A Modular Memory BIST for Optimized Memory Repair

Philipp Öhler, Alberto Bosio, Giorgio Di Natale, Sybille Hellebrand

HAL Id: lirmm-00363724
https://hal-lirmm.ccsd.cnrs.fr/lirmm-00363724

Submitted on 24 Feb 2009

A Modular Memory BIST for Optimized Memory Repair

Philipp Öhler[1], Alberto Bosio[2], Giorgio Di Natale[2], and Sybille Hellebrand[1]
[1]*University of Paderborn, Germany*
[2]*LIRMM, University of Montpellier, France*
{oehler, sybille.hellebrand}@upb.de, {alberto.bosio, giorgio.dinatale}@lirmm.fr

Abstract

An efficient on-chip infrastructure for memory test and repair is crucial to enhance yield and availability of SoCs. Most of the existing built-in self-repair solutions reuse IP-Cores for BIST without modifications. However, this prevents an optimized test and repair interaction. In this paper, the concept of modular BIST for memories is introduced, which supports a more efficient interleaving of test and repair and can be achieved with only small modifications in the BIST control.

1. Introduction

Today system-on-chips (SoC) embed several hundreds of different memory cores occupying more than 90% of the SoC chip area. The yield of the entire system is therefore dominated by the memory yield. Memory test and repair capabilities are provided to check the functionality of the memory cores and to increase yield. In the presence of manufacturing defects, redundant elements can replace the failing parts of the memory array [1].

Various built-in self-repair (BISR) schemes have been developed [2, 3, 4]. In most of these schemes test and repair coexist. Meanwhile, implementing a memory BIST is possible by using intellectual property (IP) cores or by relying on automated generation flows of CAD tools. Memory repair requires the exact failure information in the cell array, and most repair schemes are flexible in the sense that any march-like test can be used for failure retrieval. Thus the memory BIST cores can be integrated into a test and repair infrastructure without any modification.

The integrated test and repair approach presented in [4] supports an optimal built-in self-repair for memories with redundant rows and columns ("2D redundancy"). The scheme interleaves test and repair analysis to avoid large failure bitmaps. Moreover, it

follows a depth first strategy for traversing the binary tree for spare allocation. This supports a hardware implementation scaling well with the number of spares. However, backtracking in the search tree requires a restart of the complete test. A more detailed analysis shows that repeating the complete the test may lead to an unnecessarily high increase in test time. To overcome this problem, the concept of a modular march test is introduced in this paper.

2. The Modular Test Strategy

The interleaved test and repair scheme in [4] makes repair decisions as soon as faults are detected during test. The nodes in the search tree for the best repair configuration correspond to detected faults while the edges represent repair decisions (row or column). Backtracking is necessary when a chosen path cannot provide a solution or to prove that an already found solution is optimal. In Figure 1 an example is given.
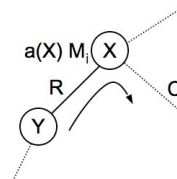
Figure 1: Partial search tree during repair analysis

The information attached to node x shows that a fault at address $a(x)$ has been detected, and the edge (x, y) indicates a row repair. If backtracking to node x occurs, then the test is restarted to determine the remaining faults for the alternative repair decision using a column. This avoids the need for large failure bitmaps, but a complete restart after each backtrack implies a high time penalty.

If a march test with march elements $M_0, ..., M_n$ is used, then the information during which march element a fault has been detected can help to reduce the overall test and repair time. In Figure 1, the fault at node x has been detected during march element M_i,

IEEE computer society

and this suggests to restart the test after the detection of the fault at address $a(x)$.

To exploit this idea properly, it must be taken into account that fault detection in march tests usually requires initialization and sensitization in earlier march elements [5]. As the initialization and sensitization conditions may have been overwritten during the previous search, the respective march elements have to be repeated, too. If initialization starts with march element M_k, $k < i$, then it is even sufficient to work with a "lightweight" version M_k* of M_k, which only contains the memory operations for initialization. M_k* can easily be compiled from the fault detection profiles of the algorithms and the considered fault models. The hardware implementation requires only small modifications of the test and repair control.

3. Experiments and Results

The same experimental set-up as in [4] has been used to analyze two different fault sets: set(a), which contains stuck-at faults, and set(b), which consists of stuck-at faults, transition faults, and coupling faults. Accordingly, two different march tests have been applied: the March X algorithm for set(a) and the March C- algorithm for set(b) [5]. The results are shown in Table 1.

Column one lists the number of randomly injected faults. The next two columns detail the results for March X and March C-. For each algorithm the average number of backtracks in the search tree (test restarts) is given in the first sub-column. The overall number of memory operations exercised during test and repair is shown in the second sub-column for the original proposal in [4]. The third sub-column shows the number of memory operations needed for the modular test strategy proposed in Section 2. Finally, sub-column four shows the ratio between both strate-

gies. Please note that the entries in sub-columns 1 to 3 are average numbers from 1000 experiments as described above.

The results are interesting in two ways. Firstly, for a small number of defects when only a few restarts are needed, the modular strategy provides only a small improvement. For a larger number of defects, when the search gets more complex, the number of exercised memory operations is reduced to approximately 70 % of the original solution in [4]. Secondly, when the test algorithm gets more complex, then the achievement is even better in most cases.

4. Conclusions

In this paper, modular memory testing has been introduced as a strategy for an improved test and repair interaction. Combined with the integrated test and repair approach described in [4], this strategy helps to considerably reduce the time penalty caused by backtracks in the search tree for spare allocation.

5. References

[1] Y. Zorian, "Embedded memory test and repair: infrastructure IP for SOC yield," Proc. IEEE Int. Test Conf. (ITC'02), 2002, pp. 340–349.

[2] C.-T. Huang, et al., "Built-in redundancy analysis for memory yield improvement," IEEE Trans. on Reliability, Vol. 52, pp. 386–399, Dec. 2003.

[3] T. Kawagoe, et al., "A built-in self-repair analyzer (CRESTA) for embedded DRAMs," Proc. IEEE Int. Test Conf. (ITC'00), 2000, pp. 567–574.

[4] P. Öhler, et al., "An Integrated Built-in Test and Repair Approach for Memories with 2D Redundancy," Proc. 12th European Test Symp. (ETS'07), 2007, pp. 91-96.

[5] A. J. van de Goor, "Testing Semiconductor Memories: Theory and Practice," ComTex Publishing, 1998.

Table 1: Experimental Results for Optimal Memory Repair

Defects	March X - set(a)				March C- set(b)			
	Test restarts	#OP original	#OP modular	Modular/ Original	Test restarts	# OP original	# OP modular	Modular/ Original
1	1.185	6.650	6.442	0.969	1.185	10.446	10.446	1.000
2	3.601	16.370	13.842	0.846	3.601	21.458	21.320	0.994
3	9.627	40.740	30.358	0.745	9.627	44.238	40.114	0.907
4	16.364	67.518	49.636	0.735	16.364	87.788	76.260	0.869
5	18.065	74.340	48.068	0.647	18.065	134.012	84.920	0.634
6	32.307	126.796	84.114	0.663	32.307	210.000	142.136	0.677
7	49.278	187.816	129.500	0.690	49.278	287.560	200.927	0.699
8	62.830	235.708	157.808	0.670	62.830	372.054	264.211	0.710
9	76.926	273.414	195.170	0.714	76.926	485.338	341.234	0.703
10	75.073	292.866	176.178	0.602	75.073	630.338	344.206	0.546
11	68.164	242.246	178.316	0.736	68.164	478.016	345.096	0.722
12	57.454	212.256	159.666	0.752	57.454	390.848	304.760	0.780