An Efficient Algorithm for Spare Allocation Problems

Hung-Yau Lin, Fu-Min Yeh, and Sy-Yen Kuo, *Fellow, IEEE*

***Abstract*—The spare allocation problem in redundant RAM is to replace faulty rows/columns of memory cells with spare rows/columns. To solve the problem, comparison-based search tree structures were used in traditional exact algorithms. These algorithms are not efficient for large problems because significant amounts of data have to be retained and copied in order to generate new partial solutions. Many data may need to be compared for the removal of each redundant partial solution. To overcome these drawbacks, an efficient algorithm is proposed in this paper. The algorithm transforms a spare allocation problem into Boolean functions, and the renowned BDD is used to manipulate them. Experimental results indicate that the proposed algorithm is very efficient in terms of speed and memory requirements. It may also be useful for problems which can be modeled as constraint bipartite vertex cover problems.**

***Index Terms*—BDD, bipartite graph, Boolean functions, exact algorithm, memory repair, RRAM, vertex cover.**

ACRONYMS,[1] NOTATIONS AND DEFINITIONS

BDD	Binary Decision Diagram
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
RAM	Random Access Memory
RRAM	Redundant RAM
SSRRAM	A spare-shared RRAM in which spare lines can be cut into segments and shared among memory blocks
SR/SC	the number of spare rows/columns in a reconfigurable memory array
Line	A line is a row (or a column) of memory cells
row/column set	The two sets of vertices in a bipartite graph are called the row set, and the column set.
Cover	A cover is a set of vertices such that every edge of a bipartite graph has at least one of its end vertices in the set.
feasible cover	A cover is said to be *feasible* if it includes at most SR vertices in the row set, and at most SC vertices in the column set.
minimal feasible cover	It has the minimal number of vertices among all feasible covers.
spare set	a set of spare rows/columns which can be used to replace the lines in the same memory block
s_m	the number of spare lines for spare set m
d_m	the number of faulty lines within the *duty* of spare set m. A faulty line is within the duty of a spare set if it can be replaced with the spare set.

Manuscript received May 29, 2005; revised November 1, 2005; November 30, 2005. This work was supported in part by the National Science Council, Taiwan, R.O.C., under Grant NSC 94-2213-E-002-082. Associate Editor: G. Levitin.

H.-Y. Lin is with the Department of Electrical Engineering, National Taiwan University, Taipei 106, Taiwan, R.O.C.

F.-M. Yeh is with the Chung-Shan Institute of Science and Technology, Taoyuan, Taiwan, R.O.C.

S.-Y. Kuo is with the Department of Electrical Engineering, National Taiwan University, Taipei 106, Taiwan, R.O.C. He is also with the Department of Computer Science and Engineering, National Taiwan Ocean University, Keelung, Taiwan, R.O.C. (email: sykuo@cc.ee.ntu.edu.tw).

Digital Object Identifier 10.1109/TR.2006.874942

[1]The singular and plural of an acronym are always spelled the same.

I. INTRODUCTION

THE speed of CPU has been increasing much faster than DRAM since the 1980s. This leads to a larger performance gap between CPU and DRAM [1]. To reduce the performance impact of the slower DRAM on a computer system, cache memory was added to the design of a system. In 1984, Motorola added 256 bytes of instruction cache memory into its MC68020 CPU. Later in 1989, Intel added 8 kilo-bytes (KB) of on-die cache memory to the 80486 CPU [2]. The Alpha 21064 CPU contained 16 KB of cache, and the PowerPC 601 CPU contained 32 KB of unified cache when they were introduced respectively in 1992, and 1993 [3]. Recent desktop CPU, such as AMD Athlon 64, Intel Pentium 4, Motorola PowerPC 7450, and IBM PowerPC 970, include at least 512 KB of on-die L2 (level two) cache. The dual-core 64-bit UltraSPARC server processor contains 512 KB of cache in each core [4]. The Power4, Power4+, and Power5 include respectively 1.41 MB, 1.5 MB, and 1.875 MB of on-die L2 caches [5]. The 64-bit 1.6-GHz SPARC processor contains 4 MB of on-die L2 cache [6]. The 1.5-GHz Itanium 2 processor has 6 MB of on-die L3 (level three) cache [7]. The 1.7-GHz Itanium 2 processor includes 9 MB of on-die L3 cache [8]. About 44% of the 5.2 million transistors in the UltraSPARC are in the on-die cache memory [9], [10], and about 86% of the 592 million transistors in the 1.7-GHz Itanium 2 processor are in the on-die L3 cache [8]. On-die cache memories have become an important driving force of chip yield as they get larger.

In addition to the increase in the size of on-die cache memory, the capacity, and the density of DRAM also grow along with the advancement of VLSI technology [1]. As the memory density increases, it becomes harder to fabricate memories containing no defect. One of the techniques for improving the yield is to replace faulty rows/columns of memory cells with spare rows/columns. Such devices are known as the RRAM or reconfigurable memory arrays. Because replacing faulty lines with

0018-9529/$20.00 © 2006 IEEE

Authorized licensed use limited to: National Taiwan University. Downloaded on January 19, 2009 at 23:48 from IEEE Xplore. Restrictions apply.

spare lines is simple and effective, this technique has been used in producing DRAM and CPU for years [10], [25], [32]–[34].

Searching for a solution which replaces all faulty cells in a memory array with spare lines is called a spare allocation problem, or a constrained bipartite vertex cover problem [11], [12]. Spare allocation problems have been shown to be NP-complete [13]. There are two categories of algorithms for spare allocation problems: heuristic algorithms, and exact algorithms. Many heuristic algorithms and exact algorithms have been proposed [11]–[30]. Though heuristic algorithms can be very fast, they cannot guarantee a solution to be found even if one exists. Exact algorithms are able to find the solution if a solution does exist. Exact algorithms seem to be preferred. However, the worst-case running time and worst-case memory space requirement in exact algorithms grow exponentially with the problem size. An exact algorithm can be separated into two stages. The first stage uses some preprocessing (or filter) algorithms to remove as many faulty cells as possible, or to stop as early as possible. The second stage usually searches for a solution by applying an exhaustive search algorithm, and it accounts for the worst-case exponential time complexity. The separation is made because an exact algorithm can usually incorporate the filter algorithms used in other exact algorithms.

Spare lines are assumed to have no faulty cells in this paper. A spare allocation problem can be easily modeled as a constrained bipartite vertex cover problem [13]. In the model, there is an edge between a vertex in the row set, and a vertex in column set iff the memory cell at the intersection of the corresponding row and the corresponding column is faulty. A set of vertices such that every edge of a bipartite graph has at least one of its end vertices in the set is called a *cover*. A set of vertices is said to be *feasible* if it includes at most SR vertices in the row set, and at most SC vertices in the column set [27]. A *minimal feasible cover* has the minimal number of vertices among all feasible covers. The most efficient exact algorithm for minimum vertex cover problems is presented in [12]. However, a minimal vertex cover is not necessarily feasible [26]. Although the algorithm in [12] may not be able to find a repair solution for a spare allocation problem, their filter algorithm to be described in Section II-D is very efficient.

To solve spare allocation problems efficiently, a two-stage exact algorithm called PAGEB is proposed. The first stage of PAGEB uses the filter algorithms described in Section II. The second stage of PAGEB transforms the remaining problem from the first stage into a set of Boolean formulas, and uses BDD (Binary Decision Diagram) [36] to solve the Boolean formulas. All solutions of a spare allocation problem are encoded in a BDD, and the optimal solution can be found by traversing the BDD only once. Depending on the definition, the optimal solution can be either the solution that uses the least number of spares, or the solution that has the lowest cost. Advancement of laser and integration technologies has allowed spare lines to be cut into segments, and shared between memory blocks [26], [37], [38]. The PAGEB algorithm can also be extended to solve the spare allocation problems in SSRRAM. Experiments using the PAGEB and branch-based exact algorithms were conducted. Experimental results indicate that the proposed algorithm can be very efficient in terms of speed and memory requirements.

II. Preprocessing/Filter Algorithms

Filter algorithms usually have polynomial time complexity. They are often used in the first stage of exact algorithms for two main purposes: 1) to stop as early as possible if the problem is known to have no solution, or 2) to filter out as many faulty cells as possible so that fewer faulty cells are left to the more time-consuming second stage. Many Filter algorithms have been presented in the literatures [12]–[14], [26], [27] and those used in PAGEB are discussed in this section.

A. The Must-Repair Algorithm

If the number of faulty cells in a row exceeds the number of spare columns, the faulty row must be replaced with a spare row in order to have a minimal feasible cover. Similarly, a faulty column must be replaced with a spare column if the number of faulty cells in the column is greater than the number of spare rows. This is called the *must-repair algorithm* because it replaces the lines that must be repaired in order to have a solution. It is often used in exact and heuristic algorithms as an initial screening step.

B. The Early-Abort Algorithm

The purpose of the early-abort algorithm is to abort program execution as early as possible if the problem is known to have no solution. The upper bound for the early-abort algorithm presented in [25] is not very useful in practice. A more practical upper bound has been proposed in [13], and it uses maximum matching [39]–[41] as the early-abort criterion. If the size of a maximum matching is greater than the number of spare lines left after the initial screening algorithm, then the problem contains no solution, and the search can be aborted early.

C. The Single-Faulty-Cell Filter

The single-faulty-cell filter is used primarily to reduce the number of partial solutions generated in the time-consuming second stage. A *single faulty cell* is a faulty cell that does not share its line with any other faulty cells [14]. It is clear that a single faulty cell must be replaced by either a spare row or a spare column. Before the second stage starts, the single faulty cells are recorded and filtered out. After the second stage finds a potential minimal feasible cover, the remaining spare lines can be easily calculated. If the number of remaining spare lines is greater than or equal to the number of single faulty cells, then a solution can be found. Otherwise, no solution exists for the potential minimal feasible cover.

D. The Gallai-Edmonds Structure Theorem

A spare allocation problem can be easily modeled as a bipartite graph. Many algorithms including maximum matching in Graph theory [42] have been developed for bipartite graphs. Hasan & Liu [26] introduced the concept of critical sets. A *critical set* is the intersection of all minimal vertex covers. Later, Hadas & Liu [27] presented *excess-k critical sets* which are conceptually similar to the critical sets. In their algorithm, the procedure of finding critical sets is interlaced with an exhaustive search algorithm. Chen & Kanj [12] observed that the Gallai-Edmonds structure theorem is a stronger version of critical sets, and showed that such interlacing is not necessary.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 19, 2009 at 23:48 from IEEE Xplore. Restrictions apply.

The Gallai-Edmonds theorem partitions the vertices of a bipartite graph into three sets: the independent set D, the intersection set A, and the perfect matching set C. The *independent set* D is the set of vertices not contained in any minimum vertex cover. The *intersection set* A is the intersection of all minimum vertex covers. The *perfect matching set* C is the set of vertices such that $C = V - D - A$, where V is the set of vertices in the bipartite graph. A bipartite sub-graph $G(C)$ induced from set C has a *perfect matching* [42]. If a spare allocation problem has a minimal vertex cover, all vertices in set A must be included in the final solution, and all vertices in D should not be included. The vertices in C are the only vertices that need to be processed in the second stage. The procedures for constructing the three sets can be found in [12], and they are briefly described here. Let G denotes a bipartite graph. Firstly, compute a maximum matching M for G. Let W be the set of unmatched vertices under M. A vertex v is in set D iff v is reachable from a vertex in W via an alternating path of *even length*. An *alternating path* is a simple path $\{v_0, v_1, \ldots, v_n\}$ such that vertex v_0 is unmatched, and the edges $[v_{2k-1}, v_{2k}]$ are in M for $1 \leq k \leq \lfloor n/2 \rfloor$. After set D has been found, set A can be computed with the following rule: set A is the set of vertices in $V - D$ such that each vertex in A is adjacent to at least one vertex in D. Set C can be computed with the equation $C = V - D - A$.

The filter algorithms in [26], [27] are less efficient than the Gallai-Edmonds theorem for the inability to separate set D from set C. While the Gallai-Edmonds theorem can be a powerful filter for some bipartite graphs, it may be of little benefit to others. A graph with a perfect matching has all of its vertices in set C [30]. Note that even though a minimal vertex cover should include all the vertices in set A, the constraints on the spares may not allow all of them to be included.

III. Exact Algorithms

The branch-and-bound (B&B) algorithm is a classic branching algorithm [13]. It is briefly discussed in this section along with its potential problem. Later in this section, the second stage of the PAGEB algorithm is described.

A. The B&B Algorithm

The B&B algorithm, a simple fault-driven approach, is the cornerstone of many exact algorithms [13], [14], [19], [26], [27]. The algorithm starts with an empty priority queue. The entries in the priority queue are sorted according to some user-defined cost functions [13]. Firstly, a faulty cell is chosen for branching. Two sub-problems (also called partial solutions) are generated by replacing the faulty cell with either a spare row or a spare column. These two partial solutions are inserted into the priority queue. Then, a partial solution at the head of the priority queue is taken for branching each time during iterations until a solution is found, or the priority queue becomes empty. If the priority queue becomes empty, then no solution can be found. For a spare allocation problem which has no solution, it has to examine all partial solutions before it concludes that no solution exists.

The B&B algorithm is similar to searching in a binary search tree except that the tree is packed into a priority queue. The number of nodes in a binary search tree, in this case the number of partial solutions, grows exponentially with the depth of the tree [43]. The exponential growth rate also plagues the B&B algorithm. It may require a lot of memory space to store all partial solutions. Even if a computer system can store all partial solutions, processing them can take a great amount of time. It is not necessary to sort the queue if an optimal solution is not required. While sorting can remove duplicates in the priority queue, and avoid redundant computation, sorting itself takes time. Millions of partial solutions may be generated for a spare allocation problem. For example, there are 2,704,156 partial solutions remaining in the unsorted queue when a solution is found for the problem with 144 faulty cells (24 effective faulty cells) [30]. Please note that the branching operations in [30] are performed on effective faulty cells. If branching operations are on faulty cells, many more partial solutions will be generated, and the performance will be much worse. Because of the exponential growth rate, the performance of a branch-based algorithm can decline dramatically with the increase of the problem size. The drawback of comparison-based branching algorithms becomes evident.

B. The Second Stage of the PAGEB Algorithm

To solve spare allocation problems more efficiently, an exact algorithm called PAGEB is devised. All filter algorithms described in Section II are used in PAGEB. Besides, the repair-most heuristic algorithm [13] can also be used in the first stage of PAGEB if an optimal solution is not required. The repair-most algorithm is a greedy algorithm, and it repeatedly replaces the line having the most faulty cells. Instead of making improvements to branch-based algorithms, the second stage of PAGEB transforms a spare allocation problem into a set of Boolean functions. BDD is used to manipulate the Boolean operations because BDD has been known to be an efficient tool for Boolean function manipulations [36], [44].

Unlike branch-based algorithms which include much information in each partial solution, a BDD node only records the minimal amount of data necessary for BDD operations. It means that each BDD node takes much less memory space than a partial solution. BDD also has a very useful property: it can merge isomorphic graphs together. The property can further reduce memory requirements because isomorphic graphs do not need to be duplicated. In addition, the property can also help boost performance. Operations performed on many graphs which are isomorphic to each other can be reduced to the operations on a single graph. Therefore, PAGEB can somewhat avoid the great number of copy and comparison operations inherent in traditional branch-based algorithms. Fig. 1 lists the pseudo-code of the PAGEB algorithm. The DF, CF, and RF will be described in detail later.

1) The Defect Function DF: A defect function is a Boolean function which encodes the locations of all faulty cells. Suppose R_i and C_i denote respectively the row and column Boolean variables of a faulty cell i. It is apparent that the following defect function encodes the locations of all faulty cells of a spare allocation problem.

$$DF = \prod_{faulty\ cell\ i} (R_i + C_i) \tag{1}$$

Authorized licensed use limited to: National Taiwan University. Downloaded on January 19, 2009 at 23:48 from IEEE Xplore. Restrictions apply.

```
PAGEB() {
  /* start of the first stage */
  Filter the original problem with the must-repair filter;
  Use the early-abort algorithm to stop early if the problem has no solution;
  Apply the Gallai-Edmonds structure theorem to filter out faulty cells;
  Use the single-faulty-cell filter if desired;
  The repair-most algorithm can be used if an optimal solution is not required;
  /* start of the second stage */
  Choose a good BDD variable ordering using a heuristic algorithm;
  Construct the BDD of the defect function DF;
  Construct the BDD of the constraint function CF;
  Construct the repair function RF with the equation RF = DF AND CF;
  Traverse RF;
  if (RF contains only one BDD node 0)
    no repair solution can be found;
  else
    report a repair solution;
}
```

Fig. 1. The PAGEB algorithm.

To replace all faulty cells, the Boolean expression inside the parentheses needs to be evaluated and found to be *true*.

2) The Constraint Function CF: A constraint function is a Boolean function encoding all combinations of faulty lines which are replaceable by spare lines. Because the constraints for rows are orthogonal to the constraints for columns, they can be considered separately. Suppose a memory array has d faulty rows. (2) encodes CF_r, the *constraint function for row*.

$$CF_r = \sum_{j=0}^{SR} C_j^d \tag{2}$$

Let the Boolean variables for the d faulty rows be $R_1, R_2, \ldots, R_d$. Among the d faulty rows, j faulty rows can be replaced, and j can range from zero up to SR. Each combinatorial function C_j^d represents a set of valid combinations of spare rows. Note that the definition of the combinatorial function is different from its conventional one. In conventional mathematics, C_j^d is the *numbe*r of valid combinations if j objects are to be picked out of d objects. In this paper, it is defined to be the summation of the *Boolean expressions* for the combinations. For example, suppose one of 3 faulty lines (say R_1, R_2, and R_3) is to be replaced by a spare line. The number of valid combinations is $C_1^3 = 3$. However, the value 3 is of little use for encoding CF_r. The Boolean expressions for the combinations ($R_1\overline{R_2}\overline{R_3}$, $\overline{R_1}R_2\overline{R_3}$ and $\overline{R_1}\overline{R_2}R_3$) are more useful.

Many Boolean expressions may be consolidated into a single Boolean expression after the summation. For example, $\overline{R_1}\overline{R_2} + \overline{R_1}R_2$ is equivalent to $\overline{R_1}$. It is desired to get the simplified Boolean expression early rather than perform redundant computation. Direct enumeration of each combinatorial function in (2) does not seem to be a practical way to compute CF_r. Decomposition of (2) is a more practical, systematic approach. Let $R_x \in (R_1, R_2, \cdots, R_d)$. It is apparent that (2) can be expanded on R_x as

$$\sum_{j=0}^{SR} C_j^d = R_x \bullet \left(\sum_{m=0}^{SR-1} C_m^{d-1}\right) + \overline{R_x} \bullet \left(\sum_{n=0}^{SR} C_n^{d-1}\right) \tag{3}$$

The symbols of the d faulty lines can be put into an array L in any order. Let the d Boolean symbols in the array L be denoted by $L_0, L_1, \ldots, L_{d-1}$. A placement function PF can be defined as

$$PF(L_k, d, SR) = L_k \bullet PF(L_{k+1}, d-1, SR-1) + \overline{L_k} \bullet PF(L_{k+1}, d-1, SR) \tag{4}$$

Note that (3) & (4) are equivalent. The meaning of (4) is intuitive. If a faulty line L_k is to be replaced by a spare line, the remaining number of spare lines will be one less than the original. If L_k is not to be replaced by a spare line, the remaining number of spare lines will be the same as the original. In both cases, L_k is removed from the set of faulty lines, and the number of faulty lines will be one less than the original. Some identical placement functions may be generated during the decomposition process of (4). Thus, the use of a hash table can avoid redundant computation in (4).

(5), and (6) are two useful properties of the placement function. These two properties are intuitive. If the number of spare lines is greater than or equal to the number of defect lines, all defect lines can be replaced and *true* is returned. If no spare line is left, then none of the defect lines can be replaced, and the symbols of these defect lines are complemented and multiplied together.

$$PF(L_k, d, SR) = true \quad if\ d \le SR \tag{5}$$

$$PF(L_k, d, 0) = \overline{L_k} \bullet \overline{L_{k+1}} \bullet \overline{L_{k+2}} \bullet \cdots \bullet \overline{L_{k+d-1}} \tag{6}$$

CF_c, the constraint function for columns, can be constructed in a similar way. The global constraint function CF is the product of the Boolean-**AND** operations between CF_r and CF_c.

$$CF = CF_r \bullet CF_c \tag{7}$$

3) The Repair Function RF: The repair function is a Boolean function which encodes all repair solutions of a spare allocation problem. Because the DF encodes the lines to be replaced, and the CF encodes the lines that the spare lines can replace, it is obvious that the repair function RF can be constructed with (8). The fact that the RF encodes all repair solutions can be easily proved.

$$RF = DF \bullet CF \tag{8}$$

Traditional exact algorithms search for the repair solution in each step of the branching process. Unlike those algorithms, the second stage of PAGEB does not search for the repair solution until the RF has been constructed. As soon as the RF is built, a repair solution can be found by traversing only once the BDD of the RF. A path from the *top variable* of a BDD to the *terminal node* 1 is called a *solution path*. All Boolean variables in a

Authorized licensed use limited to: National Taiwan University. Downloaded on January 19, 2009 at 23:48 from IEEE Xplore. Restrictions apply.

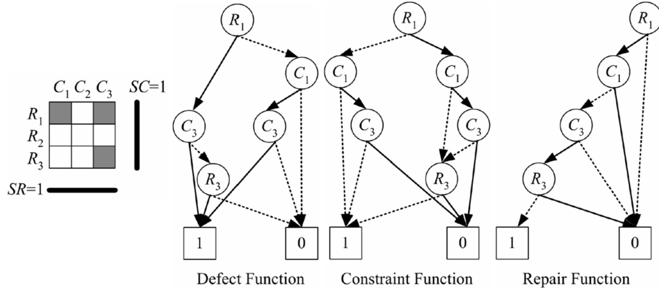

Fig. 2. A simple example.

solution path taking the 1-edge form a repair solution. In the example below, the 1-edge, and the 0-edge are drawn respectively in solid lines, and dotted lines. The 1-edge and 0-edge in a BDD can be represented by pointers in some computer languages. By following these pointers, the RF can be traversed efficiently. Finding an optimal repair solution is almost as easy as finding a non-optimal one. PAGEB is very efficient due to the fact that BDD can remove redundant nodes, combine isomorphic graphs, and have very compact representations of Boolean functions if a good variable ordering is used. Unlike other exact algorithms which stop searching when a repair solution is found, PAGEB encodes all solutions in the repair function.

4) A Simple Example: Fig. 2 shows a simple memory array with one spare row, one spare column, and three darkened faulty cells. The thick black lines represent the spare lines. The example is used for its simplicity to convey the basic idea of the second stage of PAGEB even though the solution can be found by the must-repair filter. For easier understanding, the BDD encoding method of this example is also explained with the If-Then-Else (ITE) connective [31]:

$$f = ite(x, f_1, f_0) = x \bullet f_1 + \overline{x} \bullet f_0 \tag{9}$$

where x is one of the decision variables. The functions f_1, and f_0 are Boolean function f evaluated at $x = 1$, and $x = 0$ respectively. If x & y are two variables with a variable ordering $x < y$, the following equalities hold for the operations between two ITE connectives:

$$\begin{aligned}
ite(x, G_1, G_2) \bullet ite(x, H_1, H_2) &= ite(x, G_1H_1, G_2H_2)\\
ite(x, G_1, G_2) + ite(x, H_1, H_2) &= ite(x, (G_1+H_1), (G_2+H_2))\\
ite(x, G_1, G_2) \bullet ite(y, H_1, H_2) &= ite(x, G_1h, G_2h)\\
ite(x, G_1, G_2) + ite(y, H_1, H_2) &= ite(x, (G_1+h), (G_2+h))\\
h &= ite(y, H_1, H_2)
\end{aligned} \tag{10}$$

The first step in the second stage of PAGEB is to choose a good BDD variable ordering. A variable ordering determines the level of variables in a BDD. Assume the variable ordering $(R_1 < C_1 < C_3 < R_3)$ is chosen. Then the DF can be built with (1), and its ITE form can be derived with ease.

$$\begin{aligned}
DF &= (R_1 + C_1) \bullet (R_1 + C_3) \bullet (R_3 + C_3)\\
&= ite(R_1, 1, ite(C_1, 1, 0))\\
&\quad \bullet ite(R_1, 1, ite(C_3, 1, 0)) \bullet ite(C_3, 1, ite(R_3, 1, 0))\\
&= ite(R_1, 1, ite(C_1, 1, 0) \bullet ite(C_3, 1, 0))\\
&\quad \bullet ite(C_3, 1, ite(R_3, 1, 0))\\
&= ite(R_1, 1, ite(C_1, ite(C_3, 1, 0), 0))\\
&\quad \bullet ite(C_3, 1, ite(R_3, 1, 0))\\
&= ite(R_1, ite(C_3, 1, ite(R_3, 1, 0)),\\
&\qquad ite(C_1, ite(C_3, 1, 0), 0))
\end{aligned} \tag{11}$$

Because of limited space, some derivation steps are omitted. The BDD of the DF is shown in Fig. 2. Note how the final ITE expression maps exactly to the BDD of the DF.

Before the global constraint function can be built, the CF_r and the CF_c have to be constructed with (4)–(6). There are two faulty rows in the memory array. Suppose R_1, and R_3 are inserted respectively at position 0, and 1 in the array L. The following equation shows how to build CF_r.

$$\begin{aligned}
CF_r &= PF(L_0, 2, 1)\\
&= L_0 \bullet PF(L_1, 1, 0) + \overline{L_0} \bullet PF(L_1, 1, 1)\\
&= L_0 \bullet \overline{L_1} + \overline{L_0} \bullet true\\
&= L_0 \bullet \overline{L_1} + \overline{L_0} = R_1\overline{R_3} + \overline{R_1}\\
&= ite(R_1, 1, 0) \bullet ite(R_3, 0, 1) + ite(R_1, 0, 1)\\
&= ite(R_1, ite(R_3, 0, 1), 0) + ite(R_1, 0, 1)\\
&= ite(R_1, ite(R_3, 0, 1), 1)
\end{aligned} \tag{12}$$

The CF_c can be built in a similar way. Then, the CF can be easily constructed with (7). The following equation shows the symbolic function of the CF and its ITE form.

$$\begin{aligned}
CF &= CF_r \bullet CF_c\\
&= \overline{R_1C_1} + \overline{R_1}C_1\overline{C_3} + R_1\overline{R_3C_1} + R_1\overline{R_3}C_1\overline{C_3}\\
&= ite(R_1, ite(C_1, ite(C_3, 0, ite(R_3, 0, 1)), ite(R_3, 0, 1)),\\
&\qquad ite(C_1, ite(C_3, 0, 1), 1))
\end{aligned} \tag{13}$$

It can be easily checked that the ITE form of the CF maps gracefully to its BDD representation. The clause ite(R_3, 0,1) appears twice in the ITE form. The ITE form does not make use of isomorphic graphs while the BDD technique does.

The RF can be now constructed with (8). It is easy to derive the ITE form of the RF, and the derivation detail is omitted. From the BDD of the RF in Fig. 2, there is only one solution path $R_1\overline{C_1}C_3\overline{R_3}$. Thus, the repair solution to this example problem is R_1C_3. This solution can be easily checked with the

Authorized licensed use limited to: National Taiwan University. Downloaded on January 19, 2009 at 23:48 from IEEE Xplore. Restrictions apply.

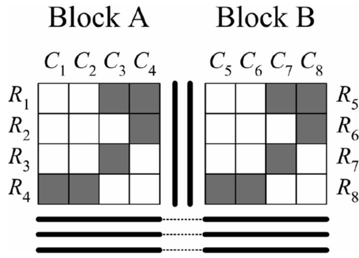

Fig. 3. A memory array with spares being cut and shared among blocks.

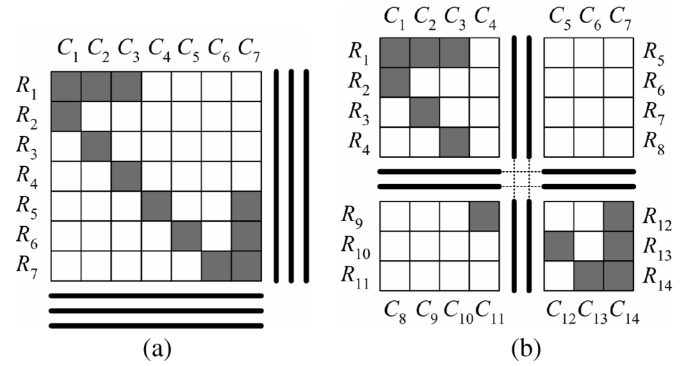

Fig. 4. Figures (a) and (b) have the same fault pattern if the four sub-arrays in (b) are joined side by side.

symbolic function. The symbolic equations above are for illustration purpose only. In a computer program, the BDD representations rather than the symbolic equations are used.

IV. Shared Spares

Advancement of laser and integration technologies has allowed spare lines to be cut into segments. Different segments of a spare line can be used to replace segments of different faulty lines. This allows more flexible use of spare lines. More faulty cells may be replaced with the same amount of spare lines, and the yield may be improved. Spare allocation problems will become more complex if the spare lines can be cut into segments, and shared among adjacent memory blocks. The reason for the higher complexity is that the union of the minimal vertex covers of all blocks is not necessarily feasible. This means that each block cannot be solved independently. Consider the memory array in Fig. 3. Blocks A & B share the two spare columns. The minimal covers for blocks A & B are (R_4, C_3, C_4), and (R_8, C_7, C_8) respectively. The union of the two minimal vertex covers is obviously not feasible because four spare columns are required.

SSRRAM is almost identical to RRAM except that the spares can be cut, and shared among memory blocks. Simple modifications can be made to the filter algorithms in the first stage of PAGEB to keep them compliant to SSRRAM, and they are omitted. Modifications to the second stage of PAGEB are discussed below. Because the global constraint function encodes how the spare lines can be used to replace faulty lines, it needs to reflect new restrictions on the shared spares. In RRAM, the CF is composed of two functions: CF_r and CF_c. These two functions are the constraints for the two sets of spare lines. The notion can be extended. In a SSRRAM, a set of spare rows (or columns) is called a *spare set* if the spares in the set can be used to replace the same memory block. For example, there are three spare sets in Fig. 3: two spare sets at the bottom, and the other between the memory blocks. Each spare set has a constraint function. The constraint function for spare set m can be computed by the equation

$$CF_m = \sum_{j=0}^{s_m} C_j^{d_m} \tag{14}$$

Equation (14) is in fact just the rewrite of (2) to emphasize the notion of spare sets. Suppose there are n spare sets, and the constraint functions for these spare sets are $CF_0, CF_1, \ldots, CF_{n-1}$. In SSRRAM, the CF is the result of performing **AND** operations on the constraint functions of all spare sets.

$$CF = \prod_{m=0}^{n-1} CF_m \tag{15}$$

Equation (15) degenerates into (7) when n equals to 2 in RRAM. The DF, and RF are still built respectively with (1), and (8). There are four solution paths in the RF of Fig. 3, and the four repair solutions are

$$\left\langle \begin{matrix} C_4R_3 \\ C_3R_2 \end{matrix} \right\rangle \left\langle \begin{matrix} C_8R_7 \\ C_7R_6 \end{matrix} \right\rangle R_1R_4R_5R_8 \tag{16}$$

Spare cutting and sharing may improve yield because the use of spares is more flexible. The same number of faulty cells may be replaced with fewer spare lines. The problem in Fig. 4(a) is a well-known problem for which the repair-most algorithm is not able to find the only solution. With PAGEB, the only solution path is correctly encoded in the RF. Removing any spare line from Fig. 4(a) makes the memory irreparable. It is easy to check that two spare rows and two spare columns are not enough to replace all faulty lines in the memory array. However, two spare rows and two spare columns are enough to cover all the faulty cells in Fig. 4(b) if the spare lines are cut and shared between adjacent blocks. The fault pattern in Fig. 4(a) is identical to that in Fig. 4(b) if the four blocks in Fig. 4(b) are joined to adjacent blocks. Fig. 4(b) has three solution paths, and the three repair solutions are

$$R_1C_{14}R_{13}R_{14}C_{11} \left\langle \begin{matrix} C_1C_2C_4 \\ C_1C_3R_3 \\ C_2C_3R_2 \end{matrix} \right\rangle \tag{17}$$

V. Experimental Results

There are almost countless fault patterns for spare allocation problems. It is not practical to experiment with all possible fault patterns. The real fault patterns are classified information in industries. Many models have been published over the years [18], [19], [45]–[48]. Defects have been shown to occur in clusters

Authorized licensed use limited to: National Taiwan University. Downloaded on January 19, 2009 at 23:48 from IEEE Xplore. Restrictions apply.

TABLE I
INFORMATION ABOUT TEST CASES

Test Set	1	2	3	4	5
(*SR*, *SC*)	(10, 10)	(20, 20)	(32, 32)	(29, 29)	(12, 12)
r	5	5	5	5	7
p_1	4×10^{-6}	6×10^{-6}	9×10^{-6}	11×10^{-6}	3×10^{-6}
p_2	0.8	0.8	0.8	0.6	0.7
p_3	0	0	0	0	0
Min faults	68	103	161	145	83
Average faults	83.06	126.85	188.83	169.17	107.76
Max faults	106	147	212	193	150
Repair-Most	8	61	86	46	24
Early-Abort	20	0	0	7	14
Has solution	80	100	100	93	86

r: number of rows/columns in a square

on wafers [45]. The model in [19] claimed to be easy to implement, and reflect the clustering effect. It has been said that the faults become more characteristic of random defects when the process and design have matured [10]. Because no fault model was presented in [10], the model in [19] was used to generate the test cases in this paper. Note that PAGEB is not restricted to spare allocation problems. To measure the general performance of PAGEB, other fault models can also be used. The model proposed in [19] uses a compounded Poisson distribution, and it is briefly described here. An $n \times n$ memory array is partitioned into squares containing $r \times r$ elements. Defects occur s-independently within the same square while occurrences of defects in different squares are statistically dependent. A square is fault prone s-independently of other squares. Let p_1 be the probability for a square to be fault prone, p_2 be the fault probability within a fault prone square, and p_3 be the fault probability within a fault resistant square. The Poisson parameter λ is specified by the criteria

$$\text{Probability}\{\text{square is fault prone}\} = p_1, \quad \lambda = p_2 \times r^2; \quad (18)$$

$$\text{Probability}\{\text{square is fault resistant}\} = \overline{p_1}, \quad \lambda = p_3 \times r^2. \quad (19)$$

Table I shows the parameters used in the model; and it also shows the minimum number, the average number, and the maximum number of faulty cells in the test sets. Each test set has 100 test cases. All memory arrays in the table are of size 1024 $\times$ 1024. The array size is not very important for PAGEB because the information does not appear in any of the Boolean functions. Note that equivalent fault patterns can be created by shifting faulty lines around in the memory array.

To compare the relative performance between a branching algorithm and PAGEB, the second stage of PAGEB is replaced with a branching algorithm. Two versions of branching algorithms are implemented: sorted queue, and unsorted queue. The unsorted version is not able to find the minimal feasible cover. The sorted version may be faster or slower than the unsorted version depending on the fault patterns. A heuristic algorithm was used to generate the static BDD variable orderings in PAGEB because it has been shown in [50] that finding the optimal BDD variable ordering is an NP-complete problem [35]. The basic

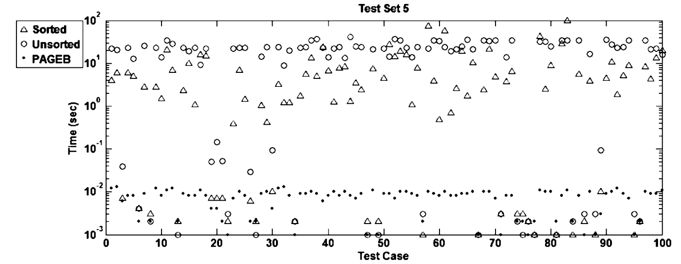

Fig. 5. Experimental results of test set 5.

idea of the algorithm is to put a group of variables as close as possible if they can determine the result of a Boolean expression. Some code snippets written by Setubal [41] were used to search for a maximum matching. The CMU BDD library [49] developed by David Long was used to handle the BDD operations. Everything else was written in the C++ programming language. All program files were compiled with GNU `g++` 3.4.3, and the optimization flag `-O2` was used. The hardware system had 1 GB of memory, and a single AMD Athlon-XP processor running at 2.2 GHz. The operating system was a Linux system with kernel version 2.6.11.

The branching algorithm was aborted if it could not find repair solutions in 100 seconds, or if it ran out of memory space. Note that the value "100 seconds" is used as the execution time in such cases. The benchmark results are summarized in Table II. The results for the test set 5 are plotted in Fig. 5 for illustration. The two branching algorithms ran out of memory or could not finish in 100 seconds for most of the test cases in the test sets 2–4. The maximum execution time of the unsorted branching algorithm is less than the sorted counterpart. However, the average execution time of the unsorted branching algorithm is larger than the sorted counterpart because the unsorted version is slower than the sorted one for most of the test cases. The sorted branching algorithm requires much less memory than the unsorted version because redundant operations are not performed. The unsorted version could not solve any test cases in the test set 3, while the sorted version was able to find repair solutions for 8 test cases. The two branching algorithms can be faster than PAGEB for only a handful of test cases. However, the small lead in time can be ignored because of imprecision in the measurement. The average execution time of PAGEB is under 1 second for all test sets. The average execution time of the branch-based algorithms is much larger than that of PAGEB. It is clear that the proposed PAGEB algorithm is much faster than the two branching algorithms.

PAGEB is able to finish all test cases in test set 4 under one second, except for test case 34. The two branching algorithms could not solve test case 34 in test set 4 within 100 seconds, while PAGEB was able to solve it in 3.64 seconds. The value 3.64 is higher than other test cases in the test set. The fault pattern of the test case was investigated, and no peculiarity was found. There is no apparent reason that PAGEB cannot solve it efficiently. The execution time dropped from 3.64 seconds to 0.22 seconds if PAGEB was compiled with the optimization flag `-O2 - mtune = athlon - xp`. The test was repeated many times, and the performance improvement was almost identical.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 19, 2009 at 23:48 from IEEE Xplore. Restrictions apply.

TABLE II
EXPERIMENTAL RESULTS (IN SECONDS)

Test Set	Unsorted			Sorted			PAGEB		
	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
1	0.001	1.515	2.359	0.001	0.423	4.696	0.001	0.006	0.010
2	0.006	82.577	100	0.002	79.007	100	0.003	0.020	0.040
3	100	100	100	0.017	93.755	100	0.004	0.048	0.154
4	0.001	93.000	100	0.001	92.003	100	0.001	0.132	3.641
5	0.001	17.738	40.624	0.001	7.665	98.972	0.001	0.007	0.013

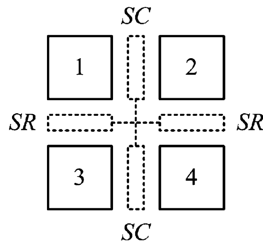

Fig. 6. The layout of the 4-blocks memory array.

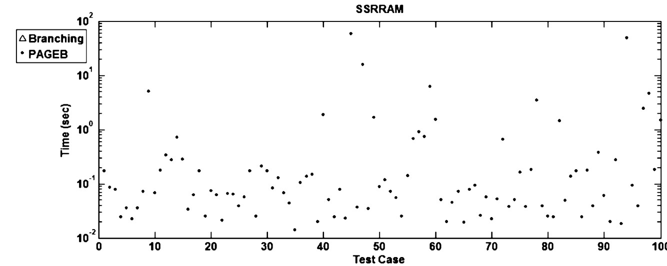

Fig. 7. Experimental results for SSRRAM.

Other test cases do not have such bizarre result. The great difference in performance for the test case may be caused by the optimized hashing mechanisms used in the algorithm.

Although the repair-most algorithm can be embedded into the first stage of the tested algorithms, it was not used in the experiment. However, Table I still lists the number of test cases reparable with the repair-most filter. It also lists the number of test cases aborted early by the early-abort filter, and the number of test cases having repair solutions. 24 out of 100 test cases in test set 5 can be repaired with the repair-most filter, 14 test cases can be aborted early, and 86 test cases have repair solutions. The Gallai-Edmonds theorem is mainly used in the first stage to reduce the problem size, but it was able to find the repair solution for one test case in test set 5.

Both PAGEB and the branching algorithm can be extended to solve spare allocation problems in SSRRAM. To compare the performance between PAGEB and the sorted branching algorithm, 100 4-blocks test cases are generated with the following parameters: $m = 3$, $p_1 = 0.000015$, $p_2 = 0.8$, $p_3 = 0$, and $SR = SC = 15$. Each block is of size 512 × 512. Fig. 6 shows the layout of the 4-blocks test cases. Spares are shared among adjacent memory blocks. Fig. 7 shows the experimental results. The sorted branching algorithm could not solve any of the 100 test cases within 100 seconds, while PAGEB could for all of them. Although PAGEB was able to finish solving most of the test cases in 1 second, test case 45 and 94 took as long as 58.14 seconds and 49.35 seconds respectively. The execution time of these two cases can be reduced to less than 2.1 seconds if the *sift* dynamic variable ordering [51] is used. The results indicate that PAGEB is much faster than the branch-based algorithm.

VI. CONCLUSION

Spare allocation problems are important not only because an increase in chip production yield by a few percentage points can have a substantial impact on the profit, but also because it may find applications in other fields. The experimental results indicate that the proposed PAGEB algorithm is very efficient in terms of speed and memory requirement. The efficiency results from the following good properties: 1) each BDD node requires much less memory space than a partial solution in branch-based algorithms, 2) BDD can merge isomorphic sub-graphs together to avoid redundant computation and reduce memory space requirements, and 3) a solution can be easily obtained by traversing only once the BDD of the repair function. With the good performance, PAGEB can also be used to check the results of other algorithms. The second stage of PAGEB transforms a spare allocation problem into Boolean function operations. Though the SAT solver can replace BDD as the Boolean functions solver, it cannot find the minimal feasible cover. In the applications requiring the minimal feasible cover, BDD is a better option than SAT. However, all BDD-based algorithms require good variable orderings to have better performance, and finding the optimal variable ordering is an NP-complete problem. It is still unknown whether the transformed Boolean functions possess any characteristic that can be used to determine an optimal or near optimal variable ordering. Finally, the proposed algorithm may be useful for other problems which can be modeled as constraint bipartite vertex cover problems.

ACKNOWLEDGMENT

The authors would like to thank H. Fernau, and J. Chen for the discussions and help.

REFERENCES

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 3rd ed. : Morgan Kaufmann, 2004.
[2] M. Rafiquzzaman, *Microprocessors and Microcomputer-Based System Design*. : CRC Press, 1995.
[3] J. E. Smith and S. Weiss, "PowerPC 601 and alpha 21064: a tale to two RISCs," *IEEE Computer*, vol. 27, no. 6, pp. 46–58, Jun. 1994.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 19, 2009 at 23:48 from IEEE Xplore. Restrictions apply.

[4] T. Takayanagi, "A dual-core 64-bit UltraSPARC microprocessor for dense server applications," *IEEE J. Solid-State Circuit*, vol. 40, no. 1, pp. 7–18, Jan. 2005.
[5] R. Kalla, B. Sinharoy, and J. M. Tendler, "IBM Power5 chip: a dual-core multithreaded processor," *IEEE Micro*, vol. 24, no. 2, pp. 40–47, 2004.
[6] H. McIntyre, D. Wendell, K. J. Lin, P. Kaushik, S. Seshadri, A. Wang, V. Sundararaman, P. Wang, S. Kim, W. J. Hsu, H. C. Park, G. Levinsky, J. Lu, M. Chirania, R. Heald, P. Lazar, and S. Dharmasena, "A 4-MB on-chip L2 cache for a 90-nm 1.6-GHz 64-bit microprocessor," *IEEE J. Solid-State Circuit*, vol. 40, no. 1, pp. 9–52, Jan. 2005.
[7] S. Rusu, H. Muljono, and B. Cherkauer, "Itanium 2 processor 6M: higher frequency and larger L3 cache," *IEEE Micro*, vol. 24, no. 2, pp. 10–18, 2004.
[8] J. Chang, S. Rusu, J. Shoemaker, S. Tam, M. Huang, M. Haque, S. Chiu, K. Truong, M. Karim, G. Leong, K. Desai, R. Goe, and S. Kulkarni, "A 130-nm triple-Vt 9-MB third-level on-die cache for the 1.7-GHz Itanium 2 Processor," *IEEE J. Solid-State Circuit*, vol. 40, no. 1, pp. 195–203, Jan. 2005.
[9] M. E. Levitt, "Designing ultraSparc for testability," *IEEE Design & Test*, vol. 14, no. 1, pp. 10–17, 1997.
[10] L. Youngs and S. Paramanandam, "Mapping and repairing embedded-memory defects," *IEEE Design & Test*, vol. 14, no. 1, pp. 18–24, 1997.
[11] H. Fernau and R. Niedermeier, "An efficient exact algorithm for constraint bipartite vertex cover," *J. Algorithms*, vol. 38, pp. 374–410, Feb. 2001.
[12] J. Chen and I. A. Kanj, "Constrained minimum vertex cover in bipartite graphs: complexity and parameterized algorithms," *J. Computer & System Science*, vol. 67, pp. 833–847, Dec. 2003.
[13] S. Y. Kuo and W. K. Fuchs, "Efficient spare allocation in reconfigurable arrays," *IEEE Design & Test*, pp. 24–31, Feb. 1987.
[14] 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. Computer-Aided Design*, pp. 323–328, Mar. 1990.
[15] R. W. Haddad, A. T. Dahbura, and A. B. Sharma, "Increased throughput for the testing and repair of RAMs with redundancy," *IEEE Trans. Computers*, pp. 154–166, Feb. 1991.
[16] C. P. Low and H. W. Leong, "A new class of efficient algorithms for reconfiguration of memory arrays," *IEEE Trans. Computers*, pp. 614–618, May 1996.
[17] ——, "Minimum fault coverage in memory arrays: a fast algorithm and probabilistic analysis," *IEEE Trans. Computer-Aided Design*, pp. 681–690, June 1996.
[18] D. M. Blough and A. Pelc, "A clustered failure model for the memory array reconfiguration problem," *IEEE Trans. Computers*, pp. 518–528, May 1993.
[19] D. M. Blough, "Performance evaluation of a reconfiguration algorithm for memory arrays containing clustered faults," *IEEE. Trans. Reliability*, vol. 45, pp. 274–284, June 1996.
[20] N. Funabiki and Y. Takefuji, "A parallel algorithm for allocation of spare cells on memory chips," *IEEE Trans. Reliability*, pp. 338–346, Aug. 1991.
[21] W. Shi and W. K. Fuchs, "Probabilistic analysis and algorithms for reconfiguration of memory arrays," *IEEE Trans. Computer-Aided Design*, pp. 1153–1160, Sep. 1992.
[22] S. Y. Kuo and I. Y. Chen, "Efficient reconfiguration algorithms for degradable VLSI/WSI arrays," *IEEE Trans. Computer-Aided Design*, vol. 11, no. 10, pp. 1289–1300, 1992.
[23] C. T. Huang, C. F. Wu, J. F. Li, and C. W. Wu, "Built-in redundancy analysis for memory yield improvement," *IEEE Trans. Reliability*, vol. 52, no. 4, pp. 386–399, Dec. 2003.
[24] D. K. Bhavsar, "An algorithm for row-column self-repair for RAMs and its implementation in the Alpha 21264," in *Proc. Int'l Test Conf.*, Sep. 1999, pp. 311–318.
[25] J. R. Day, "A fault-driven comprehensive redundancy algorithm for repair of dynamic RAMs," *IEEE Design & Test*, vol. 2, no. 3, pp. 35–44, 1985.
[26] N. Hasan and C.L. Liu, "Minimum fault coverage in reconfigurable arrays," *IEEE Fault-Tolerant Computing Symp.*, pp. 348–353, June 1988.
[27] R. L. Hadas and C. L. Liu, "Fast search algorithms for reconfiguration problems," in *Proc. Int'l Workshop on Defect and Fault Tolerance on VLSI Systems*, Nov. 1991, pp. 260–273.
[28] H. Y. Lin, F. M. Yeh, I. Y. Chen, and S. Y. Kuo, "An efficient perfect algorithm for memory repair problems," in *Proc. IEEE Symp. on Defect and Fault Tolerance in VLSI Systems*, 2004, pp. 306–313.
[29] H. Y. Lin, H. Z. Chou, F. M. Yeh, I. Y. Chen, and S. Y. Kuo, "An Efficient Algorithm for Reconfiguring Shared Spare RRAM," in *Proc. IEEE Conf. Computer Design*, 2004, pp. 544–546.
[30] H. Y. Lin, F. M. Yeh, S. Y. Kuo, and H. Fernau, BDD Variable Orderings for Spare Allocation Problems unpublished.
[31] W. S. Jung, S. H. Han, and J. Ha, "A fast BDD algorithm for large coherent fault trees analysis," *Reliability Engineering & System Safety*, vol. 83, no. 3, pp. 369–374, March 2004.
[32] G. D. Chevley, "Main memory wafer-scale integration," *VLSI Design*, pp. 54–58, Mar. 1985.
[33] J. I. Raffel, A. H. Anderson, G. H. Chapman, K. H. Konkle, B. Mathur, A. M. Soares, and P. W. Wyatt, "A wafer-scale digital integrator using restructurable VLSI," *IEEE J. Solid-State Circuits*, vol. 20, pp. 399–406, Feb. 1985.
[34] Y. Ueoka, C. Minagawa, M. Oka, and A. Ishimoto, "A defect-tolerant design for full-wafer memory LSI," *IEEE J. Solid-State Circuits*, vol. 19, pp. 319–324, June 1984.
[35] M. R. Garey and D. S. Johnson*, Computers and intractability: a guide to the theory of NP-completeness*. : W. H. Freeman, 1979.
[36] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Computers*, pp. 677–691, Aug. 1986.
[37] Y. N. Shen, N. Park, and F. Lombardi, "Spare cutting approaches for repairing memories," *IEEE Conf. Computer Design*, pp. 106–111, Oct. 1996.
[38] N. Park and F. Lombardi, "Repair of memory arrays by cutting," in *Proc. Int'l Workshop on Memory Technology, Design, and Testing*, 1998.
[39] J. E. Hopcroft and R. M. Karp, "An n5/2 algorithm for maximum matchings in bipartite graphs," *SIAM J. Computing*, pp. 225–231, Dec. 1973.
[40] Z. Galil, "Efficient algorithms for finding maximum matching in graphs," *ACM Computing Surveys*, vol. 18, no. 1, pp. 23–38, Mar. 1986.
[41] J. C. Setubal, Institute of Computing State University of Campinas, Brazil, Technical Report IC-96-09, 1996.
[42] D. B. West*, Introduction to Graph Theory*, 2nd ed. : Prentice Hall, 2001.
[43] H. R. Lewis and L. Denenberg*, Data Structures and their Algorithms*. : Harper Collins Publishers, 1991.
[44] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of an OBDD package," in *Proc. the 27th Design Automation Conf.*, June 1990, pp. 40–45.
[45] C. H. Stapper, "On yield, fault distributions, and clustering of particles," *IBM J. Research & Development*, vol. 30, pp. 326–338, May 1986.
[46] Z. Koren and I. Koren, "A unified approach for yield analysis of defect tolerant circuits," *Defect and Fault Tolerance in VLSI Systems*, vol. 2, pp. 33–45, 1990.
[47] F. J. Meyer and D. K. Pradhan, "Modeling defect spatial distribution," *IEEE Trans. Computers*, vol. 38, pp. 538–546, Apr. 1989.
[48] B. Murphy, "Cost-size optima for monolithic integrated circuits," in *Proc. IEEE*, Dec. 1964, vol. 52, pp. 1537–1545.
[49] The BDD Library [Online]. Available: http://www-2.cs.cmu.edu/afs/cs/project/modck/pub/www/bdd.html.
[50] B. Bollig and I. Wegener, "Improving the variable ordering of OBDDs is NP-complete," *IEEE Trans. Computers*, pp. 993–1002, Sep. 1996.
[51] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proc. Conf. Computer-Aided Design*, 1993, pp. 42–47.

Hung-Yau Lin received in 1998 the B.S. degree in mechanical engineering from National Taiwan University, Taipei, Taiwan. Since September 1999, he has been a PhD student in the department of electrical engineering at the same university. His research interests include computer graphics, data compression, memory repair algorithms, network reliability analysis, and operating systems.

Fu-Min Yeh received the B.S. degree in 1985 in electronic engineering from Chung-Yuan Christian University, the M.S. degree in 1992 in electrical engineering from National Taiwan University, and the Ph.D. degree in 1997 in electrical engineering from National Taiwan University. He is a deputy chief at the Electronic System Research Division of Chung-Shan Research Institute of Science and Technology. His research interests include UWB baseband design, Radar system design, hardware verification, VLSI testing, and fault-tolerant computing.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 19, 2009 at 23:48 from IEEE Xplore. Restrictions apply.

Sy-Yen Kuo is Dean of the College of Electrical Engineering and Computer Science, National Taiwan Ocean University, Keelung, Taiwan. He is also a Professor at the Department of Electrical Engineering, National Taiwan University where he is currently on leave, and was the Chairman at the same department from 2001 to 2004. He received the BS degree (1979) in Electrical Engineering from National Taiwan University, the MS degree (1982) in Electrical & Computer Engineering from the University of California at Santa Barbara, and the PhD degree (1987) in Computer Science from the University of Illinois at Urbana-Champaign. He spent his sabbatical years as a Visiting Professor at the Computer Science and Engineering Department at the Chinese University of Hong Kong from 2004–2005, and as a visiting researcher at AT&T Labs-Research, New Jersey from 1999 to 2000. He was the Chairman of the Department of Computer Science and Information Engineering, National Dong Hwa University, Taiwan from 1995 to 1998; a faculty member in the Department of Electrical and Computer Engineering at the University of Arizona from 1988 to 1991; and an engineer at Fairchild Semiconductor and Silvar-Lisco, both in California, from 1982 to 1984. In 1989, he also worked as a summer faculty fellow at the Jet Propulsion Laboratory of California Institute of Technology. His current research interests include dependable systems and networks, software reliability engineering, mobile computing, and reliable sensor networks.

Professor Kuo is an IEEE Fellow. He has published more than 240 papers in journals and conferences. He received the distinguished research award between 1997 and 2005 consecutively from the National Science Council in Taiwan, and is now a Research Fellow there. He was also a recipient of the Best Paper Award in the 1996 International Symposium on Software Reliability Engineering, the Best Paper Award in the simulation and test category at the 1986 IEEE/ACM Design Automation Conference (DAC), the National Science Foundation's Research Initiation Award in 1989, and the IEEE/ACM Design Automation Scholarship in 1990 and 1991.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 19, 2009 at 23:48 from IEEE Xplore. Restrictions apply.