Analyzing Test and Repair Times for 2D Integrated Memory Built-in Test and Repair

Philipp Öhler[1], Sybille Hellebrand[1], and Hans-Joachim Wunderlich[2]
[1]University of Paderborn Germany
[2]University of Stuttgart Germany

***Abstract*—An efficient on-chip infrastructure for memory test and repair is crucial to enhance yield and availability of SoCs. A commonly used repair strategy is to equip memories with spare rows and columns (2D redundancy). To avoid the prohibitive storage requirements for failure bitmaps and the complex data structures inherent in most algorithms for offline repair analysis, existing heuristics for built-in repair analysis (BIRA) either use very simple search strategies or restrict the search to smaller local bitmaps. Exact BIRA algorithms work with sub analyzers for each possible repair combination. While a parallel implementation suffers from a high hardware overhead, a serial implementation leads to increased test times. Recently an integrated built-in test and repair approach has been proposed which interleaves test and repair analysis and supports an exact solution with moderate hardware overhead and reasonable test times. The search is based on a depth first traversal of a binary tree, which can be efficiently implemented using a stack of limited size. This algorithm can be realized with different repair strategies guiding the selection of spare rows or columns in each step. In this paper the impact of four different repair strategies on the test and repair time is analyzed.**

I. Introduction

State of the art systems-on-a-chip (SoCs) typically devote a large percentage of the chip area to various kinds of memory cores. According to the International Roadmap for Semiconductors (ITRS) the percentage of memory in SoCs will continue to increase rapidly [6]. At the same time the shrinking feature sizes will lead to increasing parameter variabilities and a high susceptibility to defects. As memories are traditionally designed with more aggressive design rules than logic cores, they play a crucial role for the yield and reliability of a SoC. Embedding the necessary infrastructure for a built-in test and repair is essential to achieve acceptable yields and to guarantee a satisfactory availability in the field [16, 17].

Memory repair relies on spare elements at different levels of the design hierarchy. The most common form of redundancy, however, is 2D redundancy where both spare rows and spare columns are added to the memory [1, 4, 5-9, 11-15]. Usually the repair process for 2D redundancy consists of several steps. First the memory is tested, and the information about faulty elements is collected in a *failure bitmap*. Then *repair analysis* attempts to find an allocation of spare elements, such that all faults are covered at minimum cost. As a result either the memory is identified as not repairable or a *repair signature* is obtained which is the basis for soft or hard repair.

Strategies for 2D repair analysis have been investigated for more than two decades. However, the classical approaches have been developed for offline test and repair analysis and cannot be directly applied on chip [2, 4, 8, 13, 15]. Nevertheless, they provide the foundation for *built-in repair analysis* (BIRA). In particular, Kuo and Fuchs have shown that the problem of optimal 2D redundancy allocation is NP-complete, and they have also proposed a systematic branch and bound approach based on a binary search tree [8].

To avoid the storage of large failure bitmaps and complex data structures for organizing the search, most approaches for built-in repair analysis either follow only very simple search strategies or they rely on divide and conquer techniques [1, 5, 11, 14]. Divide and conquer strategies address both the memory itself, as proposed in [1], and the failure bitmap, which is replaced by a local failure bitmap in [5, 14].

CRESTA is a pioneering BIRA approach, which guarantees to find the optimal solution [7]. Similarly as the early work in [8] it is based on a binary search tree, but to reduce the search time and to simplify algorithmic control a separate *sub analyzer* is implemented for each path in the search tree. This way all possible solutions can be analyzed in parallel. However, the hardware cost grows rapidly with the number of redundancies. For a memory with r redundant rows and c redundant columns $b(r + c, r)$ sub analyzers are needed, where $b(n, k)$ denotes the binomial coefficient n over k. Processing of the $b(r + c, r)$ sub analysis tasks serially as mentioned in [12] can reduce the hardware cost, but leads to very high test and analysis times.

Recently an integrated built-in test and repair approach has been proposed which performs repair analysis concurrently with test application [10]. This way an optimal solution can be found without any failure bitmap. The basic algorithm performs a depth first traversal of a binary search tree and is implemented with a stack of size $r + c$. It is combined with a strategy to continuously reduce the problem complexity requiring two content addressable memories (CAMs) with only $2r\cdot c$ entries, each. The depth first traversal can be realized with different repair strategies guiding the selection of spare rows or columns in each step. Depending on the repair strategy the way to find the optimal solution, and particularly the number of backtracks, may vary. Since backtracking in the search tree

This work was supported by DFG grants HE 1686/2-1 and WU 245/4-1

requires a restart of the test, the repair strategy has impact on the overall test and repair time.

This paper presents an experimental study to analyze the impact of the selected repair strategy on the overall test and repair time. Before the experimental setup and the achieved results are described in detail in section III, the integrated built-in test and repair approach presented in [10] is briefly summarized in section II.

II. An Integrated Built-in Test and Repair Approach

As pointed out above, the integrated built-in test and repair approach proposed in [10] interleaves test generation and repair analysis and relies on the depth first traversal of a binary search tree, which is guided by a user-defined repair strategy. Since there is no global failure bitmap, backtracking in the search tree requires a restart of the test, and therefore the number of backtracks is directly correlated to the test and repair time. To show the interdependence between the selected repair strategy and the overall test and repair time, the following description concentrates on the algorithmic aspects. Details on the hardware implementation are explained in [10].

A. Memory Repair as a Binary Search Problem

In their early work Kuo and Fuchs have already shown how to organize the problem of 2D memory repair as binary search [8]. The underlying concepts and principles are explained with the help of the small example memory shown in Fig. 1.

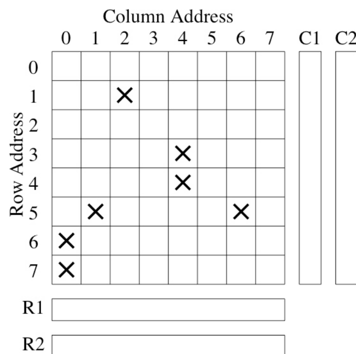

Fig. 1: Example 8×8-bit memory (2x2 spares) with several faulty locations.

The memory of Fig. 1 is equipped with two spare rows and two spare columns, and the binary search tree of Fig. 2 represents all possible repair configurations exploiting these resources. Each edge in the tree corresponds to a repair decision "row repair" (R) or "column repair" (C) for a fault in the memory. A leaf is reached when a successful repair scheme is found or no more repair resources are available. In case of a successful repair, the path from the root to the leaf provides the repair signature. The highlighted path in Fig. 2 leads to a successful repair configuration, which is the only solution for this small example. For this path the edges are also labeled with the address of the first fault covered by the spare.

For a memory with r redundant rows and c redundant columns the maximum height of the tree, i.e. the maximum length of a path from the root to a leaf node, is $r + c$. The leaf nodes in a complete search tree correspond to all solutions using all resources. As there are r rows distributed among $r + c$ spares, there are $b(r + c, r)$ leaf nodes in a tree enumerating all possible repair configurations.

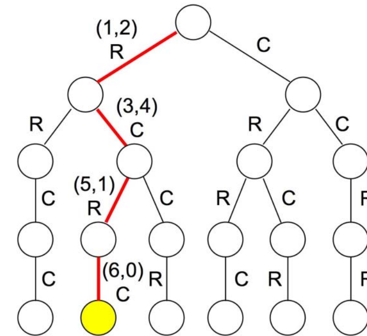

Fig. 2: Complete search tree for the memory of Fig. 1.

To reduce the size of the search tree Kuo and Fuchs propose to perform a "must repair" phase before starting the binary search [8].

Observation 1 ("must repair"): For a memory with r redundant rows and c redundant columns the following repair decisions are mandatory: If there are more than c faults in a row, then there are not enough columns to cover all the faults, and a row must be selected for repair. Similarly, more than r faults in a column require a column repair.

B. Basic Algorithm and Repair Strategies

The basic algorithm presented in [10] also uses a binary search tree, but it interleaves test and repair analysis, i.e. whenever a new fault is detected during test, a (preliminary) repair decision is made and a new node is added to the search tree. If backtracking from node w to v in the search tree is necessary, the preliminary repair decisions between the two nodes must be cancelled and the test must be restarted with the (partial) repair signature corresponding to node v.

To allow a low cost hardware implementation the binary search tree is traversed following a "depth first" strategy, because in this case a stack is sufficient, which doesn't grow larger than the height of the search tree [3]. The core of the resulting on-chip infrastructure for repair analysis is thus a repair stack with at most $r + c$ records.

The complete test and repair process, which is referred to as *basicSolve*, is illustrated for the small example memory of Fig. 1. If the test analyzes the memory row by row, then the faults are detected in the order (1, 2), (3, 4), (4, 4), (5, 1), (5, 6), (6, 0), and (7, 0). To fully determine the search process, a repair strategy must be selected. A very simple solution is to prefer one spare type, e.g. to follow a "row first" strategy. In this case the search proceeds as shown in Fig. 3, where the nodes of the tree are labeled in the order of traversal. Since rows are the preferred spare parts, they are used to repair the first two faults at addresses (1, 2) and (3, 4). Then only spare columns are left for the faults at addresses (4, 4) and (5, 1), which are detected next. When the fault at address (5, 6) is detected, no more resources are available, and the search backtracks to

node number 2 in the search tree. The test is restarted with a row repair for the fault at address (1, 2) as partial repair signature. When the fault at address (3, 4) is detected again, a column repair is selected. Then the test and search process continues with a row repair for the fault at address (5, 1) and a column repair for the fault at address (6, 0). A first solution corresponding to the leaf node number 8 is found.

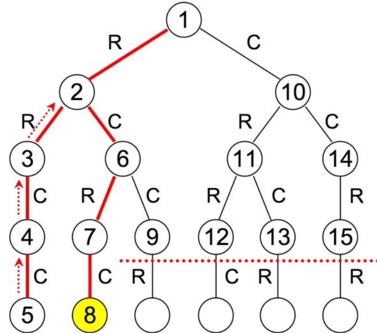

Fig. 3: Depth first traversal using the "row first" repair strategy.

To improve this solution, a repair configuration consisting of only 3 spares must be found, i.e. a path can be terminated as soon as it has reached length 3. In the example, the traversal of the rest of the tree does not reveal any other solutions.

If a balanced use of spare parts is preferred, for example to provide a flexible basis for future repairs, the repair strategy can be changed to select the spare type with the most resources. The resulting search tree is shown in Fig. 4 (in case of a tie the preference is given to rows as above).

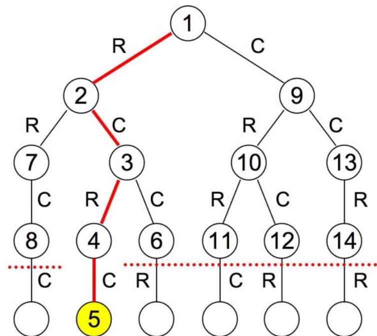

Fig. 4: Depth first traversal using the balanced repair strategy.

With the balanced repair strategy, the first path already leads to the leaf node corresponding to the only solution. A first solution is reached in one pass without any restart of the test. As explained above for the remaining search only paths of length 3 have to be considered. Comparing the two examples shows that the repair strategy can also have an impact on the number of search nodes and the number of restarts needed to reach the first or the optimal solution. Other repair strategies are for example "column first" or random.

C. Continuous Reduction of the Search Space

In the basic algorithm of section II.B, each backtrack in the search tree requires a restart of the test. To guarantee acceptable test and repair times, it is therefore crucial to keep the number of backtracks low. The must repair criterion stated in Observation 1 provides the basis to achieve this goal. However, it is not restricted to a single preprocessing step as in many other BIRA approaches. As each repair changes the number of available spares, new situations fulfilling the must repair criterion may occur after a repair step. In particular, after a must repair step other must repair decisions may be necessary. Therefore, a dynamic must repair analysis is performed as a preprocessing step and after each repair decision.

For an efficient hardware implementation an intelligent fault list manages fault addresses. It consists of a table for row addresses and a table for column addresses, each of which is realized by a small CAM of size $2r{\cdot}c$. During test, each detected fault address is compared against the contents of the two CAMs. If the number of faults with the same row address has already reached c, then, with the new fault, the must repair criterion for a row repair is fulfilled. Similarly, if the number of faults with the same column address is r, a column repair becomes mandatory. After the corresponding repair, a dynamic must repair analysis is carried out for all faults in the fault list with the updated values for r and c.

If neither a row nor a column must repair can be triggered, the row address of the new fault is stored in the row table, and the column address is entered in the column table. The maximum number of fault addresses which can be collected without invoking a must repair is $2r{\cdot}c$ [5]. Therefore it is sufficient to select $2r{\cdot}c$ as the memory size both for the row and for the column table. This observation is also useful for pruning the search tree and for the early identification of non-repairable memories. If both tables are full, and a new fault is detected without leading to a must repair, then the memory is proven to be non-repairable and the search can be stopped. The basic algorithm of Section II.B combined with the proposed strategy is called *intelligentSolve*. Similar as for *basicSolve*, the selection of a repair strategy may also have an additional impact on the number of search nodes and the number of backtracks.

III. Evaluation of Repair Strategies

In [10] it has been shown that the dynamic must repair analysis considerably decreases the test and repair time. However, the evaluation has been restricted to a "row first" repair strategy. The analysis presented in this paper focuses on the additional impact of the repair strategy on the overall test and repair time. Because a restart of the test is invoked by each backtrack, the overall test and repair time can be roughly estimated as $T = b \cdot t$, where b denotes the number of backtracks or restarts, and t denotes the duration of a single test. It should be noted that this estimation is pessimistic, as in most cases a restart will be invoked before the current test is completed.

For the analysis presented in the sequel, both the algorithm *intelligentSolve* and a version of *intelligentSolve* stopping at the first solution (*intelligentSolveFirst*) have been simulated for a 1024×1024-bit memory with 5 spare rows and 5 spare columns. For each algorithm the repair strategies "row first", "column first", "balanced" and "random" have been compared

by repeated experiments with varying numbers and distributions of random defects. The number of random defects has been linearly increased ranging from one to fifteen. A random defect can result in a single faulty cell, a faulty row or column, a “line fault” consisting of several adjacent faulty cells in a row or column, or a cluster fault affecting up to 3×3 cells. The considered distributions of defect types are listed in Table 1.

TABLE 1: DISTRIBUTION OF DEFECT TYPES.

Defects	*Distributions*		
	d_1	d_2	d_3
Row	0.10	0.10	0.10
Column	0.10	0.10	0.10
Line Fault	0.10	0.20	0.40
Cluster	0.05	0.10	0.20
Single Cell	0.65	0.50	0.20

Each experiment has been repeated 1000 times with randomly generated addresses of the faulty locations. Since all the three defect distributions show similar trends, only the results for d_2 are discussed in the following. Table 2 shows the number of restarts and the repair rates for *intelligentSolve*. The minimum value of restarts in each row is printed in bold face.

TABLE 2: RESTARTS FOR INTELLIGENTSOLVE AND DEFECT DISTRIBUTION D2

Defects	Restarts of the Test for Repair Strategy				Repair Rate
	Row First	Column First	Random	Balanced	
1	2.006	**2.001**	2.002	2.022	1.000
2	4.354	4.363	**4.326**	4.367	1.000
3	7.456	7.424	7.363	**7.290**	1.000
4	11.862	11.880	11.781	**11.401**	0.993
5	17.528	17.226	16.652	**16.045**	1.000
6	22.503	21.920	22.122	**21.876**	0.970
7	28.839	**28.378**	28.658	28.606	0.873
8	36.565	**36.298**	37.099	37.199	0.835
9	**38.030**	38.054	38.424	39.111	0.753
10	39.307	**39.272**	39.347	39.309	0.753
11	**29.934**	29.945	29.942	29.946	0.047
12	24.776	24.776	24.776	24.776	0.000
13	16.733	16.733	16.733	16.733	0.000
14	13.319	13.319	13.319	13.319	0.000
15	10.768	10.768	10.768	10.768	0.000

The strategy “random” yields the least number of restarts only in one case (2 random defects). The remaining results can be roughly divided into three classes. In the region where the defects allow a high repair rate over 90 % the balanced repair strategy provides the smallest number of restarts except for one case. When the repair rate goes down, the strategies “row first” and “column first” lead to shorter test times, and finally when no more repair is possible, all four strategies need the same time to prove this. Although there seem to be some trends in favor of specific repair strategies in certain situations, the absolute differences between the strategies are very small. However, each entry in Table 2 only shows the mean value for the results of 1000 random experiments. To get deeper insight into the behavior of the repair algorithm, it is necessary to analyze the distribution of results in more detail. For this purpose, Fig. 5 through Fig. 7 show some histograms of the results in the region where repair gets difficult, i.e. the repair rate is below 90 %.

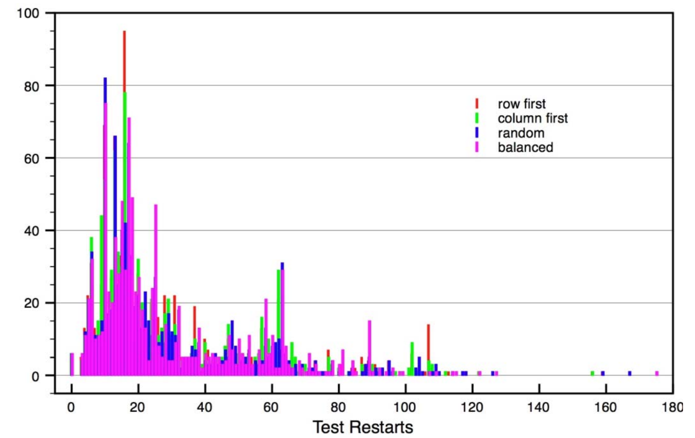

Fig. 5: Distribution of restarts for *intelligentSolve* and 7 random defects.

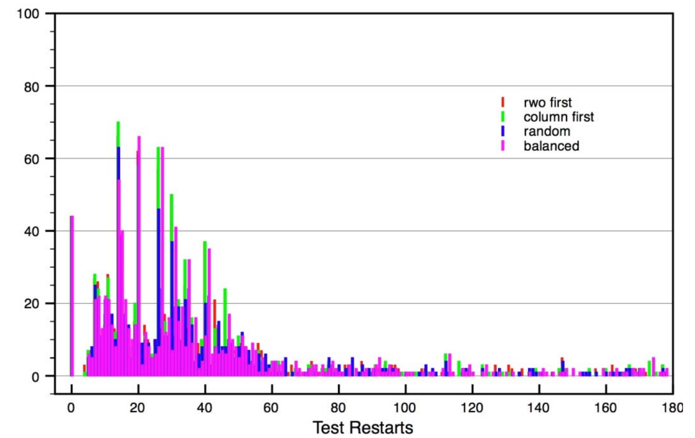

Fig. 6: Distribution of restarts for *intelligentSolve* and 9 random defects.

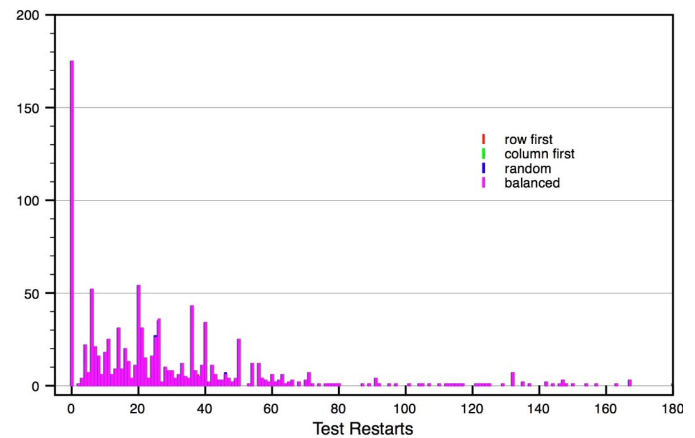

Fig. 7: Distribution of restarts for *intelligentSolve* and 11 random defects.

The histograms show the frequencies of the different results achieved during 1000 experiments. First of all it can be observed that the peaks are concentrated in the left corner of the histograms, which shows that the repair problem can be solved with a moderate number of backtracks in many cases. For example, for 7 random defects in more than half of the experi-

ments a repair solution was found with less than 20 restarts (cf. Table 2). For 11 random defects the repair gets very difficult with a repair rate below 5%. But in this case, the dynamic must repair gets very effective and the number of restarts is reduced. As illustrated in Fig. 7, in about 175 cases the memory is identified to be non-repairable without any restart.

TABLE 2: EXPERIMENTS WITH LESS THAN 20 RESTARTS FOR INTELLIGENTSOLVE (FROM 1000 EXPERIMENTS)

Defects	Number of experiments with less than 20 restarts				Repair Rate
	Row First	Column First	Random	Balanced	
7	559	542	542	540	0.873
9	403	401	395	394	0.753
11	504	504	504	504	0.047

However, regarding the impact of the different repair strategies the histograms don't show specific characteristics, and no significant interdependence between the repair strategy and the test and repair time can be confirmed for *intelligentSolve*.

Nevertheless, this is not in contradiction to the example motivating this analysis in section II.B, where the main advantage of the balanced search over the "row first" strategy was the quicker identification of a first solution. For the "row first" strategy it has already been shown in [10] that in most cases stopping at the first solution is a good alternative with respect to test and repair time and also provides good results with respect to the usage of spares. In the following the algorithm *intelligentSolveFirst* is therefore analyzed in more detail both with respect to test time and the quality of the solutions.

Similar as for *intelligentSolve* the average results cannot prove a significant difference between the different repair solutions. Therefore only the histograms for hard to repair defect constellations are discussed. The histogram for 11 random defects is not shown, because the repair rate is below 5% in this case, and most of the test and repair time is spent to identify the memory as non-repairable. This leads to a similar histogram as the one shown for *intelligentSolve* in Fig. 7. But for 7 and 9 random defects, where still a reasonable repair rate can be expected, some interesting observations can be made. As shown in Table 3, in the majority of the experiments only very few restarts are needed.

TABLE 3: EXPERIMENTS WITH LESS THAN 9 RESTARTS FOR INTELLIGENTSOLVEFIRST (FROM 1000 EXPERIMENTS)

Defects	Number of experiments with less than 9 restarts				Repair Rate
	Row First	Column First	Random	Balanced	
7	920	923	928	924	0.873
9	726	725	773	805	0.753
11	327	334	329	333	0.047

Therefore the x-axis of the histograms in Fig. 8 and Fig. 9 is cut off after 9 restarts to zoom into this region. The histograms show that in most of the cases *intelligentSolveFirst* needs only one restart to find the first solution. Furthermore, the balanced repair strategy has the highest peak for one restart and therefore offers the highest probability for a short test and repair time.

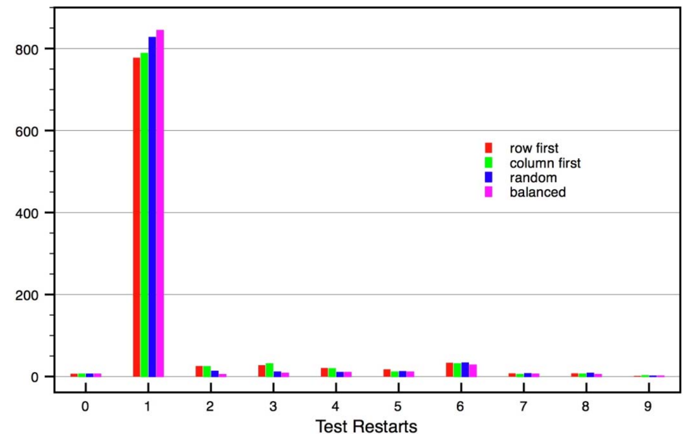

Fig. 8: Distribution of the number of restarts for 7 random defects.

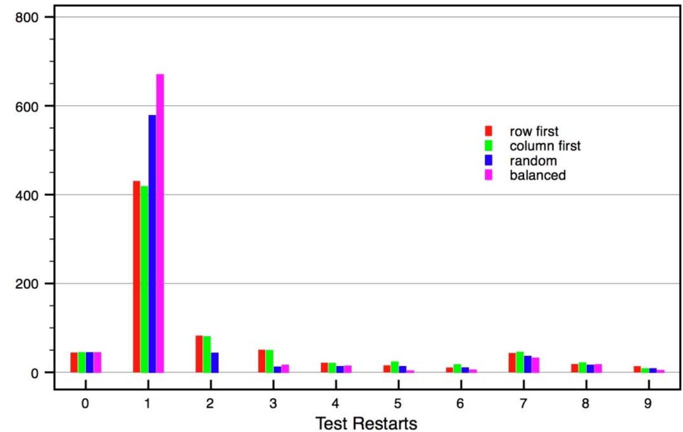

Fig. 9: Distribution of the number of restarts for 9 random defects.

In the experiments described above, also the necessary number of spares has been monitored. As expected, both the random and the balanced repair strategy on the average provide repair configurations with an equal number of rows and columns, while with "row first" and "column first" the preferred types are clearly dominating as long as not all resources are needed. As *intelligentSolve* guarantees to find the optimal solution, the overall number of spares in the optimal repair configuration is independent of the repair strategy. However, *intelligentSolveFirst* stops at the first solution, and the repair strategy may also influence the quality of the solution. Fig. 10 compares the overall number of used spares for *intelligentSolve* and *intelligentSolveFirst* with the 4 different repair strategies. The curves representing the average results normalized to the results of *intelligentSolve* (optimal solution) allow two observations. Firstly, the solutions provided by *intelligentSolveFirst* need at most 20% more spares than the optimal solution. Second the balanced repair strategy leads to the best solutions in all cases. Again the average numbers can show some trends, but a better picture is obtained by analyzing absolute repair qualities in more detail. For more than 8 random

defects both *intelligentSolve* and *intelligentSolveFirst* need 9 or 10 spares in most cases. For 7 and 8 random defects there is already a wider spectrum of results, and Fig. 11 shows the histogram for 7 random defects as an example.

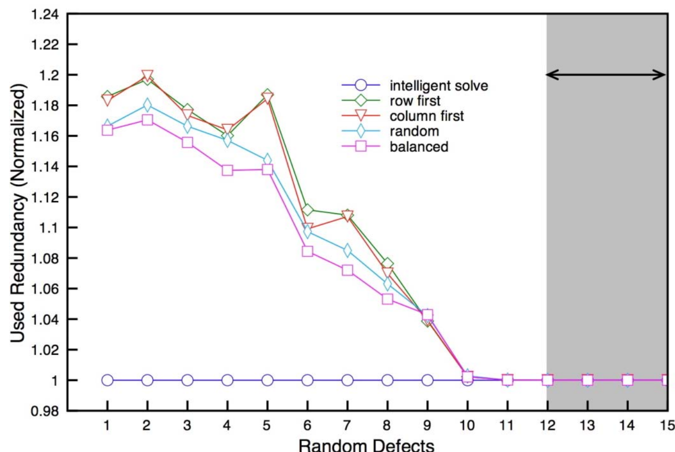

Fig. 10: Used resources for intelligentSolve and 7 random defects.

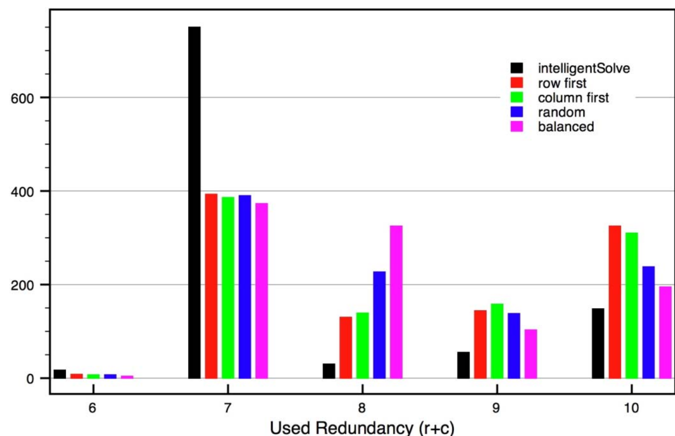

Fig. 11: Used resources for *intelligentSolveFirst* and 7 random defects.

In the majority of the experiments (750/1000) the optimal solution found by *intelligentSolve* requires 7 spare elements. Using *intelligentSolveFirst* with a balanced repair strategy cannot provide exactly the same quality, but in around 700 of 1000 experiments a solution with 7 or 8 spares is possible. In contrast to that, "row first" can find a solution with 7 or 8 spares only in around 520 cases. These results just put a spotlight on the absolute quality of the repair solutions, and a generalization is difficult, but they already show that the differences indicated by the average numbers are not negligible.

IV. Conclusions

The integrated built-in test and repair approach proposed in [10] interleaves test and repair analysis to support an optimal solution for 2D memory repair at low hardware cost. It is based on the depth first traversal of a binary tree, and the test and repair time is mainly determined by the number of backtracks. The traversal is guided by a repair strategy, and different strategies can lead to different repair times. The experimental analysis of the strategies "row first", "column first", "random" and "balanced" in this paper shows that a careful selection of the repair strategy becomes important when the search is stopped at the first solution to reduce the repair time. In particular for defect constellations where the memory is hard to repair but a reasonable repair rate can still be expected, the balanced repair strategy can help to find a solution in very short time. Furthermore, balanced repair more often provides solutions close to the optimal solution. If not all spares are needed, balanced repair also provides a more flexible basis for future repairs in the field.

References

[1] D. K. Bhavsar, "An algorithm for row-column self-repair of RAMs and its implementation in the Alpha 21264." Proc. IEEE Int. Test Conf. (ITC), Atlantic City, NJ, USA, pp. 311-318, September 1999.

[2] J. R. Day, "A fault-driven, comprehensive redundancy allocation algorithm." IEEE Design & Test of Computers, Vol. 2, No. 3, pp. 35-44, June 1985.

[3] E. Horowitz, S. Sahni, S. Rajasekaran, "Computer Algorithms in C++." New York: Computer Science Press, 1998 (2nd printing).

[4] W.-K. Huang, Y.-N. Shen, and F. Lombardi, "New approaches for the repair of memories with redundancy by row/column deletion for yield enhancement." IEEE Trans. on CAD of Integrated Circuits and Systems, Vol. 9, No. 3, pp. 323-328, March 1990.

[5] C.-T. Huang, C.-F. Wu, J.-F. Li, and C.-W. Wu, "Built-in redundancy analysis for memory yield improvement." IEEE Trans. on Reliability, Vol. 52, No. 4, pp. 386–399, December 2003.

[6] International Technology Roadmap for Semiconductors, ITRS 2005 Edition, http://www.itrs.net/Links/2005ITRS/Home2005.htm

[7] T. Kawagoe et al., "A built-in self-repair analyzer (CRESTA) for embedded DRAMs." Proc. IEEE Int. Test Conf. (ITC), Atlantic City, NJ, USA, pp. 567-574, October 2000.

[8] S.-Y. Kuo and W. K. Fuchs, "Efficient spare allocation in reconfigurable arrays." Proc. 23rd ACM/IEEE Design Automation Conf., DAC, Las Vegas, NV, USA, pp. 385–390, June 1986.

[9] J.-F. Li, J.-C. Yeh, R.-F. Huang, and C.-W. Wu, "A built-in self-repair design for RAMs with 2-D redundancy." IEEE Trans. on VLSI Systems, Vol. 13, No. 6, pp. 742–745, June 2005.

[10] P. Oehler, S. Hellebrand, and H.-J. Wunderlich, "An Integrated Built-in Test and Repair Approach for Memories with 2D Redundancy." Proc. Eur. Test Symp. (ETS), Freiburg, Germany, May 2007

[11] M. Ottavi et al., "Simulation of reconfigurable memory core yield." Proc. 14th ACM Great Lakes Symp. on VLSI 2004, Boston, MA, USA, pp. 136–140, April 2004.

[12] S. Shoukourian, V. Vardanian, and Y. Zorian, "An Approach for Evaluation of Redundancy Analysis Algorithms." Proc. IEEE Memory Technology, Design and Testing Workshop (MTDT'01), San Jose, CA, USA, pp. 51-55, August 2001.

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

[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." Proc. Design, Automation and Test in Europe (DATE 2006), Munich, Germany, pp. 53-58, March 2006.

[15] C.-L. Wey and F. Lombardi, "On the repair of redundant RAMs." IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol. 6, No. 2, pp. 222–231, March 1987.

[16] Y. Zorian, „Embedded memory test & repair: infrastructure IP for SoC yield." Proc. IEEE Int. Test Conf. (ITC), Baltimore, MD, USA, pp. 340-349, October 2002.

[17] Y. Zorian, S. Shoukourian, "Embedded-Memory Test and Repair: Infrastructure IP for SoC Yield." IEEE Design & Test, Vol. 20, No. 3, pp. 58-66, May/June 2003.