ELSEVIER
Theoretical Computer Science 197 (1998) 171–188
Theoretical Computer Science

Testing and reconfiguration of VLSI linear arrays[1]

Roberto De Prisco [a,*], Angelo Monti [b], Linda Pagli [c]

[a] *Laboratory for Computer Science, Massachussetts Institute of Technology, 545 Technology Square, 43-368, Cambridge, MA 02139, USA*
[b] *Dipartimento di Scienze dell'Informazione, Università degli studi di Roma "La Sapienza", via Salaria 113, 00198 Roma, Italy*
[c] *Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy*

Received March 1995; revised April 1996
Communicated by G. Aussiello

Abstract

Achieving fault tolerance through incorporation of redundancy and reconfiguration is quite common. In this paper we study the fault tolerance of linear arrays of N processors with k bypass links whose maximum length is g. We consider both arrays with bidirectional links and unidirectional links.

We first consider the problem of testing whether a set of n faulty processors is catastrophic, i.e., precludes reconfiguration. We provide new testing algorithms which improve and generalize known testing algorithms. For bidirectional arrays we provide an $O(kn)$ time testing algorithm and for unidirectional arrays we provide an $O(n)$ time algorithm for the case $k=1$, and an $O(kn \log k)$ time algorithm, for the case $k>1$.

When the fault pattern is not catastrophic we study the problem of finding an *optimal* reconfiguration of the array. We consider optimality with respect to two parameters: the size of the reconfigured array and the number of redundant links to activate. Considering optimality with respect to the size of the reconfigured array, we prove that the problem is NP-hard in the strong sense if the bypass links are bidirectional, while it can be solved in $O(kng)$ time if the bypass links are unidirectional. Considering optimality with respect to the number of bypass links to activate, we prove that the problem can be solved in $O(kn)$ time if the bypass links are bidirectional, and in $O(kng)$ time if the bypass links are unidirectional. © 1998—Elsevier Science B.V. All rights reserved

Keywords: Array processors; Catastrophic fault patterns; Fault tolerance; Fault detection; Reconfiguration algorithms

* Corresponding author. E-mail: robdep@theory.lcs.mit.edu.

[1] Part of this work appeared in Proceedings of the *3rd Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science, vol. 709 (1993) 553–564.

PII S0304-3975(97)00238-7

1. Introduction

In a linear array of N processing elements, one faulty element is sufficient to stop the flow of information from one side to the other. Without the provision of fault-tolerance capabilities, the yield of VLSI chips for such an architecture would be so poor that its production would be unacceptable. A lot of research has been devoted to the design of fault-tolerant parallel architectures. The most important techniques for this purpose can be divided into two main groups. The first one does not make use of redundancy on the given architecture but tries to simulate the global functioning using the healthy part of the machine (e.g., [10, 13]). This approach uses simulation algorithms which should guarantee the same functionality with a reasonable slowdown in time. The second group includes the techniques that do add redundancy to the given architecture. This approach maintains the desired structure by isolating faults and activating certain spare links or processors (e.g., [3, 7, 9, 19, 21–23]).

Our approach belongs to the second group. We consider linear arrays with both spare processors and links. Beside the regular links connecting neighboring processing elements, extra links, called bypass links, are included in a regular fashion. These redundant links can be activated in a reconfiguration phase to bypass faulty processors. In this work we make the following assumptions:
- Only processors can fail.
- Faults are total, that is, faulty processors cannot route or compare.
- Faults are static, that is, faulty processors cannot be repaired.

Redundant processors elements are used to replace any faulty processor. Redundant links are used to bypass the faulty processors and, possibly, to reach redundant processors used as replacement.

There are essentially two different ways of allocating extra links to the given architecture, namely:

(i) There are spare communication lines that any working processor can use to bypass faulty processors. Communication is realized through switches located before and after each processor. These switches can be activated to reconfigure the array. This approach has been extensively studied [2, 3, 9, 17, 19, 22].

(ii) A fixed set of spare links is dedicated to each processor. In this case a multiplexer located inside the processor element can route messages onto one of its private, spare links in case of faults. This approach was first introduced for tree and ring architectures [12, 20], and later extended to linear arrays [4, 5, 13–16, 18].

In this work we follow the second strategy, which is more suitable for the important case of production-time reconfiguration of faulty devices [16]. Indeed, the first strategy allows a general on-line reconfiguration at the expense of a larger propagation time along the spare lines. This propagation time may become intolerably large in a fixed communication pattern, in particular, when the message must traverse a chain of switches to bypass a sequence of consecutive faulty processors.

Both strategies require that a switch (or multiplexer) be traversed at the input and output of each processing element. We assume that both the number of spare links, and the length of the longest link, are reasonably small, so that the circuitry added to each processor is simple, and the communication delays along the links are negligible. Therefore, the total propagation time depends only on the number of processors, which is fixed.

Since any processing element in the array may be faulty, each one of them has to be provided with the bypass links. The connections to these bypass links must occupy different tracks in the chip. Hence, the total area required by the interconnection network is proportional to the length of the array, the number of extra links per processor, and the length of the longest link. Finally, note that in each processor, a modest amount of circuitry (multiplexer and self-repairing control unit) must be devoted to implement the proposed routing discipline. Although, in principle, this discipline could be applied to any chip, it is clearly advisable when the functionality of a processing element is not too elementary.

This approach has some inherent limits. Under a realistic assumption that the length of the longest link is small with respect to the number of processing elements, regardless of any amount of redundancy, there are sets of faults occurring at strategic positions which affect the chip in an non-reparable way (see [13]). Such sets of faults, called *catastrophic fault patterns* (CFP for short), have been extensively studied in [4, 16, 18], but only for the specific case in which the number of faults in the pattern is exactly the length g of the longest bypass link. To have at least g faults, is a necessary condition for a fault pattern to be catastrophic [13].

Nayak et al. [16] devised an $O(g^2)$ time algorithm for testing fault patterns consisting of exactly g faults, for redundant arrays with a single bypass link of length g. This algorithm has been improved in [14] with an $O(kg)$ time algorithm that solves the problem in the more general case of $k \geqslant 1$ bypass links. The problem representation and the solution techniques presented in previous works are not easily extensible to the general case of any number m, $g \leqslant m \leqslant N$, of faults.

In this paper, following a completely different approach, we consider the more general problem of testing whether a fault pattern consisting of n faults, [2] is catastrophic. In addition, when a fault pattern is not catastrophic, we consider the problem of finding optimal reconfiguration strategies, where optimality is with respect to either the number of processors in the reconfigured array (the reconfiguration is optimal if such a number is maximized) or the number of redundant links to activate in order to reconfigure the array, i.e., the amount of work needed to reconfigure the array (the reconfiguration is optimal if such a number is minimized).

Our results are the following:

The problem of testing whether a set of n faulty processing elements is catastrophic for a redundant array with k bypass links can be solved in time $O(kn)$ when the links in the array are bidirectional, and in time $O(nk \log k)$ if $k > 1$, or in time $O(n)$ if $k = 1$, when the links in the array are unidirectional.

[2] We remark that we will actually consider, without loss of generality, fault patterns of m faults, $m \geqslant n$, subdivided into n blocks of consecutive faulty processors.

The problem of finding a reconfiguration strategy that is optimal with respect to the size of the reconfigured array is NP-hard in the strong sense, when the links are bidirectional, while it can be solved in time $O(kng)$, where g is the length of the longest bypass link, when the links are unidirectional.

The problem of finding a reconfiguration strategy that is optimal with respect to the number of redundant links used in the reconfiguration, is solvable in $O(kn)$ time when the links are bidirectional, and in $O(kng)$ time when the links are unidirectional.

We provide algorithms for all the cases in which the problem can be solved, i.e., all but the problem of finding an optimal reconfiguration strategy in arrays with bidirectional links, where optimality is with respect to the size of the reconfigured array.

This paper is organized as follows: Basic concepts and a formal definition of the problem are introduced in Section 2. A testing algorithm for arrays with bidirectional links is given in Section 3. In Section 4, a testing algorithm for arrays with unidirectional links is provided. Section 5 contains results on reconfiguration strategies that are optimal with respect to the size of the reconfigured array. Finally, Section 6 contains results on reconfiguration strategies that are optimal with respect to the number of redundant links used. Section 7 contains concluding comments and open questions.

2. Preliminaries

The basic components of a redundant linear array are the *processing elements*, or simply processors, and the *links*. There are two kinds of links: *regular* or *bypass*. Regular links exist between neighboring processors, while the bypass links connect non-neighbors processors. The bypass links are used only for reconfiguration purposes when faulty processors are detected.

More precisely, let $A=\{p_1,\dots,p_N\}$ denote a linear array of identical processing elements connected by regular links (p_i,p_{i+1}), $1\leqslant i<N$. Let $G=\{g_1,\dots,g_k\}$ be an ordered set of integers such that $g_1<g_2<\cdots<g_k$. We say that A has *redundancy* G if, for each g_t, $1\leqslant t\leqslant k$, there is a bypass link (p_i,p_{i+g_t}), $1\leqslant i\leqslant N-g_t$. Notice that the set G does not contain the regular link. We denote by g the length of the longest bypass link, i.e., $g=g_k$.

At the extremities of the array two special processors, called I (for Input) and O (for Output), are responsible for the I/O functions of the system. We assume that I is connected to $p_1,\dots,p_g$ while O is connected to $p_{N-g+1},\dots,p_N$ so that bottlenecks at the borders of the array are avoided.

Example 1. Fig. 1 shows a linear array of 20 processing elements with redundancy $G=\{4\}$.

We refer to this structure as a *redundant linear array* or as a *redundant array* or simply as an *array*. The array is called *bidirectional* or *unidirectional* according to the

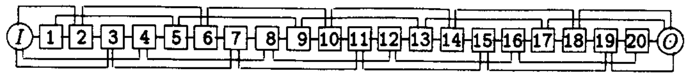

Fig. 1. A linear array of processors.

nature of its links. We admit faults occurring in the processors only (i.e., both I and O and the links always operate correctly). We refer to a processor p_i as processor i or simply as p_i.

Definition 1. For a redundant linear array A, a fault pattern F is an ordered set of pairs of positive integers $F = \{(f_1, l_1), (f_2, l_2), \ldots, (f_n, l_n)\}$, where $f_i + l_i < f_{i+1} \leqslant N - l_n + 1$, $1 \leqslant i < n$.

Each pair (f_i, l_i) identifies the block of faulty processors $p_{f_i}, p_{f_i+1}, \ldots, p_{f_i+l_i-1}$. Hence, a *faulty processor* p_z is such that $f_i \leqslant z < f_i + l_i$ for some i, $1 \leqslant i \leqslant n$. Non-faulty processors are *working* processors. A *path* from a working processor i_0 to a possibly fault processor i_{s+1} is a sequence of processors $i_0, i_1, \ldots, i_s, i_{s+1}$ such that, for each $j = 0, 1, \ldots, s$, processor i_j is a working processor connected by a link to processor i_{j+1} and $i_j = i_z$ if and only if $j = z$, $0 \leqslant j$, $z \leqslant s + 1$ (i.e., a processor is used only once). The length of the path is $s + 1$. An *escape path* is a path from I to O. We represent paths in the following way: since the flow of computation usually goes from processor i to processor $i + 1$, it is enough to indicate those processors for which the computation does not continue on the consecutive processor. Formally we give the following definition of a path.

Definition 2. A path P is represented as a triple consisting of a starting processor p_u, an ending processor p_v and a set of pairs of integers $\{(e_1, a_1), (e_2, a_2), \ldots, (e_q, a_q)\}$, where $1 \leqslant e_i \leqslant N$, $e_i \neq e_j$ if $i \neq j$ and $-k \leqslant a_i \leqslant k$, for each $i = 1, 2, \ldots, q$.

Processor e_i, $i = 1, 2, \ldots, q$, has active a link that is not the regular one. The active link of processor e_i is defined according to a_i, namely:
- if $a_i = 0$ the active link is from p_{e_i} to $p_{e_i - 1}$,
- if $a_i < 0$ the active link is from p_{e_i} to $p_{e_i - g_{(-a_i)}}$,
- if $a_i > 0$ the active link is from p_{e_i} to $p_{e_i + g_{a_i}}$.

All other processors have their regular link active. The path represented by P is the sequence of processors obtained starting from p_u and following the active links. It is obvious that this sequence must not contain a faulty processor, except, possibly, the last processor p_v. An escape path is a path P for which $p_u = I$ and $p_v = O$. In representing escape paths we will omit processors I and O. Since by activating the link a_i of the processor e_i, for $i = 1, 2, \ldots, q$, we reconfigure the system (or achieve a path from p_u to p_v), we call the set $\{(e_1, a_1), (e_2, a_2), \ldots, (e_q, a_q)\}$ *reconfiguration set*.

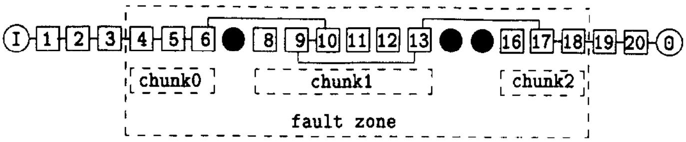

Fig. 2. A fault pattern and the chunks.

Definition 3. Given a redundant array A, a fault pattern is catastrophic for A if and only if no escape path exists.

Given a fault pattern for a redundant array A, we focus our attention on that part of A beginning at processor p_{f_1-g+1} and ending at processor $p_{f_n+l_n+g-2}$. We call *fault zone* this part of the array. Moreover, since all the processors are indistinguishable, without loss of generality, we will assume that the fault zone begins at processor p_1, i.e., $f_1 = g$.

A block of maximum length of working processors in the fault zone will be called *chunk*. More formally, we give the following definition:

Definition 4. Given an array A with redundancy G and a fault pattern F, $chunk_i$, $1 \leqslant i \leqslant n-1$, is the block of processors between $f_i + l_i - 1$ and f_{i+1}, i.e., the block $p_{f_i+l_i}, \ldots, p_{f_{i+1}-1}$. Moreover, $chunk_0$ is the block of processors $p_{f_1-g+1}, \ldots, p_{f_1-1}$, i.e., the first $g-1$ processors of the fault zone, and $chunk_n$ is the block of processors $p_{f_n+l_n}, \ldots, p_{f_n+l_n+g-2}$, i.e., the last $g-1$ processors of the fault zone.

Example 2. Consider the fault pattern $F = \{(7,1),(14,2)\}$ for a bidirectional linear array of 20 processors with link redundancy $G = \{4\}$. Then, the fault zone begins at processor p_4 and ends at processors p_{18}. There are three chunks: $chunk_0$ begins at processor p_4 and ends at processor p_6; $chunk_1$ begins at processor p_8 and ends at processor p_{13}; $chunk_2$ begins at processor p_{16} and ends at processor p_{18}. An escape path P is the sequence of processors $I, p_1, p_2, p_3, p_4, p_5, p_6, p_{10}, p_9, p_{13}, p_{17}, p_{18}, p_{19}, p_{20}, O$. The reconfiguration set that achieves this escape path is $\{(6,1),(10,0),(9,1),(13,1)\}$. Fig. 2 shows the fault pattern F, the fault zone, the chunks and the escape path P. Notice that only the active links of the processors in the escape path are drawn and that $f_1 \neq g$.

When a fault pattern is not catastrophic, we are interested in finding escape paths. Depending on the fault pattern there can exist several escape paths. We are interested in finding those escape paths that are *optimal* with respect to either the size of the reconfigured array or the number of redundant links to be activated to reconfigure the array.

In the former case, optimality is achieved when the size of the reconfigured array is maximized, i.e., when the number of processors in the escape path that reconfigures the array is maximized. In this case, an optimal escape path is called a *maximum escape path*, and a reconfiguration set that achieves a maximum escape path is called a *maximum reconfiguration set*.

In the latter case, optimality is achieved when the number of redundant links that we have to activate in order to reconfigure the array is minimized. In this case, an optimal escape path is called *minimum escape path*, and a reconfiguration set that achieves a minimum escape path is called a *minimum reconfiguration set*.

Example 3. Consider the fault pattern F given in Example 2. The escape path P of the example, has length 13. It is not a maximum escape path. Indeed, it is easy to see that the escape path $P_{\mathrm{M}} = \{(6,1),(10,0),(9,0),(8,1),(13,1)\}$ of 15 processors is a maximum escape path. Neither P nor P_{M} are minimum escape paths. The escape path $P_{\mathrm{m}} = \{(6,1),(13,1)\}$ is a minimum escape path since it uses only two redundant links and there are no escape paths that use only one link.

3. A testing algorithm for bidirectional arrays

Let A be a bidirectional array of N processors with redundancy $G = \{g_1, \ldots, g_k\}$, and let F be a set of m faults grouped into $n \leqslant m$ blocks of consecutive faulty processors. A straightforward way to test if F is catastrophic for A is to consider the graph whose vertices are the working processors and whose set of edges is given by the links between working processors, and to test if vertices I and O are connected in such a graph. By applying a standard algorithm for the test of connectivity we would obtain an algorithm with $O((N-m)k)$ time complexity. However, in the case of redundant arrays, the usual assumption is that N is much greater than m. We propose an algorithm running in $O(nk)$ time.

The idea is to represent the problem by a graph whose set of vertices is given by the chunks of working processors and whose size is linear in the number of faults, and to test the connectivity of I and O in such a graph. More formally, we construct a graph $H = (V, E)$ as follows: The set V of vertices is $\{C_0, C_1, \ldots, C_n\}$, where C_i's represent the chunks of F. We will write $p_x \in C_i$ to indicate that the processor p_x belongs to $chunk_i$. For each i and j, $0 \leqslant i, j \leqslant n$, $i \neq j$, the edge (C_i, C_j) belongs to E if and only if there are two processors, $p_x \in C_i$ and $p_y \in C_j$, and an integer t, $1 \leqslant t \leqslant k$, such that $|y - x| = g_t$, that is, if and only if the two processors are connected in A by a bypass link.

We call the graph H the *derived graph* of the fault pattern F. By definition of derived graph it follows that

Fact 5. *A fault pattern F is not catastrophic for an array A, if and only if C_0 is connected with C_n in the derived graph.*

Fig. 3 shows an algorithm, called GRAPH, which constructs the derived graph. Inputs to GRAPH are the fault pattern F and the redundancy G. The output is the derived graph represented by its adjacency lists. In addition to the code shown, the following should be noted: the adjacency list of node C_i is $L(C_i)$, for $i = 0, 1, \ldots, n$ and initially

```
GRAPH(F, G, H)
  for t = 1 to k do
    j = 1
    for i = 0 to n − 1 do
        while x_i + g_t > y_j do
            j = j + 1
        endwhile
        while j ≤ n and x_j ≤ y_i + g_t do
            L(C_i) = L(C_i) ∪ C_j
            L(C_j) = L(C_j) ∪ C_i
            j = j + 1
        endwhile
        if y_{j−1} ≥ x_{i+1} + g_t then j = j − 1
    endfor
  endfor
  H is given by L(C_i) for i = 1, ..., n
return(H)
```

Fig. 3. Algorithm GRAPH.

$L(C_i)$ is empty; the first and the last processor of chunk C_i are denoted by x_i and y_i, respectively.

Lemma 6. *Algorithm* GRAPH *constructs the derived graph.*

Proof. To prove the lemma we have to show that C_s is inserted into $L(C_i)$ and C_i is inserted into $L(C_s)$ if and only if (C_i, C_s) is an edge of the derived graph.

Assume that (C_i, C_s) is an edge of the derived graph. Without loss of generality, let $i<s$. By definition of derived graph, there exist two integers, z and t, such that $1 \leqslant t \leqslant k$, $x_i \leqslant z - g_t \leqslant y_i$, and $x_s \leqslant z \leqslant y_s$. Hence, we have $x_i + g_t \leqslant y_s$ and $x_s \leqslant y_i + g_t$.

Consider the ith iteration of the internal **for**. At the beginning of this iteration at least one among $x_i + g_t \leqslant y_{j-1}$ and $j = i+1$ holds. Hence, j is incremented until $x_i + g_t \leqslant y_j$, and for all j' such that $x_i + g_t \leqslant y_{j'}$ and $x_{j'} \leqslant y_i + g_t$, chunk $C_{j'}$ is inserted into $L(C_i)$ and chunk C_i is inserted into $L(C_{j'})$. Hence, also C_s is inserted into $L(C_i)$ and C_i is inserted into $L(C_s)$.

Assume that C_s is inserted into $L(C_i)$ and C_i is inserted into $L(C_s)$. Without loss of generality, let $i<s$. There must exist a g_t such that $x_i + g_t \leqslant y_s$ and $x_s \leqslant y_i + g_t$. This implies that there exists an integer z such that $x_i + g_t \leqslant y_i + g_t$ and $x_s \leqslant z \leqslant y_s$. Hence, $p_{z-g_t} \in C_i$ and $p_z \in C_s$, and thus, by definition of derived graph, $(C_i, C_s) \in E$. □

We remark that some redundant information may be present in the adjacency lists (i.e., an edge may appear more than once in the same list), however, this does not affect the order of magnitude of the size of the lists and thus the time complexity of the testing algorithm.

Theorem 7. *The problem of testing whether a fault pattern of n blocks is catastrophic for a bidirectional redundant array with k bypass links is solvable in time* $O(kn)$.

Proof. By Fact 5 and Lemma 6, it is sufficient to show that the algorithm GRAPH requires $O(kn)$ time. Indeed, there are well-known algorithms that test connectivity in a graph (V,E) in $O(|V|+|E|)$ time.

Since the outer **for** loop requires $O(k)$ time, we simply have to prove that the inner **for** loop requires $O(n)$ time. Let z_i be the increment of variable j at the ith iteration of the internal **for**. Clearly, the ith iteration of the **for** costs $O(z_i)$ and $\sum_{i=0}^{n-1} z_i = n$. Hence, the internal **for** requires $O(n)$ time. □

4. A testing algorithm for unidirectional arrays

In this section we study the problem of testing whether a fault pattern is catastrophic for a redundant array with unidirectional links. In this case the information (useful for the bidirectional case) about the chunks and their relations captured by the derived graph, is not sufficient because the starting and the ending points of the links connecting two chunks must be taken into account.

Example 4. Consider the fault pattern $F=\{(5,1),(7,3),(12,2),(16,1)\}$ for a unidirectional array with redundancy $G=\{5\}$. The derived graph erroneously suggests that C_3 can be reached from C_1. Indeed, F is catastrophic whereas in the derived graph there is a path from C_0 to C_4.

To cope with this problem we use a different approach and we present a solution requiring $O(n)$ time when $k=1$, and $O(nk\log k)$ time when $k>1$. Informally, the algorithm looks for all the reachable parts of the array staring from $chunk_0$. By a reachable part of the array we mean a set, called *block*, of consecutive (fault or working) processors, such that there is a path from I to each processor of the block. A block will generate new blocks if it contains working processors, i.e., if the block overlaps one or more chunks. The algorithm considers blocks and chunks in increasing order and discards them when they have been exploited to produce new blocks. The order of chunks and blocks is given by the starting position. The algorithm ends when it cannot create new blocks (because all chunks or blocks have been discarded). A fault pattern is not catastrophic if and only if there is a block that lies after the last fault.

In the following we will denote a chunk (or block) by the pair (x,y) where p_x and p_y are the first and the last processor in the chunk (or block), respectively. We say that a pair (x,y) is *minimum* in a set X of pairs if, for each (u,v) in X, $x\leqslant u$ holds, and we say that the pair is *maximum* if, for each (u,v) in X, $u\leqslant x$ holds. Fig. 4 shows algorithm TEST, which, given a fault pattern F and the link redundancy G, tests if F is catastrophic or not.

Lemma 8. *Given a fault pattern F for a linear array with link redundancy G, algorithm* TEST *returns* `true` *if and only if F is catastrophic.*

```
TEST(F, G)
begin
    S = {(f_1 - g + 1, f_1 - 1)}; B = {(f_1 - g + 1, f_1 - 1)}
    for i = 1,...,n - 1 do insert (f_i + l_i, f_{i+1} - 1) into S endfor
    while S ≠ ∅ and B ≠ ∅ do
        let (x', y') be a minimal element of B
        let (x, y) be a minimal element of S
        case
            y' < x : delete (x', y') from B
            y < x' : delete (x , y ) from S
            otherwise :
                    x̄ = max(x, x')
                    for i = 1,...,k do insert (x̄ + g_i, y + g_i) into B
                    delete (x, y) from S
        endcase
    endwhile
    if in B there is a pair (x', y'), y' + g > f_n + l_n return false
                                                    else return true
end
```

Fig. 4. The algorithm TEST.

Proof. Let (x_i, y_i) denote the ith chunk inserted into S. It is easy to show, by induction on i, that all the blocks (x'_j, y'_j) such that $x'_j \leqslant z + g_t \leqslant y'_j$, $1 \leqslant j \leqslant k$, are inserted into B, if and only if there is a path from p_{f_1-g+1} to p_z, $x_i \leqslant z \leqslant y_i$.

Assume that F is not catastrophic. Let P be an escape path. We assume, without loss of generality, that P passes through processor $f_1 - g + 1$. Since P is an escape path, it must bypass the fault zone. Let p_u and p_v be two consecutive processors in the path P (i.e., there is a link g_t between them), such that $u < f_n$ and $v > f_n + l_n$. Clearly, there is a path from processor p_{f_1-g+1} to processor p_u. Hence, a block (x', y') that contains v, i.e., $x' \leqslant v \leqslant y'$, is inserted into B. Such a block cannot be deleted anymore. Therefore, the **if** statement returns `false`.

Assume that TEST returns `false`. Then a block (x', y') with $y' > f_n + l_n$ has been inserted into B. This implies the existence of a path from p_{f_1-g+1} to a processor p_z, with $f_n + l_n < z \leqslant y'$. Such a path can be easily extended to an escape path.

Theorem 9. *The problem of testing if a fault pattern of n blocks is catastrophic for a unidirectional array is solvable in time* $O(n)$ *when the array has only one bypass link and in time* $O(nk \log k)$ *when there are* $k > 1$ *bypass links.*

Proof. By Lemma 8, it is sufficient to show that the algorithm requires time $O(n)$ when $k = 1$ and $O(nk \log k)$ otherwise.

Since chunks in S are inserted in increasing order, we can organize S as a queue, so that insertions and deletions in S and finding the minimal element of S take constant time. Hence, the first **for** requires $O(n)$ time.

The number of iterations performed in the **while** is $O(kn)$. Indeed, during each iteration there must be a deletion from S or B, and k insertions into B occur always with a deletion from S. Noting that S initially contains n elements and B is a singleton, we have that after at most $O(nk)$ iterations one among S and B is empty.

To complete the time analysis of TEST we need to analyze the time complexity of the body of the **while** loop and the final **if**.

The new blocks to be inserted into B are not produced in increasing order, hence a standard queue is not sufficient to efficiently handle set B (unless $k=1$). We organize B in k subsets B_i, $1 \leqslant i \leqslant k$, each containing the blocks produced using the link g_i. When a new block is generated using the link g_i, we insert it in the corresponding B_i. The inserted block is maximal in B_i, hence each B_i can be organized as a standard queue. Moreover, we organize the k "heads" of the queues B_i, which contain the minimal elements of each B_i, as a heap providing the minimal element of B. With this data structure, we can insert and delete in B or find the minimal element of B in $O(\log k)$ time (or constant time if $k=1$).

Each iteration of the **while** loop may require time $O(\log k)$, $O(1)$ or $O(k \log k)$ depending on the **case** inside the **while** (if $k=1$ then it requires constant time). Since the number of **while** iterations is $O(kn)$, we only have to show that the number of **while** iterations that require time $O(k \log k)$ is $O(n)$. This follows easily noting that each of these iterations requires a deletion from S, whose cardinality is initially n. Finally the **if** test may be done in $O(k)$ time by finding the maximal elements of each B_i (constant time if $k=1$). $\square$

5. Maximum escape paths

In this section we consider the problem of finding maximum escape paths. We prove that the problem is NP-hard for a bidirectional redundant array, while for a unidirectional array we provide an algorithm that finds a maximum escape path in $O(kng)$ time.

First, we take into account the case of bidirectional links. Consider the following *Maximum Reconfiguration Length* (MRL for short) problem.

Definition 10 (*MRL problem*). Given a bidirectional redundant array A consisting of N processors, with link redundancy G, a fault pattern F and a positive integer K, is there an escape path of length at least K?

The following lemma holds.

Lemma 11. *The MRL problem is NP-complete in the strong sense.*

Proof. We reduce the problem of testing whether there exists a Hamiltonian path between two given vertices of a graph (HP for short), known to be NP-complete (see [6]), to the MRL problem. Since it is easy to give a non-deterministic polynomial-time algorithm that solves the MRL problem we conclude that MRL is NP-complete.

Let $H=(V,E)$ be the input graph for the HP problem. Without loss of generality, assume that $V=\{1,2,\ldots,n\}$ and that 1 and n are the vertices to be tested.

Consider the following instance of the MRL problem. The array A consists of $N=(6n^3-3n^2-9n+8)/2$ processing elements. For $i=1,2,\ldots,n$ define

$$a_i=(n+i-2)n^2+\frac{(n-1)(n-2)+(i-1)(i-2)}{2}+1.$$

The link redundancy G has, for each edge $(i,j)\in E$, a bypass link of length $|a_i-a_j|$. Moreover, there is an additional bypass link of length $g=a_1$ (it is easy to see that this is the longest link). The faulty pattern consists of all the processing elements p_k such that $k\neq a_i$, for $i=1,2,\ldots,n$, i.e., the only non-faulty elements are $p_{a_1},p_{a_2},\ldots,p_{a_n}$. Formally, we have that $F=\{(1,g-1),(a_1+1,a_2-a_1-1),\ldots,(a_{n-1}+1,a_n-a_{n-1}-1),(a_n+1,g-1)\}$. Finally, let $K=n$.

Notice that the above MRL instance can be constructed in time polynomial in the size n of the graph and all the integers occurring in the description of the instance are polynomially related to n.

We will prove that H has a Hamiltonian path if and only if the above instance of the MRL problem admits a solution, i.e., if there is an escape path of size n. In order to prove this, we first need the following three facts.

(i) Any escape path must traverse p_{a_1} and p_{a_n}. Indeed, the first and the last block of faults consist of $g-1$ faulty elements and thus any escape path must traverse p_{a_1} and p_{a_n} because the longest link has length g.

(ii) If p_{a_i}, with $1<i<n$, is traversed by an escape path, then it must be traversed after p_{a_1} and before p_{a_n}. Indeed, let $d_{i,j}$, $1\leqslant i\neq j\leqslant n$, be the distance between p_{a_i} and p_{a_j}, i.e., $d_{i,j}=|a_j-a_i|=n^2|i-j|+|(j-1)(j-2)/2-(i-1)(i-2)/2|$. Since (for $i\neq j$) it holds that $n^2|i-j|<d_{i,j}<n^2|i-j|+n^2$, then (for $1\leqslant i\neq j,u\neq v\leqslant n$), we have $d_{i,j}=d_{u,v}$ if and only if $\{i,j\}=\{u,v\}$.

(iii) Graph H is isomorphic to the graph consisting of the non-faulty elements p_{a_i}, $i=1,2,\ldots,n$ and their incident links. Indeed, since $d_{i,j}<d_{1,n}<g$, $1\leqslant i\neq j\leqslant n$, processors p_{a_i} and p_{a_j} are connected by a bypass link, if and only if vertices i and j are connected by an edge in graph H. Moreover, since no other two working processors are at a distance $d_{i,j}$, this bypass link connects only p_{a_i} and p_{a_j}.

Now, we can prove that there is an escape path of length at least $K=n$ if and only if there is a Hamiltonian path between vertices 1 and n in the graph H.

Assume that there is an escape path of size $K=n$. Since in A there are exactly K working processors, each processor is involved in the escape path. Since all the working processors are traversed, by (i)–(iii) we conclude that there exists a Hamiltonian path between vertices 1 and n in H (recall that by the definition of path each processor can be traversed at most once).

Conversely, given a Hamiltonian path between vertices 1 and n in H, by (iii) it corresponds to a path from p_{a_1} to p_{a_n}, which traverses once all the non-faulty processing elements of A. This path can be easily extended to an escape path of size $K=n$ connecting I to p_{a_1} and p_{a_n} to O by means of the longest bypass link.

```
MAXIMUM_SET(F, G, P)
  for i = f_1 - g + 1 to f_n + l_n + g - 2 do
      LENGTH[i]=undefined; REC_SET[i]= ∅
  endfor
  LENGTH[f_1 - g + 1]=0; i = f_1 - g + 1
  while i < f_n + l_n + g - 2 do
    if p_i is a working processor and LENGTH[i] is defined then
      for h ∈ {1} ∪ G do
        if LENGTH[i]+1 >LENGTH[i + h] or LENGTH[i + h] is undefined then
            LENGTH[i + h]=LENGTH[i]+1
            if h > 1 then
              Let t be the integer s.t. h = g_t
              REC_SET[i + h]=REC_SET[i] ∪ {(i, t)}
            endif
        endif
      endfor
    endif
    i = i + 1
  endwhile
  P=REC_SET[f_n + l_n + g - 2]
return(P)
```

Fig. 5. The algorithm MAXIMUM_SET.

Therefore, we can test if there exists a Hamiltonian path between vertices 1 and n in H by testing if there exists an escape path of size at least K for the array A. □

The strong NP-completeness of MRL clearly implies the strong NP-hardness of the problem of finding a maximum escape path for a bidirectional array.

When the array is unidirectional, the problem of finding a maximum escape path is "easy" and can be solved in $O(kng)$ time.

Fig. 5 shows an algorithm, called MAXIMUM_SET, which, given the redundancy of a unidirectional array A and a non-catastrophic fault pattern, constructs a maximum reconfiguration set for A. Before analyzing algorithm MAXIMUM_SET we remark that in the code of MAXIMUM_SET we use, for the sake of simplicity, assignments of sets to the REC_SET's, whose cost is proportional to the cardinality of the sets. However, we can construct the REC_SET's by means of pointers, so that each assignment takes constant time. Hence, though we use assignments of sets, we consider that each assignment takes constant time.

Lemma 12. MAXIMUM_SET *is correct and constructs a maximum reconfiguration set in* $O(\ell k)$ *time, where* ℓ *is the number of working processors in the fault zone.*

Proof. Let us define the set $B[s] = \{i \mid (i = s - g_t,\ t = 1, 2, \ldots, k$ or $i = s-1)$ and p_i is not faulty$\}$. Observe that, since the array is unidirectional, we can reach processor p_s only from one of $\{p_i \mid i \in B[s]\}$.

Let z be an integer such that, $f_1 - g + 1 < z \leqslant f_n + l_n + g - 2$. We want to prove the following invariant: at the iteration of the **while** for which $i = z$, LENGTH$[z]$ is the length of a longest path from processor $f_1 - g + 1$ to processor z, and REC_SET[z] is a reconfiguration set that achieves a path (from p_{f_1-g+1} to p_z) of such a length. we prove the invariant by induction. It is easy to see that the invariant holds for $z < f_1$, for which LENGTH[z] $= z - (f_1 - g + 1)$.

Suppose that the invariant holds for any $j < z$. Consider the iterations of the **while** in which i was equal to j with $j \in B[z]$ (notice that if such a set is empty, no path to p_z exists). The algorithm has already considered the path to p_z passing through p_j: this path has been discarded if it was shorter than an already considered path, while, it has been stored in REC_SET[z] and its length in LENGTH[z], if it was the longest among all the already considered paths. Hence, once $i = z$, all the possible paths have been considered and the longest one has been recorded.

Thus, LENGTH[$f_n + l_n + g - 2$] is the length of a longest path from processor $f_1 - g + 1$ to processor $f_n + l_n + g - 2$, and REC_SET[$f_n + l_n + g - 2$] is a reconfiguration set that achieves a path of such a length. Moreover, since the array is unidirectional, any maximum escape path must pass through $f_1 - g + 1$ and $f_n + l_n + g - 2$. This means that REC_SET[$f_n + l_n + g - 2$] is a maximum reconfiguration set.

The complexity of MAXIMUM_SET is easily computed: the first **for** takes $O(\ell)$ time and the **while** with the nested **for** takes $O(\ell k)$ time. Hence, the algorithm runs in time $O(\ell k)$. □

The next lemma states that if the chunks of a fault pattern are enough big, then the fault pattern can be "splitted" into several fault patterns which can be considered separately.

Lemma 13. *Let $F = \{(f_1, l_1), \ldots, (f_n, l_n)\}$, be a fault pattern for a unidirectional array with redundancy G. If there exist integers $j_1, j_2, \ldots, j_s$, with $j_1 < j_2 < \cdots < j_s < n$, such that chunk$_{j_i}$, $1 \leqslant i \leqslant s$, has more than $2g - 4$ processors then F is a CFP if and only if at least one among the fault patterns $F_1 = \{(f_1, l_1), \ldots, (f_{j_1}, l_{j_1})\}$, $F_2 = \{(f_{j_1+1} + l_1, l_{j_1+1}), \ldots, (f_{j_2}, l_{j_2})\}, \ldots, F_s = \{(f_{j_{s-1}+1}, l_{j_{s-1}+1}), \ldots, (f_n, l_n)\}$ is a CFP.*

Proof. For the sake of contradiction, assume that F is catastrophic and none among $F_1, F_2, \ldots, F_s$ is catastrophic. Since *chunk*$_{j_i}$ has more than $2g - 4$ processors any escape path for F_i ends before the beginning of any escape path for F_{i+1}. Then concatenating the escape paths of $F_1, F_2, \ldots, F_s$ we can construct an escape path for F, contradicting the hypothesis that F is catastrophic. □

Theorem 14. *Given a unidirectional array with k redundant links and whose longest link has length g, the problem of finding a maximum escape path for a non catastrophic fault pattern of n blocks of faults is solvable in time* $O(kng)$.

Proof. Let F be the fault pattern. Split F into s fault patterns, $F_1, \ldots, F_s$, such that between F_i and F_{i+1}, for $i = 1, 2, \ldots, s - 2$, there is a chunk of more than $2g - 4$ processors. By Lemma 13, a maximum reconfiguration set for F is given by the union of the s reconfiguration sets obtained applying the algorithm MAXIMUM_SET to $F_1, \ldots, F_s$. Let n_i be the number of blocks in F_i, $1 \leqslant i \leqslant s$. Clearly, $\sum_{j=1}^{s} n_i = n$. Since the number of elements in the fault zone for F_i is less than $n_i g + (n-1)(2g-3) + 2g - 2$ (remember that each chunk has less than $2g-4$ processor and that F_i is not catastrophic for A, hence each block has less than g elements), by Lemma 12, constructing a maximum reconfiguration set for F_i takes $O(n_i gk)$ time. Hence, constructing a maximum reconfiguration set for F takes $O(\sum_{j=1}^{s} n_j gk)$ time, that is, $O(kng)$ time.

6. Minimum escape paths

In this section we consider the problem of finding minimum escape paths. When the array is bidirectional we provide an algorithm solving the problem in $O(kn)$ time, and for the unidirectional case we propose an algorithm solving the problem in $O(kng)$ time. First, we consider the case of bidirectional link.

Theorem 15. *Given a bidirectional array with k redundant links whose longest link has length g, the problem of finding a minimum escape path for a non-catastrophic fault pattern of n blocks of faults is solvable in $O(kn)$ time.*

Proof. By Lemma 5, an escape path exists if and only if chunks C_0 and C_n in the derived graph are connected. Since each edge in the derived graph corresponds to the use of a redundant link, the number of redundant links that we have to use in any reconfiguration set is at least equal to the length of the shortest path from C_0 to C_n in the derived graph. On the other hand, given a path from C_0 to C_n it is easy to obtain a reconfiguration set whose cardinality is exactly the length of the path. Hence, we can find a minimum escape path by finding a shortest path from C_0 to C_n in the derived graph. It is well known that this problem is solvable in time linear in the number of edges. The derived graph has at most $O(kn)$ edges, because it is constructed in $O(kn)$ time. □

Now we consider the case of unidirectional links. In this case we can use the same technique used to find a maximum escape path. Fig. 6 shows an algorithm, called MINIMUM_SET, that, given the redundancy of a redundant array and a non-catastrophic fault pattern, constructs a minimum reconfiguration set.

Lemma 16. *Algorithm* MINIMUM_SET *is correct and constructs a minimum reconfiguration set in time $O(k\ell)$, where ℓ is the number of working processors in the fault zone.*

Proof. Let us define the set $B[s] = \{i \,|\, (i = s - g_t,\ t = 1, 2, \ldots, k \text{ or } i = s-1) \text{ and } p_i \text{ is not faulty}\}$. Observe that since the array is unidirectional we can reach processor p_s only from one of $\{p_i \,|\, i \in B[s]\}$.

```
MINIMUM_SET(F, G, P)
  for i = f_1 - g + 1 to f_n + l_n + g - 2 do
      LINKS[i]=undefined; REC_SET[i]= ∅
  endfor
  LINKS[f_1 - g + 1] = 0; i = f_1 - g + 1
  while i < f_n + l_n + g - 2 do
    if p_i is a working processor and LINKS[i] is defined then
      if LINKS[i] < LINKS[i + 1] or LINKS[i + 1] is undefined then
          LINKS[i + 1]=LINKS[i]; REC_SET[i + 1]=REC_SET[i]
      endif
      for h ∈ G do
          if LINKS[i]+1 <LINKS[i + h] or LINKS[i + h] is undefined then
              LINKS[i + h]=LINKS[i]+1
              Let t s.t. h = g_t; REC_SET[i + h]=REC_SET[i] ∪ {(i, t)}
          endif
      endfor
    endif
    i = i + 1
  endwhile
  P=REC_SET[f_n + l_n + g - 2]
return(P)
```

Fig. 6. The algorithm MINIMUM_SET.

Fix an integer z, $f_1 - g + 1 < z \leqslant f_n + l_n + g - 2$. As in Lemma 12, we can prove by induction the following invariant: at the iteration of the **while** for which $i = z$, LINKS[z] is the minimum number of redundant links that any path from processor $f_1 - g + 1$ to processor z must use, and REC_SET[z] is a reconfiguration set that achieves a path using such a minimum number of redundant links.

Thus, LINKS[$f_n + l_n + g - 2$] is the minimum number of redundant links that any path from processor $f_1 - g + 1$ to processor $f_n + l_n + g - 2$ must use, and REC_SET[$f_n + l_n + g - 2$] is a reconfiguration set that achieves an escape path that uses such a number of redundant links. Hence, REC_SET[$f_n + l_n + g - 2$] is a minimum escape path.

The complexity of MINIMUM_SET is easily computed: the first **for** takes $O(\ell)$ time. The **while** with the nested **for** takes $O(\ell k)$ time. Hence, the algorithm runs in $O(\ell k)$ time. □

Theorem 17. *Given a unidirectional array with k redundant links and whose longest link has length g, the problem of finding a minimum escape path for a non catastrophic fault pattern of n blocks of faults is solvable in time* $O(kng)$.

Proof. Let F be the fault pattern. Split F into s fault patterns, $F_1, \ldots, F_s$, such that between F_i and F_{i+1}, for $i = 1, 2, \ldots, s-2$, there is a chunk of more than $2g - 4$ processors. By Lemma 13, a minimum reconfiguration set for F is given by the union of the s reconfiguration sets obtained applying the algorithm MINIMUM_SET to $F_1, \ldots, F_s$. The rest of the proof is as in Theorem 14. □

7. Summary and open questions

In this paper we studied the problem of providing fault-tolerant capabilities to parallel architectures by means of redundancy. This approach consists of adding spare processors and extra links that can be used to bypass faulty processing elements and reconfigure the architecture with no slow down in the performance.

No matter how much redundancy is provided, it is always possible to have a set of faulty elements for which no reconfiguration is possible. Such sets of faults are called catastrophic.

Before attempting any reconfiguration it is important to test whether the set of faults is catastrophic. When a set of faults is not catastrophic it is important to provide efficient reconfiguration algorithms that provide optimal reconfigurations.

In this paper we have considered linear arrays of processing elements. We have considered both the case when the array has bidirectional links and the case when the array has unidirectional links. We have provided new testing algorithms which improve and generalize previous known algorithms.

We have also considered the problem of finding optimal reconfiguration when the set of faults is not catastrophic. Optimality is considered either with respect to the size of the reconfigured array or with the amount of changes needed to reconfigure the array. We proved that when the links are bidirectional, the problem of finding optimal reconfiguration with respect to the size of the reconfigured array is NP-hard in the strong sense. In all the other three cases we provided algorithms which efficiently find an optimal reconfiguration.

In this paper the case of linear array has been extensively studied, however, some questions still remain open. For instance the given $O(kng)$ algorithm to find an optimal reconfiguration of a unidirectional array maximizing the size of the reconfigured array, is indeed pseudo-polynomial in g. Better algorithms for this problem might exist. Also, the problem of failures of the links has not been considered yet. Hence, another direction for research is to consider fault patterns consisting of both processing elements and links.

Other parallel architecture are used in practice. In particular, bidimensional arrays: memory chips are organized in this form and many existing parallel machines have a mesh architecture (see e.g., [11]) and their importance is still increasing nowadays. The approach adopted here might be useful also to study bidimensional arrays of processors.

References

[1] K.P. Belkhale, P. Banerjee, Reconfiguration strategies in VLSI processor arrays, in: Proc. Internat. Conf. on Computer Design, 1988, pp. 418–421.

[2] J. Bruck, R. Cypher, C. Ho, Tolerating faults in a mesh with a row of spare nodes, in: 4th IEEE Symp. on Parallel and Distributed Processing, Arlington, 1992, pp. 12–19.

[3] M. Chean, J.A.B. Fortes, A taxonomy of reconfiguration techniques for fault-tolerant processor arrays, IEEE Comput. 23 (1990) 55–69.

[4] R. De Prisco, A. De Santis, Catastrophic faults in reconfigurable VLSI linear arrays, Discrete Appl. Math. 75 (1997) 105–123.
[5] R. De Prisco, A. Monti, On reconfiguration of VLSI linear arrays, in: 3rd Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science, vol. 709, 1993, pp. 553–564.
[6] M. Garey, D. Johnson, Computers and Intractability, Freeman, New York, 1979.
[7] J.W. Greene, A.E. Gamal, Configuration of VLSI arrays in presence of defects, J. ACM 31 (1984) 694–717.
[8] E. Horowitz, S. Sahni, Fundamentals of Computer Algorithms, Computer Science Press, Rockville, MD, 1978.
[9] S.H. Hosseini, On fault-tolerant structure, distributed fault-diagnosis, reconfiguration, and recovery of the array processors, IEEE Trans. Comput. 38 (1989) 932–942.
[10] C. Kaklamanis, A.R. Karlin, F.T. Leighton, V. Milenkovic, P. Raghavan, S. Rao, C. Thomborson, A. Tsantilas, Asymptotically tight bounds for computing with faulty arrays of processors, in: Proc. of 31st Annual Symp. on Foundation of Computer Science, 1990, pp. 285–296.
[11] H.T. Kung, Why systolic architecture?, IEEE Comput. 15 (1982) 37–46.
[12] M.T. Liu, Distributed Loop Computer Network, Adv. Comput. 17 (1978) 163–221.
[13] A. Nayak, On reconfigurability of some regular architectures, Ph. D. Thesis, Dept. System and Computer Engineering, Carleton University, Ottawa, Canada, 1991.
[14] A. Nayak, L. Pagli, N. Santoro, Efficient construction for VLSI reconfigurable arrays, Integration VLSI J. 15 (1993) 133–150.
[15] A. Nayak, N. Santoro, Bounds on performance of VLSI processor arrays, in: 5th Internat. Parallel Processing Symp., Anaheim, CA, May 1991.
[16] A. Nayak, N. Santoro, R. Tan, Fault-intolerance of reconfigurable systolic arrays, in: Proc. 20th Internat. Symp. on Fault Tolerant Computing, FTCS'20, 1990, pp. 202–209.
[17] R. Negrini, M.G. Sami, R. Stefanelli, Fault-tolerance techniques for array structures used in supercomputing, IEEE Comput. 19 (1986) 78–87.
[18] L. Pagli, G. Pucci, Counting the number of fault pattern in redundant VLSI arrays, Inform. Process. Lett. 50 (1994) 337–342.
[19] D.K. Pradhan (Ed.), Fault-Tolerant Computing: Theory and Techniques, vols. 1 and 2, Prentice-Hall, Englewood Cliff, NJ, 1986.
[20] C.S. Raghavendra, A. Avizienis, M.D. Ercgovac, Fault tolerance in binary tree architectures, IEEE Trans. Comput. C-33 (1984) 568–572.
[21] A.L. Rosemberg, The diogenes approach to testable fault-tolerant arrays of processors, IEEE Trans. Comput. 32 (1983) 902–910.
[22] V.P. Roychowdhury, J. Bruck, T. Kailath, Efficient algorithms for reconfiguration in VLSI/WSI arrays, IEEE Trans. Comput. 39 (1990) 480–489.
[23] M. Sami, R. Stefanelli, Reconfigurable architectures for VLSI processing arrays, Proc. IEEE 74 (1986) 712–722.