On the Reconfiguration Algorithm for Fault-Tolerant VLSI Arrays

Jigang Wu and Srikanthan Thambipillai

Centre for High Performance Embedded Systems
Nanyang Technological University, Singapore, 639798
{asjgwu,astsrikan}@ntu.edu.sg

Abstract. In this paper, an improved algorithm is presented for the NP-complete problem of reconfiguring a two-dimensional degradable VLSI array under the row and column routing constraints. The proposed algorithm adopts the partial computing for the logical row exclusion so that the most efficient algorithm, cited in literature, is speeded up without loss of performance. In addition, a flaw in the earlier approach is also addressed. Experimental results show that our algorithm is approximately 50% faster than the above stated algorithm.

Keywords: degradable VLSI array, reconfiguration, fault-tolerance, greedy algorithm, NP-completeness

1 Introduction

The mesh-connected processor array has a regular and modular structure and allows fast implementation of most signal and image processing algorithms. With the advancement in VLSI (very large scale integration) and WSI (wafer scale integration) technologies, integrated systems for mesh-connected processor arrays can now be built on a single chip or wafer. As the density of VLSI and WSI arrays increases, the probability of the occurrence of defects in the arrays during fabrication also increases. These defects obviously affect the reliability of the whole system. Thus, fault-tolerant technologies must be employed to enhance the yield and reliability of VLSI/WSI arrays.

There are generally two methods for reconfiguration in fault-tolerant technologies, namely, the redundancy approach and the degradation approach. Various strategies to restructure a faulty physical system using the redundancy approach are described in many papers, e.g.,[1–12]. Degradation approach uses as many fault-free elements as possible to construct a target system. The final dimension is flexible and depends on the needs of the application. Usually, a maximum dimension is desirable. Literatures [13–15] have studied the problem of reconfiguring two-dimensional degradable arrays. They have shown that most problems that arise under the constraint *row and column rerouting* are NP-complete.

In this paper, we consider the reconfiguration problem of two-dimensional degradable VLSI/WSI arrays. It is defined as follows [13–15].

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2659, pp. 360–366, 2003.

Given an $m \times n$ mesh-connected host array H with the fault density ρ ($0 \leq \rho < 1$), integers r and c, find a $m' \times n'$ fault-free subarray T under the row and column rerouting scheme such that $m' \geq r$ and $n' \geq c$.

The latest work for this problem is the algorithm in [15], which is denoted as *RCRoute* in this paper. This algorithm is dominated by two sub-procedures named *Logical_Row_Exclusion* (*LRE*) and *Greedy_Column_Rerouting* (*GCR*), respectively. The time complexity of each sub-procedure is $O((1-\rho) \cdot m \cdot n)$. The routing manner in *GCR* leads to reconfiguring two neighboring fault-free elements lying in same physical row into the same logical column. But the related architecture does not support this kind of the routing. In this paper we point out this flaw and repair it. In addition, we also present a partial computing approach for the logical row exclusion. The new approach reduces the time complexity of *LRE* from $O((1-\rho) \cdot m \cdot n)$ into $O((1-\rho) \cdot n)$. Thus, we improve *RCRoute* in running time, without loss of performance. Experimental results show that the improved algorithm is approximately 50% faster than *RCRoute*.

2 Preliminaries

This section gives the definitions and the notations used in this paper.

In this paper, the original VLSI/WSI array that has been manufactured is called a host array. This host array may contain faulty elements. A degradable subarray of the host array, which contains no faulty element, is called a target array or logical array. The rows (columns) in the host array and target array are called physical rows (columns) and logical rows (columns), respectively. $row(e)$ ($col(e)$) denotes the physical row (column) index of element e. H (S) denotes the host (logical) array. R_i denotes the ith logical row. Using the same assumptions as in [13][14][15], in this paper two neighboring elements in the host array are connected by a four-port switch. All switches and links in an array are assumed to be fault-free since they have very simple structure.

In a host array, if $e(i, j+1)$ is a faulty element, then $e(i, j)$ can communicate with $e(i, j+2)$ directly and data will bypass $e(i, j+1)$. This scheme is called row bypass scheme. If $e(i, j)$ can connect directly to $e(i', j+1)$ with external switches, where $|i' - i| \leq d$, this scheme is called row rerouting scheme, d is called row compensation distance. The *column bypass scheme* and the *column rerouting scheme* can be defined similarly. By limiting the compensation distance to 1, we essentially localize the movements of reconfiguration in order to avoid complex reconfiguration algorithm. In all figures of this paper, the shaded boxes stand for faulty elements and the white ones stand for the fault-free elements.

3 Algorithms

In this section we point out a flaw in *RCRoute*[15] and repair it. Then we present our algorithm denoted *New_RCRoute* in this paper.

3.1 Updating *RCRoute*

For the *row and column rerouting* scheme, the latest efficient work is the algorithm *RCRoute*[15]. This greedy algorithm consists of two procedures namely *Row_First* and *Column_First*. *Column_First* is invoked after running *Row_First*. *Column_First* is identical to *Row_First* except that the roles of rows and columns are interchanged. Therefore, *Row_First* plays a key role in the description of *RCRoute*. For the detail description of *Row_First*, see [15].

There are two key sub-procedures in *Row_First*, denoted as *LRE* and *GCR*. The sub-procedure *LRE* selects one row to be excluded from the set that was previously selected and uses it to compensate for faulty elements in its two neighboring rows. Let M_i denote the maximum number of logical columns that pass through two consecutive rows R_i and R_{i+1}, where $i \in \{0, 1, \ldots, k-1\}$. Let $M_\gamma = \min_{0 \le i \le k-1} M_i$. *LRE* first calculates M_γ and selects the row R_γ, and then decides whether R_γ or $R_{\gamma+1}$ will be excluded. Let X denote the maximum number of logical columns that pass through $R_{\gamma-1}$ and $R_{\gamma+1}$. Let Y denote the maximum number of logical columns that pass through R_γ and $R_{\gamma+2}$. If $X > Y$, then row R_γ is selected for exclusion, otherwise, $R_{\gamma+1}$ is excluded.

GCR is used for finding a target array that contains a set of selected logical rows. It reroutes the fault-free elements to form logical columns. The successor of the fault free element u in R_i is selected from $Adj(u)$ in a left-to-right manner, where $Adj(u) = \{v : v \in R_{i+1},\ v \text{ is fault-free}, \quad |col(u) - col(v)| \le 1\}$. For the detailed description of the procedure, see [14, 15].

The sub-procedures *LRE* and *GCR* are executed iteratively until the row-based target array is found. *LRE* tests $O((1-\rho) \cdot m \cdot n)$ valid interconnections in the $m \times n$ host array in a row by row fashion, and *GCR* tests these valid interconnections column by column. Obviously, they have same time complexity $O((1-\rho) \cdot m \cdot n)$.

As can be seen from [15], the four-port switch model has a very simple architecture. But it is due to this simple construction that provides less functions that the switch model does not support reconfiguring two neighboring fault-free elements lying in same row into same logical column. Assume R_γ is to be excluded by subprocedure *LRE*. Then R_γ will be used to compensate for faulty elements in its two neighboring rows, $R_{\gamma-1}$ and $R_{\gamma+1}$, *i.e.*, each fault-free element $e(\gamma, j)$, $1 \le j \le n$, will be used to compensate $e(\gamma - 1, j)$ or $e(\gamma + 1, j)$ if they are faulty. After compensation, The subprocedure *GCR* will be executed on $\{R_1, R_2, \cdots, R_{\gamma-1}, R_{\gamma+1}, \cdots, R_m\}$ to find the current target array. However, it is possible that two neighboring fault-free elements in R_γ will be rerouted into same logical column by *GCR*. That is the flaw of *RCRoute*.

We correct *RCRoute* by adding the constraint $row(u) < row(v)$ into the definition of set $Adj(u)$, *i.e.*, let $Adj(u) = \{v : v \in R_{i+1}, v \text{ is fault-free}, |col(u) - col(v)| \le 1 \text{ and } row(u) < row(v)\}$. The constraint limits the elements in the result logical column, in the strictly increasing order of their physical row indices. This prevents the above conflict when *GCR* runs.

3.2 Partial Computing for Logical Row Exclusion

Assume $R_1, R_2, \ldots, R_k$ (initially, $k = m$) are the previously selected logical rows. In order to select one row, say R_l, to be excluded in the current iteration, *LRE* takes $O((1-\rho) \cdot k \cdot n)$ time to calculate $M_1, M_2, \ldots, M_{k-1}$ and takes $O(k)$ to select the minimum M_γ resulting in $l = \gamma$ or $\gamma + 1$ according to the compensation approach. In *Row_First*[15], *LRE* does not reuse the previous calculation of $M_i, 1 \leq i < k$. In fact, except for M_{l-2}, M_l and M_{l+1}, these M_i calculated in the previous iteration are also available in the current iteration as the compensations by R_l only affect R_{l-1} and R_{l+1}. We only need to update M_{l-2}, M_l and M_{l+1}. Hence, we simplify the sub-procedure *LRE* into *New_LRE* (Fig. 1 (left)) in order to avoid the repeat calculations of M_i except for M_{l-2}, M_l and M_{l+1}. The initial values of each M_i can be calculated out of the body of while-loop in *Row_First*[15]. Obviously, this simplified approach saves the running time of *RCRoute* but does not affect its harvest since the strategy for the selection of the row to be excluded has not been changed. By simple analysis, the running time of *LRE* in qth iteration of *Row_First* is reduced from $O((1-\rho) \cdot q \cdot n)$ to $O((1-\rho) \cdot n)$ since only M_{l-2}, M_l and M_{l+1} need to be updated, where $1 \leq q \leq m$. In the other hand, *New_LRE* still uses the same compensation strategies as described in *RCRoute*[15] after the row R_l is excluded. The time complexity of the compensation is $O((1-\rho) \cdot n)$. Hence, the time complexity of *New_LRE* is $O((1-\rho) \cdot n)$, which is far lower than $O((1-\rho) \cdot m \cdot n)$, the time complexity of *LRE*. Figure 1 shows the formal description of the improved algorithm.

```
Procedure New_ Logical_Row_Exclusion ( H, S, l, m );
begin
    /* select the minimal M_γ */
    Min := ∞ ;
    for i := 1 to m-1 do
        If M_i < Min then begin  Min := M_i;  γ := i  end;

    /* select R_l to delete and compensate */
    Greedy_Column_Rerouting (H, R_γ-1, R_γ+1, X);
    Greedy_Column_Rerouting (H, R_γ, R_γ+2, Y);
    if X < Y then  l := γ+1   /* delete row R_γ+1 */
             else  l := γ;  /* delete row R_γ */
    Row_Reroute(H, R_l-1, R_l, R_l+1)[6]; /* compensation with R_l*/

    /* update M_l-2, M_l-1, M_l and M_l+1 */
    Greedy_Column_Rerouting (H, R_l-2, R_l-1, M_l-2);
    M_l-1 := ∞ ;  /* M_l-1 will not be considered in the next iteration */
    Greedy_Column_Rerouting (H, R_l-1, R_l+1, M_l);
    Greedy_Column_Rerouting (H, R_l+1 , R_l+2, M_l+1);

end.
```

```
Procedure New_Row_First( H, m, n, r, c, row ,col );
/* Rerouting based on rows */
begin
     row_first_fail:= false;
     S={ R_1, R_2, ..., R_m };
     for i=1 to m-1 do /* calculate each M_i */
          Greedy_Column_Rerouting (H, R_k, R_i+1, M_i );
     Greedy_Column_Rerouting (H, S, n' ); /* for initial solution*/
      /* n'= maximum number of logical columns through the rows in S */
     m':=m;
     while (m' ≥ r) and (n' < c) do
     begin
          New_Logical_Row_Exclusion(H, S, γ, m' );
          Delete row R_γ from S;
          Greedy_Column_Rerouting ( H, S, n' );
          m':=m'-1;
     end;
     if (m' ≥ r) and (n' ≥ c)
     then
          begin  row:=m';  col:=n';  end
     else  row_first_fail:=true;
end;
```

Fig. 1. The formal description of the procedure *New_LRE* and *New_Row_First*

3.3 Main Algorithm and Complexity

For the description of the main algorithm, we first describe a procedure called *New_Row_First*, which is used to find a target array with maximum size based on the row. Let m' be the number of logical rows and n' be the number of logical columns of the target array. The current logical array is $S = \{R_1, R_2, \cdots, R_{m'}\}$. Initially, all rows in the host array are selected for inclusion into the target array. Thus, each logical row in S is also a physical row. The formal description of *New_Row_First* is shown in Fig. 1 (right).

In *New_Row_First*, the code before the while-loop initializes the data structures, calculates all M_i and gets the initial target array. The running time needed by all these steps is bounded by $O((1-\rho)\cdot m\cdot n)$. In the while-loop, *New_LRE* runs in $O((1-\rho)\cdot n)$ and *GCR* runs in $O((1-\rho)\cdot m\cdot n)$. Hence, the while-loop runs in $O((m-r)\cdot(1-\rho)\cdot m\cdot n)$, *i.e.*, the time complexity of *New_Row_First* is $O((m-r)\cdot(1-\rho)\cdot m\cdot n)$.

Similarly, we can describe a procedure *New_Column_First* to find a target array with maximum size based on the column. Its time complexity is $O((n-c)\cdot(1-\rho)\cdot m\cdot n)$.

The structure of the main algorithm, denoted as *New_RCRoute*, is the same as that of *RCRoute*. but the subprocedures have been improved. The largest array derived from procedures *New_Row_First* and *New_Column_First* is taken as the target array for H. The time complexity of algorithm *New_RCRoute* is $O(\max\{(m-r),(n-c)\}\cdot(1-\rho)\cdot m\cdot n)$, which is the lower bound of the complexity for row and column rerouting[15].

4 Experimental Results

We report our experimental results in this section. We have implemented the algorithm *RCRoute* (corrected version) and the improved algorithm *New_RCRoute* in C on a personal computer—Intel Pentium-III 500 MHZ. The implementations of the two algorithms are modified accordingly to find maximal target arrays and maximal square target arrays. In our experiments, *harvest* and *degradation*, formulated in [13],[14] and [15], are calculated for each target array. In order to make a fair comparison between *New_RCRoute* and *RCRoute*, we keep the same assumptions as in [14],[15], *i.e.*, the faults in random host arrays were generated by a uniform random generator; The fault size in host array is from 0.1% to 10% for the experiments. Both algorithms are tested with the same random input instances. The running time and the size of each target array obtained by *New_RCRoute* is compared with the corresponding array obtained by *RCRoute*[15]. Table 1 summarizes the experimental results for the random host arrays with different sizes. The improvement in running time is calculated by

$$(1-\frac{running\_time\_of\_New\_RCRoute}{running\_time\_of\_RCRoute})\times 100\%.$$

The calculations required to arrive at solutions for maximal target array encompass the solution for maximal square target array. Hence, without loss of

Table 1. The comparison of running time for 20 random instances

			Performance		Running Time		
Host Array	Fault Size	(%)	Harvest (%)	Degrad. (%)	RCRoute (s)	New_RCRoute (s)	Improve (%)
64×64	4	0.1	98.53	1.56	0.296	0.150	49.3
64×64	40	1.0	96.29	4.65	0.296	0.149	49.7
64×64	409	10.0	84.52	23.91	0.286	0.143	50.0
128×128	16	0.1	98.85	1.24	2.313	1.191	48.5
128×128	163	1.0	97.15	3.82	2.307	1.189	48.5
128×128	1638	10.0	84.61	23.84	2.213	1.152	47.9
256×256	65	0.1	99.24	0.86	18.316	9.441	48.5
256×256	655	1.0	97.56	3.41	18.160	9.369	48.4
256×256	6553	10.0	84.37	24.07	17.340	9.067	47.7
512×512	262	0.1	99.41	0.69	147.291	76.148	48.3
512×512	2621	1.0	97.92	3.06	145.004	75.081	48.2
512×512	26214	10.0	84.89	23.60	137.259	72.033	47.5

generality, we collect the running time only in the case of finding maximal target array. Table 1 shows the running time comparisons for the maximal target array. For each random instance, the running time required by *New_RCRoute* is significantly less than that required by *RCRoute*. For example, for the host array of size 256×256 with 655 fault elements, the running time for *New_RCRoute* is 9.369 seconds, while it is 18.160 seconds for *RCRoute*. The improvement in running time is 48.4%, which is nearly equal to 50%. Except for small size instances such as 64×64, increase in fault density in the host array leads to less improvement in running time as more backtracking is needed in routing. For instance, for the 512×512 host array with the 26214 fault elements, the improvement in running time is 47.5%, which is a little less than 48.3%, the improvement running time for the 512×512 host array with the 262 fault elements.

We can conclude from the analysis above that our algorithm *New_RCRoute* reduced the running time by approximately 50%, especially, for low density of faults in the host arrays, without loss of harvest.

5 Conclusions

We have presented a degradation approach for the reconfiguration in VLSI/WSI arrays under the rerouting constraint *row and column rerouting*. We have proposed a new strategy for the row selection. The new strategy binds well for high-speed realizations. For different sized host arrays, our algorithm has maintained harvest and degradation while reducing the running time by approximately 50%. Method to overcome the flaw in one of the recent contributions in this area was also made. The improved algorithms have been implemented and experimental results have been collected. These running time results clearly reflect the underlying characteristics of the improved algorithm.

Acknowledgment. We wish to express our sincere thanks to the anonymous referees for their constructive suggestions. We are grateful to Ms Chandni R. Patel for pointing out oversights in an earlier draft of this paper.

References

1. T. E. Mangir and A. Avizienis, "Fault-tolerant Design for VLSI: Effect of Interconnection Requirements on Yield Improvement of VLSI design", *IEEE Trans. on Computers*, vol. 31, no. 7, pp. 609–615, July 1982.
2. J.W. Greene and A.E. Gamal, "Configuration of VLSI Array in the Presence of Defects", *J. ACM*, vol. 31, no. 4, pp. 694–717, Oct. 1984.
3. T. Leighton and A. E. Gamal. "Wafer-scal Integration of Systoric Arrays", *IEEE Trans. on Computer*, vol. 34, no. 5, pp. 448–461, May 1985.
4. C.W.H Lam, H.F. Li and R. Jakakumar, "A Study of Two Approaches for Reconfiguring Fault-tolerant Systoric Array", *IEEE Trans. on Computers*, vol. 38, no. 6, pp. 833–844, June 1989.
5. I. Koren and A.D. Singh, "Fault Tolerance in VLSI Circuits", *Computer*, vol. 23, no. 7, pp. 73–83, July 1990.
6. Y.Y. Chen, S.J. Upadhyaya and C. H. Cheng, "A Comprehensive Reconfiguration Scheme for Fault-tolerant VLSI/WSI Array Processors", *IEEE Trans. on Computers*, vol. 46, no. 12, pp. 1363–1371, Dec. 1997.
7. T. Horita and I. Takanami, "Fault-tolerant Processor Arrays Based on the 1.5-track Switches with Flexible Spare Distributions", *IEEE Trans. on Computers*, vol. 49, no. 6, pp. 542–552, June 2000.
8. S.Y. Kuo and W.K. Fuchs, "Efficient Spare Allocation for Reconfigurable Arrays", *IEEE Design and Test*, vol. 4, no. 7, pp. 24–31, Feb. 1987.
9. C.L. Wey and F. Lombardi, "On the Repair of Redundant RAM's", *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 6, no. 2, pp. 222–231, Mar. 1987.
10. M.G. Sami and R. Stefabelli. "Reconfigurable Architectures for VLSI Processing Arrays", *Proc. IEEE*, vol. 74, no. 5, pp. 712–722, May 1986.
11. R. Negrini, M.G. Sami and R. Stefanelli, *Fault tolerance through reconfiguration in VLSI and WSI arrays*. The MIT Press, 1989
12. F. Distante, M.G. Sami and R. Stefanelli, "Harvesting through array partitioning: a solution to achieve defect tolerance Defect and Fault Tolerance in VLSI Systems", Proc. 1997 IEEE International Symp. *Defect and Fault Tolerance in VLSI Systems*, Paris, pp. 261–269, 1997
13. S.Y. Kuo and I.Y. Chen, "Efficient Reconfiguration Algorithms for Degradable VLSI/WSI Arrays." *IEEE Trans. on Computer-Aided Design*, vol. 11, no. 10, pp. 1289–1300, Oct. 1992.
14. C.P. Low and H.W. Leong, "On the Reconfiguration of Degradable VLSI/WSI Arrays", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 10, pp. 1213–1221, Oct. 1997.
15. C.P. Low, "An Efficient Reconfiguration Algorithm for Degradable VLSI/WSI Arrays", *IEEE Trans. on Computers*, vol. 49, no. 6, pp. 553–559, June 2000.