Shortest partial path first algorithm for reconfigurable processor array with faults

Jigang Wu
School of Computer Science and Technology
Guangdong University of Technology
Guangzhou , China
Email: asjgwu@gmail.com

Ningjing Liu
School of Computer Science and Software Engineering
Tianjin Polytechnic University
Tianjin , China
Email: ningjing_liu@126.com

Siew-Kei Lam
School of Computer Engineering
Nanyang Technological University
Singapore
Email: siewkei_lam@pmail.ntu.edu.sg

Guiyuan Jiang
School of Computer Engineering
Nanyang Technological University
Singapore
Email: gyjiang@ntu.edu.sg

***Abstract*—With the development of very large scale integration (VLSI) technologies, a large numbers of the processing elements (PEs) can be integrated on a single chip. The increasing density of VLSI arrays leads to the increase of the probability of PEs malfunction during normal operation of the system. Fault-tolerant techniques become a meaningful research topic to obtain fault-free logical array, in order to guarantee the system stability and reliability. In this paper, we propose a fast algorithm to reconfigure logical arrays based on the strategy of shortest partial path first extension. The algorithm selects the partial path with the minimum number of long interconnect to extend, as the partial path is often the part of the optimal logical column that is to be constructed. Thus, the proposed algorithm can rapidly generate the optimal logical column, as it is able to avoid the search for all possible paths related to the fault-free PEs, without loss of harvest. Experimental results show that the state-of-the-art can be improved up to by 38.9% in 128×128 host array with 20% faults, in terms of running time.**

***Index Terms*—Reconfiguration; VLSI array; Fault tolerance; Algorithm**

I. Introduction

With the rapid development of personal computing devices systems, designers were forced to decrease its power consumption and improve the efficiency of time as much as possible in high-performance system. In recent years, very large scale integration (VLSI) technology integrates huge number of processing elements (PEs) on a single chip in a tightly coupled fashion to process massive amount of information in parallel. However, as the density of the VLSI arrays increases, the probability of the faults occur in the arrays during fabrication also increases. There are different kinds of these faults on PEs, one is called hard faults that are caused by physical damage and its uneven lifetime, and the other is called soft faults caused by current overheat, overload, or occupation by other applications. The faulty PEs destroy the regular structure of the communication networks, they reduce the processing capabilities of the multiprocessor array, and thus the system will be affected. VLSI processor arrays must provide fault-tolerant techniques to reconfigure the system for improving the stability and dependability [1]. The fast fault-tolerant techniques for reconfiguring has been the necessary safeguards for system, especially this kind of processor arrays used in the aerospace craft.

In general, there are mainly two methods employed in reconfiguration techniques, namely, redundancy approach and degradation approach. In redundancy approach [2][3], the system will provide some spare PEs and these PEs are utilized to replace faulty PEs. The main characteristic of this approach is that the size of the arrays is fixed. However, if the spare PEs cannot replace all the fault of PEs, the method have to be gave up. In contrast with the redundancy approach, the degradation approach [4][5] doesnt have spare PEs, all PEs on the chip are treated in an uniform way in the degradation approach. This approach tries to utilize as many as possible fault-free PEs to construct a logical subarray.

This paper focuses on the degradation strategy. The previous literatures have shown that most reconfiguration problems under certain constraints are NP-complete [4]. Due to the complexity of the reconfiguration problems, many methods have been proposed to increase the utilization rate of fault-free PEs using different tracks [6][7], switches [8] and rerouting schemes [9]. A traditional degradation approach based on greedy strategy [10], named GCR (greed column rerouting), can produce a maximum target array on the selected rows. It is well known that the degradable arrays are constructed with minimizing the total interconnection length will provide for less routing cost, less capacitance, and less dynamic power dissipation, while improving the overall reliability [11]. Thus, a dynamic programming algorithm (DPA) is proposed in [12] to reduce the number of long-interconnects for decreasing power dissipation. Based on the DPA, a divide-and-conquer algorithm was proposed in [13], resulting in significant improvements in terms of the total interconnection length. DPA

needs to visit all fault-free PEs in the local area and calculate the path information for every fault-free PEs, that will lead to a relatively slow reconfiguration.

It is well known that the fast reconfiguration is very critical for the system reliability when the faults occur, especially for the systems utilized in aircraft and satellite. In view of the weakness of DPA, we propose a new algorithm to accelerate it. The proposed algorithm, named as SPPA in this paper, constructs the optimal logical column by computing the shortest path in the local area. The shorted path is expended from the current partial shortest path, rather than visiting all fault-free PEs in the area as DPA did. So the algorithm SPPA can significantly accelerate the construction of the shortest path for most cases. The harvest of SPPA is kept same as that of DPA, because SPPA is still able to find an optimum local column.

II. PRELIMINARIES

We introduce some definitions and notations to be used in the following sections. A host array (or physical array) H is the original processor array was obtained after manufacture, the faulty PEs will random distribute among physical array. A sub-array of H after reconfiguration, is called target array (or logical array) T, which doesnt contain faulty PE. Assume ρ indicates the fault density of the host array, where $0 < \rho < 1$, i.e., there are $\rho \cdot m \cdot n$ faulty PEs in an $m \times n$ host array. The rows (columns) in host array are called physical rows (columns). The rows (columns) in logical array are called logical rows (columns). An $r \times t$ target array is the logical array with r logical rows and t logical columns. In this paper, all the assumptions in architecture are the same as literature in [12][13].

Figure 1 shows an example for the fault-tolerant architecture of a 4×4 host array. In the array, each square stands for a PE, while each circle represents a reconfigurable switch. Neighboring PEs are connected to each other by a four-port switch and switching functions are provided by a single-track switch. All switches and links in the array are assumed to be fault-free. This assumption can make the interconnection between PEs relatively simple.

We can change the connection relationship among PEs by the switch states. Every PE can connect eight PEs in up, down, left, right four directions. 'Row bypass and column rerouting' is one type of rerouting constraints in many reconfiguration algorithms. As shown in Fig.1, e(i, j) indicates the PE located at the position (i, j) in the host array. If e$(i, j + 1)$ faults, the data will bypass e$(i, j + 1)$ through the internal bypass, then e(i, j) can communicate with e$(i, j + 2)$ directly. This connection scheme is called row bypass schemes.

The column rerouting scheme change the states of relative switches to form a logical column, and the fault-free PEs located in different physical rows can be connected together to form a logical column. In the column rerouting schemes, e(i, j) can connect to e$(i+1, j')$ through two external switches, where $|j - j'| \leqslant d$, and d is named compensation distance.

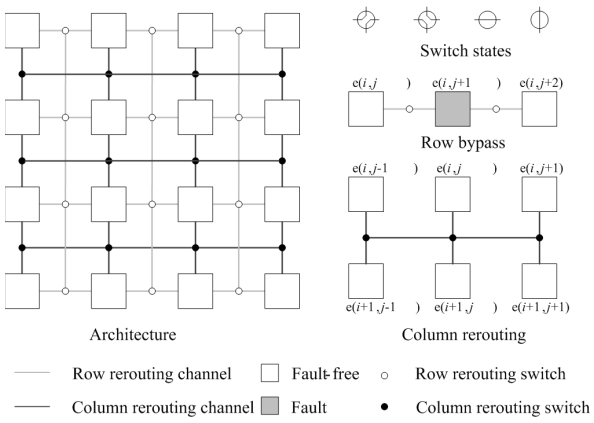

Figure 1. Switch functions and rerouting manners on a 4×4 mesh linked by four-port switch.

According to the previous literature, we also limited d to 1. For more detail, refer to [12][13].

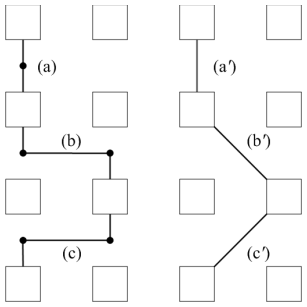

Figure 2. The three different interconnects for column rerouting scheme

There are six possible types of link-ways for a target array. They can be classified into two classes based on the number of the switches used. One is called short interconnect, which uses one switch to connect neighboring PEs, and the other is called the long interconnect, which uses two switches. Figure 2 shows three different link-ways, due to the constraint of 'row bypass and column rerouting' utilized in this paper. For more details on the fault-tolerant architecture, see [14]. In Figure 2, (a) is short interconnects, while the others are long interconnects. For the convenience of description, this paper uses a straight line (a') on behalf of the short interconnect (a), and oblique line (b'), (c') represent the long interconnect (b), (c) respectively.

Assume u indicates a PE in host array, where row(u) (col(u)) indicates the physical row (column) index of the PE u. The lower adjacent set $Adj^+(u)$ and the upper adjacent set $Adj^-(u)$ of each fault-free PE u in the row R_i is defined as follows:

1) $Adj^+(u)$=$\{v : v \in R_{i+1}$, v is fault-free and $|col(u) - col(v)| \leqslant 1\}$ $1 \leqslant i \leqslant m - 1$.
2) $Adj^-(u)$=$\{v : v \in R_{i-1}$, v is fault-free and $|col(u) - col(v)| \leqslant 1\}$ $2 \leqslant i \leqslant m$.

For arbitrary $v \in Adj^+(u)/Adj^-(u)$, v is called the lower (upper) adjacent of u. Every adjacent sets $Adj^+(u)/Adj^-(u)$ have three elements, called lower (upper) left adjacent, lower (upper) middle adjacent, and lower (upper) right adjacent, respectively.

Let $R_1, R_2, \cdots, R_m$ be the rows of the given host array. Assume B_l, B_r are two logical columns passing through each physical row of the mn host array, and they are known as the left and right boundaries of the area, respectively, where B_l is on the left of B_r. In this paper, A[B_l, B_r] indicates the area that consists of the PEs bounded by B_l and B_r (including B_l and B_r). A[B_l, B_r) indicates the same area as above including B_l but not excluding B_r. Suppose that B_l is the i-th logical column generated by GCR [15] in the left-to-right manner and B_r is the $(k-i+1)$-th logical column generated by GCR in the right-to-left manner, where k is the total number of logical columns. As pointed out in the literature [12], the boundaries B_l and B_r are not independent, i.e, there must exist at least one intersection between B_l and B_r. The area A[B_l, B_r] is the largest area available to generate the i-th local optimal logical column.

III. THE PROPOSED ALGORITHM

The algorithm DPA calculates the shortest path by visiting all fault-free PEs in local area, and they may get more than one local optimum column. In order to improve the time efficiency of reconfiguration, we propose a shortest partial path first algorithm, denoted as SPPA, that constructs a local optimal column only by visiting parts of the fault-free PEs in the local area. The algorithm was proposed based on backtracking method, together with assigning the priority to the fault-free PEs in the reconfiguration process. Unlike DPA which enumerates the all shortest paths, the proposed SPPA terminates when one shortest path is obtained, in order to save reconfiguration time.

Assume PE u is the current node prepared for expanding, when we construct a local optimal logical column in the area A[B_l, B_r] by the algorithm SPPA. $w(u)$ indicates the weight of u, i.e., the path length from the start point to the PE u. The weight of fault PE u' is assigned to ∞, i.e., $w(u') = \infty$. We set $w(u)$ to $Curw$, where $Curw$ indicates that the current weight of u. It is noteworthy that there may exist several nodes with the same weight of $Curw$.

We now briefly describe our algorithm SPPA. Initially, the algorithm visits the nodes with the $Curw$ value of 0 according to the priority of the PEs. The algorithm determines whether the search reaches the end of the logical column. If the optimal logical column is not completed based on the current weight of $Curw$, the procedure will continue its searching with the updated weight $Curw+1$, according to their priority. The proposed method repeats until getting an optimal logical column.

It is worthwhile to pointed out that, the partial path with the current weight of $Curw$ in a logical column, is the shortest partial path.

In the process of visiting the nodes, we put PE u as the current node that will be extended. It is not difficult to understand that the nodes with weight of $Curw + 1$ are obtained by visiting the upper right (left) node of PE u with the weight of $Curw$. When the upper right (left) node is the PE u, we will continue the visiting to the upper middle node, in order to obtain the nodes with the weight of $Curw + 1$.

Generally, there are some nodes with the same weight of $Curw$ in the extension of the shortest partial path. Then, we need to considering the priority of the nodes. We define the priority for these nodes as follows, in order to select someone to extend from $Curw$ to $Curw + 1$.

Assume that the shortest partial path is extended from the bottom row R_m to the top row R_1.

1) The node of the highest priority is the one that is closest to the top row R_1.
2) If several nodes have same weight of $Curw$ and they are in the same row, the node of the highest priority is the rightmost one.

When we extend the nodes from $Curw$ to $Curw + 1$, the upper right node for every node with weight of $Curw$ has a higher priority than the upper left node. This process may provide a larger area for another logical column. If there is a shortest partial path which is not extend, the proposed algorithm will not update the value of $Curw$, this is to guarantee the check for all shortest partial paths.

The proposed algorithm constructs an optimal logical column in local area A[B_l, B_r] and the weight can be calculated in the manner of bottom-to-up way. In order to conveniently understand the process of SPPA algorithm, Figure 3 shows an example to illustrate the extension of the shortest partial path on a 5×5 host array. The top row R_1 is the first line in the main array. PE01 is the intersection of B_l and B_r, i.e. the start node of the partial path. The routing process terminates when it reaches the top row R_1.

As the PEs with weight of $Curw+1$ are obtained by visiting the PEs with the weight of $Curw$, we need to backtrack to the PE with a weight of $Curw$ to extend. In the routing process, if the PE has been visited, we will not to visit it again. For the backtracked PEs, if the upper right (left) nodes was visited, we will not to backtrack it again.

As shown in Figure 3, the algorithm visits PE01 in step 1 and then directly moves to PE02. After that, the PEs with the weight of 0 has been extended. In step 2, we extend the PEs according to the priority with the weight of 0. PE02 can be extended to PE03, because PE03 does not reach the top row R_1, then we extend PE01 to obtain the PE with the weight of 1, this process can obtain PE04 and PE05. It is not difficult to find the PEs with the weight of 1 do not reach the top row R_1. In step 3, we extend the PEs according to the priority with weight of 1, to obtain the PEs with weight of 2. The node of the highest priority is PE03, and PE06 is obtained, but PE06 does not reach the top row R_1. Then we visit PE04. Because PE04 cant be extend, we obtain PE07, PE08, PE09 by sequential extension from PE05. As PE09 reach the top row R_1, the algorithm terminates.

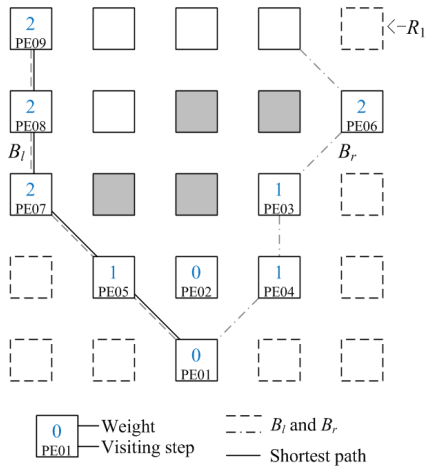

Figure 3. Example for extension of shortest partial path

In the process of constructing a shortest path, the size of $Curw$ was increased one by one and all nodes will be visited when its weight less than $Curw$. If the nodes with the weight of $Curw$ cant get the optimal solution, then the procedure will visit the PEs with the weight of $Curw+1$. When we obtain an optimal solution in the weight of $Curw$, the all nodes with the weight less then $Curw$ must have been visited. Thus, we can prove by the reduction to absurdity that the SPPA algorithm can construct an optimal path.

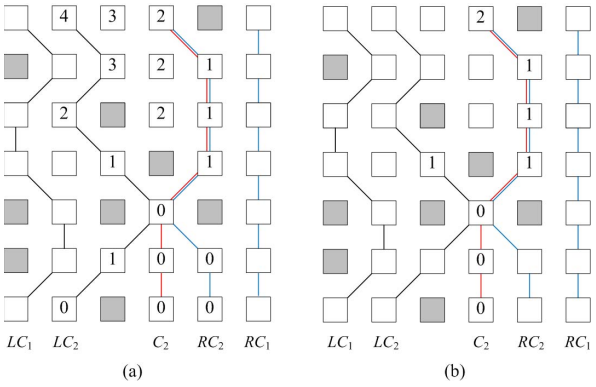

Figure 4. (a) Local optimal column by DPA, and (b) SPPA algorithm

Figure 4 shows an example to compare the algorithms DPA and SPPA in constructing the local optimal column. $LC_1, LC2_1, \cdots, LC_m$ are the logical columns generated by GCR in left-to-right manner, $RC_1, RC_2, \cdots, RC_m$ are the logical columns generated by GCR in right-to-left manner, and C_2 is the local optimal column constructed in the local area between LC_2 and RC_2, i.e. LC_2, RC_2 is the boundary B_l, B_r, respectively. The numbers indicate the weight of the PE. In the area A$[LC_2, RC_2]$, it is clear that the DPA visits all 18 fault-free PEs, while the SPPA only needs to visit 8 PEs.

In the $n \times n$ host array, assume N is the total number of fault-free PEs on the area A$[B_l, B_r]$, and m is the number of visited fault-free PEs by SPPA to generate a local optimum column on A$[B_l, B_r]$. The worst time complexity of SPPA is same as that of DPA, i.e., $O(N)$. But SPPA works on $O(m)$, and m is very close to n for most cases, although SPPA has to backtrack for the shortest path.

IV. EXPERIMENTAL RESULTS AND ANALYSIS

As mentioned in section 1, DPA has been employed by many significant works, such as ALG01 and ALG02, to refine the interconnect length of the target arrays. In this section, we compare the proposed algorithms to DPA. In order to make a fair comparisons, we keep the same assumptions as in [12][13]. The algorithms are simulated in C++ on a personal computer with Intel(R) Core(TM) i5-3470 CPU 3.20GHz and 4G RAM, and they are compared on the same random input instances. The data is collected on different sized host array with various fault densities.

The following notations are utilized for performance evaluation of the algorithms.

- $tot.len$: the total number of long interconnects.
- $tot.vis$: the total number of visited PEs.
- $imp.vis$: the improvement of the SPPA algorithm over DPA in terms of tot.vis, calculated by:

$$\left(1 - \frac{tot.vis \text{ of SPPA}}{tot.vis \text{ of DPA}}\right) \times 100\%$$

- $tot.t$: the time of constructing a logical array.
- $imp.t$: the improvement of the SPPA algorithm over DPA in terms of tot.t, calculated by:

$$\left(1 - \frac{tot.t \text{ of SPPA}}{tot.t \text{ of DPA}}\right) \times 100\%$$

Figure 5 shows the performance comparisons between DPA and SPPA for all logical columns in a host array. The data are collected on the 256×256 host arrays. The subfigures (a), (b), (c) show the number of the visited PEs for constructing each logical column. The host array is with the fault rate of $1\%, 10\%, 20\%$, respectively. The fault PEs is distributed in the array by a uniform way. The subfigures (a), (b), (c) show the time efficiency that corresponding to (a), (b), (c), respectively. It is evident that the proposed algorithm greatly reduces the number of the visited fault-free PEs for constructing a local optimal column in local area. Typically, the reconfiguration time has been reduced with the decrease of the number of visited fault-free PEs in constructing a local optimal column. For the first 20 logical columns, the improvement is clear as the corresponding local area to form the logical columns is relatively larger. For the other logical columns, SPPA and DPA are comparable in terms of the reconfiguration time.

Table 1 shows the performance comparisons of algorithms DPA and SPPA for random faults of uniform distribution, averaged over 20 random instances. The data are collected from the host array with different sizes from 64×64 to 512×512. The total number of visited PE (tot.vis) is successfully reduced, resulting in the significant improvement in terms of running time. For example, on 128×128 arrays with 10% faults, the

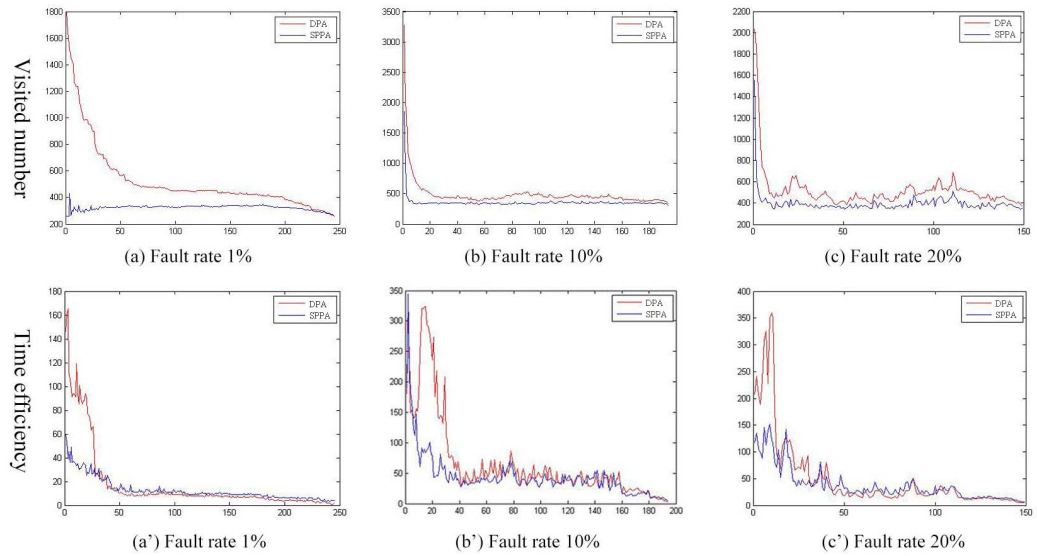

Figure 5. Performance Comparison

improvement of SPP over DPA is up to 41.7% in terms of $tot.vis$, and the corresponding improvement is 35.3% in terms of tot.t. This shows the superiority of the proposed SPPA in accelerating reconfiguration of the VLSI arrays.

The number of the long interconnects of the target arrays has a little bit increase by SPPA. For example, on the 128×128 host array with 10% faults, the target array produced by SPPA has 4633 long interconnects, that is more 25 than 4608, i.e., about 0.5% acceptable increase. This is because DPA take times to select the rightmost shortest path in order to enlarge the local area for the next logical columns, while SPPA only extends the current shortest partial path. Therefore, SPPA saves time but may reduce the size of the local area for the next logical columns, resulting in a little bit increase in the number of the long interconnects. But the target arrays produced by both algorithms have same size, and thus the harvest of SPPA is kept same as that of DPA. Briefly, the proposed SPPA significantly reduce the reconfiguration time, with an acceptable increase in the number of long interconnects, but without loss of harvest.

Table I
THE PERFORMANCE COMPARISON OF ALGORITHMS DPA AND SPPA FOR RANDOM FAULTS OF UNIFORM DISTRIBUTION, AVERAGED OVER 20 RANDOM INSTANCES

Host array		Target array	Performance							
Size	Fault	Size	$tot.len$		$tot.vis$		$imp.vis$	$tot.t$(ms)		$imp.t$
$m \times n$	(%)	$r \times t$	DPA	SPPA	DPA	SPPA	(%)	DPA	SPPA	(%)
64×64	1	64×61	488	492	6649	3861	41.9	127	118	6.8
	5	64×55	987	995	6589	3621	45.0	198	166	16.1
	10	64×48	1056	1068	5821	3379	41.9	218	186	14.6
	20	64×33	865	872	4894	2922	40.2	209	188	10.3
128×128	1	128×123	3082	3122	29759	15704	47.2	1121	923	17.6
	5	128×110	4882	4904	26055	14889	42.8	1531	1142	25.4
	10	128×96	4608	4633	23912	13950	41.7	1888	1222	35.3
	20	128×68	3708	3783	19645	12191	37.9	1934	1181	38.9
265×265	1	256×247	17991	18350	114858	63668	44.5	4518	3769	16.5
	5	256×221	23052	23375	104645	60696	42.0	6961	4768	31.5
	10	256×194	20240	20462	93402	56832	39.2	6510	4937	24.1
	20	256×136	15536	16310	77734	49543	36.3	8579	6794	20.7
512×512	1	512×496	88374	88519	454986	256436	43.6	34244	28998	15.3
	5	512×447	98374	98971	407001	243727	40.1	37665	28926	23.2
	10	512×390	84718	86064	372759	229530	38.8	34052	23916	29.7
	20	512×293	63479	64054	312820	205389	34.3	42104	34671	17.6

V. Conclusions

In this paper, we have proposed an efficient algorithm to accelerating reconfiguration for the degradable VLSI arrays with fault PEs. The proposed algorithm selects the shortest partial path to extend to an optimum logical column. Compared with the existing algorithms, the proposed algorithm successfully avoids the search for all possible shortest paths related to the fault-free PEs, resulting in the significant improvement over the state-of-the-art in terms of reconfiguration time, without loss of harvest. Simulation results show that, the improvement is from about 35% to 45% in the number of the visited PEs for all cases considered in this paper, and the state-of-the-art is accelerated up to by 38% in the best case.

Acknowledgment

This work was supported by the Specialized Research Fund for the Doctoral Program of Higher Education of China under Grant No. 20131201110002, and the Key Laboratory of Computer Architecture Opening Topic Fund Subsidization under Grant No. CARCH201303, and the Science and technology project of Guangdong province under Grant No. 2015B010129014.

References

[1] C. PoJen, L. Yao. An Efficient Reconfiguration Scheme for Fault-Tolerant Meshes, Information Sciences, 2005, 172, 309-333.

[2] I. Takanami, T. Horita. A Built-in Circuit for Self-Repairing Mesh-Connected Processor Arrays by Direct Spare Replacement, Proc. of IEEE 18th Pacific Rim International Symposium on Dependable Computing (PRDC). 2012, 96-104.

[3] L. Zhang. Fault-Tolerant Meshes with Small Degree. IEEE Transactions on Computers, 2002, 51(5), 553-560.

[4] S. Y. Kuo, I. Y. Chen. Efficient Reconfiguration Algorithms for Degradable VLSI/WSI Arrays, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 1992, 11(10), 1289-1300.

[5] Y. Fukushima, M. Fukushi, S. Horiguchi. An Improved Reconfiguration Method for Degradable Processor Arrays Using Genetic Algorithm, Proc. of IEEE 21st International Symposium on Defect and Fault Tolerance in VLSI Systems. IEEE, 2006, 353-361.

[6] I. Takanami. Self-reconfiguring of 1.5-track-switch Mesh Arrays with Spares on One Row and One Column by Simple Built-in Circuit, IEEE Transactions on Information and Systems, 2004, 87(10), 2318-2328.

[7] J. Wu, T. Srikanthan, X. Han. Preprocessing and Partial Rerouting Techniques for Accelerating Reconfiguration of Degradable VLSI Arrays, IEEE Transactions on Very Large Scale Integration(VLSI) Systems, 2010, 18(2), 315-319.

[8] J. Wu, T. Srikanthan, H. Schroder. Efficient Reconfigurable Techniques for VLSI Arrays with 6-port Switches, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2005, 13(8), 976-979.

[9] Y. Fukushima, M. Fukushi, S. Horiguchi. An Improved Reconfiguration Method for Degradable Processor Arrays Using Genetic Algorithm. Proc. of IEEE 21st International Symposium on Defect and Fault Tolerance in VLSI Systems. IEEE, 2006, 353-361.

[10] C. P. Low, An Efficient Reconfiguration Algorithm for Degradable VLSI/WSI Arrays, IEEE Trans. Computers, vol. 49, no. 6, pp. 553-559, June 2000.

[11] Y. Shin, T. Sakurai. Power Distribution Analysis of VLSI Interconnects Using Model Order Reduction, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2002, 21(6), 739-745.

[12] J. Wu, T. Srikanthan. Reconfiguration Algorithms for Power Efficient VLSI Subarrays with 4-Port Switches. IEEE Transactions on Computers, 2006, 55(3), 243-253.

[13] J. Wu, T. Srikanthan, G. Jiang, et al. Constructing Sub-Arrays with Short Interconnects from Degradable VLSI Arrays. IEEE Transactions on Parallel and Distributed Systems, 2014, 25(4), 929-938.

[14] M. Fukushi, S. Horiguchi. A Self-Reconfigurable Hardware Architecture for Mesh Arrays Using Single/Double Vertical Track Switches, IEEE Transactions on Instrumentation and Measurement, 2004,53(2),357-367.

[15] G. Jiang, J. Wu, Y. Ha, et al. Reconfiguring Three-Dimensional Processor Arrays for Fault-Tolerance: Hardness and Heuristic Algorithms. IEEE Transactions on Computers, 2015, 64(10), 2926-2939.