A novel software-based defect-tolerance approach for application-specific embedded systems

Da Cheng
Electrical Engineering Department
dacheng@usc.edu

Sandeep Gupta
Electrical Engineering Department
sandeep@usc.edu

***Abstract*—Nano-scale fabrication processes have high defect densities and variations, which lead to low yield. Traditional solutions for improving yield are based on the use of hardware redundancy, especially in memories. However, the maximum benefits of hardware redundancy in terms of yield are limited for high defect densities, because of increasing layout complexity and diminishing yield improvements beyond some level of redundancy. Previous research on memory defect tolerance aimed at repairing all defects in memories. However, this is not always necessary in application-specific systems, where programs are known a priori. In this paper we propose a software-based defect-tolerance (SBDT) approach in coordination with hardware redundancy to make use of defective memory chips with un-repaired defects for application-specific systems, and demonstrated their effectiveness using some DSP applications.**

1. Introduction

Reduction in CMOS feature sizes and increase in chip area continue to put a downward pressure on yields due to increasing levels of imperfection in fabrication processes [2]. While state-of-the-art fabrication techniques help somewhat mitigate this trend [1], overall yields are decreasing. Especially manufacturing defect free memory arrays is extremely difficult for today's memory size [10]. Memory repair is important for yield improvement not only for stand-alone memory chips with multi-Giga byte capacities, but also for memories embedded in SoC chips, since such memories occupy more than half of the area of today's typical SoCs and are expected to occupy 94% in 2014, according to ITRS projections [9]. Redundant elements and reconfiguration circuitry are hence incorporated so that memories can work despite having certain combinations of defects [2].

Typical schemes for enhancing yields for memory modules either add redundant rows and/or columns or add redundant memory arrays. For a memory array with T global redundant rows (columns), each decoded row (column) address has a dedicated $1 - to - T$ de-multiplexer cell, whose height must be less than or equal to the height of a memory cell in order to have a compact layout. However, as increasingly larger memory modules, fabricated using processes with increasingly high defect rates, require levels of redundancy beyond some limit, the height of de-multiplexers becomes greater than the height of a memory cell (one bit), which results in a wastage of circuit area and hence in a dramatic drop in layout density. Though the problem of de-multiplexers being un-scalable can be relieved by implementing local redundancy (e.g., where a set of spare rows are used to replace defective rows in a part of a memory array), such approaches require more spare rows and columns for achieving a comparable yield. In both cases, maximum benefits obtained by hardware redundancy (HR) are limited. Youhei [6] demonstrated that the yield of a 1Mb DRAM block saturates at about 0.6 due to extra routing and elements, such as decoders.

The increase in the sizes of memory modules in the last several years made it necessary to partition the memory array into several sub-arrays. One of the previous DRAM modules for IBM 110nm ASIC offering is composed of 16 1Mb DRAM sub-arrays where each sub-array has 8 redundant rows and 8 redundant columns [5]. In 2011, a 45nm embedded DRAM macro design was used in $POWER^{TM}$ processor as a 32 MB on-chip L3 Cache. This DRAM is composed of 292 Kb arrays (264 word-lines and 1200 bit-lines), each with 8 redundant rows and 4 redundant columns [8]. In such designs, the yield is limited by the yield of inter-array interconnects and the inefficiency of local spares.

In summary, hardware-based schemes to improve RAM yields cannot provide high yield and compact layout for increasingly larger RAMs fabricated in processes with increasing levels of imperfections.

In this paper, we propose a software/firmware based yield-enhancement technique to complement hardware redundancy techniques.

2. Problem Statement

This paper focuses on SoCs and other application-specific systems built around processors, such as DSP and graphics processors, that share the following characteristics.

- A large part of the chip is devoted to memories.
- The system is designed to implement a specific application.
- The system architecture is built around stored-program processors.
- Its processors do not use virtual addressing.

Traditionally defective memories are discarded if any memory module is defective and cannot be completely repaired by using available HR after post-fabrication testing. However, we may be able to guarantee completely correct operation of user programs while using chips with one or more unrepairable memory modules if we use software level methods that guarantee that two conditions are met: (1) defects only affect a few memory cells rather than cause malfunction for the entire

memory module, e.g., a failure that occurs in address decoder or some other peripheral circuit; and (2) either we do not use any part of the memory affected by the un-repaired defect, or we do use the affected part, but only in a manner that does not excite the un-repaired defect to cause errors. As we are developing such an approach, we are interested not only in the number of defects but also their locations in the memory. We are also interested in the details of the programs to be executed on the application-specific system.

One of the most important benefits of this approach is that it can be applied to existing designs, since it does not require any modifications in the hardware design or special hardware insertion. Hence, it can be particularly useful in situations where adaption of a new memory design or of a new fabrication process leads to unexpectedly low yield. Furthermore, this approach does not impose any area or performance overheads on the fraction of fabricated chips that do not have any un-repairable defects in their memory modules.

For an embedded system designed for a specific application, we have known programs. At the same time, copies of dedicated embedded chips are manufactured and tested after which (a) defect free copies or copies that can be completely repaired are marked as good, and (b) copies with un-repaired defects are identified with their defect information. We model an application program and a defective memory as described next.

A. Model of a memory with un-repairable defects

In a defective memory with given $N = n$ defects, $X_1, X_2, ..., X_n$ denote the locations of defects in the memory. As shown in Figure 1, a defective memory is viewed as $n+1$ ***partitions*** $P_1, P_2, ..., P_{n+1}$, of sizes $Q_1, Q_2, ..., Q_{n+1}$, respectively. If there are T redundant rows (columns), we will finally have at most n' unrepairable defects, where $n' \leq (n-T)$, since each redundant row (column) can repair at least one defect. As it is possible that multiple defects are located on one row (column), $n' = n - T$ is an upper-bound on the number of un-repaired defects.

Also each time a defect is repaired, the two *partitions* divided by this defect are combined into one *partition*. In some chips, $n' = 0$, i.e., the memory is *fully repaired*. In classical methodology, only chips with fully repaired memory are sold and all other chips are discarded. Our software-based approach (presented in Section 3) focuses on salvaging a fraction of *partially repaired* memory modules, i.e., chips with one or more un-reparable defects, in a manner that *guarantees that given programs will execute in a completely error-free manner.*

B. Model of a compiled user program

Common object file format (COFF) is one of DSP program formats created by assembler and linker [3]. COFF makes modular programming easier, because it encourages the user to think in terms of blocks of code and blocks of data during the development of assembly language programs. These blocks are known as ***sections*** and three scenarios are illustrated below for a program, as shown in Figure 1. (Though we use COFF in our experiments, our approach is applicable to other formats.)

- Scenario 1: A compiled program can be viewed as a single *section* S with A Bits, including machine *code* and *data*.
- Scenario 2: A compiled program is typically comprised of a number of independent procedures and data objects that can be placed independently in the address space. Hence, the program can be viewed as a collection of *code* and *data sections* $S_1, S_2, ..., S_K$ of sizes $A_1, A_2, ..., A_K$ respectively, where each *section* S_i can be placed independently in any *partition* where A_i contiguous memory locations are available.
- Scenario 3: Starting with Scenario 2, we can partition a data or code section into smaller *sections*, potentially with some performance penalty. In such a case, a compiled program can be considered as a collection of even smaller data and code *sections* with user-defined lengths. (Benefits of this will be quantified in Section 5.)

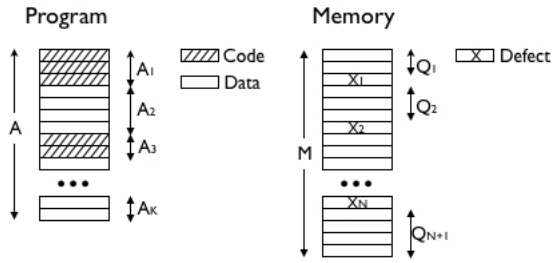


Fig. 1. Model of a Program and a defective Memory

Using the above representation for partially repaired memory and programs, we formulate three problems.

Problem 1: When a program is viewed as a single *section* as described in Scenario 1, a partially repaired memory can be used if the *section* S can be placed in any $partition$ whose size is greater than A. Note that this can occur if and only if there exists at least one *partition* with size greater than A Bits.

Problem 2: As a program is viewed as a collection of multiple code and data *sections* as described in Scenario 2, where each *section* can be placed in the memory independently, a memory can be used if and only if every program *section* is placed in some *partition* without running out of space in any *partition*. Scenario 1 is a special case of Scenario 2, with $K = 1$.

Problem 3: Similar to Problem 2, but here we have more *sections* some of which are obtained by the user partitioning some of the sections in Scenario 2 and making the corresponding changes to the program.

The rest of this paper is organized as follows. In Section 3, the design, fabrication, and integration flow of our first software-based defect tolerance approach and its two variants for utilizing HR are proposed to solve above three problems. In Section 4, a mathematical model based on order statistics is derived to estimate memory yield for a program for Problem 1. Since Problem 2 and Problem 3 are not amenable to derivation of analytical estimates of yield, in Section 5 we develop a procedure for using HR and assigning *sections*

to memory *partitions*. We also use a Monte Carlo (MC) approach to generate a number of defective memory instances to estimate yield for our proposed software-based defect-tolerance approaches for all three problems and demonstrate its benefits for example programs. Finally, we close with our conclusions.

3. Overview of Proposed Metholodogy

To place a program into a defective memory, we propose a novel software-based defect tolerance (SBDT) approach. During the use of a system with a partially-repaired memory module, compiled program files are linked in an optimized manner that avoids using any un-repaired memory cells. The location of every unrepaired memory location is identified for each fabricated copy of the chip during post-fabrication testing of the chip.

If memory modules have any HR available for repair, we use HR in conjunction with SBDT in two ways as shown in Figure 2, namely SBDT with *classical repair* (SBDTcr) and SBDT with *intelligent repair* (SBDTir).

- Classical Repair: Available HR is configured to repair defective memory in such a manner that every defective location identified during post fabrication testing has identical probability of being repaired.
- Intelligent Repair: Since how HR is used to repair different combinations of defects provides different levels of benefits in terms of memory yield[1], we develop the notion of *significance of a defect* to capture how much memory yield benefit is obtained if the defect is repaired. A heuristic procedure identifies defects to be repaired based on estimates of their *significance*, computed using the relative positions of adjacent defects along with information of the compiled programs. In a greedy manner, the most significant defect combinations are then repaired.

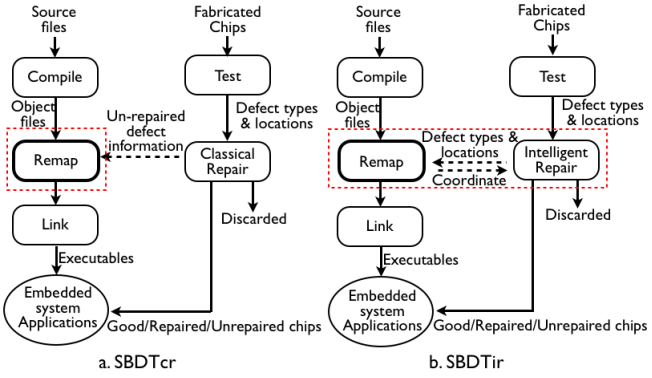


Fig. 2. SBDT approach (a) with classical repair, and (b) with intelligent repair.

After the above step, for both approaches the locations of un-repaired defects are identified and the program *sections* are optimally assigned to memory *partitions* before the compiled files are linked. The objective is to maximize the probability of fitting the program in the non-defective parts of the defective memory. As described in Section 5, a best-fit decreasing algorithm is adopted for mapping.

The proposed SBDT approaches require a direct control over memory accesses instead of managing memory through advanced operating systems. In application-specific embedded systems, one application usually runs a single program or a set of programs. Those programs, and a simple real-time operating system (RTOS), if any, require little memory management, and hence do not use virtual memory management. Furthermore, software approaches are more suitable for application-specific embedded systems because hardware techniques increase circuit complexity, while in these systems circuit simplicity is a basic issue [7]. Hence, this paper focuses on application-specific embedded systems and all experimental results are obtained from five sample programs on TI DSP6713.

4. Analytical Yield Estimates for Problem 1

This section presents a mathematical model to estimate effective memory yield for Problem 1 presented in Section 2.

4.1. Analytical yield estimation

It is easier to compute the probability that we will fail to solve Problem 1, i.e., it is easier to compute the probability of failure to place a single-section program in non-defective parts of a defective memory.

What is the probability that un-repaired defects in memory $[1, M]$ *will prevent the existence of* A *contiguous non-defective memory locations?*

First, defects are random variables uniformly distributed over $[1, M]$, yet without more than one defects at the same memory location. Assume that there are ***N defects*** at locations $X_1, X_2, X_3, \ldots, X_N$ distributed over $[1, M]$, where N is a Poisson random variable. They are sorted in an increasing-address order as $X_{(1)}, X_{(2)}, X_{(3)}, \ldots, X_{(N)}$, where $X_{(i)}$ represents the location of the i_{th} defect. Then the probability for each of these defects to be placed at a specific location $P(X_{(1)} = x_1, X_{(2)} = x_2, X_{(3)} = x_3, \ldots, X_{(N)} = x_N)$ is $\frac{1}{\binom{M}{N}}$, since every combination of defect locations is equally likely. Second, the distance between adjacent defects is less than A; distance between defects and the boundaries of the memory (i.e., locations 1 and M) is also less than A. Attributes of such problems are captured by order statistics analysis.

Event B: Locations of N random defects are sorted in an increasing order by address: $X_{(1)}, X_{(2)}, X_{(3)}, \ldots, X_{(N)}$, where $X_{(i)}$ is the i^{th} smallest-address defect in the sequence, is located over $[1, M]$. Define $X_{(0)} = 1$ and $X_{(N+1)} = M$ as the beginning and the end of the memory. *Event B* meets following conditions: (a) the distance from defect $X_{(i)}$ to defect $X_{(i-1)}$ is less than A; and (b) the distance from $X_{(i)}$ to $X_{(N+1)}$ is less than $(N - i + 1)A$. Constraints for $X_{(i)}$ are shown in Equations (1) and (2), for $i \geq 1$.

$$X_{(i-1)} < X_{(i)} \leq X_{(i-1)} + A, \tag{1}$$

and

$$M - X_{(i)} - (N - i) < (N - i + 1)A. \tag{2}$$

[1]Though the percentage of useful chips, including good chips, repaired chips, and un-repaired chips made usable by our SBDT approaches, is similar to the traditional concept of yield, it is also a function of the given program. For simplicity, we use the term "yield" in this paper.

$$P(B|N) = \sum_{Max(1,M-NA+2)}^{Min(A,M)} \sum_{Max(x_1+1,M-(N-1)A-N+3)}^{Min(x_1+A,M)} \cdots \sum_{Max(x_{N-1}+1,M-A+1)}^{Min(x_{N-1}+A,M)} \frac{1}{\binom{M}{N}}. \quad (3)$$

$$P_{IR}^{T}(B|N) = \sum_{Max(1,M-(N-1)-\lceil \frac{N}{T+1} \rceil A)}^{Min(A,M)} \cdots \sum_{Max(x_T+1,M-(N-T-1)-\lceil \frac{N-T}{T+1} \rceil A)}^{Min(x_1+A,M)} \cdots \sum_{Max(x_{N-1}+1,M-\lceil \frac{1}{T+1} \rceil A)}^{Min(x_{N-T-1}+A,M)} \frac{1}{\binom{M}{N}}. \quad (10)$$

Using the above constrains we can compute the probability of occurrence of *Event B* (i.e., the failure to solve Problem 1), given that the number of defects is N, i.e., we can calculate $P(B|N)$ by computing the conditional probability of $P(B|N)$ on each $X_{(i)}$ using the limits obtained from Equations (1) and (2). Finally we obtain $P(B|N)$ shown in Equation (3), based on Bayes' formula shown in Equation (4) and the knowledge that $P(B|X_{(1)} = x_1, X_{(2)} = x_2, ..., X_{(N)} = x_N, N) = 1$.

$$P(X_{(2)} = x_2|X_{(1)} = x_1, N) = \frac{P(X_{(1)} = x_1, X_{(2)} = x_2, N)}{P(X_{(1)} = x_1, N)}. \quad (4)$$

A. SBDT without repair

According to above deductions given N defects, the probability of a defective memory being useful in terms of SBDT is: $1 - P(B)$. As the number of defects is a Poisson random variable, the expected number of defects λ (Poisson parameter) equals to $M \times D$, where D is the probability of being defective for each cell and M is the memory size. Hence, for a given D, different numbers of defects are considered to compute yield as memory size increases. Compounding the probability for different numbers of defects, we compute yield *without repairs* in Equation (5).

$$Y_{NR} = 1 - \sum_{n=1}^{M} P(B|N = n)P(N = n), \quad (5)$$

where $P(B|N)$ is shown in Equation (3).

B. SBDT with classical repair

If there are T redundant rows (columns), we will finally have at most $N' = (N - T)$ unrepairable defects, where N' is computed assuming that each redundant row (column) can only repair one defect. As it is possible that multiple defects are located on one row (column), this value of N' is pessimistic and its use estimates a lower bound on the benefits of our approach. In this case, it can be observed that $P'(N \text{ defects}) = P(N + T \text{ defects})$, where P and P' are probability density functions before and after classical repair. And yield with T $classical$ $repairs$ can be computed in Equation (6).

$$Y_{CR}^{T} = 1 - \sum_{n=1}^{M} P(B|N = n)P(N = n + T). \quad (6)$$

C. SBDT with intelligent repair

We are interested in the probability of existence of A Bit contiguous defect-free memory locations after defects are intelligently selected and repaired by using available T redundant rows (columns). Define $X_{(0)} = 1$ and $X_{(N+1)} = M$ as beginning and the end of the memory, $X_{(i)}$, the i_{th} smallest defect in the sequence, is located in the memory in such a way that (a) the distance from $X_{(i)}$ to $X_{(i-T-1)}$ is less than A; and (b) the distance from $X_{(i)}$ to $X_{(N+1)}$ is less than $\lceil \frac{(N-i+1)A}{T+1} \rceil$ so as to meet constraints of $Event\ B$. Hence constraints for the i_{th} defect are as below.

$$X_{(i-1)} < X_{(i)} \leq X_{(i-T-1)} + A, \quad (7)$$

$$M - X_{(i)} - (N - i) < \lceil \frac{N - i + 1}{T + 1} \rceil A, \quad (8)$$

when $i \geq T + 1$. Hence, $P_{IR}^{T}(B|N)$, the Probability of *Event B* given *T intelligent repairs* (IR) is derived and shown in Equation (10). The yield can be computed as shown in Equation (11) by using Equation (10) in conjunction with Equation (5).

$$Y_{IR}^{T} = 1 - \sum_{n=1}^{M} P_{IR}^{T}(B|N = n)P(N = n). \quad (11)$$

4.2. Scaling

Complexity of numerical integration of Equation (5), Equation (6), and Equation (11) for memory size M and number of defects N is $O(M^N)$. This run-time complexity is unacceptably large even for a 1Kb program. Hence we have developed a scaling technique for programs and memories in order *to make our mathematical model computationally tractable yet computing provable upper and lower bounds*. In Section 5, we demonstrate that bounds are tight by experiments. In our scaling model, we view the memory of size M as shown in Figure 3, where the scaled memory has M' blocks, where each block represents α contiguous locations in the original memory. A defective block (marked with a circle) indicates that there was at least one defect among the original α locations in the block. The values of other parameters are scaled as:

$$M' = M/\alpha \quad D' = 1 - (1 - D)^{\alpha}$$
$$A' = A/\alpha \quad \lambda' = M'(1 - (1 - D)^{\alpha})$$

Assume that we try to place a scaled program with size A between the two defects. The best case probability of success occurs when all defects are located at *far ends* of the blocks and the worst case will be that all defects are located at *near ends* of the blocks. Hence, if A' is integer, the best case (upper bound) and the worst case (lower bound) are A' and $A' + 1$, respectively. Otherwise, $\lfloor A' \rfloor$ and $\lceil A' \rceil$ can be used to compute the best case and worst case yields. Also the smaller α is, the more accurate the approximation will be. If α equals to 1, there is no scaling and the yield is computed accurately and the upper- and lower-bounds are equal. For example, by setting scaling factor to $10,000$ Bytes/block, program P1 is scaled from $242,847$ Bytes to 24.3 blocks with defect density changed from 0.000001 to 0.076884. Then upper bound and lower bound yields can be computed using A' value of 24 and 25 blocks, respectively.

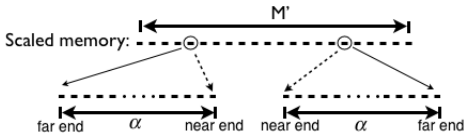


Fig. 3. Scaling

4.3. Scaling approach when HR is used for repair

To extend our scaling approach when some defective locations are repaired by HR, we assume that each block (10,000 Bytes) in the above example consists of N_R rows and N_C columns. A redundant row (column) can repair a defective memory as long as defects are located in the same row (column), in other words, only one row (column) is defective.

According to Section 4.2, for $\alpha = 10,000$ the probability of one block being defective is 0.076884. Assuming $N_R = 200$ and $N_C = 400$, probability that a defective block can be repaired by one redundant row (column) is:

P(only one defective row) = 0.073864,

P(only one defective column) = 0.073856.

Hence a defective block can be completely repaired by one redundant row (column) with a probability 0.96072 (0.96062). Thus, overall yield with T redundant rows (columns) is calculated by compounding i successful repairs (sr) with the worse case probability $P_{sr} = 0.96062$.

$$Y = \sum_{i=0}^{T} \binom{T}{i} P_{sr}^{i} (1 - P_{sr})^{T-i} Y_R^i, \tag{12}$$

where Y_R^i represents Y_{CR}^i or Y_{IR}^i for classical repair or intelligent repair, respectively.

4.4. Yield estimation based on analysis

We have computed both upper and lower bounds of yield for HR (or classical repair), SBDTcr or SBDTir. In Figure 4, we compute upper and lower bounds for all five example programs using above equations and Poisson distribution. As memory size increases, yield increases due to a better chance that a program can be placed in the defective memory. This is one of SBDT's advantages over traditional HR approaches. We also have the following observations.

(1) For Problem 1 with size beyond some limit, memory yield is close to 0.

(2) For a given memory size, yield decreases as the program size increases.

(3) For a given program, yield under the proposed SBDT approach increases as memory size increases.

5. Design and evaluation for Problems 2 and 3

In this section we develop a procedure to solve Problem 2 and Problem 3. Particularly, we develop a characterized bin packing approach for placing *sections* of code and data in memory partitions to implement SBDT approaches. Since Problem 1 is a special case of Problem 2, we also compute this approach with the order statistics based analytical model derived in Section 4.

5.1. Design: Placing program in a memory with un-repaired defects

First consider Problem 2. In this case, we have a given program in the form described in Section 2. We also have fabricated chips, each with a memory with N defects, whose locations $X_1, X_2, ..., X_N$ are provided by the post-fabrication testing algorithm. As described in Section 2, locations of these N defects can be sorted as $X_{(1)}, X_{(2)}, ..., X_{(N)}$ and they divide the memory into $N + 1$ *partitions*. The mapping problem is to place the K program *sections* into the $N + 1$ memory *partitions*. To keep symbols consistent between bin packing problem and actual program placement problem, consider (1) a program as a set of *sections* $S_1, S_2, ..., S_K$, where the *sections* are objects of sizes $A_1, A_2, ..., A_K$, and (2) the defective memory with *partitions* $P_1, P_2, ..., P_{N+1}$ as a sequence of bins of sizes $Q_1, Q_2, ..., Q_{N+1}$. Program-to-memory mapping can be directly formulated as a bin packing problem, which has the following features.

- Bin sizes $Q_1, Q_2, ..., Q_{N+1}$ may be unequal, as defects occur randomly in memory.
- Object sizes $A_1, A_2, ..., A_K$ are not necessarily less than the sizes of all bins. Hence it is possible that the bin-packing procedure will fail.
- Our objective is to pack all objects into available bins, where the number of bins is determined by the number of defects.
- Assignment of objects to bins obeys certain requirements due to relocation rules, described next.

5.1.1 Requirements imposed by relocation rules

Source files are first compiled into object files, which consist of input *sections*. The assembler treats each *section* as starting at address 0. All relocatable symbols and labels are relative to address 0 in their respective locations. Of course, all *sections* cannot actually begin at address 0 in memory, so the linker relocates *sections* by: allocating them into the memory map so that they begin at the appropriate address; adjusting label values to correspond to the new *section* addresses; adjusting references to relocated symbols and labels to reflect the adjusted label values.

If an instruction with a PC-relative field contains a reference to a symbol or label, the relative displacement is expected to fit in the instruction's field. If the displacement does not fit into the field (because the referenced item's location is too far away), the linker issues an error. For example, the linker will issue an error message when an instruction with an 8 bit, unsigned, PC-relative field references a symbol located 256 or more bytes away from the instruction [3]. Hence we must ensure that *sections* are placed in a manner that meets the above linking constraint when implementing bin packing.

5.1.2 Proposed procedure for placing program sections

Basic procedure: We have adopted the best-fit decreasing (BFD) heuristic to solve this bin packing problem. Our procedure (a) prioritizes the placement of the largest objects (sections), since they are typically the most difficult to place; (b) tries to place each object in the *bin* with the smallest unused

space in which it fits, since this typically minimizes wastage of space; and (c) takes into account the special requirements posed by relocation rules. Also note that it is possible for BFD to end with a failure for a particular fabricated copy of the chip. Such a fabricated copy of the chip is discarded and does not count as yield.

Enhancement for intelligent repair: We have developed additional heuristics to identify a combination of defects to repair that increases the probability of finding a successful mapping. First consider the special case where only one spare row (column) is available. For this special case we use the following decision process with linear complexity: If the memory's largest *partition* P_i has two adjacent *partitions* P_{i-1} and P_{i+1}, then the most significant defect is identified as the defect that separates P_i and the larger one of P_{i-1} and P_{i+1}. The available spare is used to repair this defect. From the view of bin packing, this combines the largest bin with the larger of its two neighboring bins and hence maximizes the probability that the largest object can be placed. Given that the success of bin packing is almost always limited by the largest object (demonstrated in Section 4.3), this intelligent-repair strategy typically gives the best results.

Finally, consider the general case where memory has multiple spare rows (columns) or where the user programs have more than one dominant *section* (object for bin packing). In this case, our heuristics for repair are integrated into the BFD approach for bin packing. Each time BFD fails to pack a *section* (object), every defect is checked. Assume that a defect X_i separates two *partitions* P_i and P_{i+1} with $i \in (0, N)$, then the defect X_i is repaired using one redundant row (column) if and only if combining of P_i and P_{i+1} can accommodate this *section* with minimum combined capacity $(Q_i + Q_{i+1})$. We repeat this process until all *sections* (objects) are packed or all T redundant rows (columns) are used.

5.2. Yield estimation for Problem 1

As in Section 3, we consider a compiled program as a single *section* and memory divided by defects into partitions. This is a special case of Problem 2 where only one object needs to be packed in the bin packing problem.

Yield improvement of traditional HR as well as the proposed SBDTir and SBDTcr approaches for five example programs, are explored. In each experiment, we use Monte Carlo (MC) to generate 5000 faulty versions of the memory modules. In the remainder of this section, we evaluate approaches using five executable DSP programs summarized in Table I. We obtained yield for five example programs for Scenario 1 shown in Figure 6, where the empirical results match previously captured upper and lower bounds for Scenario 1 shown in Figure 4. Hence we claim that BFD is a good heuristic in terms of both accuracy and run-time for Scenario 1. Especially when we use our heuristics for intelligent repair, the yield closely follows the upper bound obtained from order statistics model in terms of SBDTir. From Figure 6, the following can be concluded for Problem 1.

TABLE I
SIZE OF LARGEST PROGRAM SECTIONS

Program/bits	P1	P2	P3	P4	P5
Original size	1,942,776	6,860,648	7,695,576	1,292,248	20,138,656
Scenario 1	1,942,776			1,292,248	
Scenario 2		1,228,800	1,810,432		
Scenario 3		614,400	1,228,800		1,216,512

i) Memory yield is determined by the program size, which is shown in Table I. P2, P3 and P5 have yields equal to 0 because these programs are too large to fit into any memory *partition*, for given memory size, defect density and hardware redundancy.

ii) The improvement provided by SBDTcr and SBDTir compared to HR increases as memory size increases.

iii) SBDTir provides higher yield improvement than SBDTcr. This demonstrates that different defects do have different levels of inpact on yield for our appraoch.

iv) For P1 and P4, the improvement provided by SBDTcr with increasing memory size is small because the yield improvement is negated by additional defects that occur in the added memory.

Yield improvement for standard-size memory modules, i.e., whose memory size is a power-of-2, for all these techniques are shown in Table III. In this table, classical approach with given levels of HR is used as a baseline, where yield is computed as the probability of defect free chips and fully repaired chips, which is independent of the program. In the second column, *ALL* approach refers attentively to HR, SBDTcr, and SBDTir.

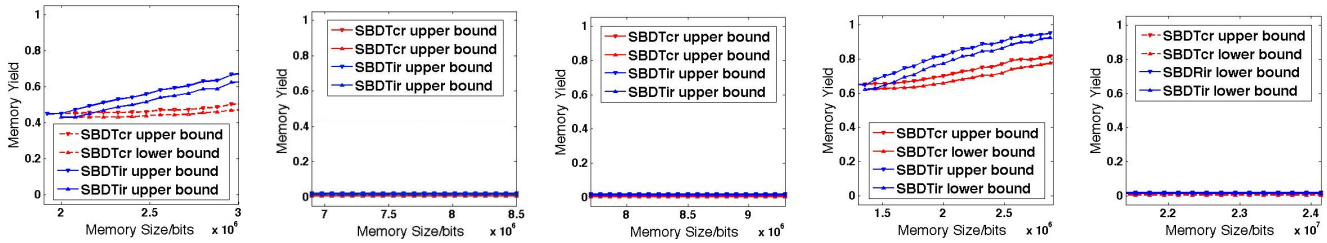
5.3. Yield estimation for Problem 2

Now we consider that program sections are of two types in Scenario 2 for each program. Code *sections* contain segments of machine code; data *sections* contain user data as well as system data, e.g., stacks. Usually, user data *sections* contain independent data arrays such that we can reasonably assume that each array is compiled into one *section*. (Due to lack of space, we do not provide the size of every *section* in the paper. Interested readers can contact us via the Program Council Chair for this information.).

Yield improvements for the traditional HR as well as our SBDTir and SBDTcr approaches are computed for all five example programs and the results are shown in Figure 6. The results lead to the following observations.

i) Yield of 100% is achieved for small programs (P1 and P4) and given memory size, the yield for larger programs (P2 and P3) increases, at the same time, yield for the program with very large sections (P5) remains close to 0%.

ii) In Scenario 2, some programs have dominating sections. When this is the case, the difference between SBDTcr and SBDTir is less than in Problem 1, as only one spare row (column) is assumed available for repair and that one single spare often fails to ensure that all large *sections* are assigned memory space in both SBDTcr and SBDTir.



(a) P1 (b) P2 (c) P3 (d) P4 (e) P5

Fig. 4. Analytical estimation of yield for Problem 1 for five example programs P1 to P5

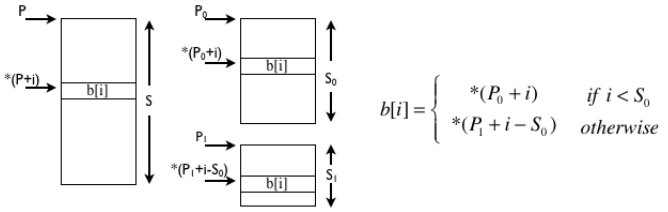


Fig. 5. Compiler-level remapping

TABLE II
CATEGORIES OF BRANCH INSTRUCTIONS

To a label:	B.S1	dotp4clasmfunc
To an address:	B.S1	0x80007A44
To a register:	MVK.S2	0x27b4,B3
	B.S1	B3

5.4. Yield estimation for Problem 3

From Table III, we can see that in Scenario 2 yield for P1 and P4 becomes 100% for all given standard-size memories. However, P2 and P3 need memories with more than twice their program sizes to reach 100% yield. More importantly P5 still has yield of 0. The relatively low yields for P2, P3, and P5 result from large data arrays used in these programs that become dominating sections. Media applications often have large data sections, such as images and video frames. For example, a 240×320 $pixel^2$ image with 16-bit color depth is an 153,600-Byte variable if stored as an array in one of the data *sections*.

In our bin packing model, large objects are difficult to pack into any specific defective memory. For programs with large data sections, this creates a bottleneck for improving memory yield. Hence we propose a data partitioning technique to address this problem for data dominant programs and a code partitioning technique for code dominant programs.

5.4.1 Data partitioning

For large data sections, partitioning can be implemented by compiler-level remapping shown in Figure 5. The compiler is enhanced to partition a large array into sub arrays where each sub array is assigned an index pointer. For subsequent bin-packing, each sub-array becomes a separate smaller data section. The program must be modified correspondingly, and a general approach is to insert a checking statement before every access to the original memory array. This check can identify the appropriate sub-array and use its index and modified offset to access the correct data item. To evaluate performance degradation of this partitioning technique, function $TIMER\_getCount()$ in library csl6713.lib is used to measure run-time of programs in terms of number of CPU cycles. Then performance of *critical routines* that are affected by partitioning are compared for cases with and without partitioning.

To identify optimal partitioned data *sections* that meet a certain yield threshold, a simple heuristic is developed based on the tradeoff between program run time and effective yield. Program run time is obtained by simulating modified assembly code on DSP6713 shown in Table III. As 100% yields are already obtained for P1 and P4, data partitioning is only performed for P2, P3 and P5. According to the results in Figure 6, we have the following observations.

i) After partitioning extra yield improvement is obtained for P2 and P3, and the amount of yield improvement is limited by the new dominating *sections*.

ii) As for Problem 2, as the number of dominating *sections* increases, the difference between SBDTir and SBDTcr decreases.

iii) Non-zero yield is obtained for P5. As its program size is much larger than others, it needs a much larger memory.

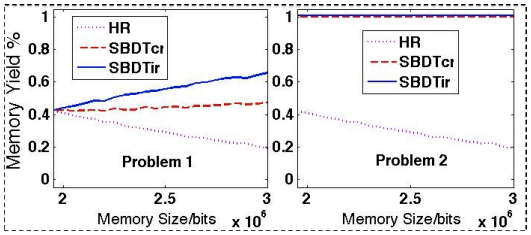
iv) Performance degradation due to data partitioning is negligible.

Yield improvement and performance degradation for standard-size memory modules are shown in Table III. For example, a "1" in the "partition" column indicates that the largest data array is partitioned once into two sub arrays and a "3" indicates that the largest data array is partitioned three times, i.e., into eight sub arrays. The last column shows the corresponding performance degradation.

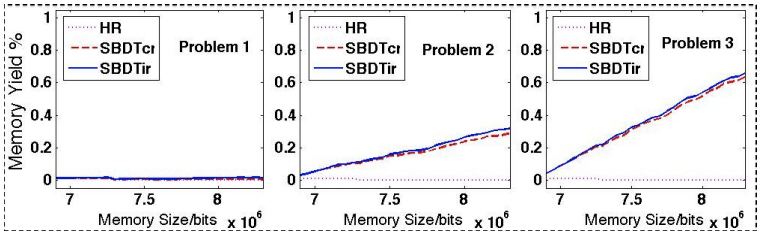
5.4.2 Code partitioning

Code partitioning technique is introduced based on a new notation, *unconditional block*. Similar to the definition of *basic block* in computer architecture, we define an *unconditional block* as a block of instructions that ends with an unconditional branch. Hence, *unconditional blocks* are independent code *sections* before compiled files are linked. It is worth mentioning that as conditional branches (i.e., BEQ) can exist inside *unconditional blocks* and jump to labels outside this *unconditional block*. That is, instructions from one *unconditional block* do not execute exactly once, in order. Hence *unconditional blocks* are not *basic blocks* by definition. In disassembled instruction streams of COFF executables, all *branch* instructions can be divided into three categories shown in Table II.

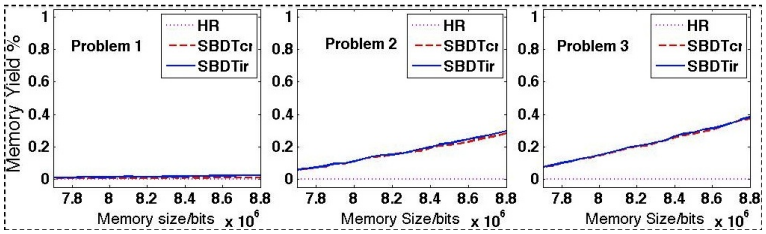
After the study of disassembled instructions of the example program "Dotp4casm" on TI DSP 6713 DSK, we have



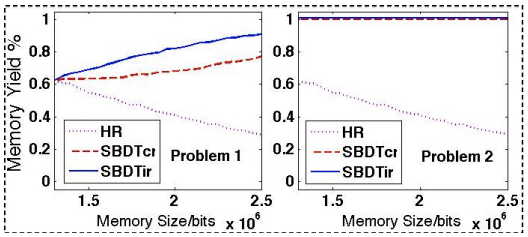
(a) Yield for P1



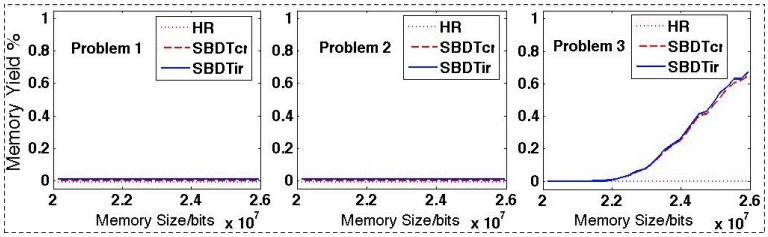
(b) Yield for P2



(c) Yield for P3



(d) Yield for P4



(e) Yield for P5

Fig. 6. Experiment results

following observations:

i) *Using only the existing unconditional branches*, the compiler is able to partition the program into blocks with a maximum size of 932 Bytes. This is encouraging in terms of bin packing algorithms, as it implies that memory yield for code-dominated programs can be always raised to 100% by appropriate code partitioning.

ii) As this approach only uses existing unconditional branches, it does **not** introduce any performance degradation or memory overhead.

6. Conclusion

This paper proposes a software-based defect-tolerance (SBDT) approach and its variants, SBDTcr and SBDTir, for application-specific embedded systems, which are demonstrated to incur trivial performance overhead and significantly improve yield for embedded systems memories.

TABLE III
YIELD FOR STANDARD SIZE MEMORY MODULES

Defect density: 10^{-6}		1 spare row (column) available						
Scenario	Approach	256K	512K	1M	2M	4M	Partition	Penalty%
Defect free	HR	38	8	0	0	0	-	-
Program 1								
1	SBDTcr	42.7	60.34	84.51	97.83	99.98	-	-
	SBDTir	46.05	81.22	97.49	99.92	100	-	-
2	ALL	100	100	100	100	100	-	-
3	ALL	100	100	100	100	100	-	-
Program 2								
1	ALL	-	-	0	0	0	-	-
2	SBDTcr	-	-	21.48	98.14	100	-	-
	SBDTir	-	-	34.33	98.74	100	-	-
3	SBDTcr	-	-	67.45	100	100	1	0.0298
	SBDTir	-	-	69.06	100	100	1	0.0298
Program 3								
1	ALL	-	-	0	0	0	-	-
2	SBDTcr	-	-	18.48	97.63	100	-	-
	SBDTir	-	-	19.14	98.64	100	-	-
3	SBDTcr	-	-	24.76	99.7	100	1	0.6166
	SBDTir	-	-	25.26	99.81	100	1	0.6166
Program 4								
1	SBDTcr	70.21	91.04	99.37	100	100	-	-
	SBDTir	84.33	98.43	99.99	100	100	-	-
2	ALL	100	100	100	100	100	-	-
3	ALL	100	100	100	100	100	-	-
Program 5								
1	ALL	-	-	-	-	-	-	-
2	ALL	-	-	-	-	-	-	-
3	SBDTcr	-	-	-	-	97.85	3	1.6176
	SBDTir	-	-	-	-	98.38	3	1.6176

References

[1] I. Koren and Z. Koren, "Defect tolerance in VLSI circuits: techniques and yield analysis: techniques and yield analysis," *Proc. IEEE*, 1819-1836, 1998.

[2] Cha, J.C. and Gupta, S.K, "Characterization of granularity and redundancy for SRAMs for optimal yield-per-area," *ICCD*,219-226, 2008

[3] Texas Instruments, "TMS320C54x Assembly Language Tools User's Guide".

[4] S. E. Schuster, "Multiple word/bit redundancy for semiconductor memories," *IEEE J. Solid-State Circuits*, vol. SSC-13, pp.698703, Oct. 1978.

[5] Barth, J E Jr, Dreibelbis, J H, and Nelson, E A,"Embedded DRAM design and architecture for the IBM $0.11-\mu m$ ASIC offering,"2002.

[6] Youhei Zenda, Koji Nakamae and Hiromu Fujioka, "Cost optimum embedded DRAM design by yield analysis," *MTDT*, 2003.

[7] Sandro Bartolini, Cosimo Antonio Prete, "An Object Level Transformation Technique to Improve the Performance of Embedded Appli-

cations," *First IEEE International Workshop on Source Code Analysis and Manipulation*, 2001.

[8] John Barth, Don Plass, Erik Nelson, Charlie Hwang, Greory Fredemn, Michael Sperling, Abraham Mathews and Toshiaki Kirihata,"A 45 nm SOI Embedded DRAM Macro for the $POWER^{TM}$ Processor 32 MByte On-Chip L3 Cache,"2011.

[9] Chao, M.C.-T., Ching-Yu Chin and Chen-Wei Lin , "Mathematical yield estimation for two-dimensional-redundancy memory arrays."

[10] Yervant Zorian and Virage Logic, "Embedded Memory Test & Repair: Infrastructure IP for SOC Yield."