A Fast Built-in Redundancy Analysis for Memories With Optimal Repair Rate Using a Line-Based Search Tree

Woosik Jeong, Ilkwon Kang, Kyowon Jin, and Sungho Kang, *Member, IEEE*

***Abstract*—With the growth of memory capacity and density, test cost and yield improvement are becoming more important. In the case of embedded memories for systems-on-a-chip (SOC), built-in redundancy analysis (BIRA) is widely used as a solution to solve quality and yield issues by replacing faulty cells with extra good cells. However, previous BIRA approaches focused mainly on embedded memories rather than commodity memories. Many BIRA approaches require extra hardware overhead to achieve the optimal repair rate, which means that 100% of solution detection is guaranteed for intrinsically repairable dies, or they suffer a loss of repair rate to minimize the hardware overhead. In order to achieve both low area overhead and optimal repair rate, a novel BIRA approach is proposed and it builds a line-based searching tree. The proposed BIRA minimizes the storage capacity requirements to store faulty address information by dropping all unnecessary faulty addresses for inherently repairable die. The proposed BIRA analyzes redundancies quickly and efficiently with optimal repair rate by using a selected fail count comparison algorithm. Experimental results show that the proposed BIRA achieves optimal repair rate, fast analysis speed, and nearly optimal repair solutions with a relatively small area overhead.**

***Index Terms*—Built-in self-repair (BISR), built-in self-test (BIST), redundancy analysis (RA), yield improvement.**

I. Introduction

As THE capacity and density of semiconductor memories have rapidly increased as a consequence of the technological progress of semiconductor manufacturing, the probability of memory faults has concomitantly increased, resulting in yield drop and quality degradation. Therefore, maintaining acceptable yield and quality has become the most critical challenges in semiconductor memory manufacturing. To achieve reasonable yield and quality, faulty cells are repaired with redundant cells; this is the most popular method for repairing embedded systems-on-a-chip (SOC) memories as well as commodity memories. Most commodity semiconductor memories are tested with external automatic test equipment (ATE) and their repair solutions are gathered from ATE. Faulty cells on a memory block are then repaired using repair solutions. However, most SOCs adopt a built-in self-test (BIST) and built-in redundancy analysis (BIRA) to test and repair their embedded memories instead of using external ATE because this method is more cost-effective. However, to test recent high density and high-speed commodity memories such as dynamic RAM (DRAM), a high-end ATE platform is required that offers fast operating speeds and high parallel testing functions. BIST and BISR schemes for commodity memories can further decrease costs by providing these functions in the chip itself instead of requiring expensive high-performance ATE. Previous researches on BIRA techniques have revealed that it still has some problems that need to be addressed to enhance its cost effectiveness.

Before we discuss BIRA, we define the repair rate because it is the most important and frequently used term in this paper. Repair rate represents the ability of an RA algorithm to find a correct repair solution and was introduced in [1]. Definitions of the repair rate and the normalized repair rate are as follows:

$$\text{repair rate} = \frac{\#\text{ of repaired chips}}{\#\text{ of total tested chips}}$$

$$\text{normalized repair rate} = \frac{\#\text{ of repaired chips}}{\#\text{ of repairable chips}}.$$

The number of total tested chips includes the number of unrepairable chips; the repair rate is influenced by the number of unrepairable chips. There are many factors that may produce unrepairable chips in semiconductor manufacturing including process variations, design originated faults, and human error. Variations resulting from these factors distort the accuracy of an RA algorithm that uses the repair rate. However, the normalized repair rate is independent of these variations. Therefore, the normalized repair rate is more appropriate for indicating the ability of an RA algorithm to obtain correct repair solutions. Optimal Repair rate was introduced in [1] and it is used when the normalized repair rate is 100%.

The three main features of BIRA are area overhead, repair rate, and analysis speed. Reduced area overhead obviously decreases the overall cost of chip production. The loss of repair rate leads to an unwanted yield drop that is not negligible for the mass production of commodity memories. So, repair rate of a BIRA should be optimal, if it is possible, to achieve an acceptable yield. High-density memory requires much longer RA times than low-density memory because of its larger analyzing

Manuscript received April 10, 2008; revised July 02, 2008 and August 27, 2008. First published March 16, 2009; current version published November 18, 2009. This work was supported by Hynix Semiconductor, Inc.

W. Jeong is with the Department of Electrical and Electronics Engineering, Yonsei University, Seoul 120-749, Korea, and also with the Product Development Division, Hynix Semiconductor, Inc., Icheon-si 467-701, Korea (e-mail: woosik.jeong@yonsei.ac.kr; woosik.jeong@hynix.com).

I. Kang and K. Jin are with the Product Development Division, Hynix Semiconductor, Inc., Icheon-si 467-701, Korea (e-mail: ilkwon.kang@hynix.com; kyowon.jin@hynix.com).

S. Kang is with the Department of Electrical and Electronics Engineering, Yonsei University, Seoul 120-749, Korea (e-mail: shkang@yonsei.ac.kr).

Digital Object Identifier 10.1109/TVLSI.2008.2005988

spaces. Faster RA speed has many advantages for BIRA because the test cost is directly proportional to the test time.

Recently, many BIRA approaches for various spare architectures have been introduced, but these are based on 2-D spare architecture [2]–[5]. Redundancy architectures of commodity DRAMs are more complex than those of embedded memories for SOCs. Most of the commodity memories including embedded memories adopt 2-D spare architecture using row and column redundancies [6]–[10]. However, the optimal redundancy allocation problem becomes NP-complete [2], [11]. From the middle of 1980s, various RA algorithms for 2-D redundancy have been developed [1], [6]–[8], [11]–[16]. Among these algorithms, repair-most (RM) algorithm [12], CRESTA [13], LRM [1], and ESP [1] are the most well-known RA algorithms for BIRA. RM is a greedy algorithm, and though its repair rate is high, it is not optimal. However, this simple algorithm has inspired the creation of other algorithms. CRESTA mainly focuses on optimal repair rate and fast analyzing speed. To detect 100% repairable chips, it simultaneously analyzes entire cases of possible solutions with several parallel subanalyzers. It takes no additional RA time after finishing fault collection during test sequence running. However, the number of parallel subanalyzers of CRESTA is exactly the same as the entire number of cases of all possible combinations of available redundancies. Its area overhead increases with an increasing number of redundancies. LRM and ESP mainly focus on minimizing the area overhead of storage requirements with simple RA algorithms. To reduce area overhead, two algorithms are designed to reduce or eliminate the ability of the failure bitmap to store faulty information. However, the repair rates of both algorithms are not optimal because of their excessive omission of faulty information. There has been much research that has attempted to solve the spare allocation problem for reconfigurable memory arrays based on the branch-and-bound (B&B) algorithm [6], [7], [15]–[18]. The B&B algorithm is a simple, fault-driven approach. Among these methods, IntelligentSolve and IntelligentSolveFirst [6] achieve both low area overhead and optimal repair rate. However, both algorithms take a lot of time to complete the RA in cases of complex fault distributions. Although IntelligentSolveFirst is faster than IntelligentSolve, its RA speed is still not fast enough for commercial production purposes. In addition, IntelligentSolveFirst does not guarantee an optimal solution.

Another important feature of BIRA is achievement of the optimal repair solution. The optimal repair solution is the minimum set of spares for a repairable memory block. It takes extra cost to repair overassigned repair solutions during the physical laser-fusing process. In some cases of postpackaging repair [19], the unallocated redundancies during the wafer level repair process can be reused to repair additional package level faulty cells. Obtaining an optimal solution for a memory block in the wafer level test can increase the probability of successfully obtaining postpackage level repair in this application. Therefore, obtaining the optimal repair solution is another important feature of BIRA.

To overcome the disadvantages of previous research, we propose a novel BIRA approach that focuses on optimal repair rate and fast analysis speed. The first idea of the proposed BIRA approach is to reduce search space by building a search tree based on line faults rather than cell faults. To build a line fault-based search tree, fault classification in must-repair faults, single faults, and sparse line faults must be preceded by RA. To support this strategy, a new fault-restoring content addressable memory (CAM) structure is proposed. This helps the proposed BIRA to distinguish between sparse line faults and single faults quickly, and it also reduces the storage requirements to store faulty information by discarding any overlapping row or column faulty addresses. The second idea of the proposed BIRA is to drop branches of the search tree that cannot be a correct repair solution by simple comparing the sum of line fail counts of a branch. By applying this strategy, a new RA algorithm named selected fail count comparison (SFCC) is proposed.

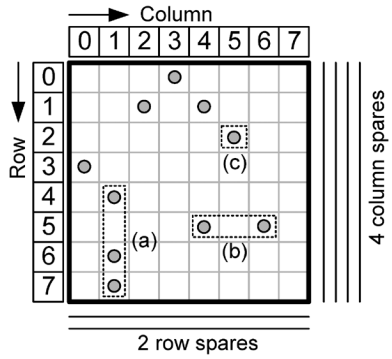

Fig. 1. Example of a memory block using 2-D spare architecture.

In this paper, we summarize the background concerning RA and fault collection in Section II. We also introduce several factors to achieve optimal repair rate in Section III. A new fault-storing hardware structure is proposed with a simple example in Section IV. A proposed RA algorithm, SFCC, is described, and a simple example is given to help understand its analysis algorithm in detail in Section V. Experimental results of the area overhead, repair rate, and RA speed are shown and compared to previous studies in Section VI. Finally, conclusions are given in the last section.

II. Background

A. Classification of Faults in a Memory Block

As we mentioned in Section I, most state-of-the-art memories, including commodity memories, adopt 2-D spare architecture. Most 2-D spare architectures for wafer-level repair consist of spare row lines and spare column lines and obey a line replacement policy. A line replacement policy dictates that any fault in a memory block has to be replaced with a spare line, even for a single fault. If a faulty cell is replaced with a spare row line in a memory block using 2-D spare architecture with a line replacement policy, all other cells sharing the row address with the faulty cell are also replaced with the same spare row line used for the faulty cell.

An example of an 8×8 memory block that has some faults and spare lines is described in Fig. 1. The memory block has two spare rows and four spare columns. All faults in the memory block have to be repaired to produce a good chip. Faults shown in Fig. 1(a) can be repaired with a spare column but not with

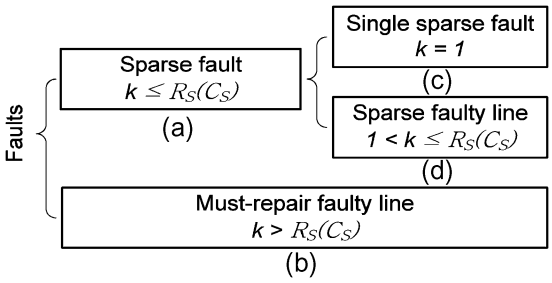

Fig. 2. Classification of faults.

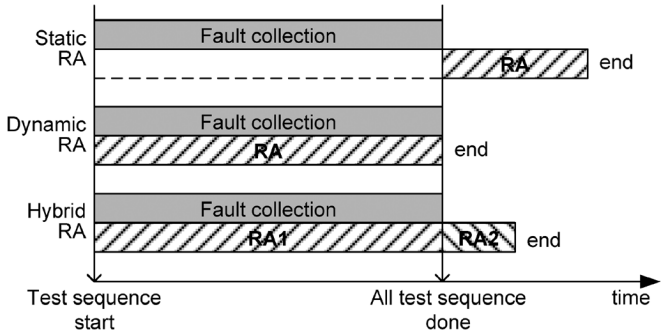

Fig. 3. Three types of RA approaches.

spare rows because the memory block has only two spare rows. However, faults such as those shown in Fig. 1(b) can be repaired with either a spare row or two spare columns. In most of the previous studies, faults were classified only into two types: must-repair faulty lines and others [1], [2], [6]. However, we classify faults into three types: single (sparse) faults, sparse faulty lines, and must-repair faulty lines. Fig. 2 shows the classification of faults.

R_S (C_S) in Fig. 2 represents the number of spare rows (columns). Let the number of faulty cells in a faulty line be k. According to the value of k, faults can be divided into three types defined as follows.

1) Single (sparse) fault: A fault that does not share a row and column address with other faults (i.e., $k = 1$), as shown in Fig. 1(c).
2) Sparse faulty line: A faulty row or column line where $1 < k \le C_S$ for a faulty row and $1 < k \le R_S$ for a faulty column, as shown in Fig. 1(b).
3) Must-repair faulty lines: A faulty row or column line where $k > C_S$ for a faulty row and $k > R_S$ for a faulty column, as shown in Fig. 1(a).

Single fault and must-repair faulty lines were introduced in previous studies [1], [6], [7], [14], [16]. Sparse faulty line faults were handled like other faults, but identification of a sparse faulty line is very important for the proposed BIRA. Classifying faults into three types helps to minimize storage requirements and enhance the RA speed of the proposed BIRA. These three types of faults are frequently used in the rest of this paper.

B. Classification of RA

For memory that uses BIST and BIRA (BISR), BIST applies test algorithms on memory cells and detects fault information. BIRA collects and restores fault information from BIST, and then analyzes the information. For commodity memories without BIST and BIRA, external ATE provides all of these functions. Fault collection and RA are two important functions for RA. RA can be classified into three types by method of fault collection and RA. Fig. 3 shows the three types of RA approaches.

The first type of RA is static RA. This type of RA requires a sufficient failure bitmap size to restore whole failures during test sequence running. After finishing the test sequence, all failure addresses are already saved in the failure bitmap, so any kind of RA algorithm can be applied. However, the main disadvantages of using this type of RA are the high area cost required to prepare the large failure bitmap and the additional search time required depending on the size of the failure bitmap. Although most commodity memories tested with external ATE use static RA, it cannot be adopted for BIRA because of the high area overhead of the failure bitmap.

The second type of RA is dynamic RA. It does not require a full failure bitmap because it analyzes every incoming failure address whenever faults are detected. When all test sequences are finished, the RA is also finished. Many BIRAs such as CRESTA and ESP adopt dynamic RA approaches. This type of RA has been developed to achieve low area overhead by not using a full failure bitmap. However, CRESTA has to multiply area overhead with an increasing number of redundancies, and ESP suffers a loss of repair rate.

The last type of RA is hybrid RA. This type of RA collects failure addresses and executes RA simultaneously for part of the failure addresses as must-repair faulty lines during test sequence running. After finishing the test sequence, the rest of the failure addresses that are not assigned to any spares yet are analyzed. Hybrid RA attempts to compensate for the disadvantages of static RA and dynamic RA. LRM and IntelligentSolveFirst use a hybrid RA. The repair rate of LRM is not optimal, but IntelligentSolveFirst achieves the optimal repair rate with a relatively low area overhead. The proposed BIRA also adopts a hybrid RA.

III. Achieving Optimal Repair Rate

A. Basic Observations for Spare Assignments

As mentioned in Section I, achieving optimal repair rate is a very important feature of BIRA. To achieve optimal repair rate for a memory block using 2-D spare architecture with a line replacement policy, three basic observations can be made for spare assignment during the RA. Some of these observations were also introduced in many previous studies [1], [6], [7], [12], [14]. The three observations are as follow.

Observation 1: A single sparse fault can be replaced with either a spare row or spare column.

Observation 2: A sparse faulty row (column) line can be replaced with a spare row (column). However, it can also be replaced with several spare columns (rows) according to the number of available spares.

Observation 3: A must-repair faulty row (column) must be replaced with a spare row (column).

According to Observation 1, both a spare row and a spare column can be a correct repair solution for a single fault. According to Observation 3, a correct repair solution for a must-repair faulty line is determined (fixed). However, a correct repair solution for a sparse faulty line is not determined. A correct repair solution for a sparse faulty line depends on its situation. Therefore, finding correct repair solutions for sparse faulty lines is the key to achieving optimal repair rate.

B. Optimal Repair Rate for Various Types of BIRA

As outlines in the previous section, analyzing sparse faulty lines is very important for achieving optimal repair rate. In the case of a static RA, all failure information is already restored in the full failure bitmap and there are no more additional faults. Thereafter, RA is executed, so sparse faulty lines can be distinguished and optimal repair rate is possible to be achieved. For a BIRA adopts dynamic type RA, however, it is impossible to distinguish sparse faulty lines from others (i.e., single faults and must-repair faulty lines). This type of BIRA executes RA during test sequence running, so the fault type may or may not be changed depending on the following fault during the rest of the test sequence. If there is a single fault at the point of running a test sequence, it can grow into either a sparse faulty line or a must-repair faulty line depending on the following faults. Similarly, a sparse faulty line at the point of running a test sequence can grow into a must-repair faulty line. However, a must-repair faulty row (column) line at the point of running the test sequence cannot be changed with a must-repair faulty column (row) line. It is impossible to predict whether a newly incoming fault at the point of running the test sequence will be a single sparse fault, a sparse faulty line, or a must-repair faulty line, until the end of all test sequences. The only exception is when the incoming fault is already included in one of the must-repair faulty lines. Therefore, to obtain optimal repair rate for a memory block with 2-D spare architecture, assignment of either a row or a column spare line for a single sparse fault or a sparse faulty line should not be made until the entire test sequence has been finished. Only a must-repair faulty row (column) line can be assigned to a row (column) spare at any time. It is obvious that if any faults are assigned to a must-repair faulty row (column) line during test sequence running, there is no need to restore all pairs of row and column addresses; only the common row (column) address should be restored. However, if there is any newly incoming fault that is not yet identified as a must-repair faulty line during test sequence running, all of its row and column addresses have to be saved to achieve a successful repair solution.

We now define three properties of failure address conservation to achieve optimal repair rate for BIRA using 2-D spare architecture with a line replacement policy.

> *Property 1*: There is no need to save all pairs of addresses but only a common row (column) address for must-repair faulty row (column) lines.

Property 1 is obvious for BIRA obeying a line replacement policy. According to the line replacement policy, if a row address is repaired by a spare row, any fault with the same row address as the repaired row address is also replaced by the line replacement policy and its column address is unimportant. Therefore, keeping only a single side address for the must-repair faulty line does not harm the repair rate of BIRA.

> *Property 2*: All pairs of row and column addresses for sparse faults have to be saved to achieve a proper repair solution during test sequence running.
>
> *Property 3*: Sparse faults must not be assigned to any spare during test sequence running.

RA is known to be NP-complete. To obtain a correct repair solution, CRESTA has parallel sub-analyzers, as many as the number of solution cases. However, most of the BIRAs have just a single RA analyzer, so RA reanalyzing is unavoidable until a correct repair solution is obtained. The repair solution for a must-repair faulty line is already determined. RA reanalyzing is needed only for sparse faults (single sparse faults and sparse faulty lines). RA reanalyzing of sparse faults is impossible without all pairs of addresses of sparse faults. If any sparse fault is determined to be a spare row or column, RA reanalyzing of the sparse fault is impossible. Therefore, Property 2 and Property 3 must be observed to achieve optimal repair rate for a BIRA with a single RA analyzer.

LRM and ESP adopt hybrid RA and dynamic RA, respectively. Both algorithms show good performance with low area overhead and a relatively high repair rate. However, their repair rates are not optimal. For LRM, the main concept is reduction of the size of the local bitmap to save area overhead. However, this reduction is not sufficient to save all fault information, and the algorithm makes a hasty spare assignment because of its lack of storage. Consequently, LRM violates Property 2 and Property 3 resulting in a loss of repair rate. To achieve an optimal repair rate using LRM, the size of the local bitmap should be taken as sufficient as $m = ((R_S+1)C_S+R_S)$ and $n = ((C_S+1)R_S+C_S)$, where m and n are the sizes of the memory block and R_S (C_S) is the number of spare rows (columns) [1]. For ESP, the main concept is to restore only orthogonal faulty addresses [1]. However, reducing storage requirements by dropping half of the addresses for nonorthogonal addresses leads ESP to violate Property 2 and it causes a loss of repair rate. Unfortunately, there is no way to achieve an optimal repair rate using ESP. Besides ESP, CRESTA also adopts a dynamic RA but the RA is not required to obey these properties because CRESTA has several parallel subanalyzers. Except for CRESTA, almost every BIRA has just a single redundancy analyzer to avoid the additional area overhead of using parallel analyzers.

Even if a BIRA observes both Property 2 and Property 3 in fault collection, it does not always guarantee optimal repair rate. For example, if the RM algorithm is adopted to analyze a memory block using 2-D spare architecture while fully observing both Property 2 and Property 3 in fault collection, its repair rate is high but not optimal. Therefore, to achieve the optimal repair rate, Property 2 and Property 3 are strictly observed and a good RA algorithm must be used for a BIRA using 2-D spare architecture with a single redundancy analyzer. LRM has to adopt a good RA algorithm to achieve the optimal repair rate even though its size of bitmap is large enough since its RA algorithm is based on RM algorithm.

Besides LRM, ESP, and RM algorithms, IntelligentSolve-First observes all three properties relatively well and it shows good performance with respect to area overhead, repair rate,

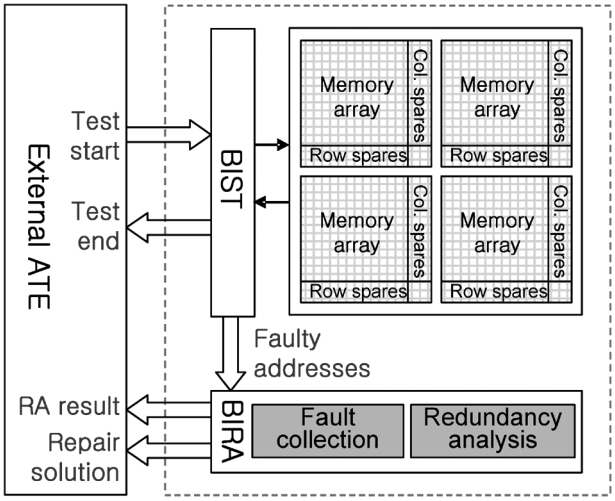

Fig. 4. Overview of memory BIST and BIRA.

and analyzing speed. The proposed BIRA also obeys all three properties to achieve optimal repair rate. In addition, the proposed BIRA improves area overhead and analyzing speed compared to state-of-the-art BIRA techniques such as Intelligent-SolveFirst.

IV. Proposed BIRA Approach

A. Overview of the Proposed BIRA

We propose a new hybrid-type BIRA approach. Fig. 4 shows an overview of memory BIST and BIRA. Even if logic circuits on the BIST and a BIRA are implemented in semiconductor memory, it is impossible to test the memory without the help of external ATE. However, it is possible to test a memory at a low cost using a simple functional ATE instead a multifunctional expensive ATE. The proposed BIRA has two functional components: 1) fault collection block during test sequence running and 2) RA block after finishing all test sequences. During test sequence running, must-repair faulty lines are assigned to the proper spare lines, and sparse faults are restored into internal CAMs that support fast address comparison between incoming faults and previous restored faults [20]. We assume that all CAM cells have passed the manufacturing test beforehand. After finishing the test sequence, SFCC analyzes sparse faults and finds a correct repair solution. In order to reduce storage requirements to restore failure information and enhance identification of sparse line faults and single faults, we propose a new fault storing structure. Also, we propose a novel RA algorithm to achieve optimal repair rate along with a fast analysis speed. A detailed description of our proposed BIRA approach follows.

B. Fault Collection

The proposed fault collection structure is based on ESP [1] and IntelligentSolveFirst [6]. Both ESP and IntelligentSolveFirst have very small area overhead of fault collection. However, ESP cannot achieve an optimal repair rate and IntelligentSolveFirst is not appropriate for building a line-based searching tree. To build a line-based searching tree quickly and minimize the area overhead of fault collection, a new fault-storing CAM structure is proposed, as shown in Fig. 5.

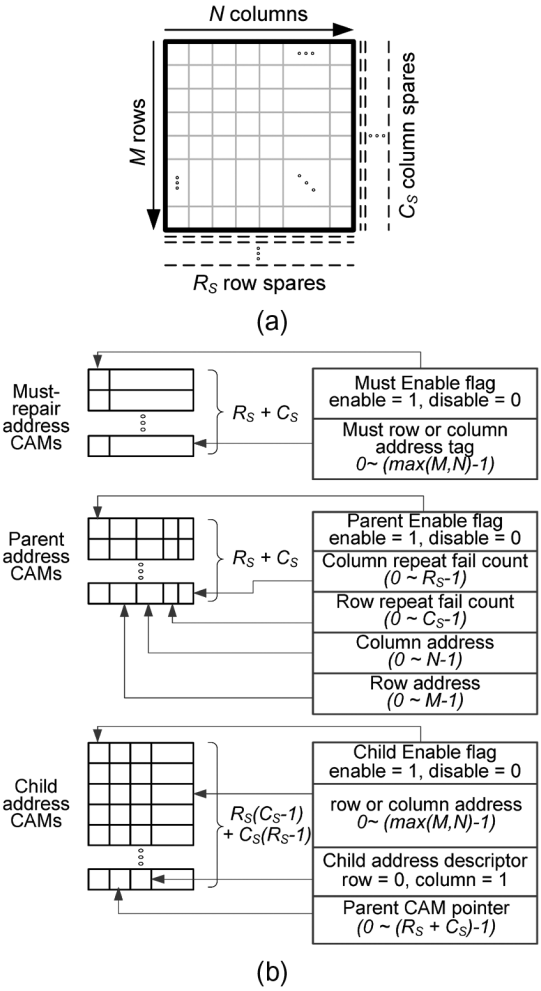

Fig. 5. Fault-storing CAM structure of the proposed BIRA.

Fig. 5(a) shows an $M \times N$ memory block using 2-D spare architecture with R_S spare rows and C_S spare columns. Fig. 5(b) shows fault-storing CAM structure of the proposed BIRA for a memory block such as Fig. 5(a). Fault-storing CAMs, consist of *must-repair address CAMs*, *parent address CAMs*, and *child address CAMs*, as seen in Fig. 5(b). The maximum number of both must-repair address CAMs and parent address CAMs is the same as the sum of the total spares (i.e., $R_S + C_S$). The maximum number of child address CAMs is $R_S(C_S - 1) + C_S(R_S - 1)$. The sum of the number of parent address CAMs and the number of child address CAMs is $2 * R_S * C_S$; the size of CAMs to restore sparse faults has already been introduced in [1] and [6].

The concept of spare pivot was introduced in [1]. The spare pivot is the first faulty address that can be a new line fault according to the following faulty addresses. When a spare pivot occurs, it does not share its row and column address with other spare pivot addresses. The proposed BIRA stores spare pivot faults into parent address CAMs. In this case, spare pivot addresses in parent address CAMs have unique row and column address only among themselves. The row (column) repeat fail count of the parent address CAM stores the number of its child address that shares a row (column) address with the parent address. If a new incoming fault occurs and shares its row (column) address with one of its parent addresses, the row (column) repeat

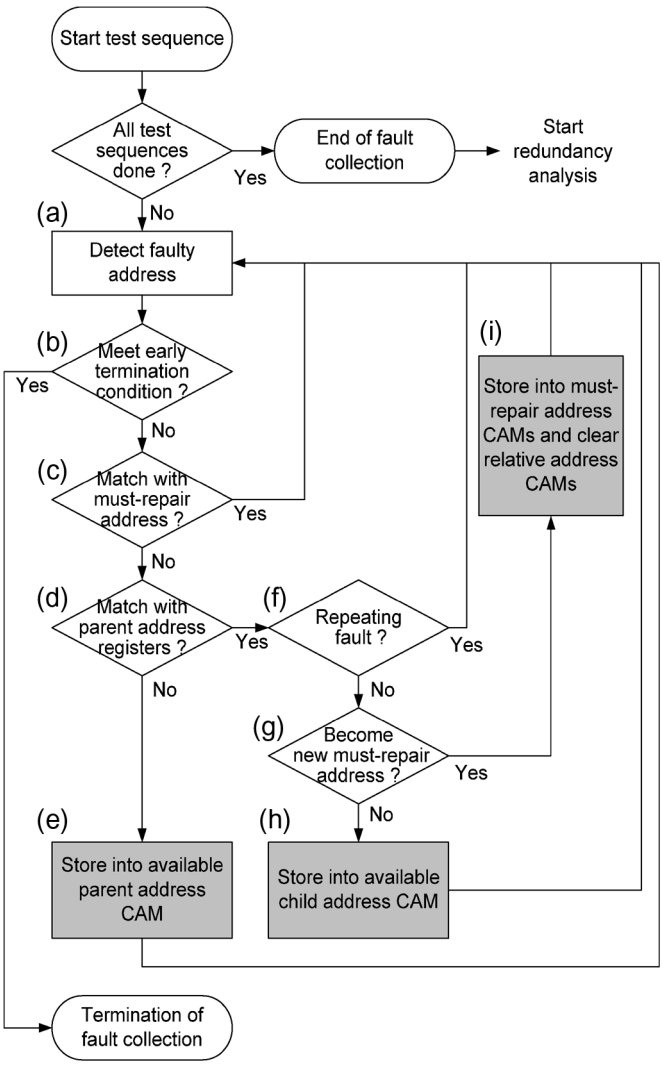

Fig. 6. Fault collection flow of the proposed BIRA.

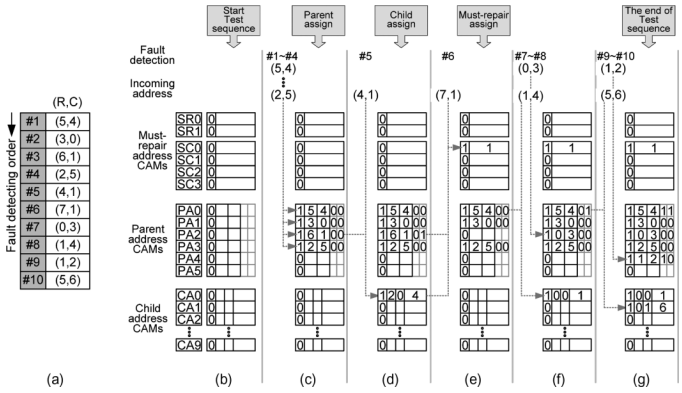

Fig. 7. Example of the fault collection process of the proposed BIRA.

fail count of its parent address is increased and the column (row) address of the incoming fault is stored in a child address CAM with the pointer of its parent CAM. If the number of row repeat fail counts of a parent address is equal to C_S, i.e., the number of faults on the row line is greater than CS because the initial value of repeat fail count is 0, the row address becomes a must-repair faulty row line. By Property 1, the row address is stored in a must-repair address CAM, and enable flags of the parent address CAM and child address CAMs, which share the row address, are initialized. Similarly, with a column line with more than R_S faults in it, the column address is stored in a must-repair address CAM.

The proposed BIRA observes the following guidelines for fault collection, as shown in Fig. 6. If a fault is detected by BIST, BIRA receives the faulty information in Fig. 6(a). Early termination conditions to speed up the BIRA analysis speed were already introduced in [1] and [6]. Before storing the incoming faulty address, BIRA checks whether the current status meets two *early termination conditions* to avoid unnecessary work in Fig. 6(b). These two conditions are as follows. R_a (C_a) of Condition 1 is the number of available spare rows (columns). These are simply calculated by subtracting the number of must-repair faulty row (column) lines and faulty spare row (column) lines from the total spare row (column) lines. Valid CAMs of Condition 2 mean valid parent address CAMs and valid child address CAMs

$$\text{Condition 1} : R_a > 0 \text{ or } C_a > 0$$
$$\text{Condition 2} : \#\text{ of valid CAMs} \leq (R_a * C_S + C_a * R_S).$$

Condition 1 means that there is at least one available spare line to replace a fault, otherwise the case is unrepairable. The maximum faults for R_a available spare rows is $R_a * C_S$ because a maximum number of faults in a sparse faulty row is C_S. Similarly, the maximum faults for C_a available spare rows is $C_a * R_S$ because a maximum number of faults in a sparse faulty row is R_S. Therefore, Condition 2 must be met for a memory block that is still repairable when it has just R_a and C_a available. If a case violates at least one of these two conditions, the memory block is unrepairable, and the BIRA terminates all subsequent sequences for the memory block. Otherwise, the incoming fault is compared with previously stored must-repair address CAMs. BIRA skips later processes when the row (column) address of the incoming fault is the same as any must-repair faulty row (column) addresses, as seen in Fig. 6(c). Then, BIRA compares the incoming faulty address with the parent address CAMs. If both the row and the column address of the incoming fault do not match with any parent addresses, as shown in Fig. 6(d), the incoming fault is stored in a parent address CAM, as shown in Fig. 6(e). In Fig. 6(f), BIRA skips later processes and returns to the state of Fig. 6(a) when the address of the incoming fault is identical to that of any previously stored sparse faults. Otherwise, BIRA checks whether the incoming faulty row or column address becomes a must-repair faulty line in Fig. 6(g). If the row (column) address of incoming fault cannot be a must-repair address, the row (column) address is stored into a child address CAM in Fig. 6(h). Otherwise, the faulty address is stored into a must-repair address CAM in Fig. 6(i). This is the end of fault collection flow for the current fault and BIRA waits the next incoming fault.

Fig. 7 shows an example of how the proposed BIRA collects faulty information for the same memory block shown in Fig. 1. The first column of Fig. 7(a) represents the detection order of the incoming failure addresses. The second column of Fig. 7(a) represents a pair of row (R) and column (C) addresses of an incoming fault. At the beginning of the test sequence, all must-repair address CAMs, parent address CAMs, and child address

CAMs are initialized, as shown in Fig. 7(b). In this example, the number of must-repair address CAMs is 6: two row must-repair address CAMs and four column must-repair address CAMs. A must-repair address CAM consists of four bits: one bit for the enable flag and three bits for addresses to express all addresses from 0 to 7. The number of parent address CAMs is 6 and the number of child address CAMs is 10. A parent address CAM consists of ten bits: one bit for the enable flag, three bits for the row CAM, three bits for the column CAM, and the last three bits for the row/column repeat fail count. A child address CAM consists of eight bits: one bit for the enable flag, three bits for the parent CAM pointer, one bit for the child address descriptor, and three bits for the child address CAM.

Now, the first fault is detected and the fault is compared with all must-repair address CAMs; however, there is nothing to be compared to in the must-repair address CAMs yet. Then, the first fault is compared with parent address CAMs and it is stored into parent address CAMs. The first four faults have been restored into parent address CAMs, as shown in Fig. 7(c). Because the first four faults did not coincide with any must-repair address CAMs, all of them are spare pivot addresses. When the fifth fault [i.e., cell (4,1)] is detected, the column address of the fault is the same as that of the third parent address CAM [i.e., cell (6,1)], so the column repeat fail count is increased to 1 and the fault is stored in the first child address CAM as in Fig. 7(d). For the first child address CAM, the enable flag is set to one. The parent CAM pointer of the first child CAM is set to 2 because the pointer of the parent address CAM starts at zero. The address descriptor is set to 0 (i.e., row address) and the child address is set to 4 because the child address to store is the row address of the cell (4,1). When the sixth fault is detected, column 1 becomes a must-repair faulty column address. Column 1 is stored into the first must-repair faulty column CAM and its related CAMs are cleared (i.e., the third parent address CAM and the first child address CAM), as shown in Fig. 7(e). The seventh fault is spare pivot and it is stored in the parent address CAM, as shown in Fig. 7(f). The eighth fault is stored in a child address CAM, as shown in Fig. 7(f). The ninth fault and the tenth fault are also stored in a parent address CAM and a child address CAM, respectively, as shown in Fig. 11(g). For the tenth fault, although it is spare pivot, its row address is shared with the first child address, its row repeat fail count is set to 1. Now all test sequences are finished. Fig. 7(g) shows the final status of fault collection CAMs of the proposed BIRA.

By following the proposed fault collection process, the proposed BIRA reduces storage requirements for storing faulty information without loss of any essential faulty information. In addition, faulty line identification and counting repeat fails for each faulty line are automatically done. Now, a detailed description of our RA algorithm follows.

C. Redundancy Analysis

The spare allocation problem is an NP-complete problem that means that there is no heuristic algorithm to solve this problem with a single analysis. Some fault-driven approaches such as B&B and IntelligentSolveFirst search for a solution by applying an exhaustive search algorithm, but they account for the worst-case exponential time complexity. RA reanalyzing is inevitable to achieve optimal repair rate for a BIRA with a single RA analyzer. Although a heuristic algorithm cannot guarantee an optimal repair rate, it can reduce analysis complexity. Generally, a fault-driven approach searches for a solution bit by bit, and the time complexity of a binary search relies on the depth of its searching tree. However, the proposed BIRA searches for a solution line by line. Because it makes the depth of the search tree much shallower, its time complexity is greatly reduced. To build an efficient line-based search tree, proper preprocessing is needed that requires a small amount of extra time and hardware overhead. Three conditions are proposed to speed up the RA speed of a line-based search tree. We now describe the RA algorithm of the proposed BIRA approach in detail.

After finishing fault collection, must-repair faulty lines have already been assigned to the proper spare lines and only sparse faults (i.e., single sparse faults and sparse faulty lines) are stored in fault-storing CAMs. Now, sparse faults need to be analyzed to obtain the final repair solution for a memory block. The main idea of the RA algorithm of the proposed BIRA is derived from the first and the second observations mentioned previously in Section III.

Sparse faults consist of two kinds of faults: single faults and sparse faulty lines. Single faults can be assigned any available spares, so the repair decision for single faults can be postponed until the repair decisions for sparse faulty lines have been determined. Therefore, finding a correct repair solution for a memory block actually depends on analyzing sparse faulty lines rather than single faults. In order to analyze sparse faulty lines effectively, we propose a new RA algorithm named SFCC. The SFCC algorithm is based on RM algorithm. The repair rate of RM is not optimal but SFCC improves the repair rate up to 100% with a fast analysis speed by building a search tree based on line faults and using a simple fail count comparison algorithm for analysis. Fig. 8 shows the proposed RA algorithm described with c-language style pseudocode.

SFCC consists of three main function modules: *get faulty line from CAM*, *descending faulty line sort*, and *fail count comparison*. SFCC focuses on fast RA analysis with an optimal repair rate. To build a line-based search tree, sparse faulty lines and single faults must be identified and SFCC algorithm searches all sparse faulty lines in the *get faulty line from CAM* function in the SFCC_RA module in Fig. 8. This function searches for all sparse faulty lines in the parent address CAMs and returns faulty line addresses. According to the row and the column repeat fail count of parent CAMs, SFCC identifies faulty lines and single faults with fail counts. SC, RBC, and CBC (number of single faults, number of row faulty lines, and number of column faulty lines, respectively) are also counted in this function. Then, SFCC checks a condition to improve RA speed is given as

$$\text{Condition 3}: ((SC + RBC + CBC) \le (C_a + R_a)) \\ (RBC \le R_a)\,(CBC \le C_a).$$

Condition 3 is *the simplicity check condition* for a memory block. If a memory block satisfies this condition, it means that it is simple analysis case. A faulty line as well as single fault can be repaired by a spare line, so a memory block, which satisfies Condition 3, can also be repaired because the sum of faulty

```
R_S, C_S : total number of spares
BUF_MAX : total number of sparse faulty line buffers
R_a, C_a : number of available spares
BUF_k : k^th sparse faulty line buffer address
EBUF_k : k^th enable flag of each sparse faulty line buffer
TYPE_k : k^th row or column type flag of each sparse faulty line buffer
BUFC : sparse faulty line count
SC : single sparse fault count
RBC (CBC) : sparse faulty row (column) line buffer count
sp_cnt : selected sparse faulty line count
-------------------------------------------------------------------------------------
SFCC_RA () {
      result =1;
      while (1) {
         (SC, RBC, CBC) = get_faulty_line_from_CAM ();
          if (Condition 3 ) { faulty_line_assign (); break; }          ← (a)
        if (((R_a==0) || (RBC==0)) && ((C_a==0) || (CBC==0))) break;
         descending_faulty_line_sort();  result =0;
         for all sp_cnt, 1≤ sp_cnt ≤ (R_a+C_a - SC) {
            fail_count_comparison (sp_cnt);
            if (result ==1) break;
         }
         if (result ==0) break;
      }
      if (result==1) { single_bit_assign (); }
      if (remain_sparse_CAM > 0) repair_fail ("Unrepairable");
       else repair_success ("Repairable");
}
get_faulty_line_from_CAM () {
      for all parent address CAMs {
           if (parent_enable ==0) continue;
           if (column_repeat_fail_count > 0) {
              update_faulty_line_buffer (column address);BUFC++; }
            if (row_repeat_fail_count > 0) {
              update_faulty_line_buffer (row address); BUFC++; }
       }
}
descending_faulty_line_sort () {
      descending_sort_of_buffer ();
      for all k, 0 ≤ k < BUFC {
         BUF_k = sparse_faulty_address;
         if (BUF_k == row_address) TYPE_k = 0;
         else if (BUF_k == column_address) TYPE_k = 1; }
      get_intersection_for_each_line ();
}
fail_count_comparison (sp_cnt) {
      Critical_fail_count = TFC - (R_a+C_a) + sp_cnt;
      for all k, max. SFC ≥ k ≥ min. SFC {
         select_faulty_lines (k); get_SFC (k);
         if (SFC (k) < Critical_fail_count) continue;
         else if (SFCR (k) ≥ Critical_fail_count) {
            faulty_line_assign (k);
            result =1; break;}
      }
}
```

Fig. 8. Proposed RA algorithm—SFCC.

lines and single faults is equal to or less than the sum of available spares. In this case, SFCC executes the *faulty line assign* function for sparse faulty lines and *single bit assign* function for single faults while skipping the rest of the time-consuming subsequent functions such as *descending faulty line sort* and *fail count comparison* in Fig. 8. We show later that this is very powerful strategy to reduce the RA time based on experiments in Section V.

If a memory block does not satisfy Condition 3, SFCC arranges sparse faulty lines in decreasing order by the number of fail counts on each faulty line in the *descending faulty line sort* function in Fig. 8. Fig. 9 shows a sparse faulty line buffer structure of SFCC for a memory block such as Fig. 5(a). The sufficient number of sparse faulty line buffers is $2*(C_S+R_S)$, where

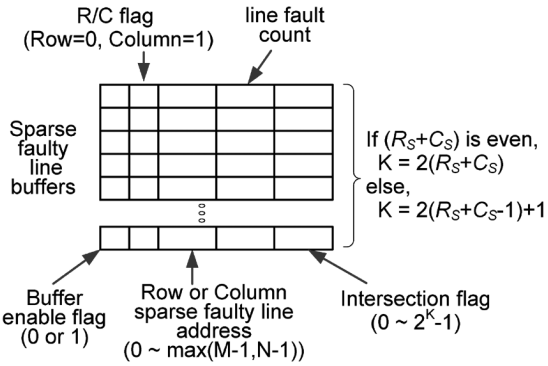

Fig. 9. Sparse faulty line buffer structure.

(C_S+R_S) is even, otherwise $2*(C_S+R_S-1)+1$. The memory block of Fig. 5(a) has C_a available spare columns and R_a available spare rows after finishing fault collection. Then, intersection flags for each faulty line are set in the *get intersection for each line* function in Fig. 8. To generate intersection flags for the sparse faulty line buffers, sparse faults except for single faults, which were already identified by *get fault line from CAM*, are checked one by one to find out which faulty line buffers share address with each sparse fault. If a sparse fault shares its row address with the ith faulty line buffer and its column address with the jth faulty line buffer, where $0 \leq (\text{both } i \text{ and } j) < \text{the maximum number of sparse faulty line buffers}$, the sparse fault is an intersection fault between two faulty lines. In this case, the jth intersection flag of the ith faulty line buffer is set to 1 and the ith intersection flag of the jth faulty line buffer is also set to 1. The jth intersection flag of the ith faulty line buffer means that the ith faulty line buffer intersects with the jth faulty line buffer.

Now, all sparse faulty lines are arranged into buffers in decreasing order of their fail counts. Then, SFCC executes the fail count comparison function. This function selects sparse faulty line combinations from one sparse faulty line to $(C_a + R_a - S_C)$ number of sparse faulty lines, where SC is the number of single faults counted already from *get faulty line from CAM*, and checks whether each combination can achieve a proper repair solution. In order for quick execution with easy confirmation of the repair possibility of each combination of sparse faulty lines, two judgment conditions are defined as follows, where *total failure count* (TFC) is the sum of sparse faults in the parent and child address CAMs. SFC is the simple sum of fail counts of sparse faulty lines for a combination. *Selected failure count real* (SFCR) is the number of faults that are covered by a combination of sparse faulty lines. It is calculated by subtracting the number of intersection faults of a combination from SFC. SFC is always greater than or equal to $SFCR$. sp_cnt is the number of selected sparse faulty lines of a combination

$$\text{Condition } 4 : SFC \geq (TFC - ((C_a + R_a) - sp\_cnt))$$
$$\text{Condition } 5 : SFCR \geq (TFC - ((C_a + R_a) - sp\_cnt)).$$

Both Condition 4 and Condition 5 are compared in the *fail count comparison* function of Fig. 8. Condition 4 is *prejudgment condition* to abort analysis of a combination and goes on to the next combination when a combination does not meet it.

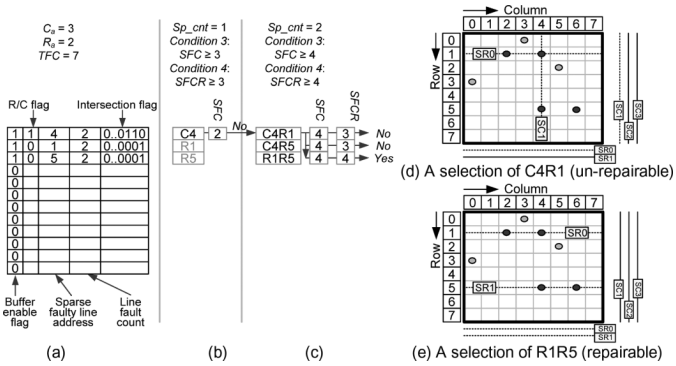

Fig. 10. Example of SFCC analysis process. (d) Selection of C4R1 (unrepairable). (e) Selection of R1R5 (repairable).

Condition 5 is *final judgment condition* of RA and a combination that satisfies the condition is the correct repair solution. If a combination of sparse faulty lines does not satisfy Condition 4, SFCC aborts analysis, otherwise SFCC checks whether the combination satisfies Condition 5 or not.

It is obvious that a spare line can replace at least one fault. If there are a number of available spare lines, at least the same number of faults can be repaired by them. Consider a combination that selects as many sparse faulty lines as sp_cnt of the total $(C_a + R_a)$ number of available spare lines to check its repair possibility. The number of unselected available spare lines is $((C_a + R_a) - sp\_cnt)$. If the combination of selected sparse faulty lines can replace as many as $(TFC - ((C_a + R_a) - sp\_cnt)$ faults, then the rest of the unrepaired faults can be repaired by unselected available spare lines. If this is the case, the combination of selected sparse faulty lines could be one of the correct repair solutions. Actually, Condition 5 must be met to obtain a correct repair solution rather than Condition 4. However, SFCC checks Condition 4 before Condition 5, because SFC is much faster to calculate than $SFCR$. If all combinations of a memory block do not meet both Condition 4 and Condition 5, the memory block is judged unrepairable memory, otherwise it is judged repairable memory.

Fig. 10 shows an example of the RA process using SFCC for sparse faults of the previous memory block in Fig. 7. As shown in Fig. 7(g), the memory block has one column must-repair address, five parent addresses, and two child addresses. Fig. 10(a) shows sparse faulty line buffers for the memory block after fault collection is finished. The maximum number of sparse faulty line buffers of Fig. 10(a) is 12 because the memory block has four spare column lines and two spare row lines (i.e., $2 * (2 + 4) = 12$). The first child address shares the column address with the first parent address (i.e., cell (4,1) for the child address and cell (5,4) for the parent address). Similarly, the second child address shares the row address with the first parent address (i.e., cell (5,6) for the child address and cell (5,4) for the parent address). Therefore, there are two sparse faulty lines in the memory block (i.e., row 5 and column 4). The first child address coincides with the row address of the fifth parent address (i.e., cell (1,4) for the child address and cell (1,2) for the parent address) and there is another sparse faulty line (i.e., row 1). Parent addresses that have no child address or no identical address with child addresses, they are single faults. As a result, there are three sparse faulty lines in the memory block (i.e., row 1, row 5, and column 4) and three single faults [i.e. cell (3,0), cell (0,3), and cell (5,6)]. Information about all sparse faulty lines is stored into sparse faulty line buffers in decreasing order of fail counts. In this example, all three sparse faults have the same fail counts, so the information is stored in increasing address order with column preference.

Intersection flags for sparse faulty lines are set by searching parent addresses that are not single faults and child addresses. In this case, the first and the fifth parent addresses and the first and the second child addresses are not single faults. For the first parent address [i.e., (5,4)], it shares both row 5 and column 4. It is the intersection fault for the first and the third faulty lines so the third bit of the first intersection flag and the first bit of the third intersection flag are set to 1. Similarly, for the first child address [i.e., (1,4)], the second bit of the first intersection flag and the first bit of the second intersection flag are also set to 1.

Fig. 10(a) indicates that column 4 has two fail counts on it. The number of available spare columns is 3 (i.e., $C_a = 3$) and the number of available spare rows is 2 (i.e., $R_a = 2$) because the memory block has one must-repair faulty column line. The total sparse fail counts of the memory block is 7 (i.e., $TFC = 7$). All preprocessing for RA of SFCC is now done. SFCC then builds a line-based search tree and starts RA.

The memory block has three single sparse faults, and three spare lines must be reserved to replace them. Thus, only two spare lines are available to repair three sparse faulty line of Fig. 10(a). Fig. 10(b) and (c) shows the process of SFCC algorithm. Fig. 10(b) shows three possible combinations for $sp\_cnt = 1$. $SFC(R)$ of all combinations for $sp\_cnt = 1$ should be greater than or equal to 3 to meet both Condition 4 and Condition 5. The first combination is C4 (i.e., column 4). SFC of column 4 is only 2, and this combination cannot meet Condition 4. Every first combination for all sp_cnt (i.e., C4 for $sp\_cnt = 1$ and C4R1 for $sp\_cnt = 2$) has the biggest SFC value because contents of faulty line buffers are arranged in decreasing fail count order. Hence, SFCC skips to execute *fail count comparison* for the rest of the selections for $sp\_cnt = 1$ [i.e., R1 and R5 of Fig. 10(b)]. Fig. 10(c) shows three possible combinations for $sp\_cnt = 2$, and $SFC(R)$ should be greater than or equal to 4 to meet both Condition 4 and Condition 5. The first combination for $sp\_cnt = 2$ is C4R1 (i.e., column 4 and row 1) and its SFC value is 4, this combination satisfies Condition 4. However, $SFCR$ of C4R1 is 3 because C4R1 combination has one intersection fault. As a result, this combination cannot meet Condition 5. SFCC continues to execute *fail count comparison* for other combinations for $sp\_cnt = 2$ until there is no satisfying combinations for Condition 4. The second combination (i.e., C4R5) also meets Condition 4 but cannot meet Condition 5. Finally, the third combination is R1R5 (i.e., row 1 and row 5) and this combination meets both Condition 4 and Condition 5. R1R5 is the correct repair solution for three sparse faults of the memory block. Fig. 10(d) shows that the combination of C4R1 cannot repair all three sparse faulty lines. Fig. 10(e) shows the memory block after replacing three sparse faulty lines by SR0 for row 1 and SR1 for row 5. Now, there is no more sparse faulty line in the memory block and SFCC aborts *fail count comparison*. Only three single faults remain

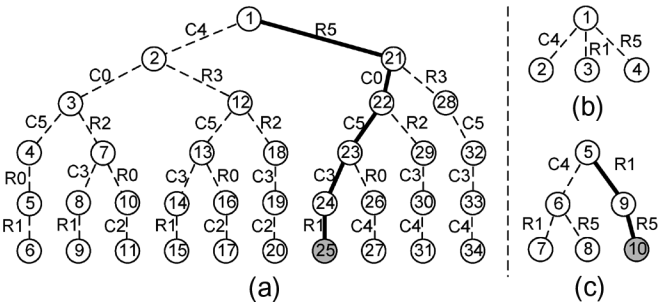

Fig. 11. Searching tree based on cell fault and on line fault.

in the memory block [i.e., cell (3,0), cell (0,3), and cell (2,5)], and they can be repaired by three available spare columns (i.e., SC1, SC2, and SC3). In this example, a correct repair solution for the memory block is column 1 (for must-repair faulty line), row 1, row 5 (for sparse faulty lines), column 0, column 3, and column 5 (for single sparse faults). This is the end of the RA using SFCC for the memory block and the proposed BIRA sends a *repair success* signal ($\mathrm{RA\ result} = 1$ of Fig. 4) to the BIST or external ATE.

Fig. 11(a) shows a search tree based on a cell fault that is used in IntelligentSolveFirst. Fig. 11(b) and (c) shows a search tree based on line faults for the example in Fig. 10 when $\mathrm{sp\_cnt} = 1$ and $\mathrm{sp\_cnt} = 2$, respectively. Fig. 11(b) has only three branches and two of them are skipped analysis because they do not meet Condition 4. Fig. 11(c) also has three branches while Fig. 11(a) has ten branches. In this case, 40% of the search space is reduced by building a search tree based on line faults and the depth of the search tree is very shallow compared with cell fault-based search tree. Consequently, building a search tree based on line faults reduces both the number of branches to search and the depth. For these reasons, SFCC is able to analyze faster than IntelligentSolveFirst.

V. Simulation Results

Area overhead, repair rate, and analysis time are the most important features of a BIRA when measuring its performance. Now, the experimental results considering these three main features of BIRA will be presented and discussed. In order to estimate the performance of BIRA, we developed a simulation tool in c-language named *RepairSim*. The overall diagram of *RepairSim* is shown in Fig. 12. Three kinds of input data are required to execute *RepairSim*: general information, memory block information, and algorithm information. According to the input information, *RepairSim* generates faulty addresses at random. Then, these faulty addresses are randomly reordered by their insertion order to save random fault generation time. After finishing RA by *RepairSim*, several output data are generated: storage requirements data, repair rate, solution efficiency, CPU time, clock cycles, and fault statistics. First, in order to compare area overhead, the storage requirements are estimated by *RepairSim*. Generally, most BIRAs include storage to restore failure information. Most commodity DRAM has more than 100 memory array blocks in a chip. Each block requires an independent fault collection module but a single analyzer can cope with all blocks. For the proposed BIRA, the area overhead of a single SFCC analyzer is below 2% of the total area overhead of BIRA for

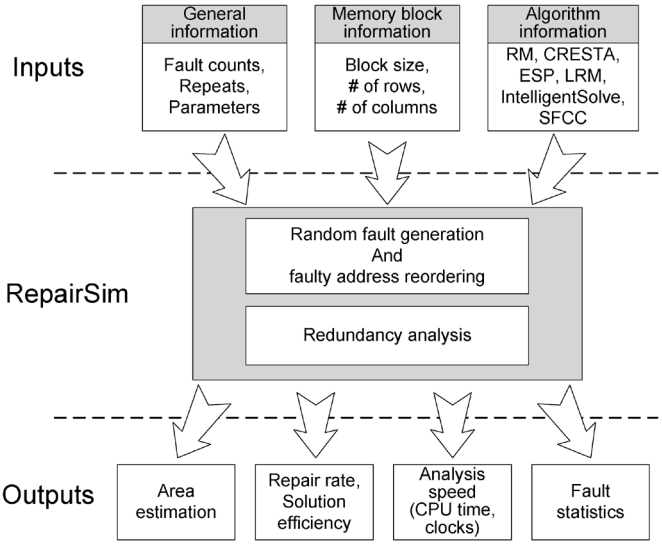

Fig. 12. Overview of RA simulator—RepairSim.

a memory assuming that the $\mathrm{number\ of\ memory\ blocks} = 100$, $M = 1024$, $N = 1024$, $R_S = 5$, and $C_S = 5$.

Therefore, the area of storage requirements is not exactly the same as the area of the whole BIRA but the area of storage cells dominates the BIRA. An $M \times N$ memory block using 2-D spare architecture, as shown in Fig. 5(a), has R_S spare rows and C_S spare columns. The equations to calculate storage requirements for each BIRA are shown below. A_{LRM}, A_{ESP}, A_{CRESTA}, $A_{\mathrm{INTELLIGENT}}$, and A_{PROPOSED} from (1) to (6) represent the number of bits required for each algorithm. Equations for LRM and ESP were introduced in [1]. Reducing the size of the local bitmap by customizing the m and n values of (1) is the main strategy taken by LRM to reduce area overhead. However, the insufficient size of the local bitmap of LRM causes loss of repair rate. To achieve optimal repair rate with LRM, the size of the local bitmap is assumed to be $m = (R_S(C_S + 1) + R_S)$ and $n = (C_S(R_S+1)+C_S)$ in (1) and a good RA analyzer must be added. All estimated area overhead equations consist of two kinds of variables: the number of spares (i.e., C_S and R_S) and the size of the memory block (i.e., M and N). We compare the required storage cells with the number of spares and the size of the memory block

$$\begin{aligned} A_{\mathrm{LRM}} &= m \times n + [(\log_2 M + 1) + (\log_2(n+1))] \\ &\quad \times m + [(\log_2 N + 1) + (\log_2(m+1))] \\ &\quad \times n + A_{\mathrm{spare\_register}} \qquad (1) \\ \text{where } m &= ((R_S + 1)C_S + R_S) \\ n &= ((C_S + 1)R_S + C_S) \\ A_{\mathrm{ESP}} &= (R_S + C_S)(\log_2 M + 1 + \log_2 N + 1) \\ &\quad + A_{\mathrm{spare\_register}} \qquad (2) \\ A_{\mathrm{CRESTA}} &= A_{\mathrm{spare\_register}} \\ &\quad \times (R_S + C_S)!/(R_S! \times C_S!) \qquad (3) \\ A_{\mathrm{INTELLIGENT}} &= 2R_SC_S(\log_2 M + \log_2 N + 1) \\ &\quad + 2R_SC_S(\log_2 R_S + \log_2 C_S) \\ &\quad + A_{\mathrm{spare\_register}} \qquad (4) \end{aligned}$$

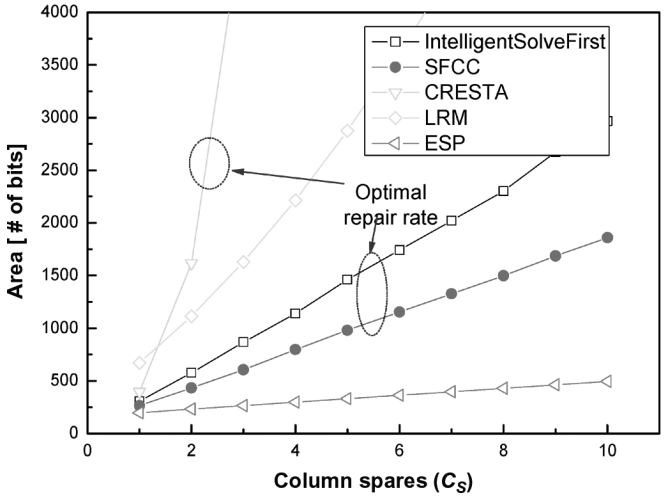

Fig. 13. Area estimation with different column spares ($M = 1024$, $N = 1024$, and $R_S = 5$).

$$A_{\text{PROPOSED}} = (R_S + C_S) \times (\log_2 M + \log_2 N + 1) + (R_S + C_S) \times (\log_2 R_S + \log_2 C_S) + (R_S \times (C_S - 1) + C_S \times (R_S - 1)) \times (\log_2 (\max(M, N)) + \log_2 (R_S + C_S) + 1) + A_{\text{spare\_register}} \quad (5)$$

$$A_{\text{spare\_register}} = [(\log_2 M + 1) \times R_S + (\log_2 N + 1) \times C_S]. \quad (6)$$

According to the area estimation results generated by *RepairSim*, ESP has the smallest storage cells compared to the other algorithms but ESP cannot achieve an optimal repair rate. Besides ESP, IntelligentSolveFirst and the proposed BIRA have relatively smaller storage cells than both CRESTA and LRM for all cases of $R_S \geq 2$ and $C_S \leq 2$, but the proposed BIRA requires much smaller storage cells than IntelligentSolveFirst. Note that (4) and (5) have the $(\log_2 R_S + \log_2 C_S)$ term for the repeat fail counter. In (4), for IntelligentSolveFirst, the repeat fail count is required for all not must-repair faults. Any faults of IntelligentSolveFirst that are not must-repair faults can be a must-repair fault according to the following faults. When a new fault is detected, a BIRA has to know the number of detected faults that share their row or column address with the fault to decide whether the fault is a new must-repair fault or not. However, the must-repair decision is impossible in a cycle without repeat fail counters. In the proposed BIRA, the repeat fail counter is required only for parent address CAMs [see (5)].

The proposed BIRA requires on average 33% smaller storage cells than IntelligentSolveFirst when $M = 1024$, $N = 1024$, $R_S = 1 - 10$, and $C_S = 1 - 10$. Furthermore, increasing both the number of spares and the memory block size makes the gap between SFCC and IntelligentSolveFirst wider in Figs. 13 and 14. Therefore, the proposed BIRA has the smallest area overhead of fault collection of all BIRAs evaluated that achieve optimal repair rates.

The next feature of BIRA that will be discussed is the repair rate. The postrepair yield of a memory depends entirely on the

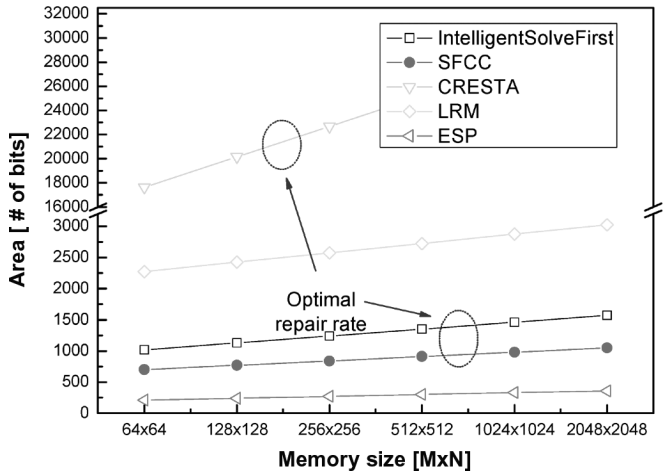

Fig. 14. Area estimation with different memory block sizes ($C_S = 5$ and $R_S = 5$).

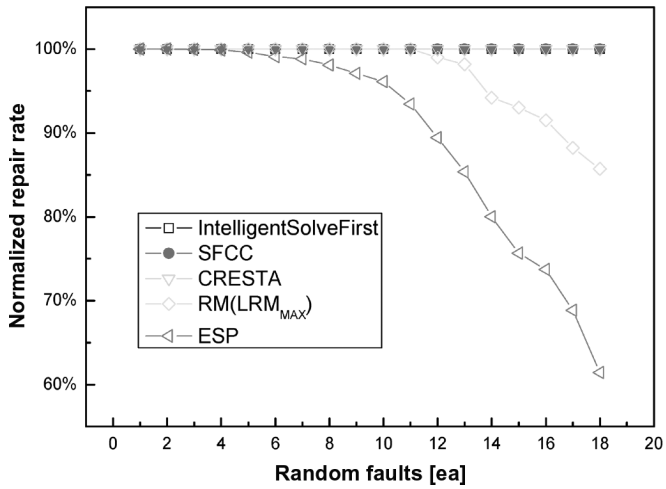

Fig. 15. Normalized repair rate of various BIRAs ($M = 1024$, $N = 1024$, $C_S = 5$, and $R_S = 5$).

ability of the BIRA to find a correct repair solution. Achievement of an optimal repair rate is very important especially in mass production of commodity DRAMs. CRESTA and IntelligentSolveFirst can achieve 100% normalized repair rates [1], [6]. The normalized repair rate of the proposed BIRA is simulated with IntelligentSolveFirst, CRESTA, RM, and ESP. The RA of LRM is based on RM and the maximum repair rate of LRM is equal to that of RM when the size of bitmap for LRM is determined as (1). Fig. 15 shows the simulated results of normalized repair rate by *RepairSim*. Although the number of spares and memory block sizes varied, the trend of the simulated results is nearly identical. Therefore, the rest of the simulations were executed for a 1024×1024 memory block with five spare rows and five spare columns. For each *x*-axis value (i.e., random fault) from 1 to 18 of Fig. 15, *RepairSim* randomly generates 5000 different sets of faulty addresses and the fault insertion order of each set of faulty addresses is randomly mixed up ten times. Therefore, a total of 50 000 different sets of faults are simulated for each *x*-axis label. According to the result in Fig. 15, IntelligentSolveFirst, the proposed BIRA, and CRESTA can all achieve optimal repair rates (i.e., 100% normalized repair rate). The repair rate of RM is almost 100% for simple cases in which

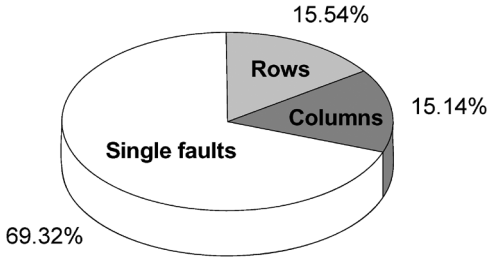

Fig. 16. Distribution of fault types ($M = 1024$, $N = 1024$, $R_S = 5$, and $C_S = 5$).

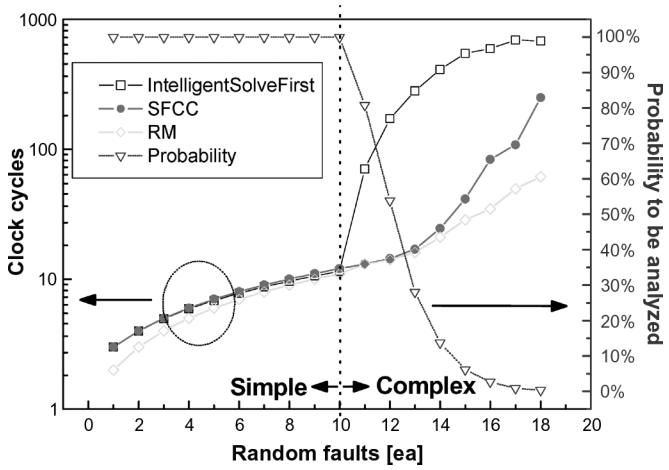

Fig. 17. Comparison of clock cycles and probability to be analyzed ($M = 1024$, $N = 1024$, $R_S = 5$, and $C_S = 5$).

the number of faults is smaller than or equal to the sum of spare rows and columns (i.e., $\text{random faults} \leq 10$) but decreases for complex cases (i.e., $\text{random faults} > 10$). The repair rate of ESP decreases severely for both simple and complex cases.

RepairSim also gives the statistical fault distribution report. Fig. 16 shows the distribution of the fault types of the random faults for only the repairable cases. According to the statistical report in Fig. 16, a random fault set generated by *RepairSim* consists of 69.32% of single faults, 15.54% of faulty rows, and 15.14% of faulty columns.

The third feature of BIRA is RA speed. To estimate analysis speed, *RepairSim* is used to model and estimate the RA execution clock cycles of the BIRA hardware. Fig. 17 shows the comparison of clock cycles between IntelligentSolveFirst, SFCC, and RM based on synthesis results by *RepairSim*. All cases where there are one to ten random faults are 100% repairable in Fig. 17 (see the right side of the *Y*-axis) because there always exist repair solutions for simple cases when the number of the total faults is less than or equal to the sum of available spare rows and spare columns [i.e., $TFC \leq (R_S + C_S)$]. But for complex cases [i.e., $\text{random faults} > 10$ ($R_S = 5$, $C_S = 5$)], the analysis speed depends on the analysis ability of BIRA. From the result of Fig. 17 (see on the left side of the *Y*-axis), SFCC analyzes slightly slower than IntelligentSolveFirst for simple cases but faster for complex cases. The average clock cycles in the RA of all analyzed cases with 1–18 random faults (593 540 cases of the total 900 000 cases) by *RepairSim* for IntelligentSolveFirst, SFCC, and RM are 196 cycles, 35 cycles, and 17 cycles, respectively. The average analysis speed of SFCC is 5.6 times faster than that of IntelligentSolveFirst for all cases requiring analysis. Fig. 18 shows simulated case statistics by *RepairSim*. Early fail of Fig. 18 shows early terminated cases that violate Condition 1 or Condition 2 during fault collection; these cases are not required to be analyzed. Condition 3 of Fig. 18 shows simulated cases that meet Condition 3. The normal flow of Fig. 18 shows that some cases do not meet Condition 3 and require complicated analysis. According to the results of Fig. 18, SFCC does not need to execute time-consuming analysis for 98.54% of all cases that require analysis. Therefore, Condition 3 greatly reduces the analysis time of the proposed BIRA. A faster analysis time can reduce the time overhead of a BIRA.

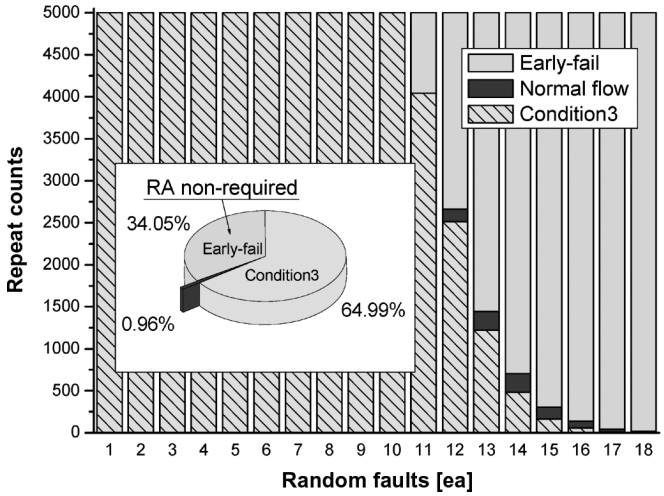

Fig. 18. Simulated case statistics ($M = 1024$, $N = 1024$, $R_S = 5$, and $C_S = 5$).

For a 256-Mb DRAM (4 bank $\times$ 4 Mb $\times$ 16 I/Os), the test time for executing a 6N March test is $2\,264\,924\,160$ ns (4 Meg cells $\times$ 6 March elements $\times$ 9 instructions for a read/write operation) assuming that the DRAM has 256 blocks and $M = 1024$, $N = 1024$, $R_S = 5$, and $C_S = 5$ at the rate of 100 MHz and all banks can be tested in parallel. The average clock cycles required for a block of IntelligentSolveFirst, SFCC, and RM are 681.1, 250.4, and 61.54, respectively. In this case, the average test time overhead of RA of these three BIRAs are $1\,743\,616$ ns (0.077%), $641\,024$ ns (0.028%), and $157\,542$ ns (0.007%), respectively, assuming the worst faulty conditions is 18 faults for all blocks. Time overheads of RA for these three algorithms are all negligible. However, the overhead of analysis time can be increased with growing the number of blocks and the number of spares but can be decreased by increasing the memory space.

In addition to these three main features of a BIRA, rate of overused spares (ROS) is also another important feature of BIRA. An ROS of 0% means a BIRA can find an optimal repair solution. Fig. 19 shows a comparison of ROS values for the different BIRAs. ROS is defined as follows. The number of optimal spares in the solution can be obtained by picking up a minimum number of spares in the repair solutions of CRESTA

$$\text{ROS} = \frac{\#\text{ of spares in solution}}{\#\text{ of optimal spares in solution}} - 1.$$

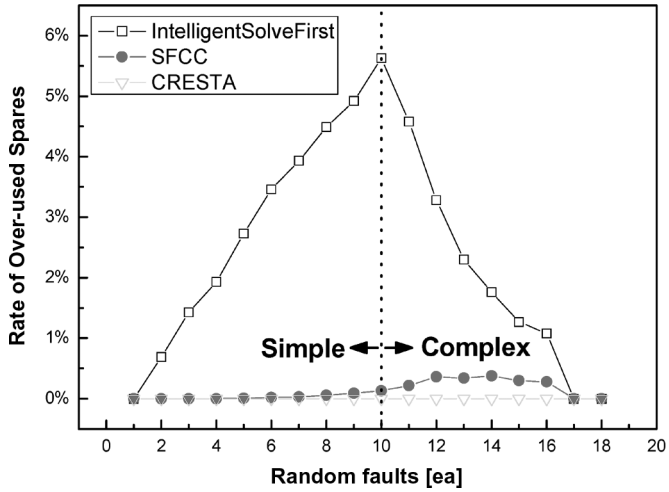

Fig. 19. Comparison of ROS ($M = 1024$, $N = 1024$, $R_S = 5$, and $C_S = 5$).

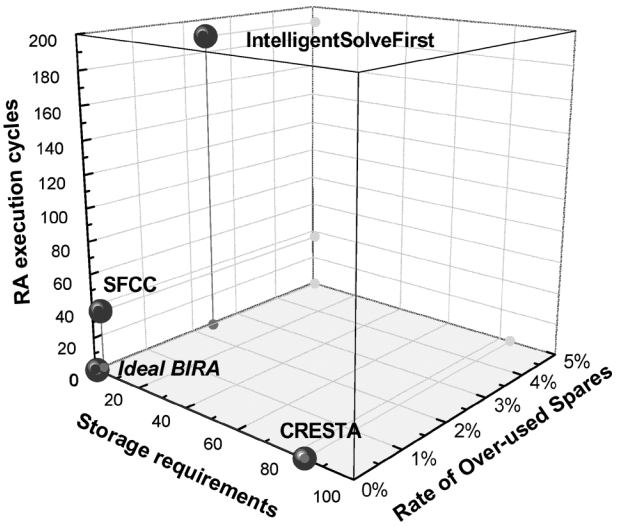

Fig. 20. Comparison of overall performance of BIRAs ($M = 1024$, $N = 1024$, $R_S = 5$, and $C_S = 5$).

The lower the ROS of a BIRA, the higher the probability of success in postpackaging repair. A nonzero ROS for a BIRA indicates that the BIRA takes an unnecessary time to repair overused spares during the physical laser-fusing process. Compared with CRESTA, IntelligentSolveFirst wastes extra spares by 2.42% average (maximum 5.6%) for the whole simulated regions. However, SFCC achieves nearly optimal repair solutions with an ROS of only 0.12% and this is acceptable for commodity memories as well as embedded memories for SOCs.

Fig. 20 shows a comparison of overall performances of BIRA such as RA execution time, storage requirements (area overhead), and ROS for IntelligentSolveFirst, SFCC, CRESTA, and ideal BIRA. The repair rates of these algorithms are all optimal; thus, the repair rate is not plotted in this figure. For ideal BIRA, it is assumed that the area overhead (storage requirements), CPU time, and ROS are all zero and its repair rate is also optimal. According to Fig. 20, CRESTA has too much area overhead and IntelligentSolveFirst does not have a fast-enough RA execution time for a memory with a large number of spares. However, SFCC has reasonable storage requirements and a fast RA execution time as well as an extremely small ROS values. Therefore, the proposed BIRA performs better than IntelligentSolveFirst and CRESTA.

VI. Conclusion

A novel BIRA approach, which builds a line-based search tree with CAM structures and analysis algorithm to support it, is proposed in this paper. Previous BIRA approaches have been pursued to achieve minimal hardware overhead and/or optimal repair rate. The proposed BIRA approach greatly reduces the area overhead with an average 33% smaller storage requirement than IntelligentSolveFirst with a guaranteed optimal repair rate. The average RA speed of the proposed BIRA (SFCC) is on average 5.6 times faster than that of IntelligentSolveFirst. The proposed BIRA approach is a viable solution for commodity memories as well as embedded memories for SOC that require optimal repair rates because it has superior area overhead, repair rate, and analysis speed compared to other state-of-the-art BIRA approaches.

Acknowledgment

The authors would like to thank all members of Computer Systems and the Reliable SOC Laboratory from Yonsei University, Seoul, Korea, for their insightful feedback and comments on drafts of this paper.

References

[1] C.-T. Huang, C.-F. Wu, J.-F. Li, and C.-W. Wu, "Built-in redundancy analysis for memory yield improvement," *IEEE Trans. Reliab.*, vol. 52, no. 4, pp. 386–399, Dec. 2003.

[2] S.-K. Lu, Y.-C. Tsai, C.-H. Hsu, K.-H. Wang, and C.-W. Wu, "Efficient built-in redundancy analysis for embedded memories with 2-D redundancy," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 4, no. 1, pp. 34–42, Jan. 2006.

[3] S. Bahl, "A sharable built-in self-repair for semiconductor memories with 2-D redundancy scheme," in *Proc. 22nd IEEE Int. Symp. Defect Fault-Tolerance VLSI Syst. (DFT)*, Sep. 2007, pp. 331–339.

[4] T.-W. Tseng, J.-F. Li, A. Pao, K. Chiu, and E. Chen, "A reconfigurable built-in self-repair scheme for multiple repairable RAMs in SOCs," in *Proc. Int. Test Conf. (ITC)*, Oct. 2006, pp. 1–9.

[5] Y.-J. Huang, D.-M. Chang, and J.-F. Li, "A built-in redundancy-analysis scheme for self-repairable RAMs with two-level redundancy," in *Proc. 21st IEEE Int. Symp. Defect Fault-Tolerance VLSI Syst. (DFT)*, Oct. 2006, pp. 362–370.

[6] P. Öhler, S. Hellebrand, and H.-J. Wunderlich, "An integrated built-in test and repair approach for memories with 2D redundancy," in *Proc. Eur. Test Symp. (ETS)*, May 2007, pp. 91–96.

[7] H.-Y. Lin, F.-M. Yeh, and S. Y. Kuo, "An efficient algorithm for spare allocation problems," *IEEE Trans. Reliab.*, vol. 55, no. 2, pp. 369–378, Jun. 2006.

[8] S.-K. Lu, Y.-C. Tsai, and S.-C. Huang, "A BIRA algorithm for embedded memories with 2D redundancy," in *Proc. IEEE Int. Workshop Memory Technol., Des., Test. (MTDT)*, Aug. 2005, pp. 121–126.

[9] S.-K. Lu, C.-L. Yang, and H.-W. Lin, "Efficient BISR techniques for word-oriented embedded memories with hierarchical redundancy," in *Proc. IEEE/ACIS Int. Workshop Compon.-Based Softw. Eng., Softw. Arch. Reuse (ICIS-COMSAR)*, Jul. 2006, pp. 355–360.

[10] R.-F. Huang, C.-H. Chen, and C.-W. Wu, "Economic aspect of memory built-in self repair," *IEEE Des. Test Comput.*, vol. 24, no. 2, pp. 164–172, Mar. 2007.

[11] J. R. Day, "A fault-driven comprehensive redundancy algorithm," *IEEE Des. Test Comput.*, vol. 2, no. 3, pp. 35–44, Jun. 1985.

[12] M. Tarr, D. Boudreau, and R. Murphy, "Defect analysis system speeds test and repair of redundant memories," *Electronics*, vol. 57, pp. 175–179, Jan. 1984.

[13] T. Kawagoe, J. Ohtani, M. Niiro, T. Ooishi, M. Hamada, and H. Hidaka, "A built-in self repair analyzer (CRESTA) for embedded DRAMs," in *Proc. Int. Test Conf.*, Oct. 2000, pp. 567–574.

[14] T.-W. Tseng, J.-F. Li, and D.-M. Chang, "A built-in redundancy-analysis scheme for RAMs with 2D redundancy using 1D local bitmap," in *Proc. Des., Autom. Test Eur. (DATE)*, Munich, Germany, Mar. 2006, pp. 53–58.
[15] S.-Y. Kuo and W. K. Fuchs, "Efficient spare allocation in reconfigurable arrays," in *Proc. 23rd ACM/IEEE Des. Autom. Conf. (DAC)*, Las Vegas, NV, Jun. 1986, pp. 385–390.
[16] W. K. Huang, Y. N. Shen, and F. Lombardi, "New approaches for the repairs of memories with redundancy by row/column deletion for yield enhancement," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 9, no. 3, pp. 323–328, Mar. 1990.
[17] D. M. Blough, "Performance evaluation of a reconfiguration algorithm for memory arrays containing clustered faults," *IEEE Trans. Reliab.*, vol. 45, no. 2, pp. 274–284, Jun. 1996.
[18] R. L. Hadas and C. L. Liu, "Fast search algorithms for reconfiguration problems," in *Proc. Int. Workshop Defect Fault Tolerance VLSI Syst.*, Nov. 1991, pp. 260–273.
[19] O. Wada, "Post-packaging auto repair techniques for fast row cycle embedded DRAM," in *Proc. Int. Test Conf. (ITC)*, Charlotte, NC, Oct. 2004, pp. 1016–1023.
[20] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE J. Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, Mar. 2006.

Woosik Jeong received the B.S. degree in control and instrumentation engineering and the M.S. degree in electronics engineering from Korea University, Seoul, Korea, in 1997 and 1999, respectively. He is currently working toward the Ph.D. degree at Yonsei University, Seoul, Korea.

His current research interests include memory testing, built-in self-test (BIST), built-in self-repair (BISR), reliability, and very large scale integration (VLSI) design. Since 1999, he has been a Product Engineer with Hynix Semiconductor, Inc., Icheon-si, Korea.

Ilkwon Kang received the B.S. and M.S. degrees in electrical and electronic engineering from Yonsei University, Seoul, Korea, in 2006 and 2008, respectively.

In January 2008, he joined Hynix Semiconductor, Inc., Icheon-si, Korea, as a Member of Technical Staff in the Dynamic RAM (DRAM) Development Division. His current research interests include memory testing, fault simulation, built-in self-test (BIST), built-in self-repair (BISR), and redundancy analysis (RA) algorithm.

Kyowon Jin received the B.S. degree in physics from Seoul National University, Seoul, Korea, in 1985.

He was involved in device engineering of bipolar technology from 1985 to 1990 and memory design from 1991 to 1998 with Gold-Star Electron, Inc., which became LG Semiconductor, Inc., in 1997. Since 1999, he has been researching and managing dynamic RAM (DRAM)/flash design and test with Hyundai Semiconductor, Inc., which became Hynix Semiconductor, Inc., Icheon-si, Korea, in 2003. His current research interests include low-cost test solution, screen ability improvement, test data analysis for mass production, discrete Fourier transform (DFT)/DFM technology, and multibit memory development technologies in DRAM/flash and next-generation memory.

Sungho Kang (S'87–M'87) received the B.S. degree from Seoul National University, Seoul, Korea, and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Texas at Austin, Austin, in 1992.

He was a Research Scientist with the Schlumberger Laboratory for Computer Science, Schlumberger, Inc., and a Senior Staff Engineer with the Semiconductor Systems Design Technology, Motorola, Inc. Since 1994, he has been a Professor with the Department of Electrical and Electronic Engineering, Yonsei University, Seoul. His current research interests include very large scale integration (VLSI) design and testing, design for testability, built-in self-test (BIST), defect diagnosis, and design for manufacturability.