Noname manuscript No.
(will be inserted by the editor)

Parallel Reconfiguration Algorithms for Mesh-connected Processor Arrays

Jigang Wu[1] · **Guiyuan Jiang**[2] · **Yuze Shen**[3] · **Siew-Kei Lam**[4] · **Jizhou Sun**[2] · **Thambipillai Srikanthan**[4]

Received: date / Accepted: date

Abstract Effective fault tolerance techniques are essential for improving the reliability of multiprocessor systems. At the same time, fault tolerance must be achieved at high-speed to meet the real-time constraints of embedded systems. While parallelism has often been exploited to increase performance, to the best of our knowledge, there has been no previously reported work on parallel reconfiguration of mesh-connected processor arrays with faults. This paper presents two parallel algorithms to accelerate reconfiguration of the processor arrays. The first algorithm reconfigures a host array in parallel in a multi-threading manner. The threads in the parallel algorithm executes independently within a safe rerouting distance. The second algorithm is based on divide-and-conquer approach to first generate the leftmost segments in parallel and then merge the segments in parallel. When compared to the conventional algorithm, simulation results on a large number of instances confirm that the proposed algorithms significantly accelerate the reconfiguration without loss of harvest.

Keywords Processor array · reconfiguration · fault tolerance · parallel algorithm.

J. Wu
School of Computer Science and Software Engineering, Tianjin Polytechnic University, Tianjin, 300387, China
E-mail: asjgwu@gmail.com

G. Jiang, J. Sun
School of Computer Science and Technology, Tianjin University, Tianjin, 300072, China
E-mail: jguiyuan@gmail.com, jzsun@tju.edu.cn

Y. Shen
College of Computer and Information Science, Northeastern University, Boston, Massachusetts, 02115, USA
E-mail: shenyuze@ccs.neu.edu

S.K. Lam, T. Srikanthan
School of Computer Engineering, Nanyang Technological University, 639798, Singapore
E-mail: assklam@ntu.edu.sg, astsrikan@ntu.edu.sg

1 Introduction

As technology advances, hundreds to thousands of processing elements (PEs) are integrated on a single chip, such as Intel's terascale processor [1] and Tilera's TILE64 processor [2], to process massive amounts of information in parallel. This has led to significant speedups and power efficiency as the processing cores can be clocked at lower frequency while meeting the performance constraints. However, due to circuit reliability issues, dependability becomes a major concern in these multi-processor systems. As such fault tolerance have become an essential component in the design of multi-processor systems in order to mitigate the dependability issues. Many works have been undertaken to tackle the challenge of dependability and fault tolerance for processor arrays (e.g. multi-processor systems).

The mesh topology is one of the most common communication infrastructure for multi-processor system due to its well-controlled electrical parameters and ease of implementation. The hardware architecture of the direct acyclic graph (DAG) and the mesh topology are compared in [3] and it was shown the mesh topology is well-suited for reconfiguration to enable fault tolerance. Fault-tolerant based reconfiguration is achieved by re-organizing the fault-free PEs in the processor array to a new mesh topology. Optical circuit switches have recently been proposed as a low-cost, low-power and high-bandwidth alternative in the design of high-performance chip multiprocessor systems [4–7]. An added advantage of these switches is that they allow for a reconfiguration of the network topology.

Two types of fault tolerance architectures, namely router-based architecture and switch-based architecture, are frequently investigated for mesh connected processor arrays. Router-based architecture [8–12] consists of network nodes connected by links in mesh topology. Each network node consists of a conventional router which is enclosed by topology switches. The topology switches are used to connect links and routers, which allows different logical topologies to be configured on top of the same physical architecture. Although, the router-based architecture is easier to reconfigure, it requires a complex router circuit which increases the hardware cost, power consumption and router faults. Moreover, the routing procedure is usually very time-consuming.

In switch-based architecture [13], single track switches are laid between neighboring PEs, and the switches are also interconnected with each other. In this type of architecture, once the connection is set up, data can be transferred through the connection without any header information; furthermore, the time delay is negligible since no routing or arbitration is needed. Hence, switch-based architecture is superior in terms of hardware cost, time delay, power consumption and probability of router/switch faults. However the challenge in switch based architecture lies in the design of efficient reconfiguration algorithms. In this paper, we focus on developing efficient reconfiguration algorithms for the switch-based fault tolerant architecture.

Generally, two distinct approaches, i.e., the redundancy approach and the degradation approach, have been investigated for switch based fault-tolerant architecture. The redundancy approaches tolerates faulty PEs by replacing them with spare PEs to reconstruct a logical array with required size [14–16], However, the chip has to be discarded if spare PEs failed in replacing all faulty ones. In the degradation approach, there is no spare PEs and all PEs are treated in a uniform way. Therefore, a degradable logical array can be formed using fault-free PEs from a faulty array under the constraint of the minimum dimension which is dependent on the requirement of applications. Many effective approaches for processor array reconfiguration have been proposed under three different rerouting constraints [18], namely 1) *row and column bypass*, 2) *row bypass and column rerouting*, and 3) *row and column rerouting elements*. They have shown that most problems that arise under these constraints are NP-complete and they also proposed some heuristic reconfiguration algorithms for these problems. An optimal algorithm named GCR was proposed in [19] to find a maximal logical array (MLA) that contains a set of the selected rows. The techniques which perform row-exclusion and compensation were proposed in [20]. A heuristic algorithm is formulated by combining the techniques with GCR to generate an approximate MLA. Recently, a dynamic programming approach was introduced for reducing power dissipation of a logical array in [21] by reducing the number of long-interconnects. In [22], reconfiguration under both row and column re-routing schemes are investigated to maximize the logical target array using 4-port switches. In [23], a genetic algorithm for the reconfiguration of degradable mesh arrays is presented for constructing logical rows/columns. A strategy based rerouting scheme is proposed to reroute the inter-connections of fault-free PEs in both row and column directions. In [24], novel techniques are presented to accelerate reconfiguration by employing flexible upper bound and lower bound for MLA. In [25,26], the reconfiguration algorithms are proposed for three-dimensional processor array. Methods that use different tracks and switches to increase harvest on the reconfigurable processor arrays are reported in [15,17,28–30]. Other approaches for faulty processor diagnoses, such as software based self-testing, functional based testing and structural scan-based testing are found in [31–36].

The reconfiguration time is one of the important criteria in real-time systems, as fatal error may occur if the deadline is missed as a result of reconfiguration. This motivates us to develop fast reconfiguration algorithms by exploiting parallelism.. In this paper, we propose two parallel algorithms to accelerate a traditional but frequently used algorithm, named GCR. The first one reconfigures a host array in parallel using the multi-threading approach, where each thread tries to form a logical column independently. These threads execute within a safe routing distance to avoid routing errors, resulting in a logical array with same size as produced by GCR. The second algorithm constructs a logical array in parallel based on divide-and-conquer approach. It constructs single logical column by generating its leftmost segments in parallel, and then it merges them in parallel. Each logical column is constructed one by one from left to right on the host array. To the best of our knowledge, this paper is the first contribution on parallel techniques for reconfiguration of degradable processor arrays.

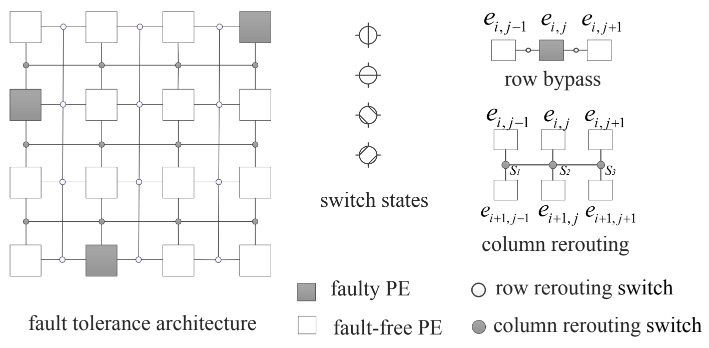
$e_{i,j-1}$ $e_{i,j}$ $e_{i,j+1}$
row bypass
$e_{i,j-1}$ $e_{i,j}$ $e_{i,j+1}$
S_1 S_2 S_3
$e_{i+1,j-1}$ $e_{i+1,j}$ $e_{i+1,j+1}$
column rerouting
switch states
fault tolerance architecture
faulty PE
fault-free PE
row rerouting switch
column rerouting switch

Fig. 1 Fault tolerant architecture and reconfiguration schemes

The rest of the paper is organized as follows. In section 2, we provide the important notations that will be used throughout the paper. We will also discuss the fault-tolerant architecture and provide a brief retrospection for the previous work. In section 3, we describe our multi-threading algorithm and in section 4, we present the divide-and-conquer algorithm. Simulation results and analysis are shown and discussed in Section 5. Finally, we conclude our work in section 6.

2 Preliminaries

2.1 Fault Tolerant Architecture and Reconfiguration Schemes

Let H indicate the physical (host) array where some of the PEs are defective. Assume the fault density of the physical array is ρ, then there are $N = (1 - \rho) \cdot m \cdot n$ fault-free PEs in a $m \times n$ physical array. A subarray constructed by changing the states of the switches is called logical/target array, denoted as T in this paper. There is no faulty PE in a logical array. The row (column) in physical array is called physical row (column), and the row (column) in logical array is called logical row (column).

Fig. 1 shows a 4×4 host array. In this fault-tolerant architecture, adjacent PEs are connected by links with four-port switches. The fault-tolerant reconfiguration is achieved by inserting several switches in the network, which allows the network to dynamically change the interconnections among PEs. Each square box in the host array represents a PE, whereas each circle represents a configuration switch. In this paper, the shaded boxes in the figures represent faulty PEs while unshaded ones represent fault-free PEs. As shown in Fig. 1, there are 4 states for each switch. Throughout this paper, $e_{i,j}(e'_{i,j})$ indicates the PE located at the position of (i, j) of the host (logical) array, where i is its row index and j is its column index. $row(u)$ $(col(u))$ denotes the physical row (physical column) index of the PE u. $u = v$ indicates that u is identical to v.

In a target array, any two neighboring PEs are interconnected through a group of physical links, and these physical links form a interconnection path between the two PEs. Since this architecture implements fault tolerance using simple switches, overlap of physical links is not allowed between any two interconnection paths, i.e., no physical link is shared by any two interconnection paths of a feasible target array. In order to generate target arrays without overlaps, two software control schemes, i.e., bypass and rerouting schemes, are utilized to guide reconfiguration algorithms. In particular, row bypass and column rerouting schemes are mostly employed in previous works. As shown in Fig. 1, if PE $e_{i,j}$ is faulty, then PE $e_{i,j-1}$ can communicate with PE $e_{i,j+1}$ and the data will bypass $e_{i,j}$ through an internal bypass link. This scheme is called *row bypass scheme*. In *column rerouting scheme*, PE $e_{i,j}$ can connect to $e_{i+1,j'}$ by changing states of relative switches (i.e., s_1, s_2 and s_3) where $|j - j'| \leq d$ and d is called the compensation distance [18–20]. If d is limited to 1, then the PE $e_{i,j}$ can connect to any one of the three PEs, i.e., $e_{i+1,j-1}$, $e_{i+1,j}$ or $e_{i+1,j+1}$, to form a logical column. The three PEs are called neighbors of $e_{i,j}$. Column rerouting scheme allows two non-faulty elements that are located in different physical columns to form a logical column. In order to reduce the complexity of the switching mechanisms and keep the cost of physical implementation low, it is necessary to maintain a small d. As same as most previous research works [19–21,23,24], d is also limited to 1 in this paper. *Column bypass* and *row rerouting* schemes can be similarly defined. With the bypass and rerouting schemes described above, a logical array can be formed from a host array by changing the states of the switches. In this paper, we study the degradable reconfiguration problem under the constraints of row bypass and column rerouting.

2.2 Problem Description and Previous Work

Problem $\mathcal{R}$. *Given an $m \times n$ mesh-connected host array with faults, find a maximum target array that contains the selected rows under the row bypass and column rerouting scheme.*

Let $R_0, R_1, \ldots, R_{m-1}$ be the rows of the given host array. Assume R_{r_0}, R_{r_1}, $\ldots$, $R'_{r_{s-1}}$ are the selected rows to construct a target array, where $0 \leq s \leq m$. Logical columns can be constructed under column rerouting scheme on the selected rows. These logical columns can be interconnected under the row bypass scheme to form a target array. A logical array T is said to contain R_{r_0}, $R_{r_1}, \ldots, R_{r_{s-1}}$ if each logical column in T contains exact one fault-free PE from each of the selected rows. On the other hand, if a physical row is not selected for inclusion into target array, all PEs in the row will be bypassed under row bypass scheme. This additional constraint on the selected rows simplifies the general reconfiguration problem, as each selected row must contribute one and only one PE to each logical column in this case. It has been proved that the problem $\mathcal{R}$ is not of NP-hard, and it is optimally solved by an algorithm named GCR (Greedy Column Rerouting) [19,20], meaning that GCR can produce maximum target array with selected rows. All operations in GCR are carried out on the adjacent neighbor sets of each fault-free PE $e_{i,j}$ in the row R_i. The set is defined as $Adj(e_{i,j})$=$\{e_{i+1,t} : e_{i+1,t}$ is unused fault-free PE and $|j-t| \leq 1$ $\}$. $Adj(e)$ consists of PEs in the next selected row and they can directly connect to e. The PEs in $Adj(e)$ are ordered in increasing columns index. The PE with the minimum column index in $Adj(e)$ is called the leftmost connectable PE for e.

Algorithm GCR constructs logical columns one by one in the left-to-right manner. Each logical column is constructed in the top-to-down manner. Without loss of generality, we assume that $R_{r_0}, R_{r_1}, \ldots, R_{r_{s-1}}$ $(0 \leq s \leq m)$ are the selected rows to construct the logical array. GCR starts by selecting the leftmost fault-free PE, say u, of the row R_{r_0} for inclusion into a logical column. Next, the leftmost PE, say v, in $Adj(u)$ will be connected to u. The process is repeated as follows: in each step, GCR tries to connect the current PE v to the leftmost PE which has not been previously examined in $Adj(v)$. If the connection is not made, then a logical column containing the current PE v cannot be formed. This leads to backtracking to the previous PE, say w, which was connected to v. The connection of w to the leftmost PE of $Adj(w) - v$ that has not been examined previously can now be attempted. The process will be repeated until either 1) a PE v in the row $R_{r_{s-1}}$ is connected to a PE in previous row $R_{r_{s-2}}$ or 2) GCR backtracks to the PE u in the row R_{r_0}. A single logical column is produced by GCR in each iteration. For the detailed description of GCR, see [20].

Algorithm GCR consists of two types of routing steps, *routing-forth* and *backtrack*, which are defined as follows,

1. *Routing-forth*: for current routing PE u, if $|Adj(u)| > 0$, a fault-free PE v can be found in $\mathcal{A}dj(u)$ such that u can directly connect to v. In this case, the routing result for current routing PE u is called *routing-forth*, as shown in Fig. 2(a), where $|Adj(u)| = 1$.
2. *Backtrack*: for current routing PE u, if $|Adj(u)| = 0$, then current routing PE cannot find a fault-free PE in $\mathcal{A}dj(u)$ to connect, and thus the algorithm has to backtrack to the previous position p. In this case, the routing result for current routing PE v is called *backtrack*. As shown in Fig. 2(b), $|Adj(u)| = 0$ because the PE v has been used for forming the previous logical column. Thus, the algorithm backtracks to PE p.

3 Parallel Reconfiguration Based on Multithread

In this section, we present our parallel algorithm PRM. It reconfigures the host array based on a multi-threading approach. Initially, we introduce some definitions to aid the understanding of the parallel algorithm. We define the process to construct a logical column staring with PE $e_{r_0,j}$, for $0 \leq j < n$, as

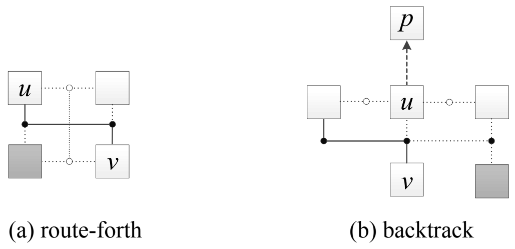

Fig. 2 Two types of routing steps.

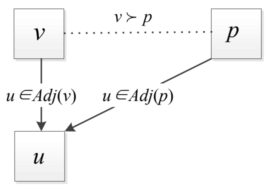

Fig. 3 Example of routing dependence.

a rerouting task denoted as t_j. Then, the algorithm GCR is comprised of n tasks, i.e., $t_1, t_2, ..., t_n$, and the n tasks are handled one by one serially.

Let S_j be the logical column constructed by task t_j. During the parallel reconfiguration process, S_j might be a partial logical column. The PE with largest row index in column S_j is called the current routing PE of column S_j. In algorithm GCR, when processing task t_j, each S_i for $0 \leq i < j$ is either a complete column or a empty column.

Given two fault-free PEs $v \in S_a$ and $p \in S_b$ such that $a < b$ and $row(v) = row(p)$. If $\mathcal{A}dj(v) \cap \mathcal{A}dj(p) \neq \phi$, then p's routing result depends on v's routing result, denoted as $v \succ p$. We call this dependence as routing dependence. As shown in Fig. 3, PE v and p belong to different physical columns, and $\mathcal{A}dj(v) \cap \mathcal{A}dj(p) \neq \phi$, thus $v \succ p$. The routing dependence indicates that v should routes before p. Otherwise, p's routing result will occupy a PE in $\mathcal{A}dj(v) \cap \mathcal{A}dj(p)$, which may results in backtracking on v. This may decreas the total number of logical columns, leading to loss in harvest.

In the parallel algorithm, the n tasks are executed in parallel by n threads. Hence, a task t_b can start before its prior task t_a $(a < b)$ is completed. The routing dependence between PEs results in relative threads dependence, which is defined as follows.

Let threads T_a and T_b $(a < b)$ construct columns S_a and S_b, respectively. Assume $v \in S_a$, $p \in S_b$ and $v \succ p$. Then T_b depends on T_a, denoted as $T_a \succ T_b$ or $dep(T_b) = T_a$. T_a is called the dependence thread of T_b.

In the parallel reconfiguration process, a thread, say T_b, must monitor its dependence thread, say T_a, to prevent incorrect routing due to threads dependence. Assume thread T_a is routing in the i-th selected row and thread T_b is routing in the i'-th selected row. The routing distance r_d between T_a and

T_b is calculated by $i - i'$. Routing under the condition $r_d < 1$ may lead to incorrect routing between two threads with dependence relationship. In order to avoid incorrect routing, thread T_b must keep a routing distance no less than 1 with its dependence thread $dep(T_b)$ (see lemma 1). This can be implemented using a conservative distance $c_d = 3$ as follows.

– If routing distance r_d between T_a and T_b is no less than 3, then T_b routes in the same way as GCR does.
– If routing distance r_d between T_a and T_b is less than 3, then thread T_b unmarks its current routing PE, say v, and backtracks to the prior PE $pre(v)$.

In this way, the routing distance r_d is no less than 1 during the parallel computation. Before starting to route, thread T_b must waits for T_a to perform a few routing steps first. During these steps, thread T_b is idle and we call this an *empty step*. Therefore, the parallel algorithm PRM consists of routing steps including *routing-forth*, *backtrack* and *empty step*.

The algorithm PRM is formed as follows. All tasks, i.e., $t_0, t_1, ..., t_{n-1}$, are regarded as n threads that will be executed in parallel. Each thread, say T_i, must monitors its dependence thread T_j $(0 < j < i)$, and routes based on the routing distance. If the dependence thread $dep(T_i)$ terminates, T_i finds a new monitor target until no active thread exists in $\{T_0, T_1, \cdots, T_{i-1}\}$.

A thread, say T_b, may terminates under 2 conditions: (1) T_b successfully forms a logical column, (2) thread T_b is routing on row E_{r_0} and needs to perform a backtrack step even if $r_d \geq c_d = 3$. Note that, performing a backtrack step from row E_{r_0} due to thread dependence will not lead to the termination of the thread. The algorithm terminates when all threads have completed the routing process. The pseudo-code of the PRM is as follows.

We present following lemma and theorem to prove that PRM and GCR can produce the same logical array.

Lemma 1. *For two dependence threads, if the routing distance between them is no less than 1 in their parallel processing, the two threads do not affect each other and can route correctly in parallel.*

Proof: For two threads T_i and T_j, assume $i < j$ and $T_i \succ T_j$. Let u and v be the current PE routing in T_i and T_j, respectively. Thus $row(u) - row(v) \geq 1$. In the next routing step, when both threads perform route-forth or backtrack, T_i will not route to a PE in $\mathcal{A}dj(v)$ from its current PE u. Also, T_j will not route to a PE in $\mathcal{A}dj(u)$ from its current PE v. This implies that T_i and T_j do not affect each other in the next routing step. If routing distance between T_i and T_j is no less than 1 in their parallel processing, the logical columns generated by T_i and T_j are identical to the ones produced by GCR. In other words, T_i and T_j can execute independently to produce logical columns. This concludes the lemma.

Theorem 1. *Algorithm PRM produces the same target array as algorithm GCR does.*

Proof: It is noteworthy that, in PRM, all tasks $t_0, t_1, ..., t_{n-1}$, are executed in parallel in the presence of thread dependences. We prove that both

```
Algorithm 1: PRM($H, r_0, r_1, ..., r_{s-1}, n$)
Input: host array $H$ of $m \times n$; index of selected rows: $r_0, r_1, \ldots, r_{s-1}$; number of
       threads $n$.
Output: Target array $T$.
begin
for each processor $i$ in $\{0, 1, ..., s-1\}$ parallel do
    $E_i \leftarrow$ set of fault-free elements in $R_{r_i}$;    /* initialization in parallel */
    for each $u \in E_i$ do
        $pre(u) \leftarrow null$;
for each processor $i$ in $\{0, 1, ..., s-1\}$ parallel do
    for each $u \in E_i$ do
        $Adj(u) \leftarrow \{v : v \in E_{r_{i+1}}, |col(u) - col(v)| \leq 1\}$;    /* initialize $Adj(u)$ */
for each processor $i$ in {1,2,..., n-1} parallel do
    $myid \leftarrow$ get processor id;  $dep(myid) \leftarrow myid - 1$;    /* initialize $dep(i)$ */
/* construct logical columns in parallel */
while threads $n$-1 is not finished do
    for each active thread $i$ in {0, 1, ..., n-1} parallel do
        PARA_Route($Adj, pre, dep$);    /* perform a routing step in parallel */
        synchronization;
end
```

```
Procedure: PARA_Route($Adj, pre, dep$)
begin
$myid \leftarrow$ get processor id;
if thread $dep(myid)$ is finished then
    $dep(myid) \leftarrow$ active thread of biggest id in {1,2,...,$myid$-1}
    if failed in find a valid $dep(myid)$ then
        $dep(myid) \leftarrow$null;
$e_0 \leftarrow$ current routing PE in thread $dep(myid)$;
$cur \leftarrow$ current routing PE in thread $myid$
if $row(e_0) - row(cur) \geq 3$ then
    if $|Adj(cur)| > 0$ then
        $q \leftarrow$ leftmost PE in $Adj(cur)$;  $pre(q) \leftarrow cur$;  $cur \leftarrow q$;  mark $q$ as used;
    else
        if $cur \notin E_{r_0}$ then $cur \leftarrow pre(cur)$; else $S_{myid} \leftarrow Null$; end thread; end
    if $v \in E_{r_{s-1}}$ then Return column $S_{myid}$; end thread; end
else
    unmarke $cur$;  $cur \leftarrow pred(cur)$;  /* backtrack caused by thread dependence */
end
```

algorithms produces same target array based on the fact that threads within dependence relationship do not impact each other during the generation of logical columns.

It is evident that thread T_0 produces the same logical column as GCR, because T_0 does not depend on any other thread.

Assume that routing results of tasks $t_0, t_1, ..., t_{i-1}$ by threads $T_0, T_1, ..., T_{i-1}$ are the same as routing results of tasks $t_0, t_1, ..., t_{i-1}$ by GCR. Also, assume that T_i monitors $T_k \in \{T_0, T_1, ..., T_{i-1}\}$, i.e., $T_k \succ T_i$, we will prove that T_k and T_i do not impact each other despite of dependence constraints.

Let e_i and e_k be the current PEs routing in threads T_i and T_k. There are two possible scenarios:

1. No backtracking occurs in T_k: in this case, $row(e_k)-row(e_i) \geq 3$ is satisfied during the parallel execution.
2. Backtracking occurs in T_k: ① if routing distance is no less than 3, then T_k will backtracks and T_i will route-forth, thus $row(e_k) - row(e_i) \geq 1$ after one routing step. ② If routing distance is less than 3, thread T_k backtracks and thread T_i also backtracks due to thread dependence, and the routing distance is kept unchanged. In addition, T_i will removes the *used-marker* from current routing PE before backtrack.

Therefore, $row(e_k)-row(e_i) \geq 1$ is satisfied at during the parallel execution regardless of whether backtracking occurs or not. Thus, threads T_k and T_i do not impact each other in column rerouting, see ***Lemma 1***. Therefore, we conclude that each thread T_j can produce the same result as GCR does in processing task t_j, for $0 \leq j < n$.

We now analyze the time complexity of algorithm PRM. Assume the host array is of size $m \times n$. In the best case, i.e., no backtracking occurs in any thread, the routing distance between any two neighboring threads is 3, thus the accumulated distance among all threads are $3(n-1)$. In addition, the last thread need an extra m routing steps to terminate. Therefore, the time complexity of the algorithm is bounded by $O(m+n)$. In the worst case, i.e., the routing step of backtrack occurs most frequently and no logical column can be formed, at most three routing steps can be performed on each PE, i.e., one routing-forth and two backtrack, thus the time complexity for the worst case is $O(mn)$.

4 Parallel Reconfiguration Based on Divide-and-Conquer

We now present a parallel reconfiguration algorithm based on divide-and-conquer, denoted as PRDC in this paper. In this section, a logical column is regarded as the connection of some partial logical columns, and each partial column is called a column segment (segment in short). Let $C_{i,j} =< e_i, e_{i+1}, \cdots, e_j >$, which indicates a feasible segment from the row R_i to R_j, $0 \leq i < j < s$, where e_k lies in R_k for $i \leq k \leq j$. e_i and e_j are called start PE and end PE of $C_{i,j}$, respectively.

PRDC involves three procedures, namely DIVIDE, CONQUER and MERGE. Initially, the original physical array is divided into some subarrays in procedure DIVIDE. Then, the leftmost column segments are generated in parallel by PRDC on the corresponding subarrays, and they are merged to form a logical column. By performing CONQUER and MERGE alternately, PRDC constructs logical columns one by one, from left to right, to form a logical array.

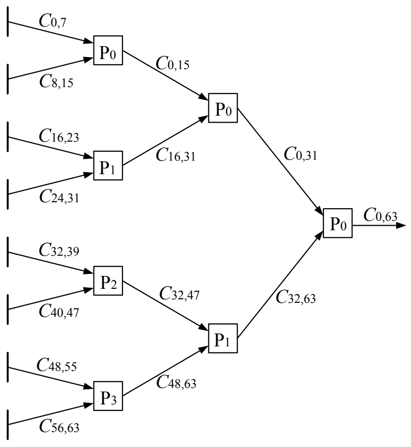

Fig. 4 Merge 8 segments using 4 processors, on a 64×64 host array.

4.1 Divide and Conquer

Assume that there are p processors labeled as $0, 1, ..., p-1$, respectively. Then the s rows selected from the original physical array is uniformly divided into p subarrays, such that each subarray contains about $\lceil s/p \rceil$ rows. Let $A_{t,b}$ indicate a subarray where t and b represent indexes of the top row and the bottom row of the subarray, respectively. The procedure DIVIDE is implemented in parallel, such that the index t_i of top row and the index b_i of bottom row are calculated by processor i $(0 \leq i < p)$. For example, a 10×10 host array can be divided into 4 basic subarrays, $A_{0,2}$, $A_{3,5}$, $A_{6,7}$ and $A_{8,9}$. The algorithm PRDC constructs a leftmost segment on each basic subarray in the same manner as used in algorithm GCR.

4.2 Merge

A complete logical column can be obtained by merging all segments constructed from the subarrays. Fig. 4 shows an example of parallel merging in which eight segments are merged using four processors. Each processor receives two segments and merges them into one larger segment for the subsequent merging stage.

Assume $C_{t,m}$ and $C_{m+1,b}$ are the leftmost segments on subarrays $A_{t,m}$ and $A_{m+1,b}$. The end PE of $C_{t,m}$ and the start PE of $C_{m+1,b}$ are denoted as e_m and e_{m+1}. Given $C_{t,m}$ and $C_{m+1,b}$, we construct $C_{t,b}$ by merging $C_{t,m}$ and $C_{m+1,b}$ according to the following routing conditions.

1. If $|col(e_m) - col(e_{m+1})| \leq 1$, i.e., e_m is able to connect e_{m+1} directly, then $C_{t,b}$ can be easily obtained by linking e_m and e_{m+1}. This type of merging is denoted as $C_{t,b} = C_{t,m} \oplus C_{m+1,b}$.

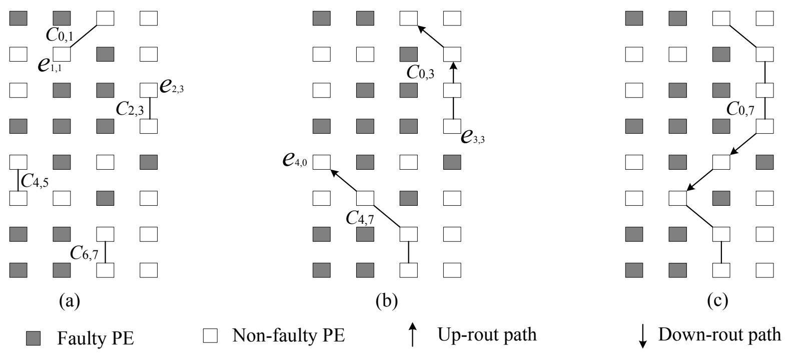

Fig. 5 (a) CONQUER: construct four column segments in parallel. (b) MERGE: construct $C_{0,3}$ and $C_{4,7}$ in parallel by merging. (c) MERGE: merge $C_{0,3}$ and $C_{4,7}$ to form $C_{0,7}$.

2. If $col(e_m) - col(e_{m+1}) > 1$, it implies that PE e_m lies to the right of PE e_{m+1}. Then, the algorithm routes down starting from PE e_m in the same manner as used in algorithm GCR, and the routing procedure terminates if it converges with a segment $C_{m+1,b}$ or arrives at row R_b. The routing process is called DOWN_ROUT.
3. If $col(e_{m+1}) - col(e_m) > 1$, i.e., PE e_m lies to the left of PE e_{m+1}. Then, the algorithm routes up starting from PE e_{m+1} towards $C_{t,m}$ until it converges with a segment $C_{t,m}$ or arrives at row R_t. The routing process is called *Up_rout*. It constructs a leftmost segment in the manner of down-to-up, instead of top-to-down as used in GCR.

The algorithm terminates if the current logical column violates the physical boundary of host array H. for more details, see the pseudo-code of PRDC.

Fig. 5 shows an example of constructing a logical column by performing CONQUER and MERGE. As shown in Fig. 5(a), four leftmost segments, i.e., $C_{0,1}$, $C_{2,3}$, $C_{4,5}$ and $C_{6,7}$, are produced in parallel by procedure CONQUER on 4 subarrays. As shown in Fig. 5(b), $C_{0,1}$ and $C_{2,3}$ are to be merged into one larger segment by procedure MERGE. Since the end PE $e_{1,1}$ in $C_{0,1}$ cannot directly connect to the start PE $e_{2,3}$ in $C_{2,3}$, and PE $e_{1,1}$ lies to the left of PE $e_{2,3}$, the algorithm performs UP_ROUT starting from PE $e_{2,3}$ to converge with $C_{0,1}$. This process terminates either it successfully converges with $C_{0,1}$ or it routes upward to row R_0 which is the first row for $C_{0,1}$. The up-routing paths are illustrated in the figure within the merged segment $C_{0,3}$. Similar up-routing is required for $C_{4,7}$. Fig. 5(c) shows the process of merging $C_{0,3}$ and $C_{4,7}$ into $C_{0,7}$. Since PE $e_{3,3}$ cannot connect to PE $e_{4,0}$ directly and PE $e_{3,3}$ lies to the right of PE $e_{4,0}$, the algorithm performs DOWN_ROUT starting from PE $e_{3,3}$ to converge with segment $C_{4,7}$. The down-routing paths and the final merged segment $C_{0,7}$ are illustrated in Fig. 5(c).

It is easy to understand that PRDC produces the same target array as generated by GCR, because PRDC still constructs the leftmost logical column but in a parallel way, instead of in sequence way as used in GCR. In addition,

Algorithm 2: PRDC($H, r_0, r_1, ..., r_{s-1}, p$)

```
Input: physical array $H$ of $m \times n$; index of selected rows: $r_0, r_1, \ldots, r_{s-1}$; number of
       processors $p$.
Output: Target array $T$.
begin
/* Divide: Divide the $s$ selected rows into $p$ subarrays in parallel */
for each processor $i$ in $\{0, 1, ..., p-1\}$ parallel do
    $Q \leftarrow s/p$;   $M \leftarrow s \ mod \ p$;
    if $i \leq M$ then
        $t_i \leftarrow i * Q + i$;   $b_i \leftarrow t_i + Q + 1$;
    else
        $t_i \leftarrow i * Q + M + 1$;   $b_i \leftarrow t_i + Q$;

while (1) do
    /* Conquer: Construct leftmost segment on each subarray in parallel */
    for each processor $i$ in $\{0, 1, ..., p-1\}$ parallel do
        $C_{t_i,b_i} \leftarrow$ Left Most Segment in $A_{t_i,b_i}$;
    /* Merge: Merge $p$ segments to form a logical column */
    $C \leftarrow$ PARA_Merge( $C_{t_i,b_i}$ for $0 \leq i < p$, $H$);
    if $C$ is an invalid column then break; end
    $T \leftarrow T \cup C$;
return $T$;
end
```

Procedure: PARA_Merge(C_{t_i,b_i} for $0 \leq i < p$, H) /* Merge p segments to form a complete logical column. */

```
Input: Original physical array $H$; $C_{t_i,b_i}$ $(0 \leq i < p)$ are segments from $p$ subarrays;
Output: a complete column $C_{0,k-1}$ merged from $p$ segments
begin
$num \leftarrow p$; /* number of segments */
while ($num > 1$) do

    for each processor $i$ in $\{0, 1, ..., \lceil num/2 \rceil - 1\}$ parallel do
        /* calculate bounds for the merged segment */
        $t \leftarrow t_{2i}$;   $mid \leftarrow b_{2i}$;   $b \leftarrow b_{2i+1}$;
        /* $C_{t,b}$ is obtained by merging segments $C_{t,mid}$ and $C_{mid+1,b}$. The end PE
        of $C_{t,mid}$ and the start PE of $C_{mid+1,b}$ are denoted as $e_{mid}$ and $e_{mid+1}$.*/
        if $|col(e_{mid}) - col(e_{mid+1})| \leq 1$ then
            $C_{t,b} = C_{t,mid} \oplus C_{mid+1,b}$.
        else
            if $col(e_{mid}) > col(e_{mid+1})$ then
                $C_{t,b} \leftarrow$ DOWN_ROUT($H$, $e_{mid}$, $C_{mid+1,b}$);   /* merge by
                down-route from PE $e_{mid}$ */
            else
                $C_{t,b} \leftarrow$ UP_ROUT($H$, $e_{mid+1}$, $C_{t,mid}$);   /* merge by up-route
                from PE $e_{mid+1}$ */

    $num \leftarrow \lceil num/2 \rceil$;

return $C_{0,k-1}$;
end
```

PRDC also involves in three types of routing steps, i.e., *routing forth*, *backtrack* and *empty step*. The *empty step* occurs in the procedure MERGE .

Now we analyze the time complexity of the algorithm PRDC. Assume p processors are available for parallel reconfiguration, then the original physical array is divided into p basic subarrays. The DIVIDE procedure runs in $O(1)$, since the p subarrays are calculated by p processors in parallel. There are about $\lceil s/p \rceil$ rows in each subarray, thus constructing a segment in procedure CONQUER requires $O(s/p)$ in the best case (where no backtracking occurs), and $O(sn/p)$ time in the worst case (where backtracking occurs most frequently). The procedure MERGE runs in $O(1)$ to connect two segments in the best case (directly merging) and thus takes $O(\log p)$ time to merge all segments. For the worst case, it runs in $O(l \cdot n)$ when merging two segments with lengths of l, where n is the number of physical columns. The MERGE runs in $\log p$ times and its i-th iteration runs in $O(l_i \cdot n)$, i.e., $O(\lceil \frac{s}{2^i} \rceil \cdot n)$. Thus, for the worst case it takes $O(n \cdot s)$ time to merge all segments, from $\sum_{i=0}^{\lceil \log p \rceil} \frac{s}{2^i} \leq 2 \cdot s$. Therefore, the running time of PRDC is bounded by $O(s \cdot n)$ in the worst case, while it executes in $O(n \cdot s/p)$ in the best case, from $O(1+n \cdot (s/p + \log p)) = O(n \cdot s/p)$.

5 Simulation Results and Analysis

Since both parallel algorithms achieve target array with same size as generated by GCR, we only evaluate the acceleration of the proposed parallel algorithms over GCR. As discussed before, routing steps are the basic operations for reconfiguration. In this section, we evaluate the two proposed parallel algorithms PRM and PRDC in terms of routing steps. We have implemented a simulator in C language to compute the routing steps and executed the parallel algorithms on a large number of randomly generated dataset that is used in [18–26]. The distribution of faults were generated by a uniform random generator, with fault density from 1% to 10%. In order to make a fair comparison, we keep the same assumptions as in [18–27]. Faults are only associated with PEs and communication infrastructure is assumed to be fault free. This assumption is justified since the switches and links use much less hardware resources when compared to the processors and are thus less vulnerable to defects.

The number of routing steps required by algorithm GCR is denoted as *G_steps*. The number of routing steps in the longest thread of algorithms PRM and PRDC are denoted as *M_steps* and *D_steps*, respectively. The acceleration of PRM over GCR is calculated as follows.

$$speedup = \frac{G\_steps}{M\_steps} \times 100\%$$

The acceleration of algorithm PRDC over GCR can be similarly calculated.

Table 1 Performance evaluation of PRM on host arrays, from 32×32 to 128×128, averaged over 40 random instances.

Host array		*ProcNum*	Acceleration		
size	fault density		*G_steps*	*M_steps*	*speedup*
32×32	1%	32	971	107	9.0
	5%		920	136	6.7
	10%		874	168	5.2
64×64	1%	64	3960	253	15.6
	5%		3794	355	10.7
	10%		3612	441	8.2
128×128	1%	128	16256	381	42.67
	5%		15413	892	17.28
	10%		14460	1047	13.81

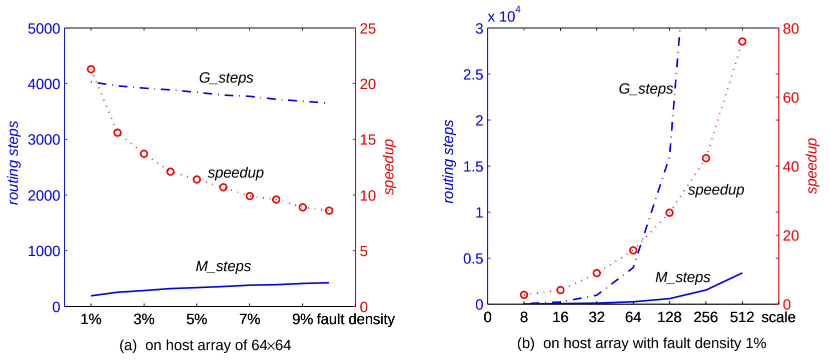

Fig. 6 Effects of fault density and array size on acceleration of PRDC, averaged over 40 random instances.

5.1 Performance Evaluation of PRM

The comparisons between PRM and GCR are presented in table 1. Data are collected on host arrays with sizes ranging from 32×32 to 128×128 and fault density ranging from 1% to 10%. In general, PRM significantly accelerates its serial counterpart. For the case of a 64×64 host array with fault density 1%, *G_steps* is 3950 and *M_steps* is 379, thus resulting in the speedup of 10.43 over GCR. In addition, the acceleration is more significant on large host arrays than on the smaller ones. It can also be observed that the speedup is more significant on less defective host arrays than on arrays with large number of defects, and this will be discussed later.

Fig. 6(a) shows the impact of fault density on the acceleration of algorithm PRM. Simulation results are collected on host arrays of 64×64 with fault density from 1% to 10%, and each case is averaged over 40 random instances. The value of *G_steps* tends to decrease with increasing fault density, while *M_steps* increases with the increasing fault density. This is because GCR only examines fault-free PEs while the increasing fault density decreases the number of

Table 2 Performance evaluation of PRDC on host arrays, from 64 × 64 to 256 × 256, averaged over 40 random instances.

Host array		*ProcNum*	Acceleration		
size	fault density		*G_steps*	*D_steps*	*speedup*
64×64	1%	32	3950	379	10.43
	5%		3764	440	8.56
	10%		3542	497	7.12
128×128	1%	64	16212	945	17.15
	5%		15322	1233	12.43
	10%		14680	1408	10.42
256×256	1%	128	64614	2374	27.22
	5%		61731	3340	18.49
	10%		59204	3855	15.36

nonfaulty PEs, which leads to the reduction in routing steps. For *M_steps*, the increasing fault density reduces the chance for a thread, say T_i, to construct a complete logical column. This leads to an increase in backtracking and rerouting steps of the thread T_i and other threads depending on T_i. Therefore, the value of *M_steps* increases with increasing fault density. In addition, speedup decreases with increasing fault density because the *G_steps* decreases while *M_steps* increases.

Fig. 6(b) illustrates the impact of increasing array sizes on the acceleration of algorithm PRM. Simulation results are collected on host arrays with size ranging from 8×8 to 64×64. The fault density is set to be 1% and each case is averaged over 40 random instances. It is easy to understand that the number of routing steps in both GCR and PRDC increases when the size of host array scales up. This is due to the fact that, the total number of PEs increases with increasing array size. In Fig. 6(b), *speedup* grows rapidly when the size of the host array increases as more threads are able to run in parallel.

5.2 Performance Evaluation of PRDC

Table 2 shows the performance comparison between PRDC and GCR in terms of *speedup*, on physical arrays with size ranging from 64 × 64 to 256 × 256 and fault densities ranging from 1% to 10%. Each host array with size of $m \times n$ is divided into $\lceil m/2 \rceil$ basic sub-arrays for our simulation, and $\lceil m/2 \rceil$ processors are employed for parallel reconfiguration. Each value in the table is averaged over 40 instances. In general, PRDC significantly accelerates its serial counterpart, i.e., algorithm GCR, which can be seen from the values of *speedup* in the table. For example, for the 256 × 256 host array with fault density 1%, algorithm PRDC achieves as much as 27 times acceleration. The value of *speedup* decreases with the increasing fault density. On host arrays with size of 128 × 128, the values of *speedup* are 17.15, 12.43 and 10.42 for fault densities 1%, 5% and 10%, respectively. This is due to the fact that, the increasing fault density makes it more difficult to merge two segments. In other words, operation UP_ROUT and DOWN_ROUT perform more routing steps to

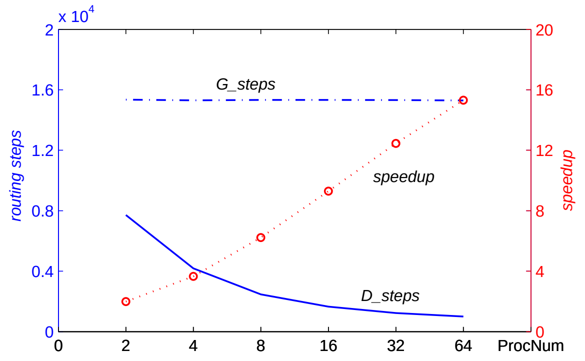

Fig. 7 Acceleration of PRDC over GCR on host arrays of 128×128 using different number of processors

merge two segments when fault density is large. In addition, the *speedup* tends to increase with increasing array size. For example, on host arrays with fault density 1%, the values of *speedup* are 10.43, 17.15 and 27.22 on host arrays with size of 64×64, 128×128 and 256×256, respectively. This is because, with the increasing size of host array, more subarrays could be obtained by dividing the original host array, which allows for more processors to be used for parallel reconfiguration.

Next,we examine the scalability of the parallel algorithms by varying the number of processors utilized in parallel reconfiguration. *ProcNum* indicates how many basic subarrays the original host array are divided into, as well as how many processors are utilized in parallel processing. In Fig. 7, simulation are conducted on host arrays with size of 128×128 and fault density of 5%. $ProcNum$ is set to 1, 2, 4, 8, 16, 32 and 64, respectively. Each value illustrated in the figure is averaged over 40 instances. The acceleration increases with the increasing $ProcNum$. For example, the *speedup* increases from 1.99 to 15.31 when $ProcNum$ increases from 2 to 64. This is because original host array is divided into more sub-arrays and this enables more processors to perform parallel computations to accelerate GCR.

6 Conclusions

We have presented two novel parallel algorithms to accelerate a conventional but widely used algorithm GCR for the reconfiguration of processor arrays. The first algorithm can implement the reconfiguration in parallel using multi-threading approach. Each thread is able to generate a logical column, and the multithreads execute within a safe rerouting distance, thus avoiding routing errors. The second algorithm consists of three procedures, Divide, Conquer and Merge. Divide and Conquer are used to split the host array into a number of subarrays and leftmost segments are generated on each subarray.

Next, the Merge connects these segments from separate sub-arrays to form logical columns. Both parallel algorithms can produce logical arrays with the same size as produced by GCR. We also provide simulation results on host arrays with different size. The first algorithm can achieve a speedup of up to 42 times on 128×128 host arrays. Notable accelerations are also demonstrated by the second algorithm.

Acknowledgements This work was supported by the National Natural Science Foundation of China under Grant No. 61173032 and No. 61070136, and Specialized Research Fund for the Doctoral Program of Higher Education under Grant No. 20131201110002.

References

1. S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. "An 80-tile sub-100-w teraflops processor in 65-nm cmos," *IEEE Journal of Solid-State Circuits,* vol. 43, no. 1, pp. 29-41, 2008.
2. S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. "TILE64-Processor: A 64-Core SoC with Mesh Interconnect," *In Digest of Technical Papers, IEEE International Solid-State Circuits Conference (ISSCC),* pp. 99, 598, 2008.
3. J. Antusiak, A. Trouv and K. Murakami, "A Comparison of DAG and Mesh Topologies for Coarse-Grain Reconfigurable Array," *Proc. of 2012 IEEE 26th International Parallel & Distributed Processing Symposium Workshops & PhD Forum (IPDPSW 2012)*, Shanghai, China, pp.220-226, May 2012.
4. P. Pepeljugoski, J. Kash, F. Doany, D. Kuchta, L. Schares, C. Schow, M. Taubenblatt, B. Offrein, and A. Benner, "Towards exaflop servers and supercomputers: The roadmap for lower power and higher density optical interconnects," *Proc. of 2010 36th European Conference and Exhibition on Optical Communication (ECOC)*, pp. 1-14, Sept. 2010.
5. S. Kamil, L. Oliker, A. Pinar and J. Shalf, "Communication Requirements and Interconnect Optimization for High-End Scientific Applications," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 21, no. 2, pp. 188-202, 2009.
6. D. Ajwani, S. Ali, J.P. Morrison, "Graph Partitioning for Reconfigurable Topology," *2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)*, Shanghai, China, pp. 836-847, May 2012.
7. L. Schares, X. J. Zhang, R. Wagle, D. Rajan, P. Selo, S.-P. Chang, J. R. Giles, K. Hildrum, D. M. Kuchta, J. L. Wolf, and E. Schenfeld, "A reconfigurable interconnect fabric with optical circuit switch and software optimizer for stream computing systems," *Proc. of 2009 Conference on Optical Fiber Communication (OFC)*, pp.1-3, 2009.
8. M. B. Stensgaard and J. Spars, "ReNoC: A Network-on-Chip Architecture with Reconfigurable Topology ," *Proc. of Second ACM/IEEE International Symposium on Networks-on-Chip(NoCS 2008)*, Newcastle upon Tyne, England, pp.55-64,2008.
9. J. H. Collet, P. Zajac, M. Psarakis, and D. Gizopoulos, "Chip Self-Organization and Fault Tolerance in Massively Defective Multicore Arrays," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 2, pp. 207-217, march 2011.
10. Z. Liu, J. Cai, M. Du, L. Yao, Z. Li, "Hybrid Communication Reconfigurable Network on Chip for MPSoC," *Proc. of 2010 24th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pp. 356-361, Perth, Australia, 2010.
11. J. H. Collet, M. Psarakis, P. Zajac, D. Gizopoulos, and A. Napieralski, "Comparison of Fault-Tolerance Techniques for Massively Defective Fine- and Coarse-Grained Nanochips," *Proc. of 16th International Conference on Mixed Design of Integrated Circuits and Systems,(MIXDES 2009)*, Lodz, Poland, pp.23-30,June 2009.

12. A. Avakian, J. Nafziger, A. Panda and R. Vemuri, "A Reconfigurable Architecture for Multicore Systems," *Proc. of 2010 24th IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW 2010)*, Atlanta, GA, USA, pp.1-8 , April 2010
13. M. Modarressi, H. Sarbazi-Azad, A. Tavakkol, "An efficient dynamically reconfigurable on-chip network architecture," *Proc. of the 47th Design Automation Conference (DAC'10)*, pp.310-313, 2010.
14. Y. Y. Chen, S. J. Upadhyaya and C. H. Cheng, "A comprehensive reconfiguration scheme for fault-tolerant VLSI/WSI array processors," *IEEE Transactions on Computers*, vol. 46, no. 12, pp. 1363-1371, Dec. 1997.
15. T. Horita and I. Takanami, "Fault-tolerant processor arrays based on the 1.5-track switches with flexible spare distributions," *IEEE Transactions on Computers*, vol. 49, no. 6, pp. 542-552, June 2000.
16. Li Zhang, "Fault-Tolerant meshes with small degree," *IEEE Transactions on Computers*, col. 51, No. 5, pp. 553-560, May 2002.
17. M. Fukushi and S. Horiguchi, "A Self-reconfigurable Hardware Architecture for Mesh Arrays Using Single/double Vertical Track Switches", *IEEE Transactions on Instrumentation and Measurement*, vol. 53, no. 2, pp. 357-367, April 2004.
18. S. Y. Kuo and I. Y. Chen, "Efficient reconfiguration algorithms for degradable VLSI/WSI arrays," *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 10, pp. 1289-1300, Oct. 1992.
19. C. P. Low and H. W. Leong, "On the reconfiguration of degradable VLSI/WSI arrays," *IEEE Transactions Computer-Aided Design of integrated circuits and systems*, vol. 16, no. 10, pp. 1213-1221, Oct. 1997.
20. C. P. Low, "An efficient reconfiguration algorithm for degradable VLSI/WSI arrays," *IEEE Transactions on Computers*, vol. 49, no. 6, pp. 553-559, June 2000.
21. J. Wu and T. Srikanthan, "Reconfiguration Algorithms for Power Efficient VLSI Subarrays with 4-port Switches", *IEEE Transactions on Computers*, vol. 55, no. 3, pp. 243-253, March 2006.
22. J. Wu and T. Srikanthan, "Integrated row and column re-routing for reconfiguration of VLSI arrays with 4-port switches", *IEEE Transactions on Computers*, vol. 56, no. 10, pp. 1387-1400, Oct. 2007.
23. M. Fukushi, Y. Fukushima, and S. Horiguchi, "A genetic approach for the reconfiguration of degradable processor arrays", in *Proc. of 20th IEEE International Symposium on Defect Fault Tolerance VLSI System.*, pp. 63-71, June, 2005.
24. J. Wu, T. Srikanthan, and X. Han, "Preprocessing and Partial Rerouting Techniques for Accelerating Reconfiguration of Degradable VLSI Arrays," *IEEE Transactions on Very Large Scale Intergration (VLSI) Systems*, vol. 18, no. 2, pp. 315-319, August 2010.
25. G. Jiang, J. Wu and J. Sun. "Non-Backtracking Reconfiguration Algorithm for Three-dimensional VLSI Arrays," in *Proc. of 2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, Singapore, pp.362-367, Dec. 2012.
26. G. Jiang, J. Wu and J. Sun, "Efficient Reconfiguration Algorithms for Communication-Aware Three-dimensional Processor Arrays," *Parallel Computing (2013)*, doi: http://dx.doi.org/10.1016/j.parco.2013.04.005.
27. L. Zhang, Y. Han, Q. Xu, X. Li and H. Li, "On Topology Reconfiguration for Defect-Tolerant NoC-Based Homogeneous Manycore Systems," *IEEE Transactions on Very Large Scale Intergration (VLSI) Systems*, vol. 17, no. 9, pp. 1173-1186, 2009.
28. N. R. Mahapatra and S. Dutt, "Hardware-efficient and Highly Reconfigurable 4- and 2-track Fault-tolerant Designs for Mesh-connected Arrays", *Journal of Parallel and Distributed Computing*, vol. 61, no. 10, pp. 1391-411, Oct. 2001.
29. I. Takanami, "Self-reconfiguring of 1.5-track-switch Mesh Arrays with Spares on One Row and One Column by Simple Built-in Circuit", *IEICE Transactions on Information and Systems*, vol. E87-D, no. 10, pp. 2318-2328, 2004.
30. J. Wu, T. Srikanthan, and Schroder Heiko, "Efficient Reconfigurable Techniques for VLSI Arrays with 6-port Switches," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 8, pp. 976-979, August 2005.
31. P. Parvathala, K. Maneparambil, W. Lindsay, "FRITS - A Microprocessor Functional BIST Method", *Proc. of IEEE International Test Conference* , pp 590-598, 2002.

32. L. Chen, S. Ravi, A. Raghunathan, S. Dey, “A Scalable Software-Based Self-Test Methodology for Programmable Processors”, *Proc. of IEEE/ACM Design Automation Conference (DAC 2003)*, pp. 548-553, 2003.
33. A. Paschalis and D. Gizopoulos, “Effective Software-Based Self-Test Strategies for On-Line Periodic Testing of Embedded Processors”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* , vol. 24, no 1, pp. 88-99, 2005.
34. F. Corno, E. Sanchez, M. Sonza Reorda, G. Squillero, “Automatic Test Program Generation - a Case Study”, *IEEE Design & Test of Computers* , vol. 21, no. 2, pp. 102-109, 2004.
35. M. Hatzimihail, M. Psarakis, D. Gizopoulos and A. Paschalis, “A Methodology for Detecting Performance Faults in Microprocessor Speculative Execution Units via Hardware Performance Monitoring”, *Proc. of IEEE International Test Conference* , paper 29.3, 2007.
36. D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan and S. Ravi, “Systematic Software-Based Self-Test for Pipelined Processors”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* , vol. 16, no 11, pp. 1441-1453, 2008.