->next_expected_fault.y);

        if (maybe_next_index != -1)
        {
            const SpookySliceExtended next_slice = spooky_slices_get(spooky_slices, (uint32)maybe_next_index);
            const Coordinates first_fault = spooky_slice_start(next_slice);

            /* Get the interval to search from the starting fault of the next_slice */
            const Interval search_interval = (Interval){
                .low = first_fault.y,
                .high = first_fault.y,
            };

            /* This is done in order to check if there is more than one slice at the same y, overlapping. */
            spooky_slices_walk(spooky_slices, search_interval, walk);
        }
    }

    if (walk_context.maybe_index != -1)
    {
        nef_iter->nef_slice_index = (uint32)walk_context.maybe_index;
        nef_iter->next_expected_fault = walk_context.maybe_nef;
    }
    else
    {
        nef_iter->state = NEF_ITERATOR_TERMINATED;
    }

    return;
}
```

The `nef_iter_next` function ensures an orderly traversal of spooky slices while preserving fault alignment despite variations in slice orientation and memory scrambling. By employing an adaptive search strategy, it efficiently determines the next fault location while maintaining logical consistency within the eMemory fault-mapping framework. The implementation of the function is shown in the Listing 3.6.

3.5 Sorting the spooky slices

Due to the limited memory environment of embedded programs it becomes unfeasible to exploit data structures with dynamic capabilities, such as self-balancing trees, hash-tables or linked lists. This becomes even more problematic when the embedded environment system is designed without any primitives for memory management, such as memory allocators, that would provide some level of flexibility for data structure design. Given the limited options for building any complex data structure, the most basic approach would be to use static arrays declared globally with a fixed amount of elements.

This limitation is very relevant in the construction for a suitable data structure that would store all the set of spooky slices. As shown in the previous sections, spooky slices need to be created, modified, and searched. Of course an implementation should take into consideration all of these aspects, by providing time effective capabilities for all the operations.

The base for the spooky slices' storage will be a static array of fixed position, this array will act as an object-pool-like structure, where instantiating a new spooky slice is performed by increasing an index pointing inside the array, this index will divide the array in allocated objects, and non-allocated objects. In order to perform fast searches among the slices there needs to be some kind ordering. Until now no ordering definition was provided for the spooky slices themselves, while it was provided for the faults that they are made out of.

A suitable abstraction for ordering can be achieved by considering the intervals of each slice, enabling their arrangement within the array. Before deriving the intervals from the spooky slices, it is essential to establish precise definitions for the intervals that will be utilized in this context.

Definition 3.5.1 (Set of intervals)**.** Let $\mathbb{I} = \{[l, h] \mid l, h \in \mathbb{N}, \, l \leq h\}$ represent the set of all intervals where l and h are natural numbers and $l \leq h$.

Definition 3.5.2 (Interval ordering)**.** Consider a and b denote two intervals within the set $\mathbb{I}$. The ordering between these intervals is then defined as follows

$$a < b \iff a_l < b_l \quad \text{or} \quad (a_h < b_h \quad \text{and} \quad a_l = b_l)$$

Definition 3.5.3 (Interval intersection)**.** Consider a and b denote two intervals within the set $\mathbb{I}$. The intersection between these intervals is then defined as follows

$$a \cap b \iff a_l \leq b_h \quad \text{and} \quad a_h \geq b_l$$

For sorting the spooky their y-interval will be considered. This can easily computable, where if the spooky slice is a horizontal slice the interval will simply have both the extremes as the shared word-line. When a spooky slice is oriented vertically the interval will range from the starting word-line to the ending word-line of the slice. In Figure 3.5 a mapping between spooky slices and their respective interval is shown.

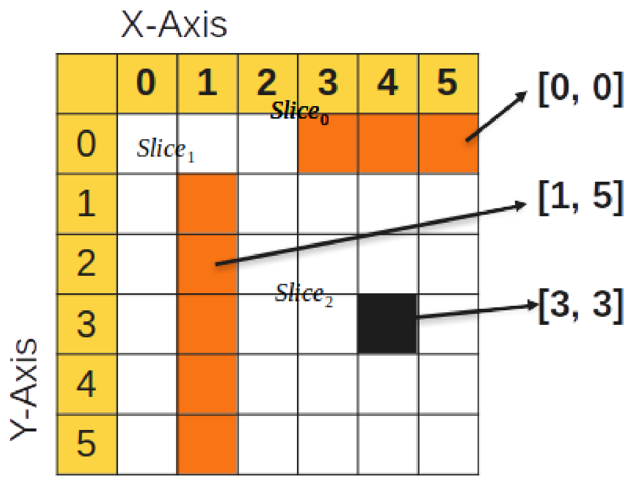

Figure 3.5: Spooky slices with their respective y-interval shown on the side.

If the array of spooky slices is maintained in a sorted order, a binary search can be performed, allowing queries to be executed in $O(\log n)$ time, where n denotes the number of elements in the array. By initiating the search from the midpoint of the array and treating the left and right traversal steps as branches in a tree structure, with the root corresponding to the central element, the sorted array can inherently be interpreted as a tree-like structure. Notably, this representation eliminates the need for explicit pointer-based linking between individual nodes, enabling a great reduction in memory occupation.

By considering the intervals as a sorting mechanism, a data structure suitable for searching is the Augmented Interval Tree [2], where each node stores an interval along with an auxiliary value, such as the maximum endpoint of all intervals in its subtree, called `max_high`. This augmentation enables efficient range queries, such as searching for all intervals, or points overlapping a given query interval, in $O(\log n + m)$ time, where n is the number of nodes and m is the number of reported matching results. Augmented interval trees are widely used in computational geometry, scheduling algorithms, and memory allocation, where efficient interval-based queries are required.

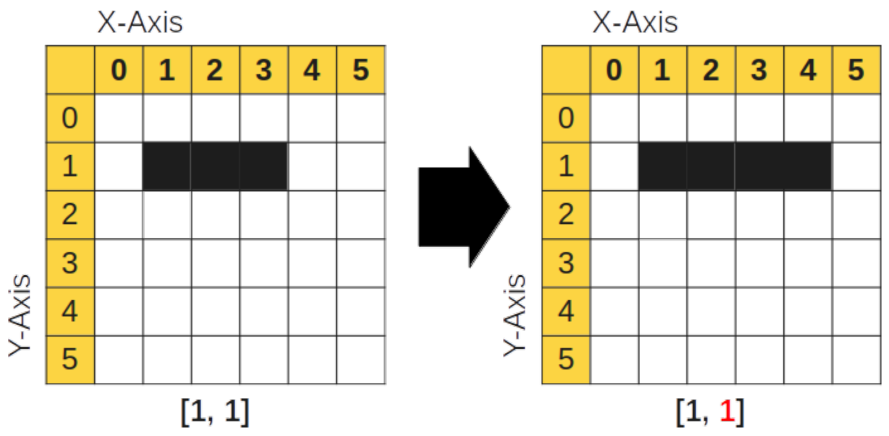

Figure 3.6: Horizontally oriented spooky slice interval modification on expansion.

Using the interval of a spooky slice for keeping track of the order come from an observation of the faults' distribution. All the faults that will create the spooky slices, will come in a structured sequence, following the order of Definition 3.3.1 thanks to the linearization of the addresses. From this observation it comes naturally that once a slice is created, it can only extend itself along increasing word-lines, or increasing bit-lines. If a spooky slice is oriented horizontally, its y-interval will never change, staying fixed for the entire duration of the test, as shown in Figure 3.6, while a spooky slice with vertical orientation, can only grow along increasing word-lines, making the low part of the y-interval stay always fixed, only the high part might increase, as shown in Figure 3.7.

It's possible to see now the power of this interpretation, this is especially true when encoding a new a fault in the spooky slices, a suitable slice that can be extended must be found first, because the interest is in single spooky slice, the timing for the operation are $O\log(n)$, in this case the m term is not present because it is equal to 1. In the Listing 3.7 an implementation of the tree traversal is shown.

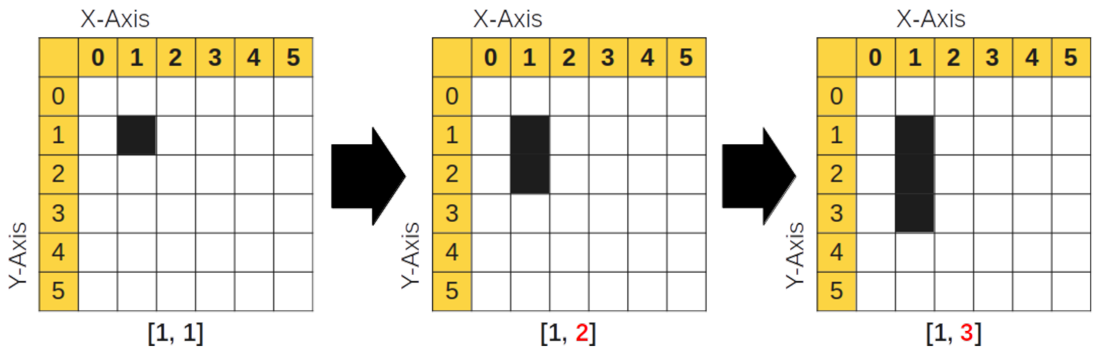

Figure 3.7: Vertically oriented spooky slice interval modification on expansion.

The function operates recursively until no spooky slice capable of extending the current fault is found. The initial phase of the function involves examining the current spooky slice within the recursion. If a match is detected, the corresponding slice index is returned. Otherwise, a comparison is performed between the lower bound of the queried interval and the `max_high` value of the left child. If `max_high` is greater than the queried interval's lower bound, the recursion precedes on the left subtree; otherwise, it follows that no potential match can exist within the left subtree, and the search continues along the right children.

Algorithm 3 High level algorithm for the difference computation given the sequences N and F.

```
1:  procedure BINARY-SEARCH($node, interval$)
2:      $NEF_k \leftarrow NEF_0 \in N$
3:      ▷ Iterate over all the faults in $N$.
4:      if $node_{slice} \cap interval$ then
5:          return $node$
6:      end if
7:      $left \leftarrow node_{left}$
8:      $right \leftarrow node_{right}$
9:      if $\nexists left$ then
10:         return BINARY-SEARCH($node_{right}, interval$)
11:     else if $left_{\max high} \geq interval_{low}$ then
12:         return BINARY-SEARCH($node_{left}, interval$)
13:     else if $\exists right$ then
14:         return BINARY-SEARCH($node_{right}, interval$)
15:     end if
16:     return $\{\}$
17: end procedure
```

Algorithm 3 provides a high-level representation of the binary search process performed on the tree, ensuring a time complexity of $O(\log n)$.

Listing 3.7: C code implementation of the recursion algorithm for finding an extendable spooky slice.

```c
static sint32 _spooky_slices_binary_search(const SpookySlices *
    spooky_slices, const Interval fault_i, const Coordinates fault,
    const uint32 current_index, const uint32 start, uint32 finish)
{
    sint32 found_slice_index = -1;

    const SpookySliceExtended current_slice =
        spooky_slices_get(spooky_slices, current_index);
    const boolean found_slice =
        spooky_slice_vertical_continuation(current_slice, fault)
        || spooky_slice_horizontal_continuation(current_slice, fault)
    ;

    if (found_slice)
        found_slice_index = (sint32)current_index;

    const uint32 left = (current_index + start) / 2;
    const uint32 right = (current_index + finish) / 2;

    const boolean go_left = start <= left && left < current_index;
    const boolean go_right = current_index < right && right < finish;

    /* Interval search */
    /* If the slice has not been found, keep going down */

    /**
     * Search left children.
     *
     * If left.max_high is less than the fault_i.low it means
     * that all the nodes in the left sub-tree will never intersect
     * the interval.
     *
     */
    if (
        found_slice_index == -1
        && go_left
        && spooky_slices_get_max_high(spooky_slices, left) >= fault_i
    .low)
    {
        found_slice_index = _spooky_slices_binary_search(
            spooky_slices,
            fault_i,
            fault,
            left,
                start,
                current_index);
    }
    else if (found_slice_index == -1 && go_right && check_right)
    {
        found_slice_index = _spooky_slices_binary_search(
                spooky_slices,
                fault_i,
                fault,
                right,
                current_index + 1,
                finish);
    }

    return found_slice_index;
}
```

During the creation phase of the spooky slices, the instantiation of a new slice, and the enlargement of a vertical spooky slice need to be taken into consideration, when adopting this sorted-array-tree interpretation of the memory. If a new spooky slice is inserted into the array, the root of the tree-structure will now be a different one, this will trigger a full traversal of the tree in order to fix all the `max_high` values, the reason for this behavior is that when the root moves to a different position, causing all the previous branches to modify their paths to different nodes, invalidating all the `max_high` values inside each node. When this situation happens, the time to fix the whole tree takes a time complexity of $O(n)$. Insertion frequency in the array may vary based on the compactness of the faults encountered in the eMemory, if the faults are compact enough, on each fault a matching spooky slice will be found in the binary search, avoiding new allocations in the array. Figure 3.8 shows an example of insertion in the tree, triggering the `max_high` fixing operations.

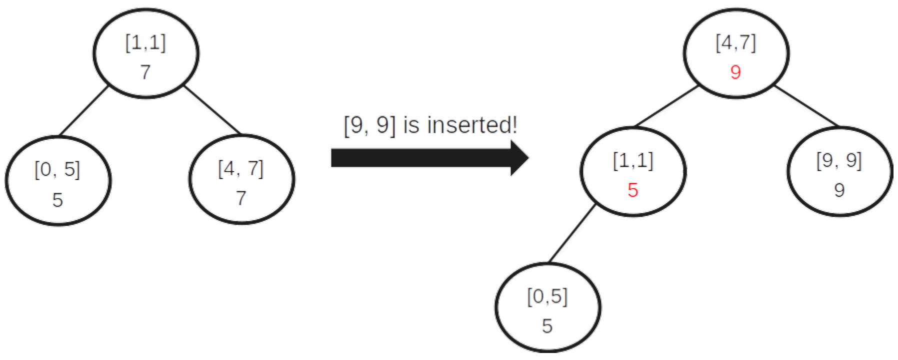

Figure 3.8: Augmented interval tree (array based) on the insertion of a new interval.

When a spooky slice will increase it size, the result of this operation might disrupt the ordering the array, while it is true that for each spooky slice the lower end of the interval will remain fixed, the high part might increase, when this happens the spooky slice needs to be swapped into a new position in order to maintain the ordering true.

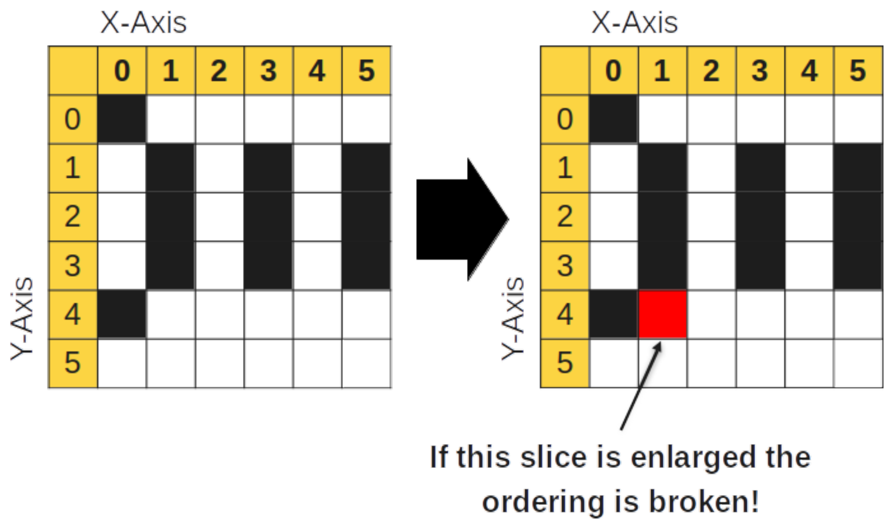

Figure 3.9: An enlargement of a spooky slice disrupts the ordering among the intervals.

When the ordering is disrupted, the corrective steps are straightforward. The process involves identifying the next interval that is greater than the last modified interval, which caused the disruption. Subsequently, the enlarged interval is swapped with the preceding interval of the newly identified one to restore the correct ordering. The steps followed to fix the example in Figure 3.9 are shown in Figure 3.10.

Of course this approach comes with downsides; for instance, if the sequence of identical intervals is long enough a lot of comparisons have to be made before the spooky slice is swapped in the correct position, this might take $O(n)$ in the worst case scenario, but a situation where all the spooky slices are identical is quite rare. After the swap is performed the `max_high` of the parent nodes might be wrong, this takes an exploration of the tree to adjust the `max_high` of the nodes along the path to the swapped item; at worse a path will stretch to one of the leafs, because the tree structure will always be balanced, the time complexity to fix an entire path to a leaf takes $O(\log(n))$.

Table 3.1 presents a comparative analysis of execution times, measured in seconds, for different algorithms used in the slice creation phase applied to increasing numbers of injected word-lines. The evaluated approaches include SLAC, a naive

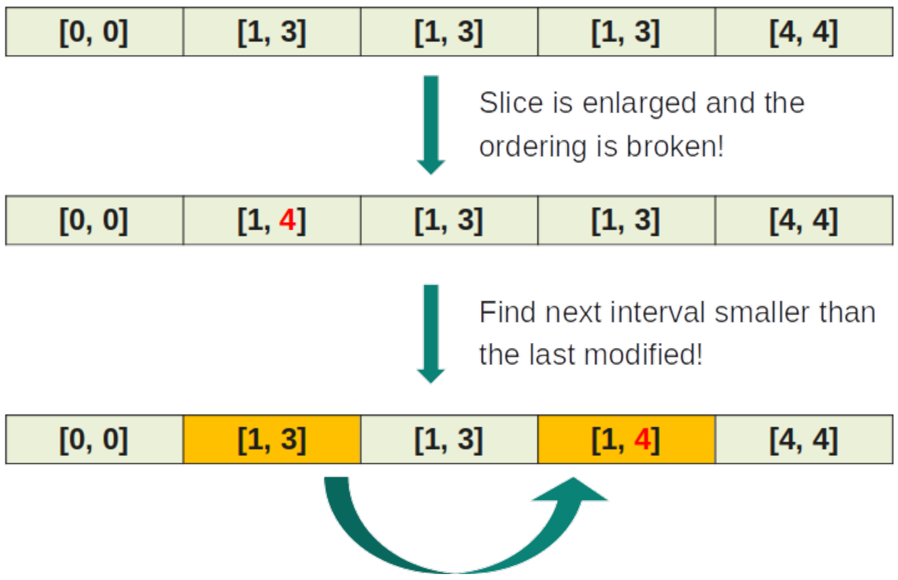
[0, 0]
[1, 3]
[1, 3]
[1, 3]
[4, 4]
Slice is enlarged and the ordering is broken!
[0, 0]
[1, 4]
[1, 3]
[1, 3]
[4, 4]
Find next interval smaller than the last modified!
[0, 0]
[1, 3]
[1, 3]
[1, 4]
[4, 4]

Figure 3.10: Steps to recreate an ordered array due to an enlarged vertical spooky slice.

Linear traversal, and the proposed Array-Tree structure. The results highlight the significant performance advantages of the Array-Tree method over the Linear approach. The time measurement are taken for the whole test time, it should be noted the SLAC timing is inserted as well for reference, but when the Delta-SLAC is enabled with linear and Array-Tree searching, the SLAC timing will be included as well.

The results clearly demonstrate that while the SLAC approach maintains the lowest execution times, the Array-Tree method significantly outperforms the naive Linear approach. As the number of injected word-lines increases, the execution time of the Linear method grows as $O(n^2)$, reaching 556 seconds for 256 word-lines. This quadratic growth suggests that the linear traversal incurs a substantial computational overhead when searching for and processing faults sequentially.

In contrast, the Array-Tree approach maintains a much lower execution time across all tested scenarios. The hierarchical nature of the Array-Tree enables efficient searching and insertion, reducing the complexity compared to a purely sequential approach. Instead of iterating through the entire dataset for each query, the tree-like structure facilitates faster lookups and updates, leading to a significant reduction in execution time. For instance, at 256 injected word-lines, the Array-Tree completes execution in 65 seconds, compared to 556 seconds required by the Linear method—a nearly 8.5× speedup.

Injected word-lines	SLAC [s]	Linear [s]	Array-Tree [s]
50	8	23	10
100	15	85	19
150	20	195	36
200	27	341	46
256	34	556	65

Table 3.1: Full test execution time of the first phase comparison between the SLAC and Delta-SLAC (linear and array-tree approach).

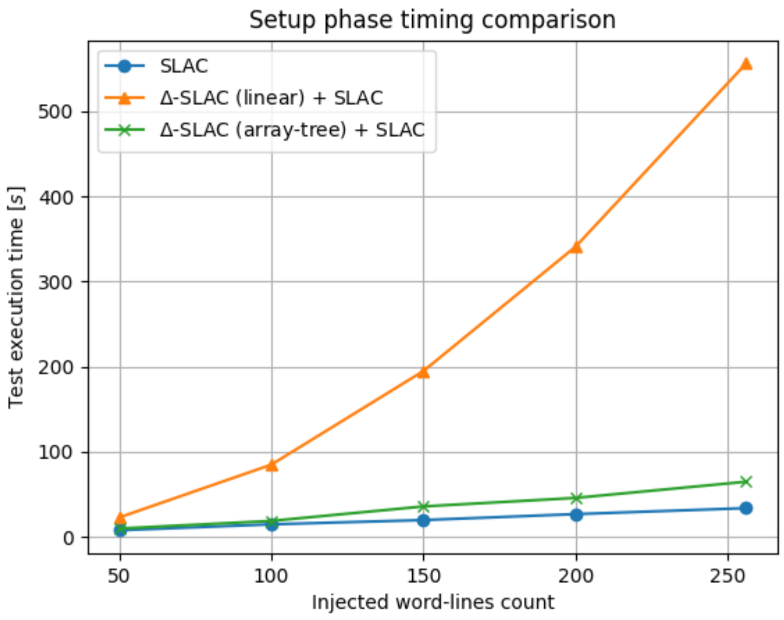

Figure 3.11: Graph representation of Table 3.1.

These results underscore the advantages of leveraging structured data representations for fault processing, particularly when dealing with large-scale fault analysis. The Array-Tree approach provides a scalable and efficient alternative to linear traversal methods, making it a suitable choice for handling high-density fault maps in eMemories.

3.5.1 Bucket subdivision

The array of spooky slices is partitioned into multiple sub-arrays called “buckets”, each corresponding to a distinct memory bank. This division is necessary due to the nature of the linearized coordinates, which only encode the bank offset without explicitly distinguishing between different banks. To overcome this limitation, each bucket of the array is assigned to a separate bank, ensuring that bank-specific spooky slices are grouped together. The implementation for the data structure containing the spooky slices can be shown in the Listing 3.8.

Listing 3.8: C code implementation of the data structures holding the spooky slices.

```c
typedef struct
{
    SpookySlice *spooky_slices;
    uint32 size;
    uint32 capacity;
} SpookySlices;

typedef struct
{
    SpookySlices base;
    SpookySlices buckets[NUM_OF_ITERATOR_BANK];
} SpookySlicesBuckets;
```

By adopting this approach, the array-tree approach remains independent for each bucket, effectively reducing the overhead associated with inserting new faults, caused by the re-computations the `max_high` values. Furthermore, each bucket operates independently, allowing the previous considerations regarding efficient search operations to remain valid within each bucket subdivision. This design choice enhances both scalability and computational efficiency. A graphical representation of the spooky slices array can be shown in Figure 3.12.

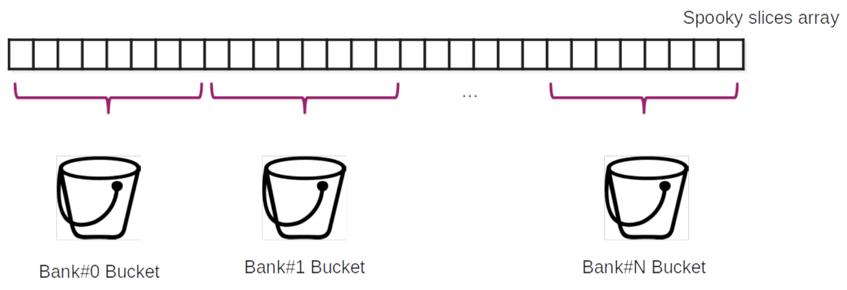

Figure 3.12: Spooky slices array subdivision in buckets.

3.6 Zero-One structures

When an eMemory is tested, as previously discussed, different patters might be injected, enabling pattern specific faults. Among the most common patterns are found: the "Solid Zeros Pattern", the "Solid Ones Pattern", the "Checkerboard Pattern". In order to create an efficient memory reduction these patterns need to be taken into consideration.

The Delta-SLAC framework initializes two essential data structures during the first two verification steps. These initial verifications, referred to as Verify-Zero (when solid zero pattern is injected) and Verify-One (when solid one pattern is injected), are fundamental to the correct functioning of the algorithm. Once this initialization phase is completed, the memory can also be tested using mixed patterns.

The necessity of these Zero/One structures arises from the additional information required when handling mixed pattern verifications. Specifically, the ability to distinguish between different types of stuck-at faults is crucial. E.g. consider the case of two faulty bit-lines:

- One bit-line exhibits a short circuit, causing bits to be permanently stuck at logic `1`.
- Another bit-line is broken, leading to bits that are persistently stuck at logic `0`.

Tracking whether a bit is stuck at logic `0` or logic `1` is essential to optimizing memory usage. Without this classification, additional storage would be required to track faults redundantly, leading to inefficiencies. Considering the above example, by running the setup-phase with a solid zeros pattern, only the shorted bit-line will be marked as faulty, becoming the set of base faults, if the second test is run with solid ones pattern the shorted bit-line in the current pattern will not be considered as faulty, because the current pattern expects a reading of `1` bit values, only the broken bit-line will be read as faulty; in this situation the algorithm will find a bit-line (shorted) that is not present, so it will be marked as a difference, and a new faulty bit-line (broken) that will be marked as well as a difference. This is limitation is due to a miss-classification of the faults by the algorithm. A graphical representation of the filtering process is shown in the Figure 3.13.

To ensure efficient fault representation, the initial verification steps must always include a Verify-Zero followed by a Verify-One. After these two steps, subsequent tests can compute differences relative to the initialized structures, thereby avoiding redundant data storage. Given that read faults are persistent across all verification steps, this approach ensures that fault tracking remains compact and efficient.

With the two data structures in place keeping track of the faulty bit stuck-at zero and one, the need for two sequences of next expected faults is required, in order to separate the faults coming from the two structures. This change does not affect the way in which the differences are computed, in fact the Algorithm 2 still holds true. The paradigm shift relies on the way in which the next NEF in the

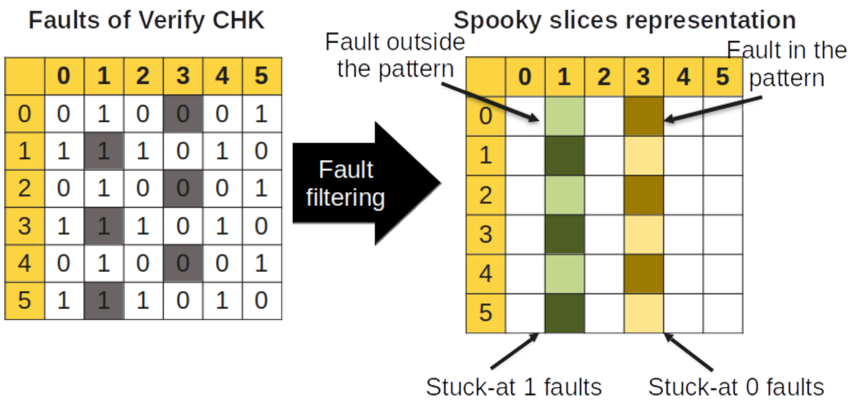

Figure 3.13: Filtering process of the faults from the zero/one structures that will not appear in the current pattern.

sequence is retrieved. To get the Next Expected Fault (global/mixed NEF) for the difference comparison, each of the individual NEF, need to be advanced until a valid fault is found, when both of them are in a valid position, only the smaller one will be used for comparison, if the Next Expected Fault is smaller than the compared fault, it will be advanced to a valid position, and then the smaller will be taken for comparison. The actual code implementation for retrieving the next NEF used for the difference computation is shown in the Listing 3.10.

Listing 3.9: Data structures that represent the mixed iterator of Next Expected Faults.

```
typedef struct
{
    Coordinates next_expected_fault;
    uint32 nef_slice_index;
    NefIteratorState state;
    CPLXDRV_ITERATOR_BANK_T bank;
} NefIterator;

typedef struct
{
    NefIterator *nef_iter_zero;
    NefIterator *nef_iter_one;
    const SpookySlicesBuckets *zero_slices;
    const SpookySlicesBuckets *one_slices;
    boolean nef_from_zero;
    Coordinates nef;
    NefIteratorState_t state;
    CPLXDRV_ITERATOR_BANK_T bank;
    SHARED_PATTERN_TYPE_T pattern;
} MixedNefIterator;
```

The `MixedNefIterator` structure holds all the information to advance the two independent Next Expected Faults: a pointer to the two Next Expected Faults and spooky slices, the actual Next Expected Fault used for the difference computation with a flag specifying if the Next Expected Fault comes from the zero-structure or the one-structure, and the current pattern injected into the memory, to filter out the faults that would not appear in the current reading. The structures design is shown in the Listing 3.9.

Listing 3.10: C code implementation for computing the next global NEF used for the difference computation.

```c
/**
 * @brief Iterates over the nefs until the iterator
 * reaches the end.
 *
 */
#define for_each_nef(nef_iter, spooky_buckets)                \
    for (nef_iter_next_with_bank(nef_iter, spooky_buckets);  \
         NEF_ITERATOR_ITERATING == (nef_iter)->state;         \
         nef_iter_next_with_bank(nef_iter, spooky_buckets))

static void mixed_nef_iter_next(MixedNefIterator *mixed_nef_iter)
{
    NefIterator *curr_nef_iter;
    const SpookySlicesBuckets *curr_slices;
    const boolean stuck_bit = mixed_nef_iter->nef_from_zero;

    mixed_nef_iter_get_curr_nef(*mixed_nef_iter, &curr_nef_iter, &
    curr_slices);

    for_each_nef(curr_nef_iter, curr_slices)
    {
        const NonLinearFault fault = dslac_derlinearize_fault(
    curr_nef_iter->next_expected_fault, curr_nef_iter->bank);

        if (stuck_bit != get_bit_value_from_pattern(fault,
    mixed_nef_iter->pattern))
            break;
    }

    mixed_nef_iter_update_nef(mixed_nef_iter);
}
```

3.7 Off-line processing

At the end of test-flow, the dumped faults will be retrieved by an external test program and stored for further data analysis. The final output will consist of, at most, two a set of faults, referred to as the base faults. These faults will be a byproduct of the setup phase of the Delta-SLAC, containing the information about the bits stuck-at `0` and `1`. All the subsequent tests will encode the newly found faults using the difference computation mechanism, relative to the base faults.

To reconstruct the original fault set, a series of off-line operations must be performed. Given the output of the n^{th} test (difference encoding) and the base faults, a decoding from the SLAC slice representation to a plain fault list must be carried out in advance. The base faults are first filtered according to a specific pattern. Once filtered, they are merged, and a bitwise XOR operation is applied against the faults detected in the n^{th} test. The result of this process yields the original set of recorded faults. A diagram of the operations is shown in Figure 3.14.

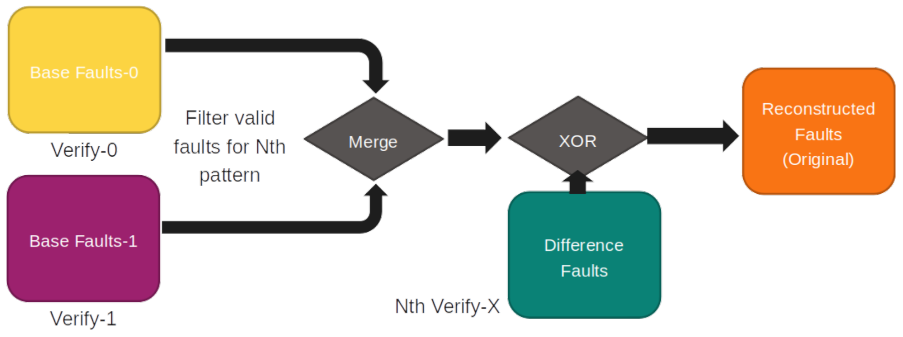

Figure 3.14: Original faults reconstruction schema.

To provide a formal proof of the correctness of the approach, following the Equation 3.6, it comes that $\mathbb{N}$ can be replaced with the set of Next Expected Faults at the n^{th} test $\mathbb{N}_n$.

Definition 3.7.1 (NEF set at the n^{th} test.)**.** Given the set of Next Expected Faults at the n^{th} test, for bits stuck-at `0`, as $\mathbb{N}_n|_0$, and the set of bits set at `1` in the pattern run during the n^{th} test as $\mathbb{P}_n|_1$, the bits stuck-at `0` that can appear at the n^{th} test will be intersection of the two sets, called $\mathbb{Z}_n$.

$$\mathbb{Z}_n = \left\{ p \mid p \in \mathbb{N}_n|_0 \bigcap \mathbb{P}_n|_1 \right\}$$

The same reasoning can be made for the set bits stuck-at `1` at the n^{th} test defined as $\mathbb{O}_n$.

$$\mathbb{O}_n = \left\{ p \mid p \in \mathbb{N}_n|_1 \bigcap \mathbb{P}_n|_0 \right\}$$

The set of Next Expected Faults at the n^{th} test $\mathbb{N}_n$ is defined as, the union of $\mathbb{Z}_n$ and $\mathbb{O}_n$.

$$\mathbb{N}_n = \left\{ p \mid p \in \mathbb{Z}_n \bigcup \mathbb{O}_n \right\}$$

Once $\mathbb{N}_n$ has been defined, in order to compute the correct differences it will just be plugged in the Equation 3.6, obtaining.

$$\mathbb{F} = \left\{ p \in \mathbb{N}_n \bigcup \Delta \mid p \notin \mathbb{N}_n \bigcap \Delta \right\} \tag{3.9}$$

3.8 Algorithm Verification

The verification of the Delta-SLAC algorithm is a fundamental step in ensuring its correctness and efficiency in fault detection and encoding. The verification process consists of multiple phases, each designed to validate different aspects of the algorithm, including its initialization and fault encoding mechanism.

The test where conducted on a real SoC with RRAM technology, by performing emulated injections of the eMemory and running a verification test to run the Delta-SLAC algorithm and store the read faults. In order to process huge amounts of data a set of programs and libraries where designed using Python to facilitate the operations. Before starting the injections the faults, need to be encoded in a suitable format for the injections, after the set of faults are ready to be injected, the “Injection Loop” phase begins. In this phase four operations are repeated over the set of faults: set the memory to the desired pattern, inject the memory with previously recorded faults, run the test, collect relevant statics and the final test dump. The diagram is shown in Figure 3.15.

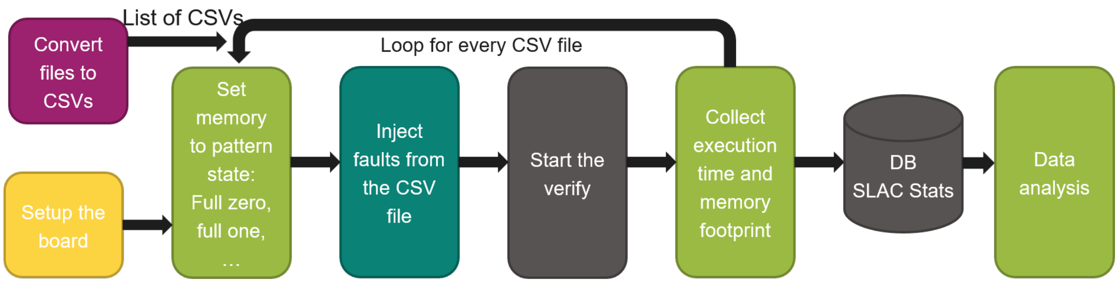
List of CSVs
Loop for every CSV file
Convert files to CSVs
Setup the board
Set memory to pattern state: Full zero, full one, …
Inject faults from the CSV file
Start the verify
Collect execution time and memory footprint
DB SLAC Stats
Data analysis

Figure 3.15: Fault injection loop.

The first phase involves the execution of a structured test sequence to initialize the algorithm. This requires performing an initial verification using two predefined test patterns: *Verify-0* and *Verify-1*. These tests establish the baseline fault set by identifying bits that are permanently stuck-at `0` or `1`. The outcome of this phase provides a reference for all subsequent tests, enabling an efficient fault encoding through differential computation.

Once the initialization is complete, the primary objective is to verify that the fault encoding correctly represents the observed errors. This is achieved by systematically applying mixed test patterns and ensuring that the algorithm correctly encodes only the changes relative to the baseline faults. The expected behavior is that each new test iteration produces an encoded fault set that can be reliably reconstructed by applying a bitwise XOR operation with the baseline. To be in the most realistic scenario, faults from production devices have been used. The list of fault files are divided into a hierarchical structure: lot, wafer, die, test ID.

When the injections are performed, this hierarchy must be taken into considerations. The injections will be grouped per device, where the first injection will be the setup of the Delta-SLAC, and from the second onwards they will all be considered with the difference computation mechanism. To each dumped output faults (Figure 3.16) a specific set of input faults (Figure 3.17) will be associated.

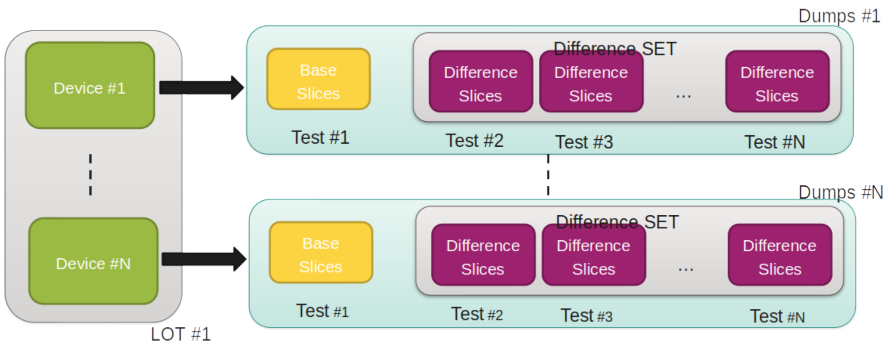

Figure 3.16: Output data of the injection loop.

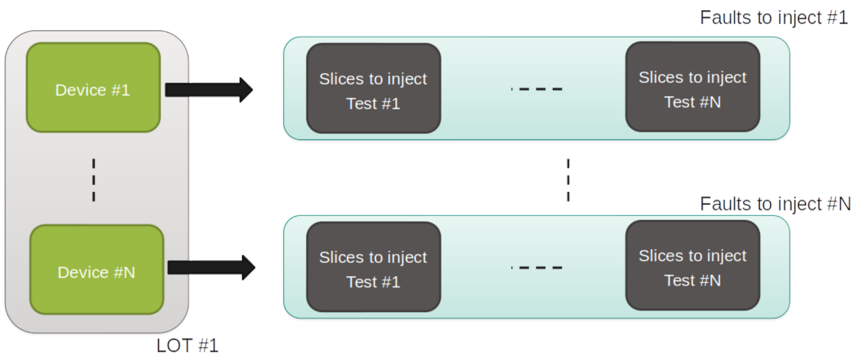

Figure 3.17: Input data of the injection loop in hierarchical style.

To confirm the correctness of fault retrieval, an off-line reconstruction process is executed. Given the output of an arbitrary n^{th} test, the base fault sets are filtered according to the specific test pattern used. These filtered sets are then merged and then a bit-wise XOR is performed with the newly detected faults to verify that the reconstructed fault map matches the original injected errors. Any discrepancy at this stage would indicate a flaw in either the encoding or retrieval mechanism, requiring further refinement of the algorithm.

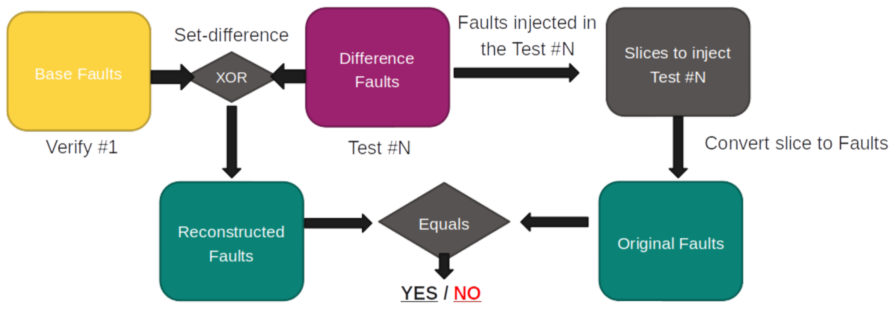

Figure 3.18: Validation schema after the fault injection loop.

Overall, the verification of the Delta-SLAC algorithm provides strong guarantees on its accuracy, correctness and effectiveness in fault encoding. The structured approach to verification ensures that the algorithm operates reliably under diverse testing conditions, making it a robust solution for fault analysis in eMemories.

Chapter 4

Experimental Results

4.1 Delta-SLAC Results on Front-End Testing Dataset

Experimental tests where performed on real SoC devices with RRAM technology. The environment set up follows the same steps discussed in the Section 3.8. The tests where performed using data coming from Front-End testing operations on real production silicon wafers. The data stored in databases was used to validate the premises that, over multiple test steps inside a test-flow, following the activation of the Delta-SLAC algorithm there would be a reduction in memory usage.

Wafer	SLAC mem. [KiB]	Δ-SLAC mem. [KiB]	Mem. saved [%]
25	50.7	1.3	97.4
24	50.4	39.6	21.5
13	153.0	54.0	64.7
18	26.0	24.8	4.7
15	26.3	24.8	5.6
14	72.4	54.2	25.0
22	189.5	118.2	37.6
21	77.8	69.6	10.5
20	25.9	17.2	33.4
17	53.6	59.6	-11.4
19	142.2	101.5	28.7
16	1.7	1.1	33.3
12	52.4	14.9	71.5
23	225.9	70.5	68.8

Table 4.1: Lot 1: 642 devices with at least one fault.

In order to perform the space occupation analysis of the SLAC and the Delta-SLAC some minor adjustments had to be performed. Specific tests had to be chosen, per device, for the setup phase of the Delta-SLAC. The test are performed on series of four different tests, these will be the final test-flow to be re-run per each device. In this test-flow the first two tests will inject the memory with a “Solid Zeros” and a “Solid Ones” patters, which will be the two setup steps for the Delta-SLAC structures. The last two tests will again inject a “Solid Zeros” and a “Solid Ones” patterns, into the eMemory. All the results taken will not consider the memory occupation of the first two steps, this is due to the fact that, even when the Delta-SLAC is enabled, the dumped faults will be equal to real faults read during those steps. During the setup phase the Delta-SLAC only initializes its structures without performing difference computation. The dumped faults during the setup phase will act as the base faults, and they will be used during the off-line operations to compute back the original faults of the successive test result dumps, by using the set of difference faults.

Table 4.1 presents a comparative analysis of memory consumption between the standard SLAC approach and the proposed Delta-SLAC algorithm across different wafers of the “Lot 1”. The table reports the cumulative memory usage in KiB for both methods, along with the percentage of memory saved by Delta-SLAC relative to the SLAC algorithm.

The results indicate that Delta-SLAC reduces memory consumption in most cases. Notably, for wafer 25, the required memory is reduced from 50.7 KiB to only 1.3 KiB, achieving a memory saving of 97.4 %. Similarly, wafers 12 and 23 exhibit reductions of 71.5% and 68.8%, respectively, demonstrating the effectiveness of Delta-SLAC in compressing fault data.

However, certain cases, such as wafers 17 and 18, show minimal or even negative savings. For instance, wafer 17 shows an increase in memory usage of 11.4%, suggesting that in some scenarios, the distribution of the faults will not remain regular across subsequent test runs. This can be attributed to the faults appeared in setup phase to not conform to a stuck-at fault model, or that the eMemory is progressively failing with the execution of the test-flow.

Table 4.2 shows results referred to “Lot 2”, which are very similar to Table 4.1, but slightly worse. By excluding the wafer with low memory occupation, and the memory saving is considered again, the maximum achieved only a 58.1 % by the wafer 11. The other wafer have a lower, or close to zero, memory saving percentage.

If a deeper inspection is taken to the “Lot 2”, a noticeable amount of devices had physical defects, showing very high rates of failure in massive blocks.

A more in depth analysis on the physical distributions of the faults can highlight the reasons why a difference computation would cause an increase in memory usage. Two set of images are provided showing the eMemory state, during the first and second reading test conducted on the device.

Wafer	**SLAC mem. [KiB]**	**Δ-SLAC mem. [KiB]**	**Mem. saved [%]**
7	1.4	0.4	68.7
3	0.5	0.3	50.0
2	12.9	12.9	0.3
10	2.4	1.7	29.3
1	4.0	1.5	63.5
8	22.2	12.4	44.2
9	3.1	2.1	31.9
4	0.2	0.5	-144.1
12	1.6	2.1	-29.4
11	25.1	10.5	58.1
5	20.7	21.1	-2.0
6	27.6	27.6	-0.2

Table 4.2: Lot 2: 298 devices with at least one fault.

Figure 4.1 shows two different eMemory states, where a prominent chunk of memory exhibits failures that can be seen throughout the duration of the two tests. The differences between the first and the second test, are some newly failing bit-lines that appear only in the second test (near the 5650th bit-line number). In fact during the second test the SLAC will use 11.9 KiB to store the information, while the Delta-SLAC only takes 0.9 KiB, caused by the newly found bit-lines.

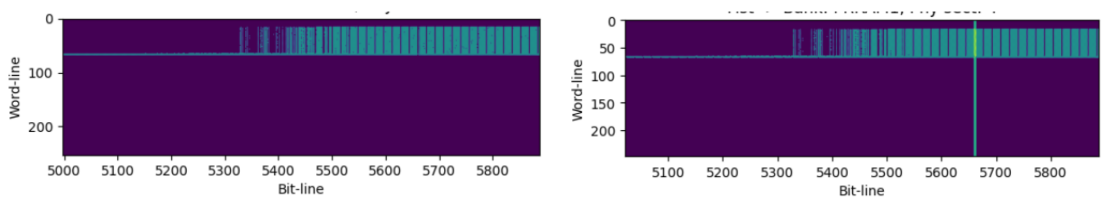

Figure 4.1: Two memory fault bitmaps of the same bank, where the majority of the faults are stuck-at, only some bit-lines appear in the second test (right), compared to the first test (left). Memory occupation: Setup 11.5 KiB; SLAC 11.9 KiB; Δ-SLAC 0.9 KiB.

Figure 4.2 shows opposite circumstance to that of Figure 4.1, in this case the memory state is very different from the two subsequent tests, in fact it is possible to see a more progressive failure of the memory along the test-flow. The few isolated bit-lines become compact block of failures during the second reading test, causing a spike in difference, resulting in 18.0 KiB of memory usage by the Delta-SLAC.

Figure 4.3 and Figure 4.4 show a comprehensive picture of the Delta-SLAC execution timings, compared to the SLAC. Both pictures clearly show that with

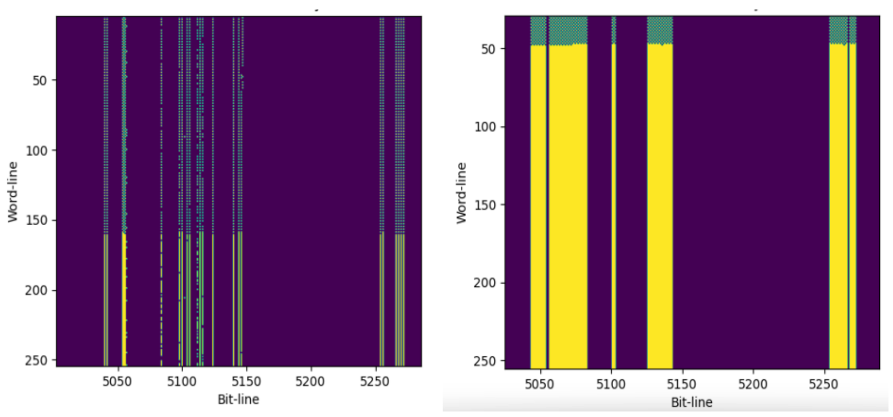

Figure 4.2: Two memory fault bitmaps of the same bank, where in second test (right) the bit-lines starts to progressively fail more than the first test (left). Memory occupation: Setup 5.0 KiB; SLAC 19.9 KiB; Δ-SLAC 18.0 KiB.

few faults the Delta-SLAC outperforms the SLAC timings, while when the number of faults surpasses the thousands thresholds, the Delta-SLAC timings becomes progressively worse.

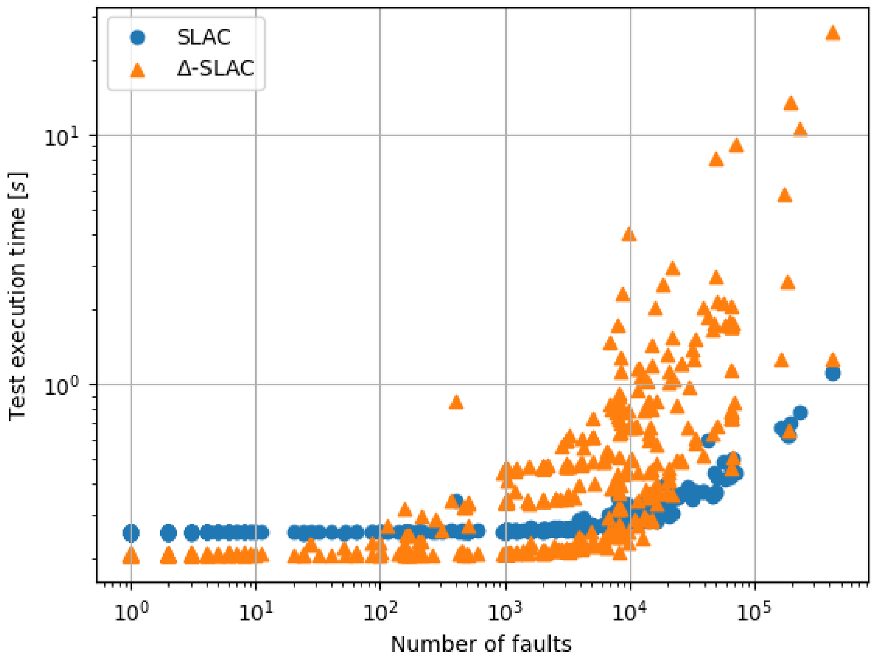

Figure 4.3: Lot 1 timing analysis of the difference computation execution timing.

It is possible to conclude that when a big chunk of the memory contains stuck-at

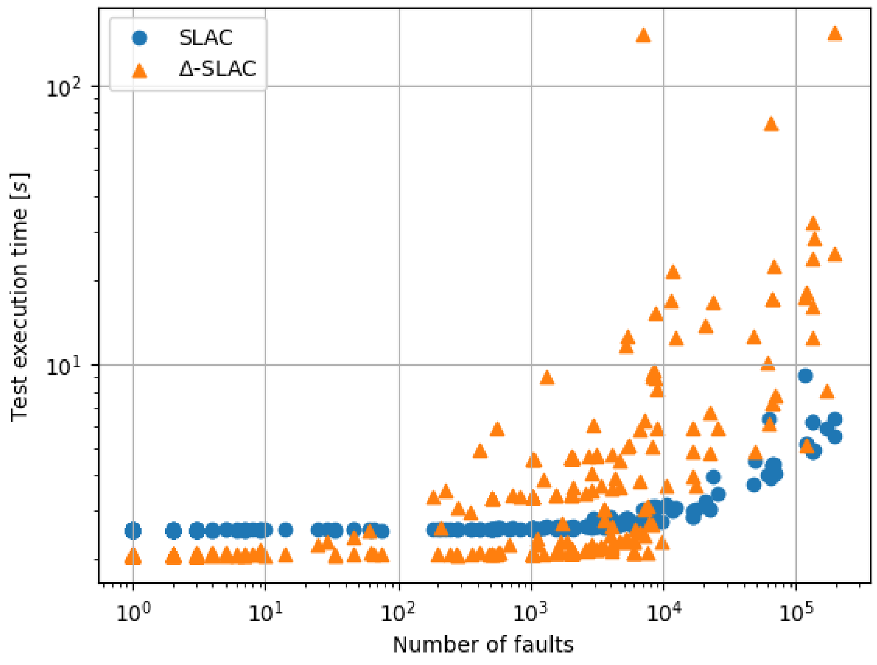

Figure 4.4: Lot 2 timing analysis of the difference computation execution timing.

failures, repeating across multiple reading tests, the memory saving is very effective. Overall, the results confirm that the Delta-SLAC is highly efficient in reducing memory requirements, particularly for wafers with a high number of stuck-at faults. The variations in savings highlight the dependence of the algorithm's effectiveness on the stuck-at faults, indicating that Delta-SLAC is most beneficial in scenarios where fault occurrences follow structured and constant distributions.

Chapter 5

Conclusions and Future Works

The Delta-SLAC algorithm presents a novel and efficient approach to fault data compression in eMemory systems. By leveraging a difference encoding mechanism, the Delta-SLAC algorithm significantly reduces the memory footprint required to store fault bitmap information, addressing a critical challenge in modern semiconductor testing. The experimental results demonstrate substantial improvements in memory efficiency, with savings exceeding 95% in optimal cases. This reduction not only minimizes storage requirements but also enhances the overall scalability of fault management systems, over long-running test-flow operations.

Through the integration of optimized data structures such as array-based augmented interval trees and array-based partitioning, the Delta-SLAC algorithm enables efficient fault processing and storage. The algorithm effectively encodes fault into a structured and searchable format, allowing for rapid access and reduced computational overhead. The use of binary search techniques within the interval tree further ensures logarithmic time complexity for fault lookups, making Delta-SLAC highly suitable for large-scale memory arrays.

Despite its advantages, the study highlights certain limitations where the Delta-SLAC may introduce additional overhead in scenarios with sparse or highly fragmented fault distributions. Future work may focus on adaptive encoding techniques that dynamically switch between difference computation and absolute representations based on fault difference set magnitude, further optimizing memory usage. Additionally, optimization on algorithm level could, and a switch interval axis comparison can improve the search timings.

In conclusion, Delta-SLAC provides a robust and scalable solution for fault data compression in embedded memories, demonstrating its potential for integration into industrial testing frameworks. Its ability to efficiently store and retrieve fault information with minimal memory overhead marks a significant advancement in memory fault analysis, paving the way for more efficient and cost-effective testing methodologies.

Bibliography

[1] G. Insinga, M. Battilana, M. Coppetta, N. Mautone, G. Carnevale, M. Giltrelli, P. Scaramuzza, and R. Ullmann. «Density-oriented diagnostic data compression strategy for characterization of embedded memories in Automotive Systems-on-Chip». In: *2023 IEEE European Test Symposium (ETS)*. 2023, pp. 1–6. DOI: `10.1109/ETS56758.2023.10174126` (cit. on p. 4).

[2] Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest, and Clifford Stein. *Introduction to algorithms.* eng. Third edition. Includes bibliographical references and index ; Here also later published, unchanged reprints of the 3rd edition. Cambridge, Massachusetts ; London, England: MIT Press, 2009, xix, 1292 pages. ISBN: 978-0-262-03384-8 and 978-0-262-53305-8 and 0-262-53305-7 (cit. on p. 34).