A machine learning-based approach to optimize repair and increase yield of embedded flash memories in automotive systems-on-chip

A Machine Learning-based Approach to Optimize Repair and Increase Yield of Embedded Flash Memories in Automotive Systems-on-Chip

A. Manzini[1,2], P. Inglese[3,4], L. Caldi[3], R. Cantoro[4], G. Carnevale[1], M. Coppetta[1], M. Giltrelli[1], N. Mautone[1], F. Irrera[2], R. Ullmann[3], P. Bernardi[4]

[1]INFINEON TECHNOLOGIES, Italy
[2]Department of Information Engineering, Electronics and Telecommunication, Sapienza University of Rome, Italy
[3]INFINEON TECHNOLOGIES, Germany
[4]Dipartimento di Automatica e Informatica, Politecnico di Torino, Italy

***Abstract*— Nowadays, Embedded Flash Memory cores occupy a significant portion of Automotive Systems-on-Chip area, therefore strongly contributing to the final yield of the devices. Redundancy strategies play a key role in this context; in case of memory failures, a set of spare word- and bit-lines are allocated by a replacement algorithm that complements the memory testing procedure. In this work, we show that replacement algorithms, which are heavily constrained in terms of execution time, may be slightly inaccurate and lead to classify a repairable memory core as unrepairable. We denote this situation as Flash memory false fail. The proposed approach aims at identifying false fails by using a Machine Learning approach that exploits a feature extraction strategy based on shape recognition. Experimental results carried out on the manufacturing data show a high capability of predicting false fails.**

Keywords: Memory Test and Repair, Machine Learning.

I. INTRODUCTION

Production testing is an important aspect in the automotive microcontroller manufacturing due to the high performance and reliability demands in challenging applications and safety aspects. Since embedded Flash (eFlash) memories represent a large percentage of the area of modern automotive microcontrollers, they significantly contribute to the overall product quality and yield.

Production testing main target is to ensure that all the devices are working within the device specification window, by means of calibration, test, and repair of defective Flash cells. The latter methods are aimed to save yield and based on redundancy structures available on-chip. Common redundancy structures are extra word-lines (WLs) or bit-lines (BLs), which are used by a specialized algorithm to replace failing cells. Redundancy resources are limited and allow to correct only a subset of all possible failures.

An important aspect of embedded memory test to save yield is the redundancy analysis and the activation of redundancy structures. Redundancy analysis is executed on-line during production test execution making use of SoC computational power in Software-assisted In-chip Self-Test (SIST) FLASH memory test phases. Error Correction Code (ECC) is – in contrast to commodity standalone memory test [1] – not applied for yield in ASIL-D Automotive System on Chip but is left for high in-field reliability.

In this context, production data is an important source of information to define an optimized sequence of screening tests to ensure fail modes coverage looking first at the most frequent fails so to reduce test time and yield loss in package tests. Moreover, those data can be used to understand technology marginalities or process deviations and to address new failure modes and related test approaches. Thus, the process of collecting data from production test and the consequent data analysis is a fundamental activity for the assessment of the next-generation devices' production status and to drive decisions about redundancy allocation strategies.

In fact, a major bottleneck in the Flash memory test and repair flow is related to redundancy management. Ideally, the redundancy allocation needs to be fast and accurate. This goal contrasts with failure bitmaps collection that may be used to compute the ideal redundancy resources allocation. Bitmap retrieval consists in downloading the coordinates of the encountered faults, which is time and memory intensive, therefore industry efforts are today oriented to limit their usage to operate the redundancy allocation. However, choosing strategies that limit the time and memory costs trades-off with allocation accuracy and may lead to false positive behaviors. In particular, two cases may show up during production volume testing, which are *ineffective repair* and *false fails*. The first case is the most harmful and leads to consider an uncorrectable failing device to be corrected by repair when it is not possible; the other is the case where a device is discarded because the repair algorithm was not able to find a suitable allocation that corrects all errors while it exists. In this case, the direct consequence is a yield loss [2].

In our approach, we deal with the false fail identification, given that the production flow should never produce ineffective repair cases. The risk of false fails becomes more prominent when a repair algorithm is oriented to avoid ineffective repair, but the automotive practice to repeat full test coverage post stress after repair takes place [3] ensures that every failing device is correctly binned and therefore discarded.

Different fail constellations may systematically appear at different technology nodes and maturity of the product. To easily adapt with respect to different scenarios, we propose a Machine Learning-based strategy that works in two phases:

1. At development time, i.e., during technology ramp-up, we collect bitmaps on a selected number of devices, which compose the training/test set; an analysis of defective fail constellations is performed to identify classes of false fail behaviors.
2. During massive production flow, we limit the datalog activity by extracting features [4] based on a coloring algorithm described in the paper; then we use a Machine Learning (ML) approach to label discarded devices as potential false fails.

The identified devices are further investigated to possibly find a redundancy allocation that can properly address a correction of the device. The advantage introduced by the approach is that a yield recovery can be obtained over production data at a reduced costs. The experimental results demonstrate that a yield recovery is achieved on production volume data at a sustainable cost.

In the paper, section II describes the general problematic and provides background about test and repair. Section III describes our approach, composed of a learning and a production assessment phases. Section IV provides experimental results collected on a significant data volume. Section V concludes the paper.

II. BACKGROUND

This background section provides the required basic concepts in the field of Flash memory test, repair and data analysis.

A. Flash Memory Test and Repair Algorithms

Current literature in the field of embedded memory test can be categorized into:

- Built-in Self-Test (BIST) methods, that are based on HW circuitries that perform the test [5][6] and collect diagnostic information [7][8]
- Software-based BIST (SBIST), where the test is directly applied by the CPU executing a firmware accessing the memory matrix [9]
- Hybrid approaches [10]: efficient eFlash memory testing can be conducted through a combination of Hardware dedicated Design for Testability (DfT) – e.g., Flash erase/program operations [1] – and taking advantage of Software-assisted In-chip Self-Test (SIST) [2].

With the latter it is possible to combine the hardware acceleration by Programmable BIST (PBIST) with the software flexibility using maximum System-on-Chip (SoC) CPU performance. As shown in Fig. 1, during SIST execution, there are two collaborating components:

- A PBIST, which:
  - Drives the Flash test execution without making use of CPU resources
  - Calculates test results to detect device misbehaviors
  - Passes test failure information to the SoC CPU
- A SoC CPU that:
  - Allocates redundancy resources according to a predefined and test time optimized repair algorithm
  - Collects failure data (i.e., bitmaps).

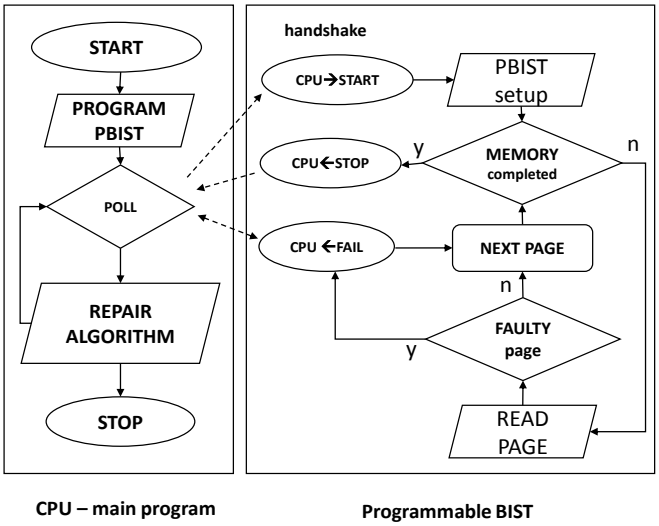


Fig. 1. SIST working principle.

As briefly introduced, bit-line and word-line oriented repair resources can be available on-chip [11]. Repair algorithms reconfigure the memory rows and columns to make the memory fully working. The approaches constitute a trade-off between topological efficiency post-repair and test time impact, especially in massively parallel testing on the ATE. Repair can be hardware-based [12] or software-based [13]. Often, there are hardware restrictions on how the repair elements can be used to replace the failing elements. These limitations restrict the number of allowed repair combinations.

Our embedded Flash memory redundancy scenario includes:

- 2 bit-lines allocated per each Flash page, composed of 256 data bits + 22 ECC check bits
- 2 n-word-lines per Flash block, where the n means that every word-line is composed of a set of (n) adjacent rows.

The repair algorithm used in production test is run by the SoC CPU, which receives the data from the PBIST. A handshake protocol is used to make the memory test and repair coexist. The CPU manages to resume after every stop due to a found fault and reprograms the PBIST to keep at-speed and back-to-back characteristics and ensure high memory test quality.

Such a repair algorithm is very fast, being based on greedy decisions. Bit-lines allocation is preferred to word-lines allocation. Once the bit-lines are exhausted, word-lines are allocated with intelligence, possibly freeing up already used bit-lines. When all the repair resources are used, any new fault causes the device to be discarded.

As shown in the experimental results, the described repair algorithm always correctly classifies uncorrectable fail configurations (i.e., there is never *ineffective repair*), thus it is very effective in screening out all defective devices. Nevertheless, an accurate assessment of the greedy repair algorithm has highlighted some limitations that may lead to a false fail classification in specific constellations of faults.

The pictures in Fig. 2 graphically explain why a faulty configuration may be classified as a false fail and how it could be corrected by an alternative use of the repair resources.

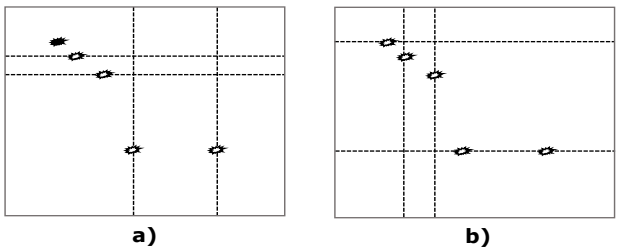


Fig. 2. a) False Fail and b) Effective Repair configuration.

In this scenario, assuming only 2 redundancy BLs and 2 redundancy WLs are still available, Fig. 2.a shows how the improper usage of redundancy elements is leading to a discarded device, while Fig. 2.b shows how the same resources can actually repair the device.

At the beginning of this research work, upon a massive inspection over production data, we were selecting suspect devices, where the identification criterion was the number of faults. If this was relatively low and the device was not repaired, the device was considered as suspect. This pioneer extraction of data provided us cases like the one depicted in Fig. 3: this specific case was identified as a suspect since the number of faults is quite low and the device was marked as not repairable by the redundancy algorithm.

Fig. 3. Example of fail constellation leading to false fail.

Further analysis on this bitmap has demonstrated that the repair algorithm is operating ineffectively for this constellation and an effective repair configuration exists.

B. Experimental setup for data analysis

Being able to properly extract and analyze data from a production flow is a fundamental prerequisite.

Flash memory test results are collected by the Automatic Test Equipment (ATE) via a proper test communication interface (e.g., JTAG). Test results – including pass/fail information and more structured data – are therefore transferred to a production database, ready to be used by product analysts and test engineers for test optimization in terms of yield, coverage, or test time.

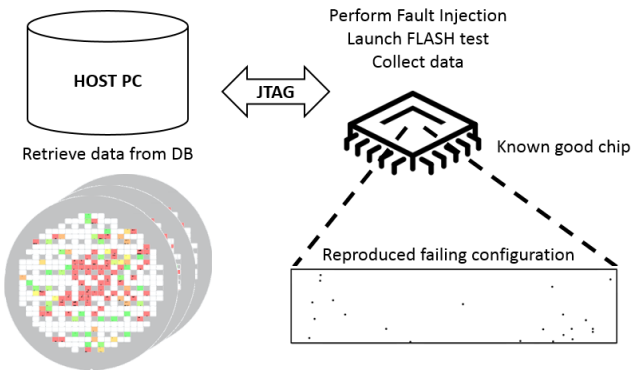


Fig. 4. Repair algorithm analysis flow

Some of the most important data collected during memory test are bitmaps. Bitmaps are the representation of memory topologic fails map and they are crucial to identify failure mechanisms.

In our flow, bitmap data are collected on-line by the SoC and transferred from the Device Under Test (DUT) to the ATE, and finally recorded into production databases; we have heavily drawn from these databases to perform big data analysis. As a preliminary stage, we were selectively reconstructing failing scenarios extracted from productive bitmap data in a lab environment for regression using a fault-free device mounted on a development board.

As shown in Fig. 4, the content of a failed memory can be purposely reconstructed to be analyzed. Production bitmap data are parsed by a script-based toolchain that translates physical fail maps into logical fail addresses. The fail constellation is then installed in a good device and afterwards the SIST repair algorithm is physically re-run, possibly with different flavors in allocating redundancy. This approach allows to run the repair algorithm on a reproduced fail scenario to be debugged and evaluated in terms of effectiveness and performance. Corner cases are particularly interesting to observe as they provide a broad set of typical fail constellations. Automation was developed on top of this measurement facility, so that the regression setup can autonomously run batches of experiments.

III. PROPOSED STRATEGIES

The proposed strategies pursue the objective of identifying false fails during volume production with the lowest possible costs. In this section it is first described how to systematically identify recurrent fail constellations leading to false fails. Then, it is discussed how to take advantage from the previous analysis to setup a Machine Learning (ML) -based environment to predict false fails in the set of devices discarded by the greedy algorithm.

The identification of a sufficient number of false fails in a subset of discarded devices is based on the following steps:

1) Identification of the lots and the chips that are the most representative in terms of fails occurrences
2) Reproduction in the experimental setup of a fail constellation
3) Execution of an exhaustive algorithm which determines whether the greedy approach has produced a false fail.

It is convenient to perform this learning phase during technology ramp-up as it is the basis of the volume data analysis. It should be conducted on a limited number of devices because it is quite time consuming. The subset of selected devices constitutes the training/test set that is used in the Machine Learning approach that is described below.

In production flow testing, an approach using the exhaustive repair algorithm would be unfeasible because of the execution time. To have a fast response about discarded devices potentially being false fails, we have devised a Machine Learning approach, based on the following points:

1) Features extraction via a coloring algorithm which returns aggregated information about fail shapes.
2) Use of a prediction model that is trained/tested using the data coming from the learning phase
3) Identification of false fails without the need of the full bitmap, but just feeding the predictor with fails' color statistics.

Fig. 5 graphically depicts the implemented scenario. The purpose of the prediction is to identify in the data volume the chips that are currently discarded but can be repaired, thus analyzing them and therefore possibly saving yield.

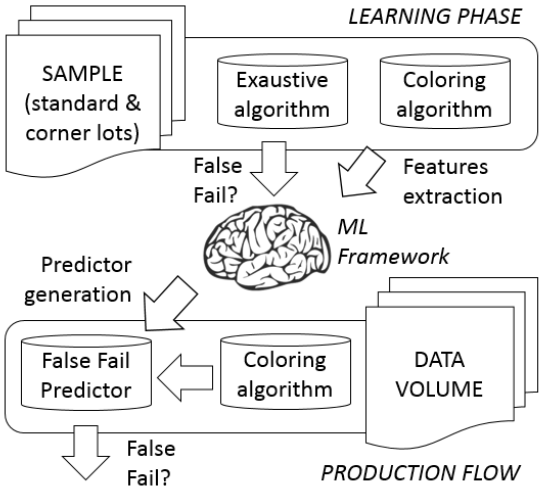


Fig. 5. Repair algorithm analysis flow.

A. Systematic identification of false fail cases

One of the most crucial aspects of this work is the availability of a reliable dataset preparation from production data. After the wafer fabrication, different electrical tests are performed at wafer level. Front-End wafer testing data have been used in this work. These are the results of the joint tuning of the manufacturing process and the Test Program screening maturity, to reduce extrinsic defects and to improve device performance.

The product development time is much shorter than the product lifetime. Then, long-term product fabrication will be affected by higher process variation than the initial production lots. A longstanding industry practice is to produce the so-called corner lots. These lots manufacturing recipes are adjusted to achieve the values of the long-term fabrication variation for key performance parameters of the product. Standard lots, instead, are manufactured considering target inline parameter limits.

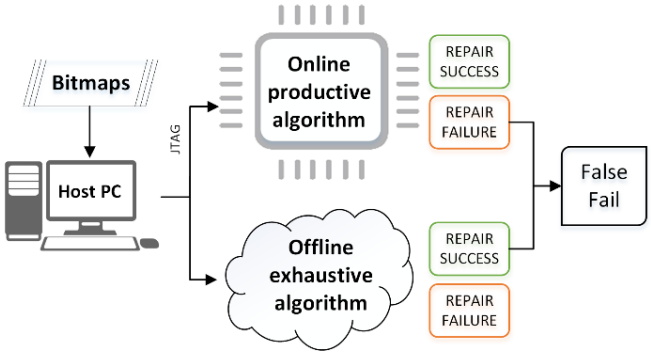


Fig. 6. Automated False Fail identification on sample data.

Sample data extracted from lots at different technology maturity grade have been used for ML environment train and test. Differently from the basic strategy of analysis described in Section II.a, which is based on manual efforts, we implemented an automated flow capable of automatically detecting false fail occurrences in the selected production samples. As depicted in Fig. 6, the chosen strategy compares the reparability verdict of the traditional algorithm to the one stated by an exhaustive algorithm, taken as reference. We developed a system capable of processing a bitmap both on-line on the chip via a SIST routine and offline on the host computer. The results of the different repair algorithms are stored, therefore enabling comparisons.

The current repair algorithm makes use of a greedy approach that allocates redundancy resources (i.e., bit-lines and word-lines) as soon as faults arise as fails during memory verification. On the contrary, the exhaustive repair algorithm first tracks the fails in a data structure representing the fail bitmap. Furthermore, it navigates through the fail bitmap and tries to repair the failing page using the available resources. This algorithm works on every *pageset* containing at least one fault: a *pageset* is defined as a set of pages (word-lines) sharing the same bit-line redundancy elements.

As a first attempt, bit-lines are allocated if they are enough to cover fails in a pageset; otherwise, the pageset is completely replaced using one of the available word-lines. After each replacement, a back-tracking step is performed: the algorithm proceeds until either all fails are managed (i.e., repair success) or redundancy resources are exhausted (i.e., repair failure); in case of failure, the algorithm is rolled-back up to the back-tracking step; then, a new attempt is made. If all possible attempts lead to a failure, then the bitmap is not repairable. The algorithm pseudo-code is shown in Fig. 7.

```
Algorithm Exhaustive Repair
  collect unrepaired fails
  while fail found do
    pageset ← next pageset containing fails
    if pageset is repairable using available bit-lines then
      allocate bit-lines BL needed to repair pageset
      repaired ← call Exhaustive Repair (recursive call)
      if repaired then return True
      else restore BL
    if pageset is repairable using word-lines then
      for wl in available word-line do
        allocate wl to repair pageset
        repaired ← call Exhaustive Repair (recursive call)
        if repaired then return True
        else restore wl
    return False
  return True
```

Fig. 7. Pseudocode of the exhaustive repair algorithm.

Wordlines include multiple pagesets. Thus, in the worst case scenario, the algorithm performs a number of attempts in the order of $O(n^m)$, where n is the number of wordlines and m is the number of redundant pagesets. Moreover, in order for the algorithm to be time efficient, a considerable amount of memory is needed to store aggregated information for the backtracking (i.e., concerning failing pagesets). Alternatively, the algorithm should re-compute the failing context dynamically, which would further increase the computational time required.

B. Feature extraction using a Coloring Algorithm

Since the goal of collecting information about false fail is to enable ML techniques, a strategic decision was taken about features extraction that could enable an effective prediction and a systematic yield analysis.

We decided to implement a "coloring" algorithm. This algorithm assigns codes (colors) to every fault encountered and returns a statistic of each color occurrence. An example is shown in Fig. 8, where a constellation scenario was colored by our algorithm, with "black" spots, "yellow" bit-oriented defects, "pink" word-oriented faults, and more.

The coloring algorithm permits to aggregate the various faults assigning to each one of them one or more "colors" that are merged together to provide a digest of the memory faults to the ML predictor. This permits to tune the Machine Learning to different use-cases, i.e., in our case, to the detection of False Fails.

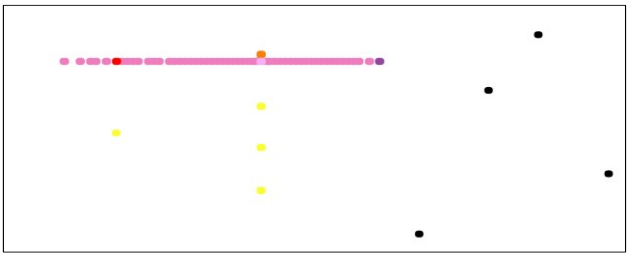


Fig. 8. Example of coloring result

This algorithm, whose pseudo-code is shown in Fig. 9, follows a similar approach than the exhaustive repair one: during the memory verification it receives the various faults as they arise, and it performs a "neighbors' search" to aggregate them. The search is done as soon as the fault information is received in order to minimize the impact on the total verification time. The neighbors' search looks for other faults in the close surroundings and, with respect to the position in the vicinity of the fault being currently analyzed, modifies the current fault color and, if needed, the neighbor's color, too. As soon as the neighbors' search ends, the new fault is saved in a hash table, which is devised to achieve a fast search within the fail set.

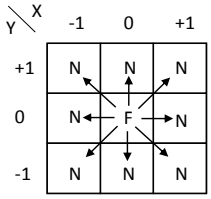
```
Algorithm Coloring
  while fail found do
    collect unrepaired fail
    page ← page containing fails
    for each fail in page do
        for each neighbor in neighborhood do
            if failing neighbor found then
                update neighbor.color
                update fail.color
            store fail
  for each color do report number of fails
```

Fig. 9. Pseudocode of the coloring algorithm.

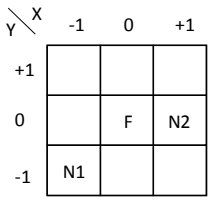
Concerning the color attribute to be assigned to each fault and its update, such value is represented using an 8-bit variable; the color update depends on the position of the found failing neighbor and it is obtained by simply operating a bit-wise logic sum (or) operation with a position specific mask.

Fig. 10 shows an example of color mask attribution with respect to neighbor position and an example of color computation where the target fault has two neighbors. A single bit within the N bits composing the mask is set to value 1, whose position is different for every neighbor position. A new fail found has "black" color, corresponding to all 0s in our codification; if another failing bit is found that is located in the considered neighbor, then the color value is updated "merging" with the proper bit mask.

At the conclusion of the test procedure, in order to provide the final digest of the colored features, an aggregation of the various colors is carried out: all the different faults are analyzed, and the total number of the various colors is extracted and used for the purposes of Machine Learning train/test and final deployment as a predictor engine.



X	Y	Color mask target fault	Color mask neighbor
0	-1	00000001	00000010
0	+1	00000010	00000001
-1	-1	00000100	10000000
-1	0	00001000	01000000
-1	+1	00010000	00100000
+1	-1	00100000	00010000
+1	0	01000000	00001000
+1	+1	10000000	00000100



Faulty position	color
F	01000100
N1	10000000
N2	00001000

Fig. 10. Example of color masks and their application.

C. ML approach to quickly predict false fails

The Machine Learning approach to develop the predictor has been structured in two phases.

The *first phase* consists in an exploratory analysis of the available data to evaluate the goodness of the extracted features set and to give feedbacks to the coloring algorithm design. For this reason, both unsupervised and supervised techniques, together with some 2D embedding methods, were used to assess if the false fails were correctly discriminated in the extracted input space [4]. During this analysis, a discrete variance in the cross-validated baseline models prediction accuracy, combined with the limited amount of data available for the early-stage product under investigation, suggested us to use some augmentation techniques on the original bitmaps. The augmenting transformations were chosen in order not to alter the characteristic fail signatures (e.g., bitwise and, or, xor among bitmaps; cropping, flipping, translation, addition of noise). By adding artificial bitmaps, we were able to obtain a more complete representation of the false fail phenomenology and more statistically significant prediction accuracies. The exploratory analysis showed that the available dataset was strongly unbalanced, in our case with a small percentage of false fail cases. Thus, we decided to use as indicator a confusion matrix: a classifying table which reports predicted labels along the rows and actual labels along the columns. The result is a square matrix with the number of correct predictions on the principal diagonal and the number of misclassifications elsewhere.

The *second phase* consists in selecting the best Machine Learning model, which could fit our requirements and constraints. We needed an efficient binary classificator, which had to be simple enough to be implemented in an automotive microcontroller and fast enough not to impact the overall test time in the production flow. The selection criteria were modeling capability, reliability, implementation feasibility, and interpretability.

IV. EXPERIMENTAL RESULTS

The approach proposed in this paper has been experimentally checked on Infineon AURIX™ SoC devices. The SoCs include multi-core TriCore processors running at 300 MHz and are used for automotive safety-critical applications. The Program Flash is composed of a number of Physical Sectors, each one 1MB wide, grouped in banks.

Each physical sector has dedicated redundancy resources and is tested independently and the production datalog marks a device as failed when one or more sectors are not repairable.

A training set was constructed by extracting features from bitmaps related to failing devices of several production lots, finally composed of 498 false fail cases, representing the 0.5% of the total sampled fail cases.

As described in III.A, the exact redundancy algorithm was off-line applied to each failing physical sector and its result was compared to the greedy algorithm running on the regression system; the sector is then labelled as real/false fail.

Aside the greedy repair algorithm, the coloring algorithm described in III.B was run on-chip to extract the features to feed the ML environment in its training phase. In terms of time overhead, if the coloring is performed by the CPU executing the repair algorithm, we measured from 1% to 8% of the test and repair time, depending on the number of faults and their location. Significantly, the coloring computation for the most recurring false fail constellations is around 3%.

Concerning the machine learning step illustrated in III.C, first of all, augmentation techniques were used to reach enough training samples and each physical sector of the augmented bitmaps was labelled as well. The augmentation processes required 10 days of CPU time. Then, among the broad set of types of ML engines and configurations, we decided to consider three models: a decision tree model, a random forest, and a feed-forward neural network. We tuned each model optimizing the returned accuracy with a grid search over the hyper-parameters space. Since we want to predict all possible false fails even tolerating a slight increase of test time, recall is used as the score metric, estimated using a 10-foldcross-validation [14]. For each proposed model, mean and variance of recall over the 10 folds are reported in Table 1.

Table 1. 10-fold cross-validation scores

Classifier	Mean	Variance
Decision Tree	98.40%	0.04%
Random Forest	98.20%	0.03%
Neural Network	95.33%	0.11%

The Decision Tree was finally selected as the candidate for on-line prediction since it showed a high score and a low variance even being the model with the fastest and simplest prediction routine. This routine can be easily implemented on the chip as a cascade of conditional branches and included into the on-line verification algorithm, running right after the feature extraction algorithm (coloring). The longest decision path in the obtained tree includes 5 nodes, i.e., it takes less than 100 clock cycles to make a prediction.

Table 2. Decision Tree Confusion Matrix

Actual \ Predicted	Real fail	False fail
Real fail	105	19
False fail	4	120

The confusion matrix produced by the Decision Tree allowed us to grade the approach. A balanced validation set was constructed picking 25% of the false fails in the augmented dataset and a comparable number of real fails. All the remaining samples were used for the training. The confusion matrix on the validation set is reported in Table 2. Coherently with the cross-validation results, we can correctly detect 98% of false fails even in the balanced validation set.

The Confusion Matrix also permits to distinctly evaluate the two failing modes of the predictor. The first happens when a false fail is missed and a repairable device is binned, which is not good for yield; the second when a false fail is erroneously recognized and the alternative repairing test flow is activated in vain, which is not good for test time. Please note that a misprediction of a real fail as false fail (12% of the cases) does not affect in any way the effectiveness of the verifying algorithm, i.e., the model predicts a suspect (false fail) but the device is discarded anyway.

V. CONCLUSIONS

This paper presents a systematic approach for eFlash repair algorithm optimization making use of a ML setup. The approach is based on real production data analysis about fail constellation bitmaps. The proposed methodology is based on a 2-step approach, with learning and production assessment based on a ML engine. Experimental results demonstrated that it is possible to predict a false fail device with a good accuracy.

REFERENCES

[1] P. Nicosia, F. Nava: “Test Strategies on Non Volatile Memories Electrical Wafer Sort on NAND, NOR Flash and Phase Change Memories”. IEEE NVSMW 2007, Page 11-18.

[2] M. Coppetta, R. Ullmann, L. Caldi, R. Cantoro, P. Bernardi, “Optimized Fail Signature Analysis for Automotive eFLASH Memories using Manufacturing Scale Data”, Testmethoden und Zuverlässigkeit von Schaltungen und Systemen, TuZ 2018

[3] U. Backhausen et al: “Robustness in automotive electronics: An industrial overview of major concerns”, 2017 IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS).

[4] I. Guyon and A. Elisseeff. “An introduction to variable and feature selection”. Journal of Machine Learning Research, 3:1157-1182, 2003.

[5] I. Voyiatzis, "Symmetric transparent on-line BIST of word-organized memories with binary adders," 2015 20th IEEE European Test Symposium (ETS), Cluj-Napoca, 2015, pp. 1-2.

[6] A. J. van de Goor, "Testing semiconductor memories — theory and practice", World Scientific Publishing Co., 1992.

[7] P. Bernardi, L. Ciganda: “An Adaptive Low-Cost Tester Architecture Supporting Embedded Memory Volume Diagnosis”, IEEE Transactions on Instrumentation and Measurement, vol. 61, no. 4, April 2012.

[8] J. Yeh, K. Cheng, Y. Chou and C. Wu, "Flash Memory Testing and Built-In Self-Diagnosis With March-Like Test Algorithms," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 26, no. 6, pp. 1101-1113, June 2007.

[9] A. Singh, D. Bose and S. Darisala, "Software based in-system memory test for highly available systems," 2005 IEEE Workshop on Memory Technology, Design, and Testing (MTDT'05), 2005, pp. 89-94.

[10] Bai Hong Fang, Qiang Xu and N. Nicolici, "Hardware/software co-testing of embedded memories in complex SOCs," ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No.03CH37486), San Jose, CA, USA, 2003, pp. 599-605.

[11] J. M. Daga, "Test and repair of embedded flash memories," IEEE International Test Conference, 2002, pp. 1219-.

[12] R. Chandramouli, "Managing Test and Repair of Embedded Memory Subsystem in SoC," IEEE Asian Test Symposium 2005, pp. 452-452.

[13] M. Schölzel, P. Skoncej, "Software-based repair for memories in tiny embedded systems," IEEE European Test Symposium, 2015, pp. 1-2.

[14] E. A. Garcia and H. He, "Learning from Imbalanced Data," IEEE Transactions on Knowledge & Data Engineering, vol. 21, no. , pp. 1263-1284, 2008.