Journal of Circuits, Systems, and Computers
Vol. 28, No. 7 (2019) 1950111 (24 pages)

DOI: 10.1142/S0218126619501111

World Scientific
www.worldscientific.com

Algorithms for Reconfiguring NoC-Based Fault-Tolerant Multiprocessor Arrays*

Jigang Wu† and Yalan Wu‡

School of Computer Science and Technology,
Guangdong University of Technology,
Guangzhou, Guangdong 510006, P. R. China
†*asjgwucn@outlook.com*
‡*wuyalan93@outlook.com; wuyalan93@126.com*

Guiyuan Jiang§ and Siew Kei Lam¶

School of Computer Science and Engineering,
Nanyang Technological University,
Singapore 639798, Singapore
§*gyjiang@ntu.edu.sg*
¶*siekei_lam@pmail.ntu.edu.sg*

Received 3 March 2018
Accepted 17 July 2018
Published 23 August 2018

This paper investigates the techniques to construct high-quality target processor array (fault-free logical subarray) from a physical array with faulty processing elements (PEs), where a fixed number of spare PEs are pre-integrated that can be used to replace the faulty ones when necessary. A reconfiguration algorithm is successfully developed based on our proposed novel shifting operations that can efficiently select proper spare PEs to replace the faulty ones. Then, the initial target array is further refined by a carefully designed tabu search algorithm. We also consider the problem of constructing a fault-free subarray with given size, instead of the original size, which is often required in energy-efficient MPSoC design. We propose two efficient heuristic algorithms to construct target arrays of given sizes leveraging a sliding window on the physical array. Simulation results show that the improvements of the proposed algorithms over the state of the art are 19% and 16%, in terms of congestion factor and distance factor, respectively, for the case that all faulty PEs can be replaced using the spare ones. For the case of finding 64×64 target array on 128×128 host array, the proposed heuristic algorithm saves the running time up to 99% while the solution quality keeps nearly unchanged, in comparison with the baseline algorithms.

Keywords: Network on chip; multiprocessor array; topology reconfiguration; fault tolerance.

*This paper was recommended by Regional Editor Piero Malcovati.
‡Corresponding author.

1. Introduction

Network on chip (NoC) is a communication subsystem on an integrated circuit, which applies networking theory and methods to on-chip communication and brings notable improvements over conventional bus and crossbar interconnections. NoC is generally regarded as one of the most promising interconnect solutions for giga-scale integrated circuits, such as many-core processors,[1,2] in which the topology determines the ideal performance of NoC whereas the routing algorithm and the flow control mechanism determine how much of this potential is realized. In recent years, many useful network topologies, such as the combined Gaussian network topology,[3] are proposed to improve the ideal performance of NoC. Mesh-connected multiprocessor array is readily employed for high-speed implementation of most signal and image processing algorithms, which is one of the most common networks for NoC due to its simplicity, scalability, structural regularity and ease of implementation.[4]

With the advance in very large-scale integration (VLSI) techniques, a single chip can be integrated with tens to hundreds of processing elements (PEs) to process massive amounts of information in parallel.[5] However, as the density of VLSI arrays increases, the probability of occurrence of the faults in the arrays during fabrication also increases. In addition, some PEs are temporally unavailable for the current application caused by their "soft faults," i.e., overheating, overload or being employed by other applications.[4] Thus, it is hard to guarantee all PEs in the NoC to be fault-free throughout their working lifetime. Moreover, without considering fault tolerance during the architecture design, the yield of many-core system may decrease to as low as 10–20%.[6] Therefore, circuit reliability becomes one of the major challenges, and also fault tolerance becomes an essential inherent characteristic of every chip design.

Generally, two types of fault tolerance architectures, namely router-based architecture[7–9] and switch-based architecture,[10–12] are extensively investigated for mesh-connected multiprocessor arrays. Compared with switch-based architecture, router-based architecture is superior in the design of reconfiguration algorithm and utilization of fault-free PEs, but it has disadvantages in hardware cost and power consumption. In addition, it is verified that providing defect tolerance capabilities on-chip via redundant PEs is more efficient than incorporating redundant circuits at microarchitecture level.[13–15] Therefore, this paper focuses on developing efficient reconfiguration algorithms for router-based architecture with redundant PEs to reduce the overall energy consumption of system.

In router-based architecture, the fault-tolerant reconfiguration algorithm for NoC-based multiprocessor arrays reorganizes the fault-free PEs to form a logical array, instead of really changing the physical interconnection among PEs. In recent years, two different dimensions, i.e., 2D[16,17] and 3D,[18,19] for mesh-connected NoC are widely investigated. For example, for the NoC-based 3D mesh, a low-overhead fault-tolerant routing scheme is presented.[18] And for the NoC-based 2D mesh, a novel topology reconfiguration algorithm is proposed in Ref. 16 which aims at higher reconfiguration

rate. In this paper, we investigate the reconfiguration algorithm for the 2D mesh-connected NoC. As we know, effective fault-tolerant techniques are essential to improve the yield of complex integrated circuits. Therefore, researchers have proposed many different optimization schemes for NoC-based fault-tolerant issues. The work in Ref. 13 presents an effective fault tolerance scheme on mesh-based NoC to solve the problems caused by faulty routers or broken links. An algorithm[20] is presented based on maximum flow for the reconfiguration problem, which optimizes the use of spare PEs with minimal impact on area, throughput and delay, and thus it significantly improves the repair rate of faulty PEs. And an enhanced approach using a minimum-cost maximum-flow algorithm is further presented in Ref. 20 by considering various PEs in practical applications. It is noteworthy that a customized simulated annealing algorithm (denoted as RSSA) is presented in the work[21] to solve the topology reconfiguration problem for NoC-based multiprocessor arrays with redundant PEs. The algorithm refines the initial topology generated by row rippling column stealing scheme (denoted as RS). However, a considerable number of long interconnection paths are generated due to column stealing technique utilized in the algorithm, which leads to an increase of overall energy consumption.

As mentioned above, we can conclude the faults into two types, i.e., "hard faults" and "soft faults." For "hard faults," caused by the physical damage or limited operational lifetime, an efficient reconfiguration algorithm must be employed to reconstruct the original-sized logical array for keeping the normal work. For "soft faults," caused by overheating, overload or being employed by other applications, minimizing the energy dissipation of logical arrays is very important. Moreover, it has been shown that running applications on large arrays with low speed does not certainly consume less energy than on smaller array with relatively higher speed, in which it should be better for some cores to be switched to sleep mode.[22–24] It means that an online application prefers a subarray with customized size instead of the same size of the original array. Thus, for the "soft faults," the logical arrays with suitable size of online application should be quickly reconstructed for the coming applications with different sizes. In conclusion, fast and efficient reconfiguration algorithms are crucial for NoC to produce high-quality available logical array which can potentially mitigate communication congestion and reduce the overall energy consumption during the operation of the system. It is also true not only for processor arrays with spare PEs,[25–28] but also for the degradable processor arrays.[25,29–31] This is because fault tolerance is essential to the reliability of the system, no matter how many PEs are faults and how often the faults occur. Therefore, these motivate us to investigate the efficient reconfiguration algorithm to generate the high-quality fault-free logical arrays with original size and the given size in this paper. The main contributions of this paper are as follows.

(i) For the high-quality target array with original size, based on our previous work,[32] we present an efficient heuristic algorithm by employing column shifting

and row bishifting operations to generate a feasible logical array with small number of long interconnection paths in reconfiguration. The logical array is further refined by a customized tabu search (TS).

(ii) For the high-quality target array with given size, instead of the original size, we define a sliding window to locate an initial logical array on the physical array. Then, we contribute two heuristic algorithms based on the sliding window to generate the high-quality target array with given size. One is directly to focus on minimizing unified metric of the target array, which can obtain a good solution quality. And the other works on finding a target array with the minimum number of faulty PEs and minimum penalty, which keeps the solution quality and saves plenty of running time.

The rest of the paper is organized as below. In Sec. 2, the definitions and the description of the problems, together with the related works, are presented. In Sec. 3, we first describe our novel algorithm to derive a maximum fault-free array with high quality from the original one. Then we introduce two algorithms for reconfiguring the topology to get a fault-free subarray with given size. In Sec. 4, we show the experimental results and the analysis on the two situations, respectively. Finally, Sec. 5 concludes the current work and provides directions for future work.

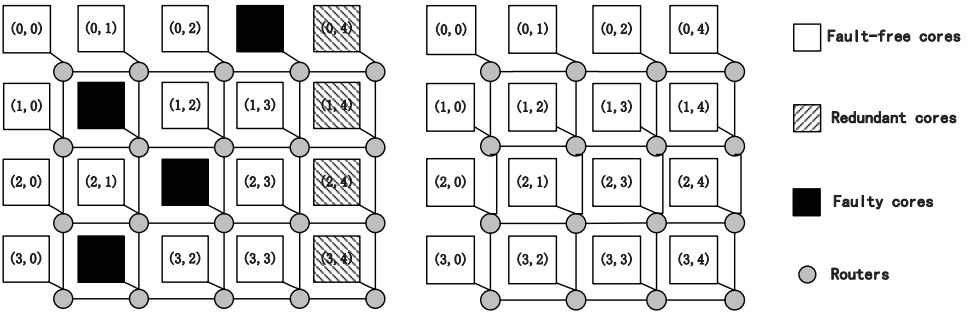
2. Preliminaries

In this section, we describe the reconfiguration architecture, related notations and definitions, followed by the reconfiguration problems to be investigated in this paper.

2.1. *Fault-tolerant architecture*

Let H denote the physical array on which some of the PEs are defective. Assume that N is the number of PEs in an $m \times n$ physical array, i.e., $N = m \cdot n$. Assume that the fault density of the physical array is ρ, then there are $(1 - \rho) \cdot N$ fault-free PEs in an $m \times n$ physical array. The rows and columns in the physical array are called physical rows and columns, respectively. T indicates the target array (logical array) which contains no faulty PEs. Assume that N' is the number of PEs in target array. A logical array is a degraded isomorphic of the physical array, and is obtained by replacing faulty PEs with redundant ones. Since the PEs in T correspond to the nodes in H, we have $|T| \leq |H|$, where $|T|(|H|)$ indicates the fault-free PEs in the target array (host array). We are interested in the mapping of PEs in the physical array to nodes in the logical array. The rows and columns in logical array are called logical rows and logical columns, respectively. Throughout this paper, $H_{i,j}$ $(T_{i,j})$ indicates the PE located at the position of (i, j) of the physical (logical) array, where i is its row index and j is its column index.

Figure 1(a) shows an example of the fault-tolerant architecture of a $4 \times (4 + 1)$ physical array. In the array, each PE is self-autonomous and is capable of



(a) Fault tolerant architecture (b) A logical array

Fig. 1. Fault-tolerant architecture.

communicating with any other PEs through the NoC infrastructure. Each square box in the host array represents a PE, whereas each circle represents a router. The black shaded boxes represent faulty PEs while unshaded ones represent the fault-free PEs. In addition, there are four redundant PEs located at the right-hand side of the physical array. Figure 1(b) illustrates a logical array obtained by replacing the faulty PEs with four redundant ones.

2.2. *Performance evaluation metrics*

To evaluate the performance of a logical topology, two evaluation metrics, i.e., distance factor (denoted as d) and congestion factor (denoted as c), are defined in Ref. 21. d is used to evaluate the communication delay between PEs, and it is defined as follows:

$$d = \frac{1}{N'} \sum_{i=1}^{N'} \frac{1}{n_k} \sum_{i=1}^{n_k} H_{k,i} \,, \tag{1}$$

where n_k is the number of logical neighbors of PE k and $H_{k,i}$ is the number of physical hops between PE k and its neighbor PE i, thus $1/n_k \sum_{i=1}^{n_k} H_{k,i}$ denotes the average hops between PE i and its neighbors. N' is the number of PEs in target array. Thus, d indicates the average physical hops between any two logical neighboring PEs. It is evident that smaller d leads to better communication performance. On a physical array without faulty PEs, the best target array is identical to the physical array with the limit of size, in which the minimum value of d is 1. However, due to the existence of the faulty PEs, d is always larger than 1.

Unlike distance factor d, the congestion factor c reflects the potential unbalance of communication traffic flow among different physical links. Let l be a physical link, for any two logically neighboring PEs, say PE u and PE v, if l belongs to the routing path between u and v according to the NoC's routing mechanism (e.g., the XY routing mechanism), we add c_l by 1. In other words, c_l indicates how many times

the link l is utilized by routing paths on the target array. c is defined as the standard deviation of c_l of all links:

$$c = \sqrt{\frac{\sum_{l=1}^{L} c_l - \overline{c_l}}{L-1}}, \tag{2}$$

where L is the total number of physical links and $\overline{c_l}$ is the average c_l for $0 < l \leq L$. Evidently, small c value implies balanced communication loads, and the increase of c will possibly lead to unbalanced communication traffic flow.

The d and c might be conflicted with each other during optimization, hence the unified metric (denoted as u in this paper) is defined as

$$u = \omega_c \cdot c + \omega_d \cdot d\,, \tag{3}$$

where ω_d and ω_c are two weight factors for optimization, and $\omega_c + \omega_d = 1$, $0 < \omega_c < 1$, $0 < \omega_d < 1$.

2.3. *Problems and latest works*

The problems investigated in this paper are as follows.

Problem $\mathcal{R}$. On router-based architecture, given an $m \times (n+k)$ mesh-connected homogeneous processor array H, which contains r faulty PEs and $m \cdot k$ redundant PEs ($r \leq m \cdot k$), find an $m \times n$ target array which contains no faulty PEs and the unified metric u of the target array is minimized.

With the development of techniques in NoC, a large number of PEs can be integrated on a single chip. But the target array required by the application generally is much smaller than the provided host array. This leads us to investigate the following problem, instead of reconfiguring an $m \times n$ target array for the application which only requires a smaller target array.

Problem $\mathcal{R}^c$. On router-based architecture, given an $m \times (n+k)$ mesh-connected homogeneous processor array H, which contains r faulty PEs and $m \cdot k$ redundant PEs ($r \leq m \cdot k$), find a $p \times q$ ($p \leq m$ and $q \leq n$) target array which contains no faulty PEs and the unified metric u of the target array is minimized.

The problem $\mathcal{R}$ has been proved to be NP-hard problem.[24] Noting that the solution of problem $\mathcal{R}$ is a particular solution of the problem $\mathcal{R}^c$, we conclude that the problem $\mathcal{R}^c$ is also NP-hard.

Recently, a heuristic algorithm is presented in Ref. 21 to solve the problem $\mathcal{R}$. A row rippling column stealing algorithm is utilized to reorganize the fault-free PEs into a logical regular topology. Therefore, RS algorithm tries to maintain the physical regularity of the virtual topologies in row unit and in column unit. To simplify the problem, without loss of generality, Ref. 21 assumes that one column of spare PEs is considered in the mesh or torus topology. If a row contains only one

faulty PE, the faulty PE is replaced by its right neighbor, meaning that the faulty PE moves to the right position and then it is replaced by the next neighbor, and so on, this process continues until the faulty one is transferred to the end of the row. When a row contains more than one faulty PEs, i.e., faulty PEs are more than the spare ones in this row, the rightmost faulty PE is replaced using the spare one, which is named as column stealing. The other faulty PEs within the row are replaced with the PEs immediately beneath them, from the next row. Similarly, the next row will steal fault-free PEs from its next row. The process repeats until a row, say row R_i, cannot find enough fault-free PEs from its next row to steal, then it steals fault-free PEs from the redundancy row, which locates above the row where column stealing started. Usually, the redundancy row is far from the current row R_i, we call this type of stealing as backward stealing. Backward stealing leads to great loss in communication performance in terms of c and d of the resultant topology. In Ref. 21, the topology produced by row rippling column stealing algorithm is further refined by a customized simulated annealing algorithm. For more details, see Ref. 21.

3. Reconfiguration Algorithms

This section presents algorithms for solving the problems $\mathcal{R}$ and $\mathcal{R}^c$.

3.1. *Algorithms for problem* $\mathcal{R}$

For problem $\mathcal{R}$, we present a novel reconfiguration algorithm consisting of two sub-algorithms. The first one is a heuristic algorithm to generate a logical array S by performing column shifting and row bishifting operations. The second one is to optimize the initial target array S by a customized tabu search approach.

3.1.1. *Heuristic algorithm*

Before introducing the proposed algorithm, we first define two types of operations, column shifting and row bishifting. Column shifting is essentially a cyclic shifting on a column segment. Formally, let $S_{i,k}$ denote the PE located at the position (i, k) of array S, where i is the row index and k is the column index. Then, the sequence $\langle S_{i,k}, S_{i+1,k}, \ldots, S_{j-1,k}, S_{j,k}\rangle$ is a column segment lying between row S_i and row S_j of target array S. The column shifting operation from $S_{j,k}$ to $S_{i,k}$ is performed as follows:

$$S_{i,k} \leftarrow S_{i+1,k}, S_{i+1,k} \leftarrow S_{i+2,k}, \ldots, S_{j-1,k} \leftarrow S_{j,k}, S_{j,k} \leftarrow S_{i,k}\,,$$

where $S_{i,k} \leftarrow S_{i+1,k}$ indicates that PE $S_{i,k}$ is replaced by PE $S_{i+1,k}$. The row bishifting operation on row S_i performs in a way that all PEs in row S_i move toward two directions, i.e., fault-free PEs move to left-hand side and faulty PEs move to right-hand side. In other words, it splits S_i into two parts such that the left part contains only fault-free PEs and the right part consists of faulty PEs. Note that the procedure

of row bishifting is stable, which means that the order among faulty PEs or the order among fault-free PEs is kept unchanged.

The proposed heuristic algorithm based on column shifting and row bishifting, denoted as CRS, works in the following way to solve problem $\mathcal{R}$.

Step 1: Target array S is initialized as host array H. Then, CRS performs row bishifting operation on row S_i for $0 \leq i < m$, and row S_i will be marked as a redundancy row if $|S_i| > n$.

Step 2: For each row, say S_i $(0 \leq i < n)$, whose fault-free elements are less than n, i.e., $|S_i| < n$, CRS locates the first faulty PE, say $S_{i,j}$, in row S_i and finds the nearest redundancy row, say S_r, to row S_i. Then algorithm CRS performs column shifting from $S_{r,j}$ to $S_{i,j}$ on sequence $\langle S_{i,j}, \ldots, S_{r,j} \rangle$, and then employs row bishifting operation on row S_r to make the faulty PE shifted from row S_i to the right-hand side of row S_r. After the two operations, the redundancy mark will be removed from row S_r if $|S_r| \leq n$. By far, one iteration of step 2 is done. If $|S_i| < n$, CRS continues to perform another iteration on S_i until $|S_i| = n$ is satisfied, otherwise, the algorithm proceeds to check row S_{i+1}. Step 2 repeats until all rows in S satisfy the above condition, i.e., $|S_i| \geq n$ for $0 \leq i < m$.

The pseudocode of CRS is shown in Algorithm 1.

Step 1 of the CRS runs in $O(n)$ for each row to move all faulty PEs to the right-hand side of the row. Thus, the algorithm CRS takes $O(m \cdot n)$ time for processing the entire array. In step 2, CRS performs one operation of column shifting and one operation of row bishifting for each faulty PE in the array, where column shifting operation runs in $O(m)$ time and the row bishifting operation runs in $O(n)$ time. Therefore, step 2 runs in $O(k \cdot (m + n))$, where k is the number of faulty PEs in the $m \times n$ physical array. As a result, the CRS runs in $O(k \cdot (m + n)) + O(m \cdot n)$, i.e., $O(k\alpha + N')$, where $\alpha = \max\{m, n\}$.

3.1.2. *Optimization by tabu search*

TS is one of the traditional heuristic-based algorithms to search for the global optimal solution for NP-hard problems.[33,34] It is possible to find a global optimum by using tabu search, although the global optimum is not guaranteed by this method. In this sub-subsection, we customize a TS algorithm, denoted as CRS-TS, to refine the heuristic solution generated by CRS.

Generally, TS works in an iterative way with five primary parameters: initial solution, neighborhoods structure, evaluation metric, tabu list and termination criteria. It starts from an initial solution, and iteratively moves from one potential solution s to an improved solution s' in the neighborhood of s, until the termination criterion is satisfied. In each iteration, a number of neighbor solutions are generated from the current solution. All the neighbors are then examined according to the

Algorithm 1. CRS

Input: An $m \times (n + k)$ physical array H.
Output: An $m \times n$ target array T.

```
1: S := H;
   /*Let S_i be the set of PEs located in ith row in S, and let |S_i| be the number
   of fault-free PEs in S_i*/
2: for i := 0 to m − 1 do
3:    Move all faulty PEs in row in S_i to the right-hand side of the row;
4:    if |S_i| > n then
5:       Mark row S_i as a redundancy row;
6:    end if
7: end for
8: for i := 0 to m − 1 do
9:    if |S_i| < n then
10:      while |S_i| < n do
11:         Find the first faulty PE in row S_i, say S_{i,j};
12:         Find a redundancy row, say S_r, which is the nearest to row S_i;
13:         Perform column shifting operation from PE S_{r,j} to PE S_{i,j};
14:         Perform row bishifting operation on row S_r;
15:         if S_r = n then
16:            Remove the redundancy row mark for S_r;
17:         end if
18:      end while
19:   end if
20: end for
21: T := S;
```

evaluation metric and the best neighbor is selected. Then the algorithm proceeds by transiting from current solution to the best neighbor, which is called a move. Note that a move may decrease the quality of the current solution. In order to avoid possible cycling and go beyond local optimum, tabu search introduces the notion of tabu list to forbid the recently visited solutions.[33] In other words, performing a move which is already in tabu list is not allowed. For more details on tabu search, see Ref. 34.

Initial solution. In this algorithm, CRS-TS starts with heuristic solution generated by CRS. Initial solution is in the format of array, where each element in the array stands for a PE.

Neighborhood structure. A neighbor solution is generated by exchanging any two nonfaulty nodes in the current feasible solution. All possible neighbor solutions will be built for searching the best one.

Evaluation. All neighbors are evaluated according to unified metric u defined in Sec. 2. Then the algorithm moves to the best neighbor that is not in tabu list and with minimum u. ω_c and ω_d are set to 0.1 and 0.9, respectively, which is the same as in Ref. 21.

Tabu list. The tabu list is used to prevent the search from cycling between solutions by storing the recently performed moves. In this paper, our tabu list has a fixed size. When the list is full, the oldest element of the list is replaced by the new element. Also, an aspiration criteria is utilized so that, if a tabu move generates a better solution than all the feasible solutions obtained so far, its tabu status is neglected.

Termination criteria. Stopping rules may be a fixed number of iterations or a fixed number of CPU time or a fixed number of consecutive iterations without an improvement in the best objective function value, etc. In this paper, the stopping rule is to fix the number of iterations, and it is set to $2mn - m - n$ for an $m \times n$ host array.

The pseudocode of CRS-TS is shown in Algorithm 2.

Algorithm 2. CRS-TS

Input: An initial solution S_{init} generated by CRS.
Output: Refined solution T_{best}.

$S_{\text{cur}} := S_{\text{init}}$; /*current local solution */
$S_{\text{best}} := S_{\text{init}}$; /*the best-so-far solution*/
for $i := 1$ to M **do**
  Generate neighbor set Q for solution S_{cur};
  Evaluate all neighbors in set Q by the unified metric u;
  /*select the best neighbor for the next move of the algorithm*/
  while 1 **do**
    Select a neighbor solution, say S_{neib}, of the minimum u from set Q;
    if $S_{\text{neib}} \notin TabuList$ or $u(S_{\text{neib}}) < u(S_{\text{best}})$ **then**
      Put S_{neib} in $TabuList$;
      $S_{\text{cur}} := S_{\text{neib}}$;
      if $u(S_{\text{neib}}) < u(S_{\text{best}})$ **then**
        $S_{\text{best}} := S_{\text{neib}}$;
      end if
      break;
    else
      Remove S_{neib} out of set Q, find a neighbor solution of the minimum u from set Q;
    end if
  end while
end for
$T_{\text{best}} := S_{\text{best}}$;

3.2. *Algorithms for problem* $\mathcal{R}^c$

This subsection presents two algorithms for topology reconfiguration. In other words, given an $m \times n$ physical array with r faulty PEs, we try to find a $p \times q$ nonfaulty subarray, such that the subarray is an approximate tightly-coupled one which has the lowest unified metric u.

Definition 1. A *sliding window* is a rectangular subarray, which contains a fixed number of PEs. The boundary of the sliding window consists of all horizontally adjacent PEs and vertically adjacent PEs to the window.

Figure 2 shows a sliding window on a host array. Each square represents a PE, and black ones represent faulty PEs. The area bounded by the dashed line is the sliding window, and the PEs shown by shaded block form the boundary of the sliding window.

3.2.1. *Unified metric-based algorithm*

In this sub-subsection, we propose an algorithm for problem $\mathcal{R}^c$. The algorithm utilizes a sliding window to check each feasible position and then to select a window with minimum unified metric u. The algorithm, denoted as UMA in this paper, works with the following steps.

Step 1: Create an initial $p \times q$ sliding window that is located in the upper left corner of the host array. For example, Fig. 3(a) shows the initial position of the sliding window.

Step 2: Find all faulty PEs in the sliding window, and then replace each faulty PE with its nearest nonfaulty one that lies out of the window. To implement the PE replacement, the operation of row/column shifting presented in algorithm CRS is employed to shift the nonfaulty PE to the boundary of the sliding window. After shifting operation, the algorithm replaces the faulty

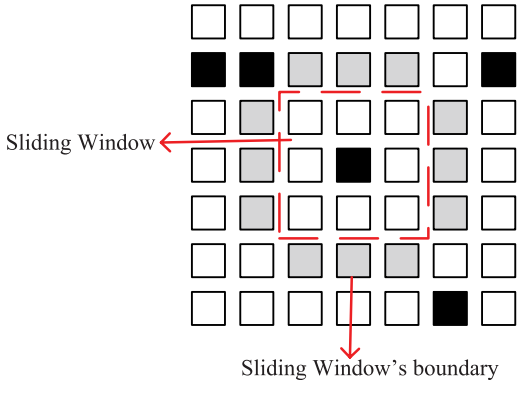


Fig. 2. The boundary of the sliding window.

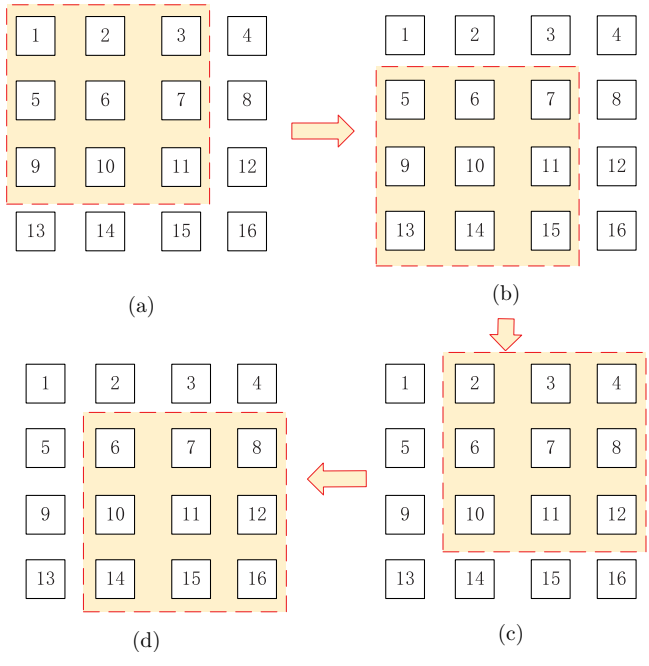


Fig. 3. The movement of sliding window.

PE with the nonfaulty PE. For the advantage of the shifting operation, see Fig. 4. If the value of u is smaller than the current value, the best window and the currently minimum u will be updated.

Step 3: Move the sliding window from top to bottom, and then from left to right in the host array, each move passes one row or one column. Figure 3 shows an example of moving sliding window on 4×4 host array, the sequence of steps for the movement is $a \rightarrow b \rightarrow c \rightarrow d$. The sliding window moves repeatedly (step 2) until all positions have been checked in the host array. Finally, the target array with minimum u is found.

Step 4: If the window is a rectangle, i.e., $p \neq q$, the algorithm rotates the window and then runs for one more time, in order to check whether there exists a better solution on the rotated window. After that, the algorithm will choose a better one from the two solutions, which are derived from original window and the rotated one respectively, as the final solution.

It is worthwhile to point out that the algorithm can terminate if there exists a fault-free subarray which is equal to or larger than the target array. Then the fault-free subarray will be the final solution. For the PE replacement, if the algorithm directly uses shifting operation to replace faulty PE with target fault-free PE, it is possible for the fault-free PEs within the window to be shifted out of the window. Thus, the

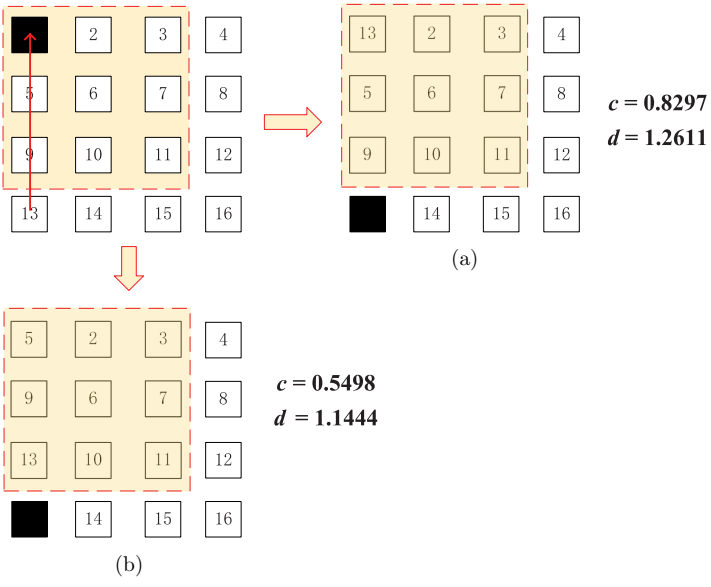


Fig. 4. Impact of shifting operations on the congestion factor c and distance factor d.

algorithm initially moves the fault-free PE to the boundary of the window, and then exchanges the faulty PE with the fault-free one, in order to keep the fault-free PEs still in the window.

In Fig. 4, directly using the fault-free PE numbered 13 to replace the faulty PE in the window results in the layout as shown in Fig. 4(a), with relatively high distance factor d and congestion factor c, in comparison to the layout shown by Fig. 4(b) which is derived from column shifting operation on the PEs numbered 5, 9 and 13. That is why we employ the shifting operation, rather than directly using PE replacement.

The presented algorithm UMA could generate an acceptable solution for the problem $\mathcal{R}^c$. The target is to minimize the unified metric u. The sliding window enumerates all possible positions in the worst case, to select the best window with the minimum u as the final solution, although the computing time may increase for large problems. The pseudocode of UMA is shown in Algorithm 3.

The sliding window moves $2 \cdot (n - q + 1)(m - p + 1)$, i.e., $O(N)$ times in algorithm UMA, where $m \cdot n$ is the size of the original array and $p \cdot q$ is the size of the sliding window. Finding the locations of faulty PEs in a sliding window takes $O(p + q)$ time, because a new window only increases a new row/column in comparison with the previous windows. For each faulty PE within the sliding window, UMA performs only one shifting operation and only one exchanging operation. The shifting operation runs in $O(m + n)$ time and the exchanging operation runs in $O(1)$ for each faulty PE in the sliding window. Thus, UMA runs in $O((p + q) \cdot (m + n))$

Algorithm 3. UMA

Input: $m \times n$ physical array.
Output: $p \times q$ target array T_{best}.

Procedure Sub-UMA($H, p, q, T_{\text{best}}^{p\times q}$)
for $i := 0$ to $n - 1$ **do**
  for $j := 0$ to $m - p$ **do**
    $H' := H$;
    for each faulty PE within F **do**
      Find the corresponding nonfaulty PE P_{nearest} which is nearest and out of F for the faulty PE;
      Replace the faulty PE with the corresponding P_{nearest} by shifting and exchange operations;
    end for
    if $u(F) < u(T_{\text{best}}^{p\times q})$ **then**
      $T_{\text{best}}^{p\times q} := F$;
    end if
    Move F down by one row;
  end for
  Move F up to the top side of H';
  Move F right by one column;
end for
end procedure

Generate a $p \times q$ sliding window F that is located in the upper left corner of H;
$T_{\text{best}} := F$;
Sub-UMA($H, p, q, T_{\text{best}}^{p\times q}$);
$T_{\text{best}} := T_{\text{best}}^{p\times q}$;
if $p \neq q$ **then**
  Sub-UMA($H, q, p, T_{\text{best}}^{q\times p}$);
  The better one from the current T_{best} and $T_{\text{best}}^{q\times p}$ is set to the final T_{best};
end if

time for each sliding window. Therefore, the time complexity of UMA is $O(\alpha \cdot \beta \cdot N)$, where $\alpha = \max\{m, n\}$ and $\beta = \max\{q, p\}$.

3.2.2. *Minimum fault-based algorithm*

We now present a fast greedy algorithm, named FGA, to deal with the problem $\mathcal{R}^c$. FGA also utilizes sliding window to find a best subarray with the minimum number of faulty PEs and the minimum penalty (see the following Definition 2). Note that

the subarray with the small number of faulty PEs can be simply reconfigured. This motivates us to design a greedy algorithm, initially finding the subarray with the minimum number of faulty PEs, then reconfiguring it to get an approximate tightly-coupled target array.

Definition 2. The shortest distance between the faulty PE i and the boundary of sliding window is defined as the penalty of PE i, denoted as P_i. Let P_{total} indicate the penalty of the sliding window. P_{total} is defined as $P_{\text{total}} = \sum_{i=1}^{x} P_i$, where x represents the number of faulty PEs in the sliding window.

Algorithm 4. FGA

Input: $m \times n$ physical array.
Output: $p \times q$ target array T_{best}.

Procedure Sub-FGA$(H, p, q, F_{\text{best}}^{p\times q})$
for $i := 0$ to $n - q$ **do**
  for $j := 0$ to $m - p$ **do**
    $H' := H$;
    Calculate the number of faulty PEs $x(F)$ and the penalty $P_{\text{total}}(F)$;
    if $x(F) < x(T_{\text{best}}^{p\times q})$ **then**
      $F_{\text{best}}^{p\times q} := F$;
    end if
    Move F down by one row;
  end for
  Move F up to the top side of H';
  Move F right by one column;
end for
$F_{\text{best}}^{p\times q}$:= the sliding window with minimum P_{total};
end procedure

Generate a $p \times q$ sliding window F that is located in the upper left corner of H;
$F_{\text{best}} := F$;
Sub-FGA$(H, p, q, F_{\text{best}}^{p\times q})$;
$F_{\text{best}} := F_{\text{best}}^{p\times q}$;
if $p \neq q$ **then**
  Sub-FGA$(H, q, p, F_{\text{best}}^{q\times p})$;
  The better one from the current F_{best} and $F_{\text{best}}^{q\times p}$ is set to the final F_{best};
end if
T_{best} := the resultant array based on F_{best};

The greedy algorithm FGA is similar to the algorithm UMA in outline. It also utilizes the sliding window to accomplish the reconfiguration of a subarray. Unlike UMA, the greedy algorithm focuses on minimizing the penalty and the number of faulty PEs. The main steps are as follows.

Step 1: Create an initial $p \times q$ sliding window that is located in the upper left corner of the host array.
Step 2: Check each PE in the sliding window to find the faulty PEs, then compute and save the number of faulty PEs and the penalty of the sliding window.
Step 3: Move the sliding window from top to bottom, and then from left to right in the host array, each move passes one row or one column. The sliding window moves repeatedly (step 2), until all positions have been checked in the host array. In the end, the sliding window with minimum number of faulty PEs, together with the minimum penalty, will be chosen as the target array.
Step 4: If the window is a rectangle, i.e., $p \neq q$, the algorithm rotates the window and then runs one more time, in order to check if there exists a better solution on the rotated window. After that the algorithm will choose a better one from the two solutions, which are derived from the original window and the rotated one, respectively, as the final solution.

The pseudocode of FGA is shown in Algorithm 4.

The sliding window moves $2 \cdot (n - q + 1)(m - p + 1)$, i.e., $O(mn)$ times in algorithm UMA, where $m \cdot n$ is the size of the original array and $p \cdot q$ is the size of the sliding window. For each sliding window, FGA calculates the number of faulty PEs and the penalty of the window, which runs in $O(p \cdot q)$ time. Thus, the time complexity of FGA is $O(m \cdot n \cdot p \cdot q)$, i.e., $O(p \cdot q \cdot N)$.

4. Simulation Results

We keep the same assumptions and model of the array as in Refs. 21 and 32 in our simulations. The faulty PEs are randomly distributed in the host array, and the faults are only associated with PEs. The communication infrastructure is assumed to be fault-free. Simulations have been conducted on a large number of randomly generated instances with fault density ranging from 1% to 30%.[21,32] All algorithms are implemented in C++ and they run on an Intel Xeon 3.2-GHz computer with 32-GB RAM.

4.1. *Results for problem* $\mathcal{R}$

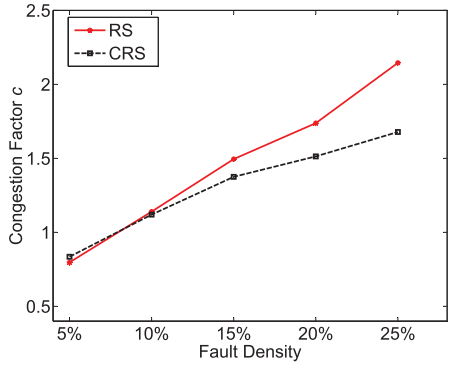
In this subsection, the proposed algorithms are compared with the row rippling column stealing algorithm,[21] and the customized simulated annealing algorithm,[21] in terms of distance factor d and congestion factor c. As discussed in Sec. 2, d is used to evaluate the communication delay between PEs, and c reflects the potential

unbalance of communication traffic flow among different physical links. The improvements in d and c of our algorithm CRS over algorithm RS are calculated based on the following formulas:

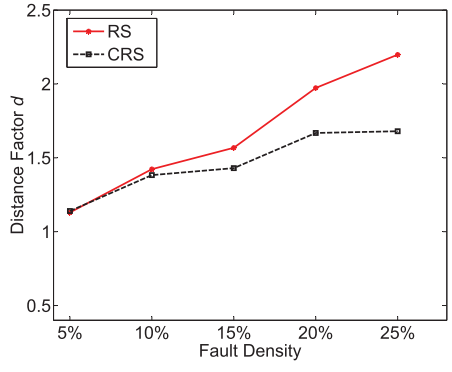
$$\text{imp}_c\ (\%) = \frac{c_{\text{RS}} - c_{\text{CRS}}}{c_{\text{RS}}} \times 100\,, \tag{4}$$

$$\text{imp}_d\ (\%) = \frac{d_{\text{RS}} - d_{\text{CRS}}}{d_{\text{RS}}} \times 100\,. \tag{5}$$

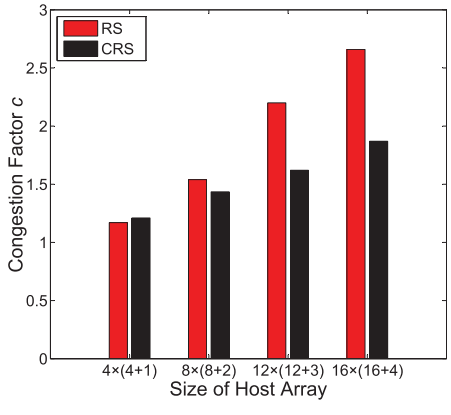
For problem $\mathcal{R}$, it needs to find a maximum fault-free subarray during reconfiguration using redundant PEs. We conduct three sets of experiments. In the first experiment we compare algorithm RS and CRS. Figures 5(a) and 5(b) show the performance comparisons between algorithms RS and CRS, on the size of



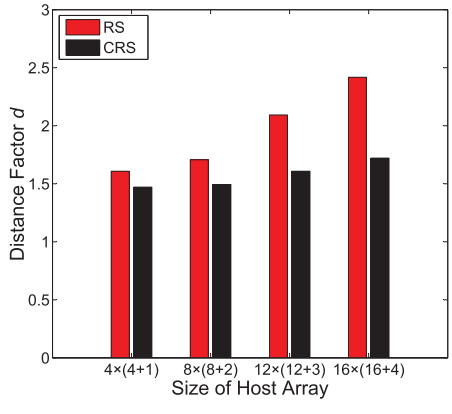
(a) Congestion factor c on $10 \times (10 + 3)$ arrays.



(b) Distance factor d on $10 \times (10 + 3)$ arrays.



(c) Congestion factor c on fault density of 20%.



(d) Distance factor d on fault density of 20%.

Fig. 5. Performance comparisons between RS (baseline) and CRS (new algorithm) in terms of c and d, averaged over 20 random instances.

$10 \times (10 + 3)$ host arrays, i.e., the host array has three redundancy columns. The fault density ranges from 5% to 25%.

CRS tends to outperform RS with the increasing fault density. For example, on host arrays with 5% faulty PEs, the value of c is 0.797 for CRS and is 0.835 for RS. Then, with the increasing of fault density, c increases much slower for CRS than for RS. This is due to the fact that algorithm RS performs backward stealing techniques to replace faulty PEs with fault-free PE in the next row, when a physical row does not contain enough fault-free PEs. However, algorithm CRS redistributes fault-free PEs using a shifting operation, such that the neighboring PEs in physical array can be utilized to form the target array, in order to reduce the values of c and d. Figures 5(c) and 5(d) illustrate the comparisons between two algorithms, in terms of c and d, on host arrays with fault density of 20%. The sizes of host arrays are set to $4 \times (4 + 1), 8 \times (8 + 2), 12 \times (12 + 3)$ and $16 \times (16 + 4)$, respectively. Both c and d of the two algorithms are very close for small arrays, but CRS clearly outperforms RS with the increasing size of host array. This is because that a physical row R_i needs to steal fault-free PEs from a redundancy row R_r in backward stealing of RS, the row R_i may be far from R_r on a large host array.

Table 1 shows comparisons of algorithms CRS-TS and RSSA on host arrays with size ranging from 4×4 to 16×16 and fault density ranging from 10% to 30%. RSSA first generates an initial topology using RS and then performs a simulated annealing algorithm to refine the initial topology, while CRS-TS first generates an initial topology using CRS and then employs TS to refine the initial topology produced by CRS. Generally, the average improvement of CRS-TS over RSSA is 19% for congestion factor c and is 16% for distance factor d. Both CRS-TS and RSSA achieve desirable c and d on host arrays with small fault density. However, with the increase of fault density, algorithm CRS-TS is clearly better than RSSA. For example, for

Table 1. The performance comparisons of algorithms CRS-TS (new algorithm) and RSSA (baseline).

Host arrays			Target arrays					
Size	Spare PEs	ρ (%)	c_{RSSA}	$c_{\text{CRS-TS}}$	imp_c	d_{RSSA}	$d_{\text{CRS-TS}}$	imp_d
4×4	4×1	10	0.965	0.927	2.888	1.108	1.097	0.829
	4×1	20	1.040	1.013	1.775	1.462	1.438	1.275
	4×2	30	1.262	1.109	10.807	1.160	1.137	1.428
8×8	8×1	10	1.087	0.991	7.180	1.420	1.347	4.639
	8×2	20	1.516	1.219	18.602	1.680	1.419	15.205
	8×3	30	1.823	1.397	22.795	2.081	1.555	24.922
12×12	12×2	10	1.240	1.122	8.990	1.348	1.278	5.078
	12×3	20	2.091	1.447	29.820	2.043	1.546	23.975
	12×4	30	2.442	1.592	34.333	2.761	1.783	35.258
16×16	16×2	10	1.595	1.294	17.835	1.593	1.422	10.103
	16×4	20	2.584	1.567	38.746	2.397	1.627	31.812
	16×5	30	2.921	1.804	37.816	3.219	1.962	38.925

8×8 host arrays with fault density increasing from 10% to 30%, distance factor d by CRS-TS increases slowly from 1.347 to 1.555 while it grows rapidly from 1.420 to 2.081 for RSSA, implying that CRS-TS is more scalable than RSSA. This is because the improvement of algorithm CRS over RS increases with the increasing fault density, and TS is superior than simulated annealing algorithm utilized in algorithm RSSA. In addition, the improvements in terms of c and d are more significant on relatively large physical arrays, due to the same reason discussed above. For example, on physical arrays with fault density of 20%, the improvements in terms of c are 1.8%, 18.6%, 29.8% and 38.7% for physical arrays with sizes of $4 \times 4, 8 \times 8, 12 \times 12$ and 16×16, respectively.

4.2. *Results for problem* $\mathcal{R}^c$

For problem $\mathcal{R}^c$, this subsection shows the performance comparisons for the algorithms UMA and FGA on different host arrays with different fault densities. We collect the execution time, the value of unified metric u and the number of faulty PEs to be replaced during the reconfiguration.

In the simulation, a baseline algorithm, named RAM, is employed in this subsection to evaluate the performance of the proposed algorithms. RAM randomly chooses the position of the sliding window and chooses the fault-free PEs out of the window. In detail, for an $m \times n$ host array and a $p \times q$ target array, the algorithm randomly chooses a coordinate point (r_x, r_y) $(0 \leq r_x < m - p, 0 \leq r_y < n - p)$. This point is treated as the upper left corner of the $p \times q$ sliding window. For the sliding window, the algorithm randomly chooses fault-free PEs out of the window to replace the fault PEs in the window. The algorithm iterates for k times, where k depends on the sizes of the host array and the target array. Finally, the algorithm chooses the subarray of the lowest u as its output.

Figure 6 shows the comparison of three algorithms in terms of u. As discussed in previous sections, u is the combination of congestion factor c and distance factor d. It can reflect the inner communication distance of reconfigured logical topology, and the balance state of traffic loads. Figure 6 shows the results of u for three algorithms in different-sized host arrays, the fault density is set to 10% and the size of target array is fixed to 8×8. The host array is set to $16 \times 16, 32 \times 32, 64 \times 64$ and 128×128. From Fig. 6, it can be concluded that the algorithms FGA and UMA are able to achieve much smaller u in comparison to the algorithm RAM on different-sized host arrays. This is because, the number of the faulty PEs to be replaced in the proposed two algorithms is clearly smaller than that in the algorithm RAM. Noting that a new communication path needs to be established whenever a faulty PE exists, we conclude that the more faulty PEs are replaced, the more communication paths are generated. Therefore, the subarray with less number of faulty PEs will bring a tightly-coupled logical topology with smaller u. With the increase of the size of host array, the value of u tends to decrease for all of the three algorithms. This is because,

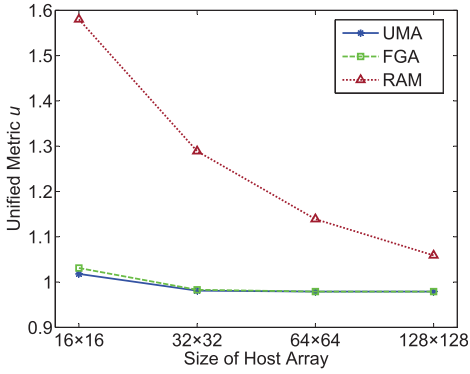


Fig. 6. Performance comparison between three algorithms in terms of unified metric on different host arrays with 10% fault density.

with the increase of the size of host array, the fixed size of target array becomes relatively smaller and smaller. Therefore, more and more good positions (with less faulty PEs) can be provided to the sliding window. The values of u for UMA and FGA become close to each other. For example, on 16×16 host array, u is 1.018 for UMA and is 1.031 for FGA, respectively. This is because, for a fixed fault density of 10%, by finding a fixed relatively small 8×8 target array, both algorithms can find a better position for the sliding window.

To further investigate the performance of the algorithms, a group of experiments have been conducted. Figure 7 shows the influence of different fault densities. The comparison of the three algorithms in terms of u is shown in each subfigure of Fig. 7. With the increasing fault density, the value of u increases for all the three algorithms. For the case of 8×8 target array, the values of u on 128×128 host arrays are smaller than on 16×16 host arrays. On the other hand, the size of target arrays will also impact the value of u on the given 128×128 host array. For example, u is smaller on 8×8 target arrays than on 64×64 target arrays. This is because, for larger host arrays, it is relatively easy to find a valid sliding window, resulting in feasible

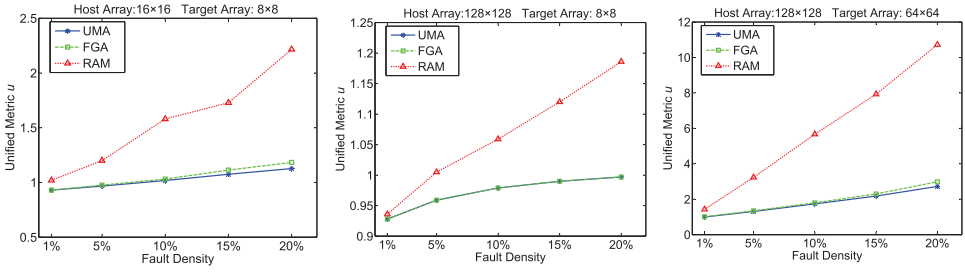


Fig. 7. Performance comparison between three algorithms in terms of unified metric on different fault densities with different array sizes.

Table 2. Comparisons in execution time.

Host arrays		Target array	Execution time (ms)		
Size	ρ (%)	Size	UMA	FGA	RAM
16×16	1	8×8	9.90	0.13	205.18
	5		9.18	0.14	208.76
	10		8.99	0.15	201.32
	15		8.98	0.16	193.84
	20		8.01	0.14	180.60
128×128	1	32×32	47301.28	37.49	27614.95
	5		42842.45	33.26	46020.10
	10		40674.65	37.41	45902.28
	15		39923.14	35.98	43454.45
	20		37182.30	38.15	41833.43
128×128	1	64×64	14721.42	43.86	26705.77
	5		15462.81	44.28	35350.41
	10		16868.78	44.72	33959.96
	15		18797.49	45.34	31664.16
	20		21725.70	46.39	29297.63

solutions. On the host array with a given size, it is more and more difficult to find a fault-free subarray with the increasing target array. This is because more faults need to be replaced by fault-free ones, which leads to the increase of u. Algorithm UMA obtains smallest u among the three algorithms, as it directly focuses on minimizing the value of u. In addition, with the increasing fault density, the number of faulty PEs that need to be replaced increases, leading to higher u for all the three algorithms.

Table 2 shows the running times of the three algorithms. It is clear that FGA runs much faster than the other two algorithms on arrays with different sizes and fault densities. This is because FGA utilizes a sliding window to find the subarray with minimal faulty PEs and minimum penalty instead of utilizing the row/column shifting operations. Searching the faulty PEs and calculating the penalty is quicker than performing the operations of row/column shifting. On the other hand, while obtaining the highest qualities of solutions among the three algorithms, the UMA also outperforms the baseline approach in running time for most cases as shown in the table, with an average improvement of 25.94%. In addition, by comparing the second group of instances ($128 \times 128 \rightarrow 32 \times 32$) and the third group of instances ($128 \times 128 \rightarrow 64 \times 64$), it can be observed that the improvement of UMA over RAM tends to grow with the increase in target array size.

5. Conclusions

Due to the fact that more and more cores are integrated into a single chip along with the ever-increasing circuit density, PEs are becoming more vulnerable to faults

during chip fabrication as well as the process of running computation-intensive programs. The existence of faulty PEs has inherently changed the structured physical topologies into undesired ones, which not only causes communication congestion and delay, but also leads to significant extra energy consumption. We have presented algorithms to construct an $m \times n$ target array on an $m \times (n + k)$ processor array with faults. The proposed algorithms are able to efficiently replace faulty PEs with the redundant ones, utilizing the novel shifting operations proposed in this paper. The customized tabu search can significantly refine the initial target array generated by the previous algorithm. Moreover, we have also proposed heuristic algorithms to find $p \times q$ target array for $p \leq m$ and $q \leq n$. Simulation results show that the algorithm CRS tends to outperform RS, in both distance factor d and congestion factor c. The average improvement of CRS over RS is 21.8% for c and 23.6% for d on the $10 \times (10 + 3)$ host array with fault density of 25%. The improvements of algorithm CRS-TS over RSSA in terms of c and d are more significant on relatively large physical arrays or on the physical array with high fault density. The average improvement of CRS-TS over RSSA is 19% for c and is 16% for d. On the other hand, for constructing arrays with given size, the proposed algorithms UMA and FGA perform better than the algorithm RAM in terms of the unified metric and the number of faulty PEs for different host arrays. For the case of 16×16 host array with 10% faulty PEs and 8×8 target array, the obtained u values are 1.018, 1.031 and 1.580, for UMA, FGA and RAM, respectively. Our additional simulation results show that the u generated by UMA and FGA are nearly equal.

Acknowledgments

This work is supported by the National Key R&D Program of China under the Grant No. 2018YFB1003201 and National Natural Science Foundation of China under the Grant No. 61672171. It is also supported in part by the Guangdong Natural Science Foundation under the Grant No. 2018B030311007 and Major R&D Project of Educational Commission of Guangdong under the Grant No. 2016KZDXM052.

References

1. W. J. Dally and B. Towles, Route packets, not wires: On-chip interconnection networks, *Proc. Design Automation Conf.* (2001), pp. 684–689.
2. L. Benini and G. De Micheli, *Networks on Chips: Technology and Tools* (Morgan Kaufmann, 2006).
3. Y. Wu, J. Zhao, D. Chen and D. Guo, Modeling of Gaussian network-based reconfigurable network-on-chip designs, *IEEE Trans. Comput.* **65** (2016) 2134–2142.
4. J. Wu, T. Srikanthan, G. Jiang and K. Wang, Constructing sub-arrays with short interconnects from degradable VLSI arrays, *IEEE Trans. Parallel Distrib. Syst.* **25** (2014) 929–938.
5. S. Borkar, Thousand core chips: A technology perspective, *Proc. ACM/IEEE Design Automation Conf.* (2007), pp. 746–749.

6. E. Sperling, Turn down the heat please (2007), http://www.edn.com/article/CA6350202.html.
7. J. H. Collet, P. Zajac, M. Psarakis and D. Gizopoulos, Chip self-organization and fault tolerance in massively defective multicore arrays, *IEEE Trans. Dependable Secur. Comput.* **8** (2011) 207–217.
8. L. Zheng, J. Cai, D. Ming, Y. Lei and L. Zan, Hybrid communication reconfigurable network on chip for MPSoC, *Proc. IEEE Int. Conf. Advanced Information Networking and Applications* (2010), pp. 356–361.
9. D. Xiang and Y. Zhang, Cost-effective power-aware core testing in NoCs based on a new unicast-based multicast scheme, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* Vol. 1 (2011), pp. 135–147.
10. M. Modarressi, H. Sarbazi-Azad and A. Tavakkol, An efficient dynamically reconfigurable on-chip network architecture, *Proc. Design Automation Conf.* (2010), pp. 166–169.
11. G. Jiang, J. Wu, Y. Ha, Y. Wang and J. Sun, Reconfiguring three-dimensional processor arrays for fault-tolerance: Hardness and heuristic algorithms, *IEEE Trans. Comput.* **64** (2015) 2926–2939.
12. G. Jiang, J. Wu and J. Sun, Efficient reconfiguration algorithms for communication-aware three-dimensional processor arrays, *Parallel Comput.* **39** (2013) 490–503.
13. Y. C. Chang, C. T. Chiu, S. Y. Lin and C. K. Liu, On the design and analysis of fault tolerant NoC architecture using spare routers, *Proc. Asia and South Pacific Design Automation Conf.* (2011), pp. 431–436.
14. S. Murali, D. Atienza, L. Benini and G. D. Michel, A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip, *Proc. ACM/IEEE Design Automation Conf.* (2006), pp. 845–848.
15. E. Schuchman and T. N. Vijaykumar, Rescue: A microarchitecture for testability and defect tolerance, *ACM SIGARCH Comput. Archit. News* **33** (2005) 160–171.
16. Z. X. Wu, J. X. Wang, J. Y. Zhang and X. Y. Wang, Exploration of a reconfigurable 2D mesh network-on-chip architecture and a topology reconfiguration algorithm, *Proc. IEEE Int. Conf. Solid-State and Integrated Circuit Technology* (2012), pp. 1–3.
17. R. Akbar, A. A. Etedalpour and F. Safaei, An efficient fault-tolerant routing algorithm in NoCs to tolerate permanent faults, *J. Supercomput.* **72** (2016) 4629–4650.
18. J. Zhou, H. Li, T. Wang and X. Li, Loft: A low-overhead fault-tolerant routing scheme for 3D NoCs, *Integr. VLSI J.* **52** (2016) 41–50.
19. K. N. Dang, M. Meyer, Y. Okuyama and A. B. Abdallah, A low-overhead soft-hard fault-tolerant architecture, design and management scheme for reliable high-performance many-core 3D-NoC systems, *J. Supercomput.* **73** (2017) 2705–2729.
20. Y. Ren, L. Liu, S. Yin, J. Han and S. Wei, Efficient fault-tolerant topology reconfiguration using a maximum flow algorithm, *ACM Trans. Reconfig. Technol. Syst.* **8** (2015) 19-1–19-24.
21. L. Zhang, Y. Han, Q. Xu, X. W. Li and H. Li, On topology reconfiguration for defect-tolerant NoC-based homogeneous manycore systems, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **17** (2009) 1173–1186.
22. M. K. Bhatti, M. Farooq, C. Belleudy and M. Auguin, Controlling energy profile of RT multiprocessor systems by anticipating workload at runtime, *Proc. Symp. Architectures Nouvelles de Machines* (2009), pp. 1–12.
23. M. A. Awan and S. M. Petters, Energy-aware partitioning of tasks onto a heterogeneous multi-core platform, *Proc. Real-Time and Embedded Technology and Applications Symp.* (2013), pp. 205–214.

24. V. Legout, M. Jan and L. Pautet, Scheduling algorithms to reduce the static energy consumption of real-time systems, *Real-Time Syst.* **51** (2015) 153–191.
25. R. Negrini, M. G. Sami and R. Stefanelli, *Fault Tolerance Through Reconfiguration in VLSI and WSI Arrays* (The MIT Press, 1989).
26. V. P. Roychowdhury, J. Bruck and T. Kailath, Efficient algorithms for reconfiguration in VLSI/WSI arrays, *IEEE Trans. Comput.* **39** (1990) 480–489.
27. T. Horita and I. Takanami, Fault-tolerant processor arrays based on the 1.5-track switches with flexible spare distributions, *IEEE Trans. Comput.* **49** (2000) 542–552.
28. P. J. Chuang and L. C. Yao, An efficient reconfiguration scheme for fault-tolerant meshes, *Inf. Sci.* **172** (2005) 309–333.
29. C. P. Low, An efficient reconfiguration algorithm for degradable VLSI/WSI arrays, *IEEE Trans. Comput.* **49** (2000) 553–559.
30. J. Wu, T. Srikanthan and X. Wang, Integrated row and column rerouting for reconfiguration of VLSI arrays with four-port switches, *IEEE Trans. Comput.* **56** (2007) 1387–1400.
31. L. H. Ran, N. Shrivastava, R. G. Melhem and C. L. Liu, Optimal reconfiguration algorithms for real-time fault-tolerant processor arrays, *IEEE Trans. Parallel Distrib. Syst.* **6** (1995) 498–510.
32. C. Wang, J. Wu, G. Jiang and J. Sun, An efficient topology reconfiguration algorithm for NoC based multiprocessor arrays, *Proc. IEEE Int. Conf. High Performance Computing and Communications and 2013 IEEE Int. Conf. Embedded and Ubiquitous Computing* (2013), pp. 873–880.
33. U. Benlic and J. K. Hao, An effective multilevel tabu search approach for balanced graph partitioning, *Comput. Oper. Res.* **38** (2011) 1066–1075.
34. F. Glover and R. Marti, *Tabu Search* (Springer US, 2006).