POLITECNICO DI TORINO
Repository ISTITUZIONALE

SIERRA—Simulation environment for memory redundancy algorithms

Original
SIERRA—Simulation environment for memory redundancy algorithms / Scionti, Alberto; Mazumdar, Somnath; DI CARLO, Stefano; Hamdioui, Said. - In: SIMULATION MODELLING PRACTICE AND THEORY. - ISSN 1569-190X. - STAMPA. - 69:(2016), pp. 14-30. [10.1016/j.simpat.2016.08.008]

Availability:
This version is available at: 11583/2649948 since: 2016-09-19T11:18:20Z

Publisher:
Elsevier

Published
DOI:10.1016/j.simpat.2016.08.008

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

12 October 2024

SIERRA – Simulation Environment for Memory Redundancy Algorithms

Alberto Scionti[a], Somnath Mazumdar[b], Stefano Di Carlo[c], Said Hamdioui[d]

[a]*Istituto Superiore Mario Boella (ISMB), Torino, Italy*
[b]*University of Siena, Dept. of Information Engineering and Mathematics, Siena, Italy*
[c]*Politecnico di Torino, Dept. of Control and Computer Engineering, Torino, Italy*
[d]*Delft University of Technology, Computer Engineering Group, Delft, The Netherlands*

Abstract

Extreme-scale computer systems take advantage of large arrays of general-purpose multicore processors coupled with specialized manycore accelerators. In order to support complex applications and correctly feed such processing elements, increasingly larger memory cores are integrated at different levels of the hierarchy. However, the adoption of increasingly aggressive manufacturing processes makes the memory sub-system particularly sensitive to faults. Error correcting codes (ECCs) allow the memory to recover from faults at run-time without interfering with the application execution. However, due to the loss of performance introduced every time an error must be corrected, the persistence of faults requires a more radical repair approach in which faulty cells are physically replaced by spare ones. Memory redundancy analysis (MRA) algorithms are used to drive the allocation process of spare resources. Many one-dimensional and two-dimensional MRAs have been proposed, but tools for evaluating their recovering capability are still not well established. This paper presents SIERRA, a simulation environment for precisely evaluating the repair efficiency of an MRA considering different fault signatures and faulty memory configurations. Our simulation engine provides a precise estimation of the MRA quality by analyzing the behavior of the MRA on several faulty memory configurations. To this end, different parameters such as the area of the memory blocks and the defect density are taken into account. The evaluation of the quality of an MRA takes into account its repairing capability, the power consumption derived from its execution, and the area overhead. Thanks to the use of a database for storing information, our tool is able to speed-up the simulation process by distributing it among several nodes. All these features make SIERRA essential in supporting the design of next-generation high-performance computers.

Keywords: Software simulator, Computer memory, Redundancy algorithms, Repair memory, BISR structure.

1. Introduction

Next generation of extreme-scale and high-performance computers are expected to execute very complex applications with an unheard-of level of performance. An exa-FLOPS machine will be able to perform up to 10^{18} floating point operations per second. Such level of performance requires the massive adoption of specialized processing elements, ranging from general-purpose multicore processors to specialized manycore accelerators (e.g., GPGPUs, FPGAs, etc.). Similarly, an increasing amount of memory and storage will be integrated at all levels of the memory hierarchy in order to correctly feed this large number of processing elements [1]. To this end, different kinds of semiconductor memory devices will be used, each exploiting aggressive manufacturing technologies and architectures. For instance, the International Technology Roadmap for Semiconductors (ITRS) [2] indicates that the area devoted to memory blocks will exceed 90% of the whole chip area in future designs.

The International Exascale Software Project (IESP) roadmap [3] has identified the most relevant issues that must be solved in order to increase performance of current high-performance systems by a factor of 10^3. Among the others, the reliability of the whole system, as well as power and energy issues are the main concerns to deal with.

Email addresses: `scionti@ismb.it` (Alberto Scionti), `mazumdar@dii.unisi.it` (Somnath Mazumdar), `dicarlo@polito.it` (Stefano Di Carlo), `s.hamdioui@tudelft.nl` (Said Hamdioui)

In this context, semiconductor memories represent a critical building block. In a computer system equipped with millions of cores, components' failure becomes the norm rather than the exception [28]. Since the memory layout is regular and dense, memory cores are highly sensitive to manufacturing defects, as well as to failures that emerge during operational activity. Soft-errors are generally caused by the interaction of high-energy particles with the silicon substrate, thus producing a flip in a cell of the memory device. Conversely, hard faults may manifest during operational activity of the component. They are caused by undetected manufacturing defects, physical stress due to working conditions (e.g., presence of high power dissipating hot-spots), device aging, etc. Conventionally, ECCs are popular techniques used for recovering from soft-errors. Parity bits are stored along with information bits, allowing to detect and correct erroneous information. However, if the fault persists, a different approach has to be activated. To this purpose, memory blocks are equipped with spare elements (e.g., spare rows, spare columns, spare cells, etc.) that can be programmed, using a dedicated circuitry, to replace faulty elements. The map of the faulty memory cells is analyzed resorting to dedicated *memory redundancy analysis* (MRA) algorithms, which are able to identify the most efficient allocation schema for the redundant elements. Generally, MRAs are designed to exploit two or more categories of spare elements (e.g., spare rows and columns, spare rows and cells, etc.) [37, 38, 39, 45]. The problem of finding the best allocation sequence of spare resources is known to be an NP-complete problem [4].

In this complex scenario, differently from standalone memory chips that can be efficiently tested and repaired resorting to standard Automatic Test Equipments (ATEs), embedded memories need to be equipped with embedded logic for memory test and repair. Focusing on the memory repair logic, to address this problem, several MRAs whose hardware implementation introduce small area overhead have been proposed [5, 6, 7]. However, the MRA repair capability is influenced by the specific defect distribution, which is a characteristic of the manufacturing technology. Thus, tools to early estimate the efficiency of an MRA, when applied in a specific technological scenario, are crucial for properly choosing the best repair allocation strategy and to maximize the reliability of the overall memory subsystem.

This paper presents a tool named SIERRA which provides a general and flexible environment to precisely evaluate the efficiency of an MRA through simulations. In particular SIERRA provides the following contribution:

- It allows to simulate the application of an MRA on a very high number of faulty memory configurations;
- It automatically generates faulty memory configurations using realistic fault models and realistic defect distributions. To this end, faults arising in spare elements are allowed to occur;
- It is coupled with an external defect simulator [8, 9] or a memory fault simulator [10, 29] to assess the capability of the test-repair solution to recover from hard faults;
- It measures the efficiency of the analyzed MRA in terms of repair rate, power consumption, and area overhead;
- It provides a high-level language to easily describe any generic MRA;
- It distributes simulations on a set of computing nodes to speedup the MRA design space exploration;
- It exports results in a graphical way, so that the MRA behavior can be easily captured.

The paper is organized as follows: Section 2 discusses the mathematical formulation of the memory repair problem, and recent works on the analysis and simulation of MRAs. Section 3 details about the general architecture of the proposed tool, the way of modeling the memory array blocks, and the fault detection sequences. This section also provides information regarding the way SIERRA generates and optimally stores faulty memory models. A dedicated high-level description language used to describe the behavior of an MRA is presented in section 4. This section also illustrates the way MRAs are simulated and their efficiency is computed. A validation campaign of the SIERRA capabilities is provided in section 5. Finally, section 6 outlines the main contributions of the work and concludes the paper.

2. Background

From the repair standpoint, a $R \times C$ faulty memory array can be modeled as a *fault bit map* (FBM), i.e., a $R \times C$ binary matrix whose asserted elements identify the faulty cells. For the sake of simplicity, let s_r and s_c be the number

of available spare rows and spare columns respectively (the problem formulation remains valid in case of other types of spare elements, such as single cells or memory blocks). The memory repair problem consists in finding, if possible, a set S_r of rows and a set S_c of columns to repair, such that $\forall \text{FBM}_{i,j} = 1 \Rightarrow i \in S_r \vee j \in S_c$, and for which $s_r \geq |S_r| \wedge s_c \geq |S_c|$. This problem has been demonstrated to be NP-complete by transforming it in polynomial time to the bipartite-graph clique problem [4]. Therefore, suboptimal heuristics are usually employed to solve it. Since memory devices can be arranged in multiple banks, resources can be also shared among banks. In that case, the repair problem becomes easier.

Memory redundancy analysis or shortly memory redundancy algorithms (MRAs) are algorithms devoted to analyze the sequence of faulty cells, and to allocate spare resources according to a specific scheme. MRAs are generally enough simple to be implemented as a circuit. Such circuits are referred to as *Built-in Repair Analysis* (BIRA) or *Built-in Self-Analysis* (BISA). Along with *Built-in Self-Test* (BIST) and *Built-in Self-Diagnosis* (BISD) circuits, they provide hardware solutions to test and repair memory blocks directly on the chip, thus avoiding the usage of expensive external testers. BIST circuits apply a sequence of read/write operations to the memory cells in order to excite and detect faulty behaviors [47, 48]. The order in which the operations are applied strictly depends on the specific test algorithm that is implemented. BISD circuits add the capability of providing the address of the faulty cells, as well as an identification of the fault type [35, 36]. Once a spare resource has been selected and allocated, a *Built-in Self-Repair* (BISR) circuit effectively replaces the faulty cell(s). BISR circuits replace faulty cells by reconfiguring the address decoders by means of a content addressable memory (CAM) or embedded fuses. To summarize, an MRA algorithm can be implemented as a dedicated circuit directly on the memory chip. This circuit is composed of an analysis module (BIRA) and a repair module (BISR), which operate in conjunction with a BISD/BIST solution that provides the address of the faulty cells found during the test phase. Figure 1 depicts the organization of such testable and repairable memory block.

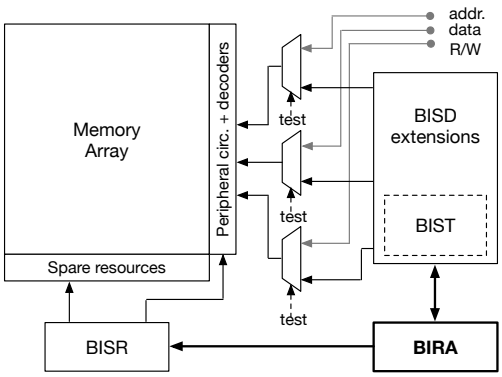

Figure 1: The organization of a testable and repairable memory array equipped with a BISD/BIST diagnostic module, a BIRA module for allocating spare elements, and a BISR circuit for reconfiguring decoders and peripheral memory circuits.

Some simulation tools for evaluating the efficiency of an MRA have been developed and proposed by Virage Logic [11, 12], National Tsing Hua University [13, 14, 15, 16, 17], and others [4, 19, 20, 21, 22]. One of the main drawbacks of the proposed approaches is that, to reduce the simulation complexity, they limit realistic faults into memory devices, and do not enable the MRA evaluation under an elevated number of different faulty memory configurations. In fact, injected defects are generally assumed to only lead the creation of single faulty cells, single faulty rows, or single faulty columns. Also the order in which the faulty addresses are presented to the MRA is limited to a linear/sorted order (e.g., from address $\langle 0\text{x}0\ldots 000\rangle$ to address $\langle 0\text{xF}\ldots \text{FFF}\rangle$). These limitations may lead to incorrect results. Nakahara et. al [23] state that, even a simple MRA can achieve fairly good performance if a simple failure often occurs. Furthermore, these tools are designed to evaluate the MRA efficiency only in terms of FMB repair rate, while power consumption and area overhead are generally not considered.

Huang et al. [24] propose an MRA evaluation framework used to optimize the insertion of shared BIRA+BISR structures. However, the simulation environment is limited to a small set of fault types and the simulation approach is time-consuming. Moreover, the framework has been validated using a relatively small number of faulty memories

considering only an over-simplified fault distribution.

Sehgal et al. [22] present a significant step forward, by proposing a tool for memory array yield analysis. Although provided results show accurate MRA efficiency estimations, the tool still presents significant limitations. First, both fault-free and faulty memory configurations are generated during the analysis. Since fault-free configurations do not require the execution of the MRA, they should be avoided. Furthermore, only three MRAs have been implemented and simulated. From this standpoint, although the authors claim that other redundancy algorithms can be simulated, no flexible mechanisms are described to support this important extension. Finally, area and power overhead estimation is also omitted.

Chao et al. [25] presented a purely statistical analysis methodology, from which deriving the yield estimation of a memory, given a selected MRA. The advantage of the methodology is that it only requires scalable mathematical computations, avoiding time-consuming simulations. However, although the yield estimation provided by the approach is fairly accurate for initial design evaluations, it presents many limitations: (i) the effective behavior of the MRA (especially when it is coupled with a test algorithm) is not taken into account, (ii) only a single defect distribution is used to drive the statistical analysis, (iii) only single cell, single row and single column signatures, along with a limited number of faulty memory configurations are used, and (iv) no area and power overhead estimation is included in the efficiency computation.

Recently, INFORMER [18] has been proposed as a fully automated tool for helping high-level designers to estimate memory reliability metrics rapidly and accurately. Unlike previous attempts, it integrates a detailed model for the area and power of the memory under analysis. The main limitation of the tool is represented by the need of running circuit-level simulations. In fact, this kind of simulations are time-consuming and limit the capability of the tool of exploring a large number of memory configurations. Moreover, it has been tailored for SRAM circuits, thus making its use not possible in case of different memory architectures (e.g., DRAMs). FaultSim [34] has been recently proposed as a fast simulation tool for the evaluation of the reliability of a memory system. It is a fast and configurable tool, where Monte Carlo simulations are used to assess the reliability of the memory system. Similarly to INFORMER the tool is tailored to simulate correction codes such as BCH-1 and ChipKill, and no easy way to implement new ECC codes and repair schemes is provided, unless to modify the simulator.

Given this premise, the simulation environment we propose represents an attempt to provide an automated solution to most of the limitations of the analyzed simulation tools and methodologies.

3. SIERRA Architecture

SIERRA is a highly modular and composable simulation environment. To effectively design such a tool we took into account a set of requirements we considered fundamental to precisely estimate the efficiency of an MRA. In the following we shortly list the most important ones:

1. *Fast quantitative evaluation of the MRA efficiency*: to take into account different types of faults that can arise in a memory device, the simulation environment has to analyze a high number of faulty memory configurations, keeping the required computations as simple as possible. Information regarding area overhead and power consumption of the repair circuitry must be included in the efficiency evaluation.
2. *Simulation speedup*: when possible, apply techniques to speedup the simulation process (e.g., distributing memory configurations to analyze among several simulation nodes, skipping simulations of theoretically unrepairable faulty memory configurations, etc.).
3. *Simulation of realistic faults*: only realistic faults (i.e., those that can be simulated using a circuit-level memory model or are part of manufacturing yield data) must be considered to avoid over– or under– estimation of the MRA efficiency [41]. Conversely, all fault models that have not been observed on real devices, can not be tested, or whose behavior can not be reproduced through electrical simulations must be avoided.
4. *Flexible memory array and redundancy circuit configuration*: used memory models must be highly configurable (e.g., partitioning of the array in banks, definition of private and shared redundancy elements, etc.) to reproduce working conditions of real devices.
5. *Simulation of both automatically generated and custom faulty memory configurations*: faulty memory configurations should be automatically generated by the simulation environment. However, the user should be able to define custom configurations known to be hard to solve with the MRA under analysis.

6. *Additional information available when MRAs fail to repair a faulty memory configuration*: whenever MRAs fail on a faulty memory configuration, information regarding the locations of consumed redundancy resources are also useful to know whether they have been efficiently allocated or not.

In order to satisfy these requirements, we implemented SIERRA around a standard database management system (DBMS). It serves as the main storage repository, keeping all the information regarding faulty memory configurations and simulation results in a structured form. Using a standard DBMS it is possible to decouple the actual data storage from the simulation engine (i.e., it is possible in this way to distribute the access requests to the database across various nodes without interfering with simulation logic). A memory fault simulator (i.e., a circuit-level memory defect simulator), as well as an area and a power consumption estimation tool, can be plugged into SIERRA, providing a more precise evaluation of the MRA efficiency. Simulation flexibility is obtained by decoupling the MRA description from its actual execution. To this end, a compiler translates a high-level MRA description into a bytecode representation that is subsequently executed by a virtual processor integrated into the tool.

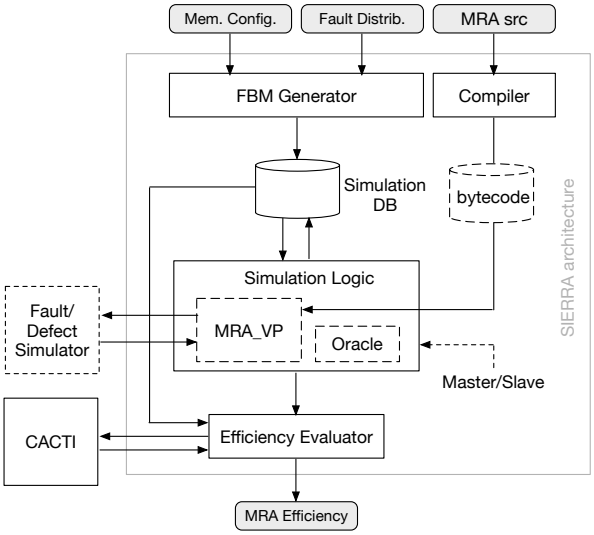

Figure 2: SIERRA architecture: the fault bit map (FBM) generator is in charge of generating faulty memory configurations, similarly a compiler translates a high-level MRA description into a bytecode. A virtual processor (MRA_VP) is responsible for the execution of the bytecode, providing the information required to evaluate the MRA efficiency.

Figure 2 shows the internal organization of SIERRA. It receives three main information:

1. The characteristics of the memory to repair: to represent real devices, SIERRA enables the specification of the memory array in terms of the number of rows, number of columns, and banks. To this end, for each bank, the range of rows and columns belonging to it is specified. Similarly, the set of spare elements is given by specifying the type of the resource (spare rows, spare columns, single cells, etc.), and marking them as private for the bank or shared among all the banks.
2. The distribution of the defects in the memory array and their characteristics obtained either from technological data or from circuit-level simulations.
3. A high-level description of the memory redundancy algorithm under analysis, by means of a dedicated language. The language is compiled to produce an executable bytecode and executed many times (one time for each FBM to analyze).

The evaluation of the MRA is performed resorting to an extensive simulation campaign, where a large set of randomly generated and user provided FBMs are used. The execution of the MRA on a given FBM is compared with the execution of the *Oracle* allocation algorithm. This is a memory redundancy analysis algorithm designed to explore all possible assignments of the spare resources in order to identify the best allocation sequence (worth to note that albeit it is optimal in terms of repair rate, it is not possible to implement the Oracle behavior in hardware due to its

large area and power consumption overheads). To better assess the efficiency of the MRAs under analysis, SIERRA provides an interface for interacting with external tools such as CACTI [26] and a memory defect/fault simulator [8, 9, 10]. The former allows SIERRA to consider the area and power efficiency of the MRA, while the latter provides a more precise estimation of the efficiency by simulating the memory test algorithm providing a realistic detection order of the faulty memory cells. The result of this massive simulation campaign is graphically exported in a 3-D plot. The sections that follow will detail the characteristics of the different modules composing our simulation environment.

3.1. Statistical FBM generation

SIERRA supports both the usage of user-defined memory configurations (e.g., hard to repair faulty memory configurations) and the automatic generation of fault bit maps, starting from the distribution of physical points of failure. A physical point of failure (hereafter indicated simply as a fault) identifies the location within the memory device where a defect or a stress condition occurred causing an erroneous behavior of the memory. Faults caused by defects or stress conditions can affect one or more cells. From the memory repair standpoint, it is important to know the number of cells that are affected by a faulty behavior. Previous works demonstrated through electrical simulations that spot defects may cause misbehavior of a single cell, as well as of multiple cells [41, 42, 8]. Similarly, parasitic capacitive variations that may arise between adjacent cells, as well as in the single cell, may lead to faulty behaviors [40]. Other works have demonstrated that defects occurring in peripheral memory circuits (e.g., address decoders) can be modeled as equivalent faults in the memory array [43]. Fault signatures are used to abstract the behavior of a faulty cell, and each fault may generate different signatures (i.e., clusters of faulty cells) in an FBM. SIERRA considers eight fault signatures, here referred to as *fault classes* (FCs), namely: single cell fault (SC), double cell fault in a row (DCR), double cell fault in a column (DCC), quadruple cell fault (QC), single row fault (SR), single column fault (SC), double row fault (DR), and double column fault (DC) (see Figure 3). Fault classes SR, SC, DR, and DC are linear classes since they affect a full row/column or a cluster of two rows/columns. Their specific length may change from a minimum number of cells to the full length of the row/column. Users can define the list of locations (i.e., cells, rows, columns, etc.) within the memory array that are affected by a fault, according to the fault signatures described above.

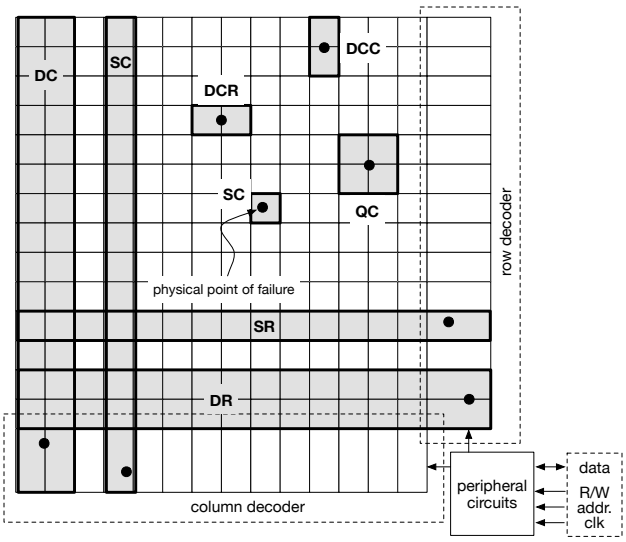

Figure 3: An example of a single bank memory with an instance of each FC type showing the different bit map signatures. The effect of faults in the peripheral circuits are reflected by equivalent faults in the memory array and/or in the row/column decoder.

The number of instances of each FC (I_{fc}) injected into a FBM is computed as follows:

$$I_{fc} = \frac{\delta \cdot A^{mem} \cdot P_{fc}}{N_{fc}} \tag{1}$$

Equation 1 takes into account: (i) the overall density of physical faults (δ) obtained from a selected fault distribution such as Poisson, Polya-Eggenberger, Gamma, Negative Binomial, Uniform, etc. [44, 46], (ii) the area of the memory array and row/column decoders (A^{mem}), (iii) the probability of occurrence of the FC (P_{fc}) obtained from external sources such as manufacturing data, and (iv) the number of possible instances of the selected fault class in the FBM (N_{fc}).

Since each FBM is generated by randomly selecting the location of the desired FCs, SIERRA can not guarantee that the generated configuration will be repairable given the available amount of spare elements. Each FBM is therefore preliminary categorized into one of two classes: *theoretically repairable FBMs* (TRFBMs) or *theoretically unrepairable FBMs* (TUFBMs). This operation is performed by simulating the application of a two phases optimal MRA, as proposed in [27]. This algorithm, referred as the Oracle, is used in our simulation as a golden comparison of other suboptimal MRAs. SIERRA enables the user to specify the number of TRFBMs to generate throughout a dedicated parameter (Gen_{tr}) in the configuration file, while a second parameter (Gen_{ef}) establishes an upper bound on the number of generation attempts required to reach Gen_{tr}, thus avoiding infinite loops. The application of the Oracle is one of the most computationally intensive tasks performed by our simulation environment. To reduce the computation time, SIERRA performs a preliminary analysis of the generated FBM counting the number of independent FC instances, i.e., instances of a fault class that do not share rows or columns. If their number is greater than the number of available spare elements, the FBM can be marked as TUFBM without requiring the execution of the Oracle.

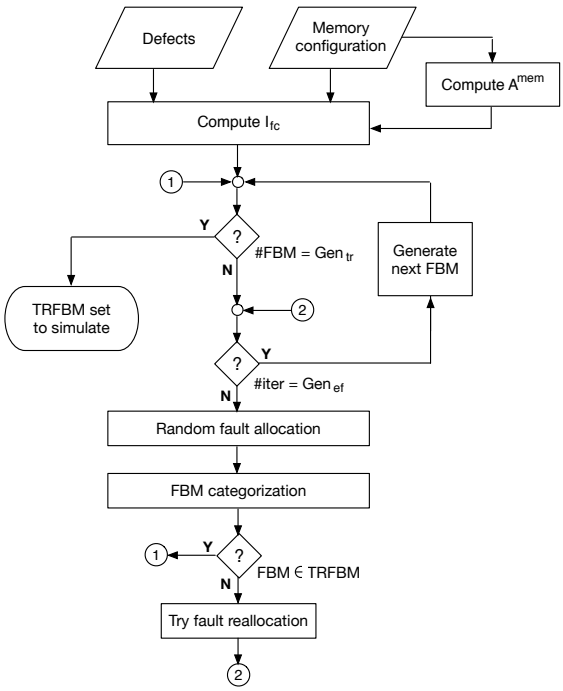

Figure 4: The flowchart describing the sequence of operations performed to generate the required set of theoretically repairable FBMs (TRFBM).

Figure 4 shows the flowchart used to generate the set of required TRFBMs. Since spare elements are simply additional storage elements (single cells, rows, columns, etc.) of the memory array, faults may arise in these elements as well. SIERRA takes this aspect into account, thus including these elements in the fault injection process. Whenever a spare element is marked as faulty, the number of available spare elements of that type is decremented by one.

3.2. Optimized FBM representation

Generating a large set of FBMs representing real large memories may produce an elevated amount of data that must be stored and processed. This may have a negative impact on the performance of the MRA analysis. To cope with this problem SIERRA uses an optimized FBM representation, where only the positions of faulty elements are

stored in the internal database (generally, the number of faulty cells is much lower than the whole size of the memory array).

Resource					
ID_FBM	Res-ID	Bank	Type	Oracle	MRA
int	*int*	*int*	*int*	*bool*	*bool*

Simulation							
ID_FBM	Bank	Row	Col	Oracle	Res-IDo	MRA	Res-IDm
int	*int*	*int*	*int*	*bool*	*int*	*bool*	*int*

Result				
ID_FBM	REP_{oracle}	REP_{mra}	$FAULTS_{oracle}$	$FAULTS_{mra}$
int	*bool*	*bool*	*int*	*int*

```
ID_FMB  = identifier of the FBM

Res_ID  = identifier of the spare resource

Bank    = memory bank identifier

Type    = spare resource type
          (e.g., column, row, etc.)

Oracle  = if true, the resource or the action is
          used or made by the Oracle

MRA     = if true, the resource or the action is
          used or made by the MRA

Row     = the row address of a faulty cell

Col     = the column address of a faulty cell

Res-IDo = the identifier of one of the available
          resources, that is used by the Oracle

Res-IDm = the identifier of one of the available
          resources, that is used by the MRA

REPoracle = if true, the current FBM has been
          repaired by the Oracle

REPmra  = if true, the current FBM has been
          repaired by the MRA

FAULTS oracle = the number of faulty cells
          repaired by the Oracle

FAULTSmra = the number of faulty cells
          repaired by the MRA
```

Figure 5: The organization of the simulation database. Three main tables are used to manage the experiments: the *resource* table describes the set of redundant resources available for the MRA and the Oracle to repair the memory, the *simulation* table maintains the description of the faulty memory configuration, and the table *result* collects efficiency results for both the Oracle and the MRA.

Figure 5 shows the organization of this database supporting simulations. It contains three main tables: (i) the *resource* table that describes the spare resources that both the MRA and the Oracle can use to repair the faulty memory configurations, (ii) the *simulation* table that contains the locations of the faulty cells and allows to keep track of the repairing process, and (iii) the *result* table that collects repairing metrics for the MRA and the Oracle. More specifically, redundant resources are described by their type (e.g., single cell, single row, etc.), their unique identifier, and two boolean flags that specify if the Oracle and the MRA have used that spare element during the repairing process.

The simulation table contains only the list of faulty cells, while two boolean flags (i.e., Oracle and MRA) are set if the corresponding repairing process (i.e., either the MRA or the Oracle) has allocated a spare element that covers the faulty cell. For instance, if there is a faulty cell located at row 1 and column 2 and the repairing process uses a spare row replacing the entire row 1, then the corresponding flag is set. An integer identifier of the spare resource is used to keep track of the allocation scheme (this information is kept separated for the Oracle and the MRA). Having faulty cells specified by splitting their address as the $\langle bank, row, column \rangle$ ease the process of generating FBMs; in fact the FBM generator can easily detect the case of FC instances that are completely screened by other instances, thus avoiding their actual placement. Finally, the result table allows collecting the repairing metrics for both the Oracle and the MRA. To this end, both the Oracle and the MRA have a boolean flag that is set if they succeeded to repair the FBM. Additionally, the number of repaired faults and the number of repaired cells are stored. All the three tables allow managing different FBMs at the same time by setting a unique identifier (ID_FBM) for each fault bit map.

3.3. Fault detection sequence

To properly analyze the efficiency of an MRA, the information contained in the FBM must be complemented with the information regarding the way the test algorithm identifies faulty cells. In fact, whenever the full FBM cannot be exported and provided to the MRA (as in the case of most BIRA solutions), the order in which the test algorithm detects the faulty cells may impact the repairing capability. To this end, SIERRA provides both an interface for an external memory defect/fault simulator, and an embedded way to scan the FBM reproducing the effect of the test algorithm. In the former case, the test algorithm is simulated on a dedicated tool [8, 9, 10, 29] using the faulty memory configuration exported by SIERRA. It resorts to a standardized format (XML file) for specifying the location of faulty cells and their associated fault models. In the latter case, SIERRA supports three detection orders for the selected FCs, as follows:

- *Ascending detection order* (O_a): the simulator starts the analysis from the fault class with the minimum number of faulty cells.
- *Descending detection order* (O_d): the simulator starts the analysis from the fault class with the maximum number of faulty cells.
- *Random detection order* (O_r): a random sequence is used to detect faulty elements in the FBM.

In all cases, different instances of the same FC are selected randomly. Each of the detection orders has its own characteristics, allowing the designer to identify lower and upper bounds in the repairing efficiency of the analyzed MRA. Detecting large fault clusters first (O_d) makes easier to repair the model, which may cause an over-estimation of the MRA efficiency. Conversely, late detection of large fault clusters (O_a) increases the repair complexity, which may underestimate the MRA efficiency. If FCs are randomly detected, the average MRA performance is captured.

4. MRA Description and Simulation

SIERRA exposes a high-level programming language interface enabling the designer to describe the behavior and the hardware resources used by any generic MRA [30]. By means of this language called Redundancy Analysis algorithM Language (RAM-L), the designer defines the set of resources used by the MRA in terms of variables (scalars and arrays), while the behavior is expressed resorting to general high-level statements, e.g., while-do loops, if-else statements, etc.

MRA structure in RAM-L (example)

```
# hardware resource declaration
resource
    register spare_cnt;
    array spare_res_no[2];
    register r_addr, c_addr;
    register spare_row, spare_col;
end
# algorithm description
behavior
    spare_res_no[0] := n_spare_rows;
    spare_res_no[1] := n_spare_cols;
    r_addr := row_address;
    c_addr := col_address;
    if ((r_addr = -1) AND (c_addr = -1))
    then
        exit;
    else
        skip;
    fi
    if (spare_cnt < (spare_row + spare_col)
    then
        if (spare_res_no[0] = 0)
        then
            if (spare_res_no[1] > 0)
            then
                spare_cnt := spare_cnt + 1;
                allocate_spare_col;
            else
                exit;
            fi
        else
            spare_cnt := spare_cnt + 1;
            allocate_spare_row;
        fi
    else
        exit;
    fi
end
```

Figure 6: The representation in RAM-L of a simple memory redundancy algorithm. A fixed allocation strategy ({R-R-... C-C-... }) is applied to repair faulty cells.

A generic RAM-L program is composed of two main sections: (i) the first part declares the variables (storage resources) used by the MRA, and (ii) the second part encodes the heuristic behavior. Figure 6 shows an example of MRA description using the RAM-L language. The code in the example assumes that both spare rows and columns are available for repairing faulty cells. It describes a simple repair strategy that first tries to repair detected faulty cells

using spare rows (line 31—34). Once all the spare rows have been consumed, it tries to repair the memory array using spare columns (line 22—30). The simulation of an MRA is a fault driven process. Therefore, special keywords are reserved to allow the MRA to interact with the simulator itself, in order to obtain specific information at run-time. The internal state of the simulated MRA can be exported to the simulation environment to share information about available spare resources and the address of the detected faulty elements. For instance, the address of the next faulty cell to repair is obtained by using *row_address* and *col_address* keywords. These keywords allow the simulator to assign the row address and column address to two MRA variables. Similarly, the MRA can obtain the current number of available spare resources (*n_spare_rows*, *n_spare_cols*). Keywords are also reserved to specify MRA actions. Possible actions are: interrupt the test and repair process, skip to repair the current faulty cell, repair the faulty cell with a specific spare resource, reset the test and repair process, or repair more than one faulty cell at a time, as required by most state-of-the-art MRAs [31, 32]. Furthermore, some MRAs require the ability to undo previously executed spare allocations and re-execute the fault detection sequence. Required actions are then represented by a pair of internal variables (i.e., BIST and BIRA) within the simulation engine. Table 1 shows the full list of supported actions.

Similarly to the VHDL and Verilog behavioral coding style, the RAM-L language has the capability of describing how hardware resources are used by MRAs. However, differently from HDL languages, our RAM-L is not designed to support concurrency. In fact, all actions performed by the MRA are simulated sequentially, even when the MRA structure allows their parallelization. For instance, the CRESTA algorithm ([33]) employs a set of parallel analyzers to repair the memory, each implementing a specific allocation sequence. The simulation of such MRA is performed sequentially, one analyzer at a time. Other differences with respect to standard HDL languages are the absence of time notion, and a limited set of data types. A RAM-L code does not contain any indication of the signals exchanged by actual hardware structures, and in particular there is no way to specify a clock signal. The RAM-L inability of describing signals is the main reason for which MRAs are simulated sequentially. In fact there is no possibility to define events (i.e., changes in the state of a signal) driving the concurrent activation of one or more hardware structures described by RAM-L code. Concerning data types, our RAM-L language supports a very limited number of types. Each resource can be either a scalar value, or an array. In any case, values encoded by variables are only integer numbers. Although this appears as a stringent limitation, it is important to highlight that the class of simulated algorithms only deal with discrete resources. To this end the number of faulty cells, as well as the number of available spare resources can be easily represented with an integer number. Finally, it is worth noting that RAM-L supported constructs are neither designed to be synthesized using a technology library, nor to be mapped onto an FPGA device.

Table 1: Set of actions supported by the simulation environment.

BIST	BIRA	Action description
0	0	Terminate the current repair session
0	1	Force the same faulty address to be detected
0	2	Undo the last spare allocation
0	3	Reset the allocation of spare elements
1	0	Detect the next fault without repair the current faulty element
1	1	Repair the current faulty element, and detect the next one
2	0	Require the reset of the fault detection subsystem

Whenever the RAM-L code is executed, first SIERRA compiles the program producing an internal bytecode representation. Then, the simulation engine executes the bytecode by means of an internal virtual processor (MRA_VP – see Figure 7). The virtual processor implements a stack architecture. Four virtual registers are implemented to execute the bytecode. A private read-only memory block (code segment) is reserved for storing the sequence of virtual instructions. Each virtual instruction is composed of two fields: (i) the OPCODE field stores the specific operation code of the instruction encoded as an integer value, while (ii) the ARGUMENT stores an optional integer parameter associated with the executed instruction. A read-write memory block (data segment) is used as a stack for storing scalar/array variables and the output values generated by the instruction execution. The code and data segments represent the virtual memory (VMEM) associated with the MRA_VP. VMEM is dynamically allocated within the host memory, depending on the result of the compilation process (i.e., number and type of variables), and its organization reflects the variable declaration order in the RAM-L program. Since the memory requirements for the simulation

of an MRA may change at run-time, the MRA_VP can dynamically expand the size of the VMEM. The MRA_VP can execute 28 different bytecode instructions. Five instructions are devoted to implement a mechanism for allowing the MRA to directly interact with the simulation environment (i.e., to set the content of the BIST and BIRA internal simulator variables).

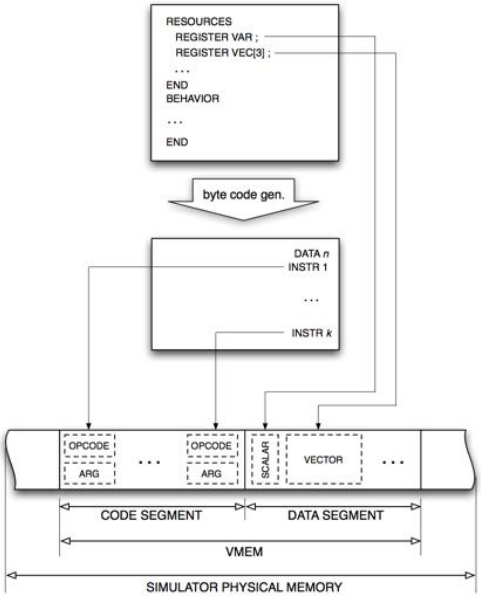

Figure 7: Run-time compiling process with different data structures allocated in the simulator memory.

During the simulation, the MRA communicates with the simulation engine to receive the address of the next faulty cell, thus emulating the test procedure of the memory device. Every time the MRA receives a new fault to handle, it applies its internal heuristic in order to take a proper action. At the basis of the simulation process, SIERRA records the set of faulty cells that have been repaired by the instantiation of a spare element (e.g., a spare row, a spare column, etc.). Whenever a redundant element is allocated, all the faulty elements that share the same row i, the same column j or both with the redundant element are marked as repaired. For instance, if a faulty cell $F_{i,j}$ is detected and a spare row $R = i$ is allocated, then all the faulty cells $F_{i,_}$ which are located on the same row i are automatically repaired. The action performed by the simulated MRA may also include to temporarily store the information and wait for further information. To support this behavior and speedup the simulation, all actions are mapped as SQL queries to the simulation database. The MRA simulation terminates when all faults have been repaired, or there are still faults to repair but the spare resources are exhausted. At the end of the simulation, the repairing efficiency is computed and saved along with that obtained from the execution of the Oracle.

4.1. Distributed simulations

The design space to explore can be very large, due to the number of different memory arrangements and configurations of spare elements to use for assessing the capability of a given MRA. Furthermore, to correctly evaluate the efficiency of the MRA, a very large number of FBMs must be simulated (generally $\geq 1,000$ FBMs), while the adoption of an external fault simulator can greatly slow the simulation process. Therefore, to reduce the time spent on simulations and allowing the designer to explore as much as possible redundancy configurations, SIERRA can work in a distributed fashion. Figure 8 shows the organization of the nodes in the distributed context.

Each simulation machine (namely simulation node) runs an instance of SIERRA. One of the nodes is marked as the *master*, while the others become the *slaves*. The master node is in charge of generating the set of FBMs, as well as copying the configuration of spare elements for the various slaves. It also provides the bytecode associated to the MRA under analysis to all the slaves. Instead of simulating different MRAs at one time, we preferred to design SIERRA in order to analyze one MRA on multiple FBMs, giving a more detailed view of its real capabilities to the designer. Once

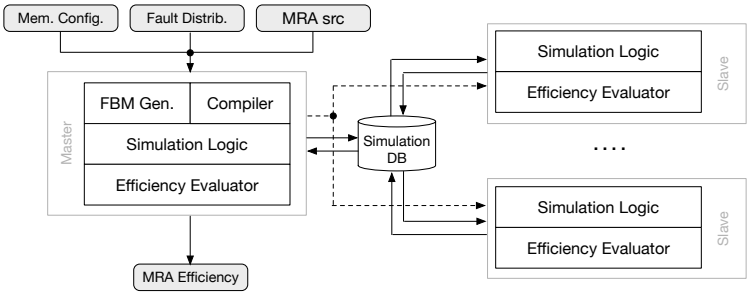

Figure 8: SIERRA running in a distributed environment. The master server coordinates the simulation on multiple nodes, where slaves are in charge of executing the MRA on a specific FBM instance.

all nodes received all required information, the setup phase is completed, and the simulation can start. It is worth to note that each node may eventually run its own copy of the memory fault simulator while the master is still responsible for summarizing the simulation results and to provide the efficiency of the MRA. During the simulation phase, each node emulates the MRA behavior on a specific FBM by issuing SQL queries to the simulation database (we leveraged on the fact that a modern DBMS is optimized to allow concurrent accesses from a large number of clients). During the simulation phase, also the master node is in charge of simulating the MRA on a given FBM. All nodes perform the simulation by executing the MRA bytecode and the Oracle strategy on the assigned FBM. At the end of the simulation, all statistics on the MRA execution are combined with the area and power overhead (using CACTI tool), in order to calculate the effective efficiency. To this end, the memory configuration is conveniently exported in a XML file, which contains both the organization of the memory array and the technology to use (e.g., 32 nm). In fact, the technology parameter influences directly both the area occupation and the power consumption of the memory. CACTI has been modified to parse this XML file and to generate the corresponding internal memory model. The memory is treated as a bit-oriented array, so that single cells can be addressed. By modeling the memory in this way, it is possible to extend the array to include the storage cells used by the MRA to perform repairing actions. Since MRA logic cannot be directly synthesized into actual hardware structures for a subsequent area and power consumption evaluation, we conveniently modeled logic as an additional 5% of storage space required by the MRA. However, it is worth to note that in a MRA the majority of the area occupation and power consumption can be ascribed to storage elements, as indicated by [17]. Thus, our method provides an over-estimation of the area consumed by the MRA logic.

4.2. MRA efficiency evaluation

SIERRA evaluates the efficiency (E) of a given MRA by simulating the action of a test algorithm and the repairing process on the full set of generated TRFBMs (see subsection 3.1).

$$E = w_1 \cdot E_R + w_2 \cdot E_A + w_3 \cdot E_P \tag{2}$$

According to equation 2, the efficiency is computed as the weighted sum of three main components: (i) the ability of the MRA to repair the full set of TRFBMs – E_R, (ii) the area overhead of the BIRA circuit corresponding to the RAM-L code of the MRA – E_A, and (iii) the power consumption of the BIRA circuit corresponding to the analyzed MRA – E_P. Each of these components is expressed as a fraction, thus its value ranges between 0 and 1. The weights associated with the three efficiency components can be set by the designer, and reflect their importance for the specific applications (e.g., for mission critical servers it is more important to protect data against memory errors and thus repairing memory faults, rather than reducing power and area overheads of the BIRA circuit). More specifically, the first component is computed as the ratio between the number of repaired TRFBMs and their total number:

$$E_R = \frac{F^R}{F^{tot}} \tag{3}$$

For each FBM that fails to be repaired, SIERRA computes two additional metrics that can be used to better understand the limitations of the MRA: the fault classes repair rate (RR_{FC}) and the faulty cells repair rate (RR_{cell}). The former is

the number of completely repaired FCs divided by the number of all injected FCs, while the latter is the ratio between the number of repaired faulty cells and the number of all faulty cells injected in the memory array. Moreover, SIERRA provides the location of all FCs and the location of all replaced elements. Since FCs are randomly allocated, their signatures may be fully or partially overlapped. Only non-completely overlapped FC instances are counted when computing the additional metrics.

The expression for the other two efficiency metrics are as follows:

$$E_A = \frac{A^{mem}}{A^{mem*}} \qquad E_P = \frac{P^{mem}}{P^{mem*}} \tag{4}$$

where A^{mem} and P^{mem} are respectively the area and power consumption of the memory array, and A^{mem*} and P^{mem*} are the area and power consumption of the memory array equipped with the BIRA circuitry, respectively.

During the evaluation of an MRA, E_R usually converges to a specific value. This implies that the difference between two consecutive measures of E_R (ΔE_R) tends to zero. SIERRA exploits this feature to early stop the simulation whenever the repairing capability of the MRA reaches the desired accuracy. ΔE_{win} is the number of FBMs to analyze before updating the value of ΔE_R. Whenever the computed value is lower than a user defined threshold (ΔE_{lim}), the simulation ends since the repairing capability of the MRA has been computed with the required accuracy. Setting ΔE_{lim} to a large value usually causes the evaluation to end before the convergence of E_R. This can be used to quickly obtain a rough approximation of the MRA capability of repairing faulty memories. Conversely, setting a low ΔE_{lim} provides more accurate results at the cost of additional computing effort.

5. Experimental Results

SIERRA has been validated with an extensive experimental campaign. Three state-of-the-art MRAs with different repairing capabilities and hardware resource utilization have been selected for the experiments. To evaluate their performance, a large set of FBMs has been analyzed, considering four probability distributions of the fault classes. Such distributions, showed in Table 2, have been derived from those used in [21, 31], reflecting different application scenarios.Without loss of generality, the efficiency of algorithms has been evaluated considering a single bank memory architecture with different configurations in terms of the size of the memory array and number of available spare resources. In addition, we used only spare rows and spare columns as redundant elements (the type of available spare resources only influences the behavior of the MRA). We performed two set of experiments. In the first set we only considered the capability of each MRA to repair faulty models (repair rate), irrespective of the area and power consumption. In the second set of experiments we estimated the effective efficiency of the three MRAs, by including the area and power estimation. To this purpose, in the first set of simulations we tuned the weights associated with the efficiency components (equation 2) in such a way the contribution of E_A and E_P was zero. This allowed SIERRA to avoid the calculation of the area and power for the three MRAs.

All simulations have been performed on a group of five workstations equipped with an Intel Core i7™ microprocessor running at 3.0GHz, 24GiB of main memory and the Linux operating system (the kernel was the 3.13.0-65-generic version). Workstations have been connected through a standard 1Gb/s Ethernet link. The DBMS (MySQL) has been exectued on one of these workstations.

5.1. Simulated MRAs

Three state-of-the-art MRAs (namelly MRA_1, MRA_2, and MRA_3), which differs each other for the amount of storage space required, logic complexity, and their repairing capabilities, have been used in our experiments as a test vehicle for validating SIERRA. They are briefly described in the following.

- **MRA_1**: it has a simple design aimed at limiting the circuit area overhead. MRA_1 is a fault-driven redundancy algorithm (i.e., it interleaves test and repair analysis) derived from the repair approach presented in [33]. In order to keep the area consumption under control while preserving an acceptable repair rate, a set of concurrent blocks (called analyzers – $a_1, \ldots, a_n$ respectively) are implemented. We used 4 analyzers. a_1 implements the {R-R-... C-C-... } allocation sequence and it is executed first, while a_2 implements the {C-C-... R-R-... } sequence. Similarly, a_3 and a_4 implements respectively the {R-C-... R-C} and {C-R-... C-R} allocation strategies. If a_1 fails to repair the memory, then a_2, a_3 and a_4 are used in sequence. In this way, MRA_1 is able to replicate the

priority selection mechanism, as in the original approach. Given these characteristics, this redundancy scheme is better suited for small memory devices.

- **MRA**$_2$: it works with a fault-driven approach. Different allocation sequences are modeled as a path from the root to one of the leaves of a tree data structure (i.e., edges are the allocated elements, while nodes represent different states of the algorithm) [21, 31]. A depth-first analysis allows the dynamic exploration of the branches, by selecting a repairing sequence. A leaf is reached when a successful repairing allocation sequence is found or no more redundant resources are available. Although the tree has a finite size, increasing the number of spare resources means increasing the time required to complete the analysis, as well as the storage resources. This MRA thus represents a good trade-off between hardware complexity and repairing capabilities.

- **MRA**$_3$: it directly derives from the optimal redundancy analysis algorithm employed by the Oracle. The algorithm implements a force-repair step followed by a fixed repairing strategy, instead of an exhaustive analysis of the remaining faults. In this way the algorithm performs a close-to-the-optimum analysis in a shorter time. However, it presents higher hardware resources demand, requiring a counter for each row/column of the memory array. Differently from the other two MRAs, the repairing decision is taken on the basis of the data collected during the test, that are stored in the row and column counters.

Table 2: Fault class probability distributions used for the evaluation of SIERRA using three state-of-the-art MRAs.

Fault class	**D**$_1$	**D**$_2$	**D**$_3$	**D**$_4$
Single cell fault (SC)	0.390	0.350	0.500	0.600
Double cell fault in a row (DCR)	0.120	0.110	0.100	0.100
Double cell faul in a column (DCC)	0.240	0.230	0.100	0.100
Quadruple cell fault (QC)	0.000	0.060	0.030	0.030
Single row fault (SR)	0.040	0.020	0.100	0.100
Single column fault (SC)	0.140	0.140	0.100	0.000
Double row fault (DR)	0.030	0.050	0.035	0.035
Double column fault (DC)	0.040	0.040	0.035	0.035

5.2. MRAs efficiency analysis

Table 2 shows the four probability distributions used in the experimental campaigns to assess the efficiency of the three selected MRAs. In particular, distributions D_1 and D_2 are representative of the end-of-production repairing process. In this case, manufacturing process defects tend to produce large clusters of faulty cells. D_1 and D_2 are therefore characterized by a lower probability of single cell faults (SC) compared to D_3 and D_4. The reduction of the probability of SC is spread on fault classes with a higher number of faulty cells (i.e., DCR, DCC, QC, SC, SR, DC and DR). During operational activity of the memory device, latent defects or external factors may induce new faulty elements. In these cases, single faulty cells appear more frequently than clustered faults. Distributions D_3 and D_4 (see Table 2) model this situation. They assure the SC is predominant among the other fault classes. For all distributions, we considered a number of injected defects per FBM in the range of 1 to 10 that fairly simulates the high density of defects that next generation nanoscale devices will face. Simulations have been performed by considering memory configurations that span from small arrays of 256×256 cells to arrays of 4096×4096 elements, all three fault detection orders (see Subsection 3.3), and the generation of 1,000 FBMs for each configuration. Redundancy configurations used for the simulations were in the range of ($s_r = 1, s_c = 1$) to ($s_r = 5, s_c = 5$), where s_r and s_c are respectively the number of spare rows and spare columns.

Figure 9 provides a graphical representation of the repair rate obtained by the three MRAs. Plots are organized into a grid, in which the result of each MRA is reported by column, while rows of the grid represent the application of different fault class probability distributions. Each plot summarizes the number of repaired FBMs over the 1,000 FBMs analyzed for the specific MRA, when spare elements and injected defects are varied. The repair rate is represented by means of two concentric bars. The red bar shows the number of TRFBMs that have been generated (thus repaired by the Oracle), while the white bar reports the number of FBMs that have actually been repaired by the MRA. Taking

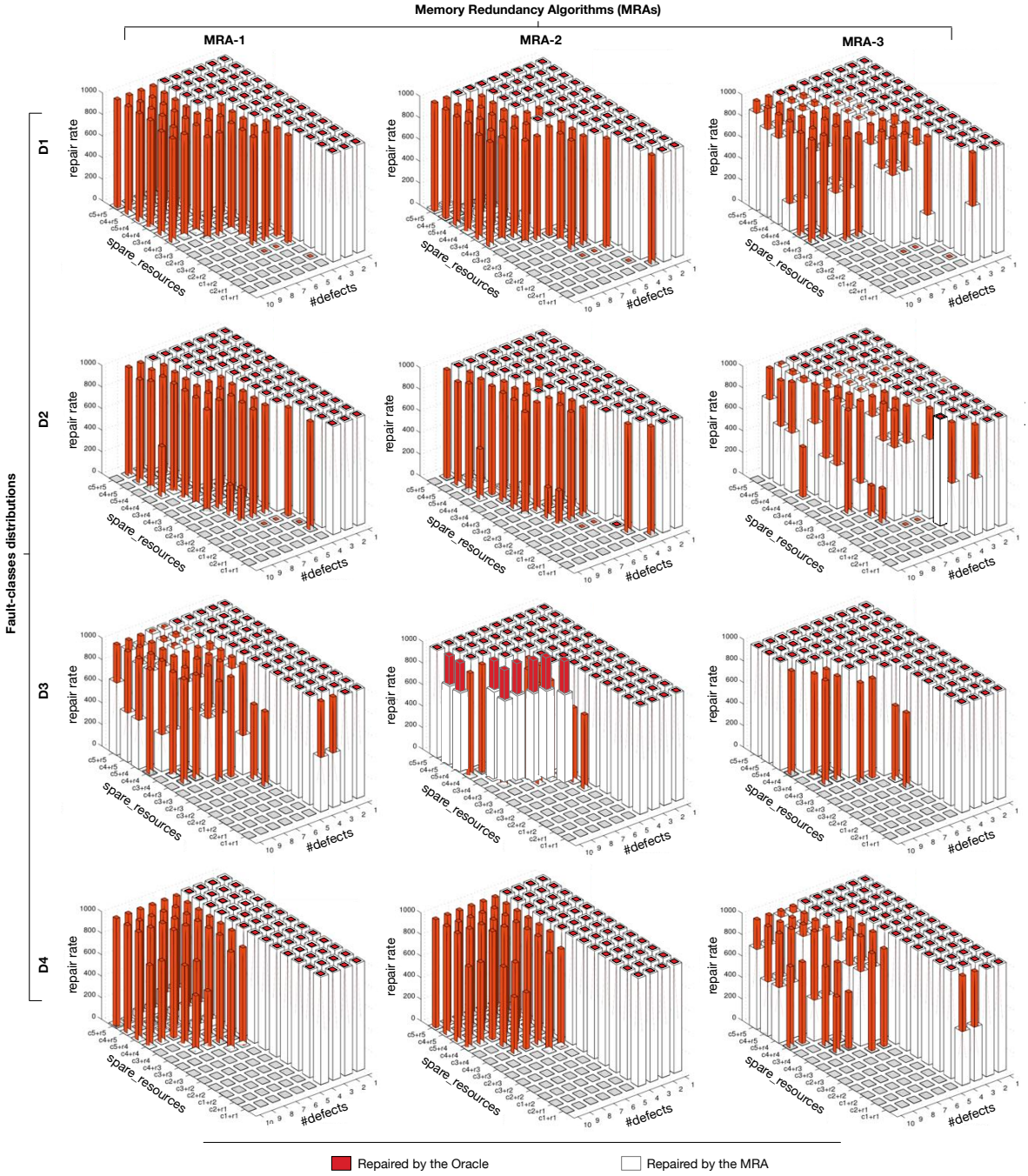

Figure 9: Experimental evaluation of the repair rate of three state-of-the-art MRAs using fault class distributions from D_1 to D_4. The results are presented, for each distribution D_i, by rows (e.g., plots for distribution D_2 are presented in the second row). For each row, the plots refer to MRA_1 (column 1), MRA_2 (column 2), and MRA_3 (column 3). Due to space limitation, only results for the largest array are presented (4096×4096).

into consideration the first two distributions (i.e., D_1 and D_2), the performance of MRA_1 quickly decreases with the number of injected defects. What is, however, more surprising is that a similar behavior remains valid also for MRA_2 which is more complex in terms of implementation. This result can be partially ascribed to the use of the random fault detection order, and demonstrates how different parameters may strongly impact the final yield. Differently, the repairing rate of the MRA_3 remains in general close to the optimum (Oracle). However, its performance is negatively counterbalanced by the high hardware cost.

Considering the other two distributions (i.e., D_3 and D_4), additional single faults (SC class) are injected. When the simple allocation strategy of MRA_1 is applied to distributions D_3 and D_4, the failure rate increases. This behavior can be ascribed to the fact that the algorithm behaves in a blind way, allocating spare resources in a fixed sequence. Conversely, the performance of MRA_2 is more in line with the expectations: it fails when the number of defects is higher. A global analysis of the algorithm behavior suggests to use this redundancy algorithm as a good trade-off between performance and hardware cost (i.e., it is more convenient when it is employed with large memory arrays [30] so that the hardware cost is counterbalanced by its efficacy). Similar to the prior case, MRA_3 maintains a global behavior close to that of the Oracle. In addition, the result analysis shows an important aspect: MRA behavior is not only influenced by the distribution of faulty elements in the memory array, but also by the detection sequence. This makes the capability of SIERRA of interacting with an external memory fault simulator an important key feature for determining the actual efficiency of the redundancy algorithm under analysis, particularly once a fair set of spare elements for the MRA has been determined.

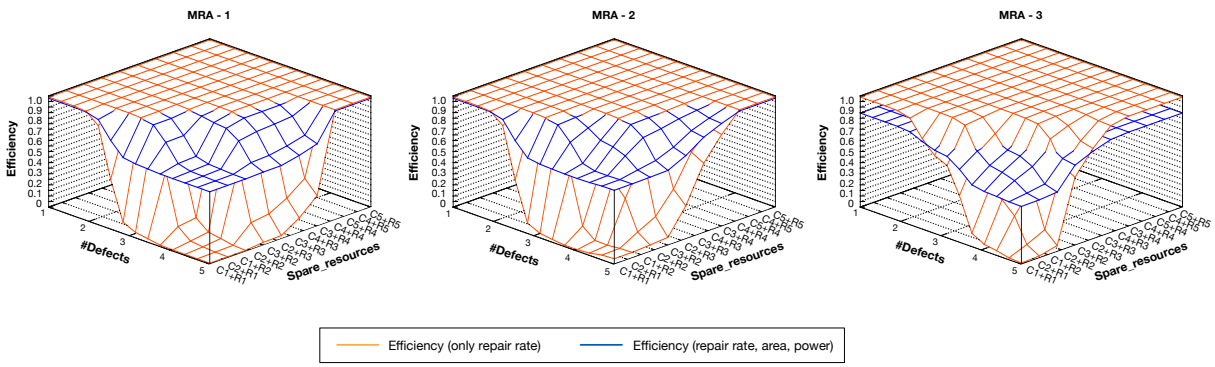

Figure 10: Experimental evaluation of the efficiency of three state-of-the-art MRAs using fault class distributions D_2. Memory array is set to the largest configuration (4096×4096). Each graph shows how the repair rate (orange) and efficiency (blue) change by modifying the number of available spare resources and the number of injected defects.

To assess the capability of SIERRA to measure the efficiency of the MRAs also in terms of area and power consumption, we performed a second set of simulations by setting the three weights in equation 2 equally. This second campaign of experiments was intended to evaluate how the structures (storage and logic) required by each MRA can influence the efficiency w.r.t. the only repair rate capability. Figure 10 shows the simulation results when faults follow the D_2 distribution (similar consideration are still valid for the other three fault distributions). All the results have been normalized to range between 0 and 1. As expected, the lower repair rate the MRA_1 exhibits is compensated by a more higher efficiency thanks to a reduced set of resources and a simpler control logic. When the number of defects is low the efficiency computed both considering the repair rate alone, and the repair rate with area and power consumption are close to 1. Increasing the number of defects causes the repair rate to rapidly decrease towards 0, while the efficiency falls close to 0.6. Moving to MRA_2 performance, we note that the lower efficiency in terms of area and power consumption is compensated by a higher repair rate. Globally, the MRA_2 performs similar to the MRA_1. In fact, for a high number of defects the repair rate is slightly higher than in the case for MRA_1. However, MRA_2 requires more storage resources and a more complex logic leading the efficiency to fall close to 0.6. More interesting is the case for MRA_3. The repair rate is in general close to the Oracle repair rate, decreasing very slowly. However, the high demand for storage resources and a complex logic makes the efficiency to appear lower than the efficiency of MRA_1 and MRA_2, even in the case of a low number of defects.

5.3. Simulation times

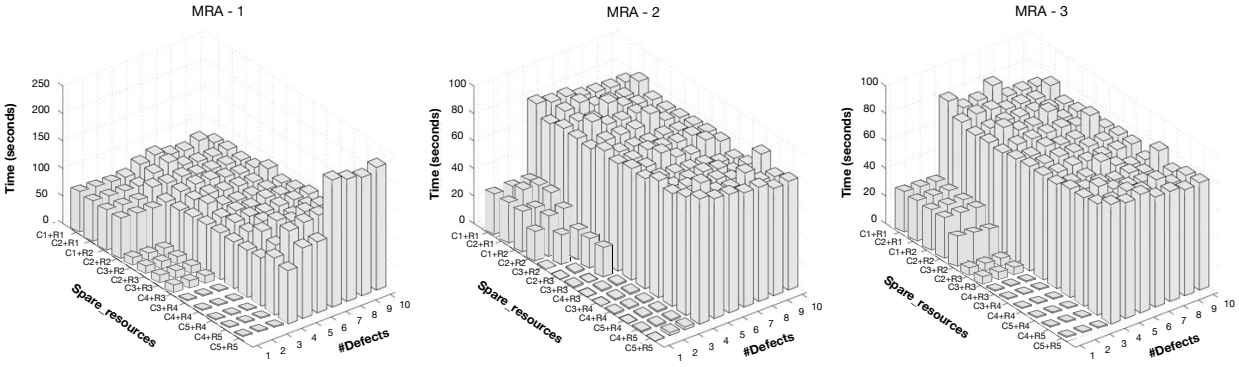

Figure 11: Experimental evaluation of the simulation time for the three state-of-the-art MRAs using fault class distributions D_2. Memory array is set to the largest configuration (4096×4096).

Looking at the simulation time, we observed that the overall computation time is slightly influenced by the simulated system (i.e., the MRA and the memory configuration). In general, the simulation of 1,000 FBMs was always completed in a range between 60 and 100 seconds on a single simulation node, confirming the efficient implementation of the simulation engine. Given a memory configuration, the evaluation of the area and power using CACTI requires less than 1 second to be performed. Accounting also the time to export the memory configuration, the overall simulation time remains within a range between 60 and 100 seconds. Figure 11 shows the distribution of times required to complete the analysis of 1,000 FBMs when the number of defects and the spare resources change. As a general consideration, the simulation time is higher for more complex MRAs, albeit it is not directly correlated to the effective time required by the hardware circuit to complete the analysis (in fact, the simulation is always sequential). In addition, the capability of distributing the simulation, allows SIERRA to efficiently scale (almost linearly) with the number of computing nodes. Moving from one to five simulation nodes, we completed the two experimental simulation sets in one fifth of the time w.r.t. the execution of SIERRA on a single node. This features becomes valuable in an industrial context, where a large number of memory configurations have to be examined.

6. Conclusions

In this paper, we presented SIERRA, a tool able to evaluate the repairing efficiency of an MRA while considering different faulty memory configurations (including realistic fault distributions), area and power consumption of the BIRA circuitry. The outcome of SIERRA is a detailed description of the MRA efficiency obtained in realistic conditions. An experimental evaluation demonstrated the efficiency of the tool and the benefits provided by the analysis of its results in assessing real capabilities of MRAs. Furthermore, being able to distribute the simulation effort and scaling with the increase of the number of computing elements, is a desired feature.

Three different memory redundancy algorithms have been simulated with SIERRA, using four realistic fault distributions. The simulation results confirmed the growing importance of simulating the behavior of MRAs using realistic sets up, and in association with a memory fault simulator. Given the expected continuous increase of memory devices placed at different levels of future high-performance computing systems, along with an increasing rate of failure due to the adoption of more sophisticated manufacturing technologies, there is a growing interest in finding efficient allocation schemes for redundant resources. On the basis of these observations, we can conclude that SIERRA represents a valuable instrument towards the generation of new heuristics for BIRA circuits that can provide very high repairing efficiency with reduced costs.

References

[1] Jacob B., "The 2 PetaFLOP, 3 Petabyte, 9 TB/s, 90 kW Cabinet: A System Architecture for Exascale and Big Data", *IEEE Computer Architecture Letters*, 2015.

[2] *International Technology Roadmap for Semiconductor*, http://www.itrs.net.
[3] J. Dongarra, et al., "The international exascale software project roadmap", *Int. J. High Perform. Comput. Appl.*, 25 (1) (2011) 3–60.
[4] S-Y Kuo, W. K. Fuchs, "Efficient Spare Allocation for Reconfigurable Arrays", *IEEE Trans. on Computers*, no. 2, vol. 41, pp. 221–226, 1992.
[5] J-F Li, J-C Yeh, R-F Huang, C-W Wu, "A built-in self-repair scheme for semiconductor memories with 2-d redundancy", *Proc. Of IEEE Intl. Test Conference*, pp. 393–402, 2003.
[6] T-W Tseng, J-F Li, A. Pao, K. Chiu, E. Chen, "A Reconfigurable Built-In Self-Repair Scheme for Multiple Repairable RAMs in SOCs", *Proc. Of IEEE Intl. Test Conference*, 2006.
[7] Rei-Fu Huang, et. al., "Economic Aspects of Memory Built-in Self-Repair", *IEEE Design and Test of Computers*, no. 2, vol. 24, pp.164–172, Mar. 2007.
[8] S. Di Carlo, G. Politano, P. Prinetto, A. Savino, A. Scionti, "Genetic Defect Based March Test Generation for SRAM", *Springer Berlin Heidelberg*, pp. 141–150, 2011.
[9] S. Di Carlo, P. Prinetto, A. Scionti, Z. Al-Ars, "Automating defects simulation and fault modeling for SRAMs", in *High Level Design Validation and Test*, *IEEE International Workshop*, vol., no., pp.169–176, 19-21 Nov. 2008.
[10] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, "Specification and Design of a New Memory Fault Simulator", *Proc. 11th Asian Test Symp. (ATS02)*, pp. 92–97, 2002.
[11] S. Shoukourian, V. Vardanian, Y. Zorian, "An approach for evaluation of redundancy analysis algorithms", in *Proc. Of IEEE Intl. Workshop on Memory Technology, Design and Testing*, pp. 51–55, 2001.
[12] V. Vardanian, et. al., "A Tool for evaluation and comparison of redundancy allocation algorithms for Static Random Access Memories", in *Proc. Of Intl. Conference on Computer Science and Information Technologies*, pp. 391–394, 2003.
[13] Tsu-Wei Tseng, Jin-Fu Li, Da-Ming Chang, "A Built-In Redundancy-Analysis Scheme for RAMs with 2D Redundancy Using 1D Local Bitmap", *Design, Automation and Test in Europe, (DATE'06)*, vol. 1, pp. 53–58, march 2003.
[14] Rei-Fu Huang, Jin-Fu Li, Jen-Chieh Yeh, Cheng-Wen Wu, "A Simulator for Evaluating Redundancy Analysis Algorithms of Repairable Embedded Memories", *Memory Technology, Design and Testing (MTDT 2002), Proceedings of the 2002 IEEE International Workshop on*, pp. 68–73, 2002.
[15] Shyue-Kung Lu, Yu-Chen Tsai, Hsu C.-H., Kuo-Hua Wang, Cheng-Wen Wu, "Efficient Built-In Redundancy Analysis for Embedded. Memories With 2-D Redundancy", *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, no. 1, vol. 14, pp. 34–42, Jan. 2006.
[16] Rei-Fu Huang, Jin-Fu Li, Jen-Chieh Yeh, Cheng-Wen Wu, "Raisin: Redundancy Analysis Algorithm Simulation", *Design and Test of Computers, IEEE*, no. 4, vol. 24, pp. 386–396, July-Aug. 2007.
[17] C.-T. Huang, C.-F. Wu, J.-F. Li, C.-W. Wu, "Built-in redundancy analysis for memory yield improvement", *IEEE Trans. on Reliability*, no. 4, vol. 52, pp. 386–399, 2003.
[18] Ganapathy S., Canal R., Alexandrescu D., Costenaro E., González A., Rubio A., "INFORMER: An Integrated Framework for Early-stage Memory Robustness Analysis", *Proceedings of the Conference on Design, Automation & Test in Europe (DATE'14)*, pp.1–4, no. 33, 2014.
[19] C-L Wey, F. Lombardi, "On the Repair of Redundant RAM's", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, no. 2, vol. 6, pp. 222–231, 1987.
[20] L. Anghel, N. Achouri, M. Nicolaids, "Evaluation of Memory Built-In Self Repair Techniques for High Defect Density Technologies", *Proc. Of IEEE Pacific Rim Intl. Symposium on Dependable Computing*, pp. 315–320, 2004.
[21] Ohler P., Hellebrand S., Wunderlich H.-J., "Analyzing Test and Repair Times for 2D Integrated Memory Built-in Test and Repair", *IEEE Design and Diagnostics of Electronic Circuits and Systems (DDECS'07)*, pp. 1–6, April 2007.
[22] Sehgal A., Dubey A., Marinissen E. J., Wouters C., Vranken H., Chakrabarty K., "Redundancy modeling and array yield analysis for repairable embedded memories", *Proc. Computers and Digital Techniques, IEE*, no. 1, vol. 152, pp. 97–106, 2005.
[23] S. Nakahara, K. Higeta, M. Kohno, T. Kawamura, K. Kakitani,"Built-in self-test for GHz embedded SRAMs using flexible patterngenerator and new repair algorithm", *Proc. Of IEEE Intl. Test Conference*, pp. 301–310, 1999.
[24] C.-S. Hou, et al., "Memory Built-In Self-Repair Planning Framework for RAMs in SoCs", *Transaction on Computer-Aided Design of Integrated Circuits and Systems, IEEE*, no. 11, vol. 30, pp. 1731–1743, November 2011.
[25] Chao, M.C.-T., Ching-Yu Chin, Chen-Wei Lin, "Mathematical yield estimation for two-dimensional-redundancy memory arrays", *Computer-Aided Design (ICCAD), IEEE/ACM*, pp. 235–240, November 2010.
[26] *CACTI: an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model*, http://www.hpl.hp.com/research/cacti
[27] J. R. Day, "A Fault-Driven, Comprehensive Redundancy Algorithm", *Design and Test of Computers, IEEE*, no. 3, vol. 2, pp. 35–44, 1985.
[28] M. Snir, et al., "Addressing failures in exascale computing", *The International Journal of High Performance Computing Applications*, Vol. 28(2), pp. 129–173, 2014.
[29] C. Wu, C. Huang, C. Wu, "RAMSES: a fast memory fault simulator", *International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 165–173, 1999.
[30] A. Scionti, "Defect oriented memory test and repair at the nanometer design scale", *Ph.D Dissertation*, Politecnico di Torino, March 2011.
[31] P. Ohler, S. Hellebrand, H-J Wunderlich, "An Integrated Built-in Test and Repair Approach for Memories with 2D Redundancy", *Proc. Of IEEE European Test Symposium*, 2007.
[32] R. J. McPartland, et al., "SRAM Embedded Memory with Low Cost, FLASH EEPROM-Switch-Controlled Redundancy", *Proc. Of IEEE Custom Integrated Circuits Conference*, 2000.
[33] T. Kawagoe, J. Ohtani, M. Niiro, T. Ooishi, M. Hamada, H. Hidaka, "A built-in self-repair analyzer (CRESTA) for embedded DRAMs", *Proc. Of IEEE Intl. Test Conference*, pp. 567–574, 2000.
[34] J. Nair Prashant, D. A. Roberts, Q. K. Moinuddin, "FaultSim: A Fast, Configurable Memory-Reliability Simulator for Conventional and 3D-Stacked Systems", *ACM Trans. Archit. Code Optim.*, Vol. (12), No. (4), pp. 44:1–44:24, December 2015.
[35] R. P. Treuer, V. K. Agarwal, "Built-in self-diagnosis for repairable embedded RAMs", *IEEE Design & Test of Computers*, Vol. (10), pp. 24–33, June 1993.

[36] C.-W. Wang, C.-F. Wu, J.-F. Li, C.-W. Wu, T. Teng, K. Chiu, and H.-P. Lin, "A built-in self-test and self-diagnosis scheme for embedded SRAM", in *Proc. Ninth IEEE Asian Test Symp. (ATS)*, Taipei, pp. 45–50, December, 2000.

[37] D. K. Bhavsar, "An algorithm for row-column self-repair of RAMs and its implementation in the alpha 21264", in *Proc. Int. Test Conf. (ITC)*, pp. 311–318, 1999.

[38] N. Park, E. Lombardi, "Repair of memory arrays by cutting", in *Proc. IEEE Int. Workshop pn Memory Technology, Design and Testing (MTDT)*, San Jose, pp. 124–130, August, 1998.

[39] S.-K. Lu, C.-H. Hsu, "Built-in self-repair for divided word line memory", in *Proc. IEEE Int. Symp. Circuits and Systems (ISCAS)*, pp. 13–16, 2001.

[40] S. D. Carlo, A. Savino, A. Scionti and P. Prinetto, "Influence of Parasitic Capacitance Variations on 65 nm and 32 nm Predictive Technology Model SRAM Core-Cells", in *17th Asian Test Symposium, (ATS)*, Sapporo, 2008, pp. 411–416.

[41] Rei-Fu Huang, Yung-Fa Chou and Cheng-Wen Wu, "Defect oriented fault analysis for SRAM", in *12th Asian Test Symposium, (ATS)*, 2003, pp. 256–261.

[42] Rodrguez-Montanes R., Arumi D., Manich S., Figueras J., Di Carlo S., Prinetto P., Scionti A., "8T SRAM Defective Cell with Open Defects", in *IEEE 25th Conference on Design of Circuits and Integrated Systems (DCIS)*, Lanzarote, ES, 17-19 Nov. 2010. pp. 492–497.

[43] A. J. van de Goor, S. Hamdioui and R. Wadsworth, "Detecting faults in the peripheral circuits and an evaluation of SRAM tests", *Proceedings of the International Test Conference (ITC)*, 2004, pp. 114–123.

[44] C. H. Stapper, A. N. McLaren and M. Dreckmann, "Yield Model for Productivity Optimization of VLSI Memory Chips with Redundancy and Partially Good Product", in *IBM Journal of Research and Development*, vol. 24, no. 3, pp. 398–409, May 1980.

[45] Minsu Choi and Nohpill Park, "Dynamic yield analysis and enhancement of FPGA reconfigurable memory systems", in *IEEE Transactions on Instrumentation and Measurement*, vol. 51, no. 6, pp. 1300–1311, Dec 2002.

[46] A. Choi, N. Park, F. J. Meyer, F. Lombardi and V. Piuri, "Reliability measurement of fault-tolerant onboard memory system under fault clustering", *Instrumentation and Measurement Technology Conference, (IMTC/2002). Proceedings of the 19th IEEE*, 2002, pp. 1161–1166 vol.2.

[47] A. Benso, A. Bosio, S. Di Carlo, G. Di Natale and P. Prinetto, "Automatic March tests generation for static and dynamic faults in SRAMs", *European Test Symposium (ETS'05)*, pp. 122–127, 2005.

[48] A. Benso, A. Bosio, S. Di Carlo, G. Di Natale and P. Prinetto, "March Test Generation Revealed", in *IEEE Transactions on Computers*, vol. 57, no. 12, pp. 1704–1713, Dec. 2008.