In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS-29)*, Madison, WI, Jun. 15-18, 1999.

Efficient Network-Flow Based Techniques for Dynamic Fault Reconfiguration in FPGAs*

NIHAR R. MAHAPATRA
mahapatr@cse.buffalo.edu
Department of Computer Science & Engineering
State University of New York at Buffalo
Buffalo, NY 14260-2000

SHANTANU DUTT
dutt@eecs.uic.edu
Department of Electrical Eng. & Computer Sc.
University of Illinois at Chicago
Chicago, IL 60607-7053

Abstract

In this paper, we consider a "dynamic" node covering framework for incorporating fault tolerance in SRAM-based segmented array FPGAs with spare row(s) and/or column(s) of cells. Two types of designs are considered: one that can support only node-disjoint (and hence nonintersecting) rectilinear reconfiguration paths, and the other that can support edge-disjoint (and hence possibly intersecting) rectilinear reconfiguration paths. The advantage of this approach is that reconfiguration paths are determined dynamically depending upon the actual set of faults and track segments are used as required, thus resulting in higher reconfigurability and lower track overheads compared to previously proposed "static" approaches. We provide optimal network-flow based reconfiguration algorithms for both of our designs and present and analyze a technique for speeding up these algorithms, depending upon the fault size, by as much as 20 *times. Finally, we present reconfigurability results for our FPGA designs that show much better fault tolerance for them compared to previous approaches—the reconfigurability of the edge-disjoint design is 90% or better and 100% most of the time, which implies near-optimal spare-cell utilization.*

1 Introduction

Field programmable gate arrays (FPGAs) consist of a large array of programmable cell or logic blocks (CLBs) and interconnects that can be reconfigured to implement a wide variety of application logic. They are commonly used for the development of prototype systems and their early introduction to market and also as emulators to verify and test designs. An important criterion in the design of FPGAs is *fault tolerance*, which is the ability to retain full or partial functionality in the presence of CLB or interconnect faults that arise during fabrication or operation. Being able to tolerate fabrication faults means higher yield and lower costs for the manufacturer. Tolerance for operational faults translates into higher reliability and reduced downtime for the user.

*S. Dutt was supported partly by a grant from Xilinx Corp. and partly by Darpa Grant # F33615-98-C-1318.

Two different approaches have been used to provide fault tolerance in FPGAs. The first is to reroute the user's circuit to avoid faulty cells and/or interconnects, using spares or other unused cells instead [5, 8, 9, 10, 11]. Requiring the layout tools to perform a new routing for every fault places a heavy burden on the user, who must also keep track of all the different routings possible on different chips for a given circuit design. The other approach, adding spare rows and/or columns of cells, is intended for reconfiguration at the factory, making the technique transparent to the user. To reconfigure around a faulty row, fuses are burned at the factory such that non-faulty rows are remapped to include the spare row. For the faulty row to be transparent, it is necessary to maintain the original connectivity between the rows on either side of the faulty one. One method of doing this is to employ longer wiring segments in the vertical channels, but then extra *tracks* must be added to these channels to retain the original routing flexibility [6].

Of the previous work done in this area, the method of interest to this research is the *node covering* method for cell fault tolerance [3]. The model of FPGA considered is shown in Fig. 1 and consists of an $n_1 \times n_2$ rectangular array of CLBs. Wiring tracks run along channels between adjacent rows and columns of CLBs to support routing of nets connecting different CLBs. In a *single-spare row (1S-R) design*, there is an additional row of spare CLBs at the bottom of the array—the spare in column j is denoted $s^b_{*,j}$, where "b" stands for "bottom"; a *single-spare column (1S-C) design* is defined similarly with the spare column to the right. A *single-spare row-column (1S-RC) design* has a spare row of CLBs at the bottom and a spare column of CLBs to the right. Finally, a *double-spare row-column (2S-RC)* design has spares on all four sides as in Fig. 1. The spare CLBs and the unused track segments provide the necessary reconfiguration capability for the array.

In the node-covering method, a faulty FPGA is reconfigured by finding for each fault a *covering sequence*, which is an ordered sequence of CLBs beginning with the faulty CLB and ending in a spare CLB such that each CLB re-

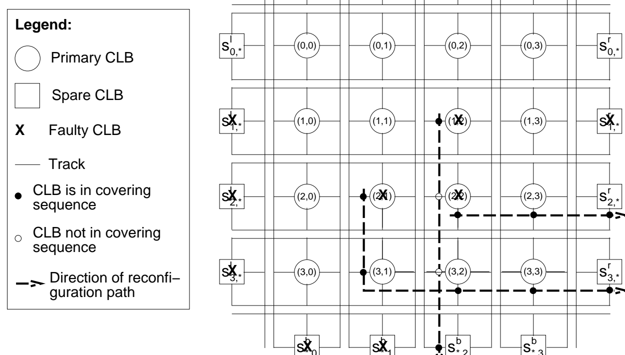

Figure 1: FPGA reconfiguration model showing reconfiguration paths for faults $(1,2)$, $(2,1)$, and $(2,2)$.

places or "covers" the preceding CLB in the sequence by taking over its functionality as well as connectivity. A CLB v_2 that covers another CLB v_1 needs to connect to nets originally connected to v_1, and this is accomplished by using track segments adjoining intermediate CLBs on a path from v_1 to v_2 called the *covering path*. A *reconfiguration path* for a fault u is a path from u to a spare through a superset of CLBs in its covering sequence obtained by concatenating all covering paths between consecutive CLBs in the covering sequence. For example, in Fig. 1, the covering sequence for fault $(1,2)$ is $((1,2), s^b_{*,2})$, while the corresponding reconfiguration path is $\langle(1,2),(2,2),(3,2),s^b_{*,2}\rangle$. Since in the above model a CLB can cover another CLB only via the track segments adjoining intervening CLBs, reconfiguration paths always consist of adjacent CLBs and hence are *continuous*. Covering sequences, on the other hand, may be *discontinuous*—this is the case in the above example in which CLBs $(2,2)$ and $(3,2)$ are *skipped* in $(1,2)$'s reconfiguration path.

In the *static node-covering* method, reconfiguration paths for faults are predetermined and are independent of the set of faults [3]. For instance, in a 1S-C FPGA, reconfiguration paths go straight to the spare CLB on the right and only one fault per row can be tolerated—similar to the reconfiguration path for fault $(2,2)$ in Fig. 1. A *cover* cell that replaces a *dependent* cell must be able to duplicate the functionality of the latter. In an FPGA, all cells are identical, so configuration data for the dependent cell can simply be transposed to the cover cell. The cover cell must also be able to duplicate the connectivity of the dependent cell with respect to the rest of the array. This is accomplished by ensuring that each net connected to a cell through a channel segment also includes the corresponding channel segment—a *cover segment*—bordering the cover cell (see Figs. 2(a) and (b)). Cover segments are included in a net in one of two ways. First, segments in the net may already be in positions to act as covers. In case the above condition does not hold, additional segments, termed *reserved segments* (RSs), should be attached to the net to provide covers. Essentially, segments are reserved to act as covers at all branch points of the circuit netlist. Thus this method requires neither the factory nor the user to generate new routing maps to reconfigure around faulty cells or wiring, as is required by [5, 8, 9, 10, 11]. Instead, the original configuration data can be reused. Also, no explicit additional tracks are needed in the channels in order to avoid the loss of connection flexibility seen in [6]. The additional wiring segments used to support reconfiguration paths cause a track overhead in the routing—for a number of benchmark circuits, the track overhead is found to be 34% [3]. This results in retaining total functionality with reduced routability or total routability with reduced functionality.

To retain total functionality with as little overhead as possible, a *dynamic node-covering* method was proposed in [13]. In this method, reconfiguration paths are dynamically determined depending upon the fault set and RS insertions are made only along channels where reconfiguration paths pass. This results in higher reconfigurability and lower overhead. Two types of dynamic designs are considered: *node-disjoint* and *edge-disjoint* FT FPGA designs. In the first type, only node-disjoint rectilinear reconfiguration paths can be supported. Thus in Fig. 1, reconfiguration paths for faults $(2,1)$ and $(2,2)$ can both be supported—note that the reconfiguration path for $(2,1)$ shown in Fig. 1 can not be supported in the static node-covering method since it is bent. However, reconfiguration paths for $(1,2)$ and $(2,2)$ and/or $(2,1)$ can not be supported in the node-disjoint dynamic-FT method, since the former intersects the latter two. The second type of FPGA design can support any set of edge-disjoint rectilinear reconfiguration paths, and hence can support reconfiguration paths for faults $(1,2)$, $(2,2)$, and $(2,1)$ in Fig. 1 all simultaneously. Thus it provides higher yield/reliability compared to the first type. We discuss the dynamic node covering method in the next section.

In order to maximally utilize the spare CLBs, rectilinear reconfiguration paths from faults to spares need to be optimally determined (i.e., if there exist such paths for all faults, they need to be found). We present network-flow based reconfiguration algorithms for both node-disjoint and edge-disjoint FPGA designs that determine such paths in Sec. 3. In previous work, network-flow based reconfiguration algorithms have been proposed for VLSI/WSI arrays in [12]. Since our dynamic node covering method is meant to be used online, reconfiguration speed is important. In Sec. 4, we present an effective technique for speeding up our reconfiguration algorithms that exploits the regular array structure of the flow graph and that provides as much as 20 times speedup. In Sec. 5, we analyze this speedup technique and verify the analysis empirically. Sec. 6 presents reconfigurability results for node- and edge-disjoint and single- and double-spare FPGA designs. Finally, we conclude in Sec. 7.

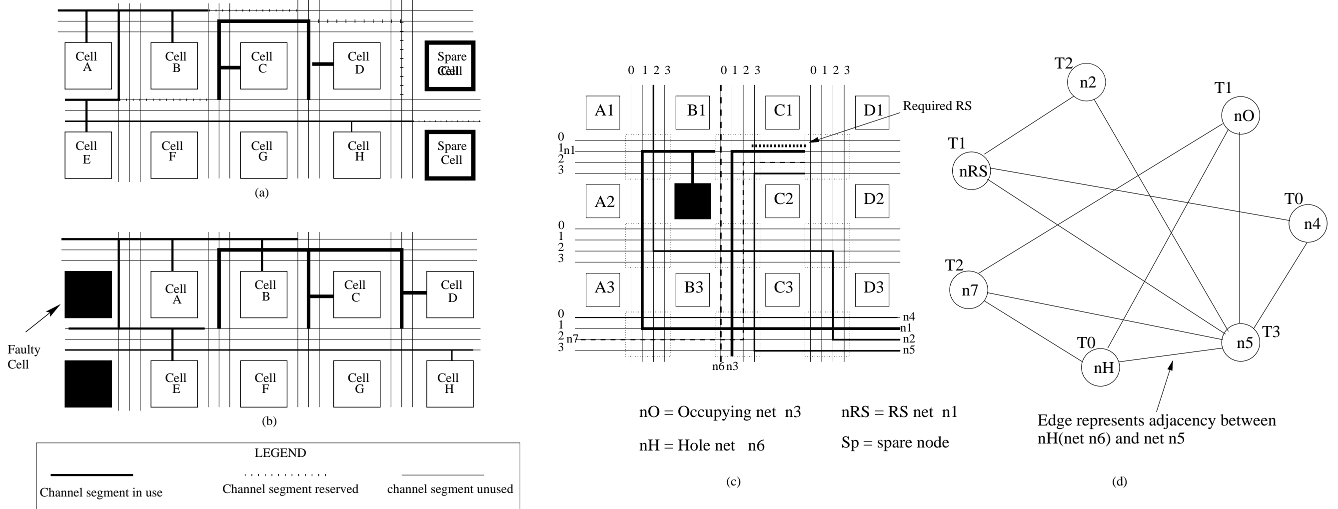

Figure 2: Fault tolerance using the static node covering method in a 1S-C FPGA: (a) Cover segments associated with nets in the FPGA. (b) Reconfiguration around faulty cells A and E using cover segments and (static) straight reconfiguration paths to the right. Reconfiguration using the dynamic node covering method: (c) Presence of occupying net (O-net) n3 prevents a straightforward insertion of a reserved segment in the RS channel. Therefore the O-net must be moved to another track, possibly bumping other nets, to make room for the reserved segment. (d) Overlap graph used to determine transition tree for FPGA circuit in [c].

2 Background: Dynamic Node Covering

The dynamic FT method [13] was developed to reduce the track overhead incurred in implementing the static method. It is similar to the static method except that reconfiguration paths for tolerating faults are not predetermined, instead they depend upon the particular set of faults to be reconfigured around.[1] The fact that reconfiguration paths in the dynamic method depend upon the fault set means that the assignment of cover cells to dependent cells is not static. This also means that RS insertions are required only along the actual reconfiguration paths used, in contrast to requiring RSs at all branch points for nets along the whole FPGA. By not statically reserving wire segments, having flexibility for reconfiguration paths, and requiring RS insertions only along such paths, a better utilization of the unused wiring segments and tracks is obtained. This leads to a much lower track overhead and better fault tolerance for the dynamic FT method compared to the static one.

The dynamic nature of the method requires the actual position of the fault to be known to identify the RSs required. Since the routing for the FPGA has already been completed, this brings into question the availability of the wire segment where an RS has to be inserted. This is discussed next.

2.1 Reserved Segments and Occupying and Hole Nets

For any required RS, if the required wire segment where the RS is to be inserted is vacant, i.e., no net has been routed through it, the RS insertion is done by including this segment in the net that is connected to the dependent cell, so that its connectivity can be taken over by the cover cell. If the required wire segment is occupied by another net, i.e., some other net has already been routed through that wire segment, then the RS insertion cannot be made directly because the required segment is not available. As shown in Fig. 2(c), the net requiring the RS extension is termed the *RS net* and the net occupying the required wire segment is termed the *occupying net* (O-net). The existence of O-nets for the required RSs gives rise to a requirement that there should be at least as many vacant segments as required RSs in the channel segment where these RSs have to be inserted. When there is a vacant segment on a section of the channel requiring the RS (the *RS channel*) on a certain track T_i, any net that passes through the channels adjacent to the RS-channel but does not pass through the RS-channel on the same track T_i is referred to as a *hole net*.

The problem can now be stated as follows. The RS net and O-net are on the same track. The RS net has to be extended by one segment (*RS-insertion*). This requires the O-net to move out of its current track. Let a *transition* be defined as the movement of net n_i on a track T_j to another track T_k. This transition may result in the net n_i bumping into one or more nets on track T_k. These nets will have to move out of their current track T_k, giving rise to a transition for each of them. This gives rise to a transition sequence which will finally terminate in "spare" nodes, which are vacant segments of appropriate lengths to which a bumped net can move in without bumping any other net. When such a vacant segment is part of on occupied track then no spare track overhead is incurred, whereas when it is part of a spare track then

[1] Note that for edge-disjoint reconfiguration paths that skip nodes (i.e., in which covering can take place between non-adjacent cells), RSs of segment length > 1 are required. Specifically, if u covers v and is at a distance of t from it, then RSs of length up to t are required for u to connect to nets originally connected to v.

an overhead of an extra track is incurred. Clearly, the set of transitions takes on a tree structure, termed a *transition tree*, with the spares forming the leaf nodes. The RS insertion is successful if a transition tree rooted at the corresponding O-net can be found whose leaves are spare nodes, such a transition tree is termed a *converging transition tree*. We next briefly discuss a model that can be used for efficiently determining a converging transition tree.

2.2 The Overlap Graph

The *overlap graph* (OG) is a graph representation of the circuit routing on the FPGA. This graph is an undirected graph with the circuit nets represented by the nodes of the graph. In the overlap graph $OG(V, E)$, the set of nets $N = \{n_1, n_2, \ldots, n_m\}$ in the circuit routing is represented by a set of nodes/vertices $V = \{n_1, n_2, \ldots, n_m\}$. We use $n_i^{T_j}$ to denote a net n_i on track T_j. There exists an edge between n_i and n_j in the OG iff nets n_i and n_j share a *channel*[2] in the FPGA. Figs. 2(c) and (d) show nets n_2 and n_{RS} having an edge between them because they are routed through a common channel on the left of the faulty cell. Also, the net $n_6^{T_0}$ is routed through the adjacent channel in the area of the faulty cell but not in the channel where the RS is required. The wire segment on track T_0 in the RS channel is vacant. So n_6 is termed the hole net.

The overlap graph can be used as an effective model in the evaluation of the required transition tree. Since $OG(V, E)$ represents the circuit routing on the FPGA, the movement of nets involved in a transition tree can be seen as a tree in the graph. Obviously, a converging transition tree should end with the final transitions of nets into tracks where they do not bump into any other nets, i.e., they end in spare segments, which correspond to the spare (leaf) nodes in the transition tree.

For a given circuit routing and a specific cell fault, it is possible to identify the RS net, the O-net, and the H-net in the OG. The O-net has to be moved out of its current track to accommodate the RS. Since the transition $n_i^{T_j \to T_k}$ of net n_i from track T_j to another track T_k may result in the net bumping into one or more of its children on that track, we associate a heuristic cost with each net transition. The heuristic attempts to measure the cost of bumpings—bumping into a large net is more expensive than bumping into a small net or a number of small nets with total length (in terms of number of track segments) the same as or smaller than the length of the large net. The heuristic is used as a guide in creating the transition tree by choosing those net transitions that are less costly. This reduces the branching factor of the tree path and hence the amount of perturbation of nets, thus making it more likely for the transitions to converge with a minimal overhead of spare tracks. Three different transition cost heuristics have been evaluated in [13]. For the best heuristic, the track overhead for tolerating one fault per row (the same fault pattern as in the static method), and for a total of four faults, for a number of benchmark circuits, is only 16% and 4.5%, respectively. This establishes the superiority of the dynamic FT approach over the static one.

[2] A channel is the set of all track segments between two adjacent switch-boxes of the FPGA.

To tolerate arbitrary fault patterns, however, straight reconfiguration paths will no longer be sufficient. Reconfiguration paths with possibly multiple bends will be needed and they may also intersect each other at nodes (edge-disjoint paths). Intersection of two paths at a node has the effect of one of the paths "skipping" the intersecting node. Optimal or near-optimal techniques for finding such reconfiguration paths in a faulty FPGA are desirable for maximal fault tolerance. Once these paths are determined, reconfiguration will be achieved by inserting RSs as required along these paths using the techniques of [13] for determining converging transition trees. In the rest of this paper, we develop fast optimal algorithms for determining reconfiguration paths in a faulty FPGA.

As mentioned earlier, there are two types of reconfiguration paths, *node-disjoint* and *edge-disjoint*. In the next section, we present network-flow based algorithms to determine reconfiguration paths of these two types.

3 Reconfiguration Algorithm

First, define a *FPGA flow graph* for a FT FPGA as follows.

Definition 1 *A* FPGA flow graph *for a faulty FT FPGA has a vertex corresponding to each primary cell in the FPGA and has two unidirectional, unit-capacity edges directed in opposite directions between vertex pairs which have corresponding cell pairs that are either east-west or north-south neighbors in the FPGA. Vertices in the flow graph corresponding to faulty primary cells in the FPGA are sources. The flow graph has an additional vertex which is a sink (representing nonfaulty spares in the FPGA) which has edges incident on it from vertices corresponding to neighbors of nonfaulty spare cells in the FPGA. Vertices that are neither sources nor sink have unbounded capacities for an edge-disjoint FPGA design and unit capacities for a node-disjoint FPGA design.*

Assuming the 7×5 array of primary cells inside rectangle ABCD in Fig. 3(a) represents an FPGA and that it is a 2S-RC design (i.e., it is surrounded by spares on all sides), the corresponding FPGA flow graph would be as in Fig. 3(b). Note that there is no edge in Fig. 3(b) corresponding to the faulty spare on the right in row 9 of Fig. 3(a). We are now ready to demonstrate the equivalence between the FPGA reconfiguration and maximal flow problems.

Theorem 1 *The problem of determining a maximal set of reconfiguration paths to reconfigure faults in the node- and*

edge-disjoint FT FPGA designs is equivalent to the problem of determining a maximal flow in the corresponding FPGA flow graph.

Proof: It is clear from the definition of FPGA flow graph (Def. 1) that a flow in the flow graph corresponds to a rectilinear reconfiguration path from a fault to a spare in the actual FPGA hardware. The fact that all edges in the FPGA flow graph have unit capacities implies that reconfiguration paths must be edge disjoint. Vertices that are neither sources nor sink have unbounded capacities in the flow graph for an edge-disjoint FPGA design to allow edge-disjoint reconfiguration paths that may not be node disjoint, i.e., that may be intersecting. However, these vertices have unit capacities for a node-disjoint FPGA design to allow only node-disjoint reconfiguration paths. Thus by solving the maxflow problem for this flow graph, a set of reconfiguration paths to tolerate the maximum number of faults possible can be found. □

Figs. 3(a) and (b) show the correspondence between reconfiguration paths in the FPGA and flows in the FPGA flow graph.

Theorem 2 *A maximal set of reconfiguration paths to tolerate as many faults as possible in the node- and edge-disjoint FT FPGA designs can be found by applying the lift-to-front preflow-push algorithm of [2] to the corresponding FPGA flow graph.*

Proof: This follows from Theorem 1 and the fact that the lift-to-front preflow-push algorithm finds a maximal flow in the flow graph to which it is applied [2]. □

In our simulations, we used the $O(|V|^3)$ lift-to-front preflow-push algorithm of [2], which runs in $O(N^3)$ time since there are $\Theta(N)$ vertices in the flow graph. Since the flow graph under consideration is planar, we can use a more efficient, although not as simple to code, maxflow algorithm meant especially for planar graphs given in [7] that runs in $O(N^{1.5} \log N)$ time.

Reconfiguration paths obtained using the above approach may either be supportable with the unused or spare tracks and track segments or extra tracks may be needed, which cause a track overhead. Clearly, it would be desirable to judiciously utilize the spare tracks and track segments and to minimize the track overhead. This is accomplished by associating with each edge from a vertex u to a vertex v in the flow graph discussed above a cost that estimates the track-overhead cost of inserting RSs in order for v to cover u. This cost can be determined as follows. Let $RS(v, u)$ be the set of RSs that need to be inserted for v to cover u, and let $O(v, u)$ be the set of corresponding occupying nets. Then the cost of arc (u, v) in the flow graph will be the sum of the minimum or best transition costs of the nets in $O(v, u)$, and is given by

$$\sum_{n_i \in O(v,u)} \min\{cost(n_i^{T_j \rightarrow T_k}) | T_k \neq T_j\}$$

where $cost()$ is a heuristic cost function. Then a set of reconfiguration paths that results in the maximum possible number of faults being reconfigured and at the same time also incurs minimum track overhead can be found by solving the mincost flow problem on this modified flow graph [1].

4 Speeding Up Reconfiguration

In this section, we present an effective technique to significantly speed up the reconfiguration algorithm. Fast reconfiguration is needed especially in long-life mission critical systems possibly operating in hazardous and/or unmaintained environments (e.g., spacecrafts, satellites, remote-sensing stations) where multiple faults can accumulate over the life of the mission. The speedup technique discussed applies to both node- and edge-disjoint FPGA designs, the only difference being that vertices that are neither sources nor sinks in the flow graph have unit capacities in the former case and unbounded capacities in the latter case.

4.1 The Technique for Double-Spare Designs

For simplicity of exposition, we first consider the reconfiguration algorithm for the double-spare case in which there are spares on all four sides of the FPGA. Figure 3(a) shows a 14×8 array of primary cells with a 10-fault pattern including a faulty spare in the rightmost column in row 9; assume for the moment that it is a double-spare design, although the figure actually depicts a single-spare design with a spare column on the right and a spare row at the bottom. Define the *circumscribing rectangle* corresponding to a fault pattern as the set of primary cells in the rectangle circumscribing all primary-cell faults and all primary cells adjacent to spare faults. This is rectangle (A, B, C, D) in Fig. 3(a).

The reconfiguration problem in the double-spare case is essentially equivalent to finding reconfiguration paths for all faults to unique boundary cells of the circumscribing rectangle, since from there the paths can be extended straight outward from the rectangle to spares in the same row (for boundary cells on the left and right sides) or column (for boundary cells on the top and bottom sides). For example, for the fault $(9, 4)$ in Figure 3(a), we first find a path to the boundary cell $(11, 4)$, and from there extend it straight outward to the spare in the bottom row and the same column. Although not shown in Figure 3(a), paths formed till boundary cells on the left and top sides of the circumscribing rectangle are extended outward to the spares on the left and top sides of the array, respectively. Thus, the reconfiguration problem in the double-spare case is equivalent to solving the maxflow problem in the subgraph of the original graph discussed in Sec. 3 consisting of vertices and edges confined to the circumscribing rectangle and in which boundary-cell

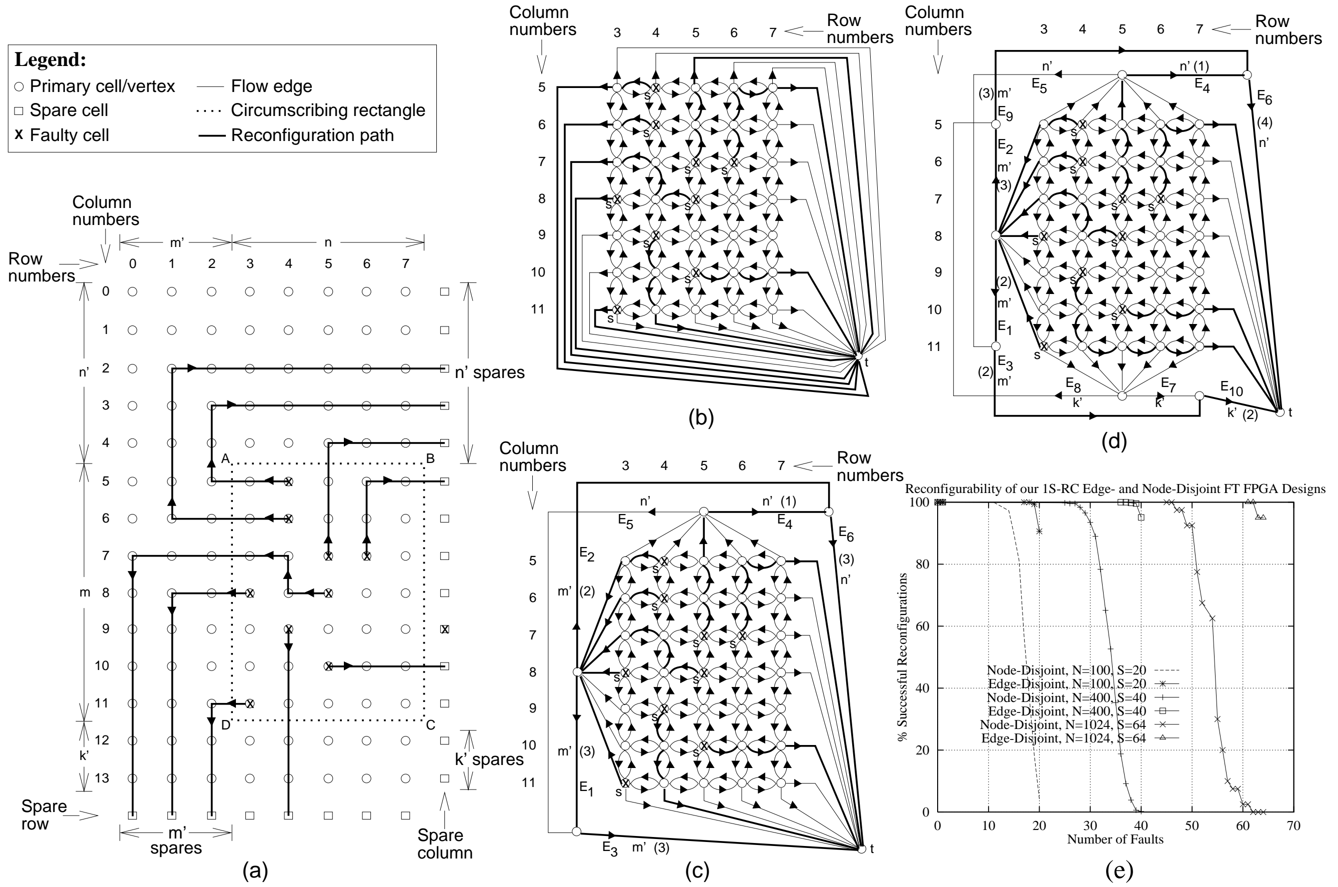

Figure 3: (a) An example fault reconfiguration showing the basis of the fast reconfiguration algorithm with the circumscribing rectangle identified with dotted lines. (b) The flow graph for the fast reconfiguration algorithm in the double-spare row-column case. (c) The flow graph for the fast reconfiguration algorithm in the single-spare row-column case. (d) The flow graph in the single-spare column case. (e) Reconfigurability of edge- and node-disjoint FT FPGA designs for arrays of 100, 400, and 1024 cells for the single-spare row-column case.

vertices are connected to the sink. The maxflow graph for the above fault pattern is shown in Figure 3(b). Note that, as before, a faulty spare (e.g., the spare fault in row 9) means that there is no corresponding edge in the maxflow graph. By reducing the number of vertices in the equivalent maxflow problem, we can speed up the reconfiguration algorithm. We will analyze the expected amount of speedup and give empirical speedup results in Sec. 5.

4.2 The Technique for Single-Spare Row-Column Designs

Next, consider the case of 1S-RC FPGA designs. The additional constraint here is that there are no spares on the left and top sides of the array, and hence paths formed till boundary cells on these sides of the circumscribing rectangle will need to bend to access spares on the right and bottom sides of the array. This is depicted in Figure 3(a) for the example fault pattern considered above. In general, let the circumscribing rectangle be of dimensions $m \times n$ and let it located a distance of m' from the left and n' from the top boundary of the array; $m = 7$, $n = 5$, $m' = 3$ and $n' = 5$ in Fig. 3(a). From our observation above, flow graphs for the double- and single-spare cases, depicted in Figs. 3(b) and (c), respectively, should be the same, except that the latter should incorporate the additional constraint on routing reconfiguration paths from the left and top boundary points of the circumscribing rectangle to spares on the right and bottom. How this additional constraint is incorporated in Fig. 3(c) by edges E_1 – E_3, all of capacity m', and edges E_4 – E_6, all of capacity n', is explained next.

Of all reconfiguration paths exiting the left side of the circumscribing rectangle, a maximum of m' paths may go to the bottom spares (since there are only m' cell columns to the left of the circumscribing rectangle to route these paths)—this restriction is modeled by edge E_1, and a maximum of m' paths may bend and access the spares on the right—this is modeled by edge E_2. Similarly, of the paths exiting the top side, a maximum of n' paths each may access spares on the right and bottom sides of the array—these restrictions are modeled by edges E_4 and E_5, respectively. Furthermore, since there are only m' bottom spares to the left of the circumscribing rectangle, no more than m' paths

exiting the left and top sides can access these spares—this is modeled by edge E_3. Edge E_6 models a similar restriction for spares on the right. The numbers in parenthesis next to the above flow edges in Fig. 3(c) represent the actual flow amount in them for the example fault pattern considered in Fig. 3(a). Thus solving maxflow on the above graph is equivalent to solving the reconfiguration algorithm in the 1S-RC case.

4.3 The Technique for Single-Spare Row/Column Designs

Finally, consider the case of single-spare row or column FPGA designs (see Fig. 3(d)). Here we show how our technique is applied for 1S-C FPGA (one in which there is a spare column on the right); application for 1S-R FPGA is similar. Assume that the circumscribing rectangle of Fig. 3(a) is located a distance of k' from the bottom boundary of the array. Reconfiguration paths exiting the rectangle on the top, left, and bottom sides must bend and access the n' and k' spares on the right to the top and bottom, respectively, of the array. By arguments similar to that made for Fig. 3(c), it is clear why the pairs of edges (E_1, E_2), (E_4, E_5), and (E_7, E_8) in Fig. 3(d) should have capacities m', n', and k', respectively. Of the reconfiguration paths exiting the bottom and left sides of the rectangle, only a maximum of m' can access the top spares—this is modeled by edge E_9; and of these and the reconfiguration paths exiting the top side of the rectangle, only a maximum of n' can access the top n' spares—this is modeled by edge E_6. By symmetry, it is easy to see how edges E_3 and E_{10} model the remaining flow constraints. It should be noted in Fig. 3(d) that since only two spares can be accessed at the bottom of the array (because $k' = 2$), the reconfiguration path for fault $(8, 5)$ in Fig. 3(a) has been diverted upwards and to the right in Fig. 3(d), and that for fault $(9, 4)$ has been diverted to spare $s^r_{11,*}$.

5 Speedup Analysis

We now estimate the expected speedup from using the above technique over the original algorithm of Sec. 3 for a $\sqrt{N} \times \sqrt{N}$ array assuming that faults are distributed randomly across the array. Recall that the time complexity of the reconfiguration algorithm is determined by the number of vertices in the circumscribing rectangle of the fault pattern. In the following, we determine, for a given fault size, the probabilities for circumscribing rectangles of different sizes. Consider an $i \times j$ rectangular subset of the array spanning rows i' through $i' + i - 1$ and columns j' through $j' + j - 1$. Given a fixed-size random fault pattern F, the probability of the event $E_0 \equiv E[i' : i'+i-1, j' : j'+j-1]$ of F lying within this rectangle is

$$p_{i,j} = P(E_0) = \frac{\binom{i \cdot j}{|F|}}{\binom{N}{|F|}} = \frac{(i \cdot j)!\,(N - |F|)!}{N!\,(i \cdot j - |F|)!}, \quad \text{for } i > 0,\ j > 0,\ |F| \leq i \cdot j, \text{ and}$$
$$= 0, \text{ otherwise.} \qquad (1)$$

Note that we do not use i' and j' to index the probability $p_{i,j}$ since it is independent of the location of the circumscribing rectangle. Let $E_1 \equiv E[i' : i' + i - 2, j' : j' + j - 1]$, $E_2 \equiv E[i'+1 : i'+i-1, j' : j'+j-1]$, $E_3 \equiv E[i' : i'+i-1, j' : j' + j - 2]$, and $E_4 \equiv E[i' : i' + i - 1, j' + 1 : j' + j - 1]$. Then, the probability that the fault pattern is circumscribed by the above rectangle, i.e., the probability that all $|F|$ faults are confined to the above rectangle and at least one fault lies on each boundary of the rectangle, is

$$q_{i,j} = P(E_0) - P(E_1 \cup E_2 \cup E_3 \cup E_4). \qquad (2)$$

The second probability on the right hand side of Eq. 2 is given by the inclusion-exclusion formula [14]

$$P(E_1 \cup E_2 \cup E_3 \cup E_4) = S_1 - S_2 + S_3 - S_4, \qquad (3)$$

where S_k is the sum of the probabilities of any k events among the E_i's, $1 \leq i \leq 4$, occuring simultaneously. For example,

$$S_2 = P(E_1 \cap E_2) + P(E_1 \cap E_3) + P(E_1 \cap E_4) + P(E_2 \cap E_3) + P(E_2 \cap E_4) + P(E_3 \cap E_4). \qquad (4)$$

Note that the probabilities of intersecting events can be computed from Eq. 1. For example,

$$\begin{aligned} P(E_1 \cap E_2) &= P(E[i' : i' + i - 2, j' : j' + j - 1] \cap E[i' + 1 : i' + i - 1, j' : j' + j - 1]) \\ &= P(E[i' + 1 : i' + i - 2, j' : j' + j - 1]) \\ &= p_{i-2,j}. \end{aligned} \qquad (5)$$

Thus we can compute $q_{i,j}$ in Eq. 2 using Eqs. 1 and 3.

Let the function $\Gamma(|V|)$ denote the average complexity of the maxflow reconfiguration algorithm on a $|V|$-vertex flow graph. Then, since there are $(\sqrt{N} - i + 1) \cdot (\sqrt{N} - j + 1)$ different possible locations for a $i \times j$ circumscribing rectangle in a $\sqrt{N} \times \sqrt{N}$ array, the average complexity of the fast reconfiguration algorithm for a given fault size $|F|$ is

$$T(N, |F|) = \sum_{i,j=1}^{\sqrt{N}} (\sqrt{N} - i + 1) \cdot (\sqrt{N} - j + 1) \cdot q_{i,j} \cdot \Gamma(i \cdot j), \qquad (6)$$

and the speedup obtained is

$$S(N, |F|) = \frac{\Gamma(N)}{T(N, |F|)}. \qquad (7)$$

We collected speedup results, averaged over ten runs, for the $O(N^3)$ reconfiguration algorithm of Sec. 3 for edge-disjoint FPGA designs. Table 1 depicts empirical speedups obtained for a 1024-cell node-disjoint FPGA array for five

different fault sizes, viz., $|F| = 2, 4, 8, 16, 32$. It also gives corresponding analytical speedups obtained from Eq. 7 assuming the average complexity of the reconfiguration algorithm is $\Gamma(N) = \Theta(N^{1.8})$. Assuming this average complexity gives us the best fit for the analytical speedup with the empirical one for $|F| = 2$ among $\Theta(N^3), \ldots, \Theta(N^{1.9})$, $\Theta(N^{1.8})$, $\Theta(N^{1.7})$, etc. As can be seen, the empirical and analytical speedups are in good agreement over the entire range of $|F|$ values. Note that by using the fast reconfiguration algorithm, we are able to solve the reconfiguration problem in a fraction of the time taken by the regular algorithm. Clearly, as is evident from Table 1 also, speedups will be higher for smaller than larger fault sizes, since the circumscribing rectangles will be smaller in the former compared to the latter case. For the same reason, it is clear that, for a given fault size, speedups will be higher for larger arrays than smaller ones. Moreover, since defects on an FPGA are likely to be localized to small regions, faults are more likely to be clustered than uniformly distributed across the array, so that the circumscribing rectangles will be smaller. Hence, speedups obtained will be more than that predicted by the above analysis.

Fault Size	Empirical Speedup	Analytical Speedup
2	23.05	22.43
4	3.34	4.55
8	2.13	2.07
16	1.25	1.38
32	1.10	1.13

Table 1: Empirical and analytical speedups obtained for a 1024-cell edge-disjoint FT FPGA array for five different fault sizes assuming in the analysis that the average complexity of the reconfiguration algorithm is $\Theta(N^{1.8})$.

6 Reconfigurability of Edge- and Node-Disjoint FT FPGA Designs

Here we present reconfigurability results for the edge- and node-disjoint FPGA designs obtained by using the maxflow-based reconfiguration algorithm of Sec. 3 and Monte Carlo simulations averaged over 300-1000 samples. In Fig. 3(e) we plot the percentage of successful reconfigurations for various fault sizes for the single-spare row-column and double-spare (not shown) row-column cases, respectively, of the above designs. We denote the number of primary cells by N and the number of spares by S. Note that the reconfigurability of the edge-disjoint design is 100% for all fault sizes except those of size $S-2$ or greater—of course it is possible to tolerate only faults of sizes S or less. Even for these large-sized faults, the reconfigurability is 90% or better and in most cases close to 100%. Furthermore, this is much better than the reconfigurability of the node-disjoint design for large fault sizes. These results also demonstrate that the spare-cell utilization in the edge-disjoint design is close-to-optimal.

7 Conclusions

We presented dynamic node covering based FT FPGA designs for increasing yield and for use in mission-critical systems. Two types of designs were considered: one supporting rectilinear node-disjoint and the other supporting rectilinear edge-disjoint reconfiguration paths. By having reconfiguration paths depend upon actual faults and inserting RSs only along actual reconfiguration paths, the dynamic method incurs much less overhead for the same reliability compared to the static method. To facilitate fast online reconfiguration, a speedup technique that speeds up reconfiguration time by as much as 20 times, depending upon the fault set, was presented. The amount of speedup obtainable on average using the technique was analyzed and verified empirically. Finally, we presented reconfigurability results for both the node- and edge-disjoint FT FPGA designs. These show that the reliability achievable via the dynamic node covering approach is indeed high and very close to the best possible with a given number of spares. Future work will explore minimization of track overhead for several real FPGA circuits by incorporating transition costs in the flow graph and solving the resulting mincost flow problem.

References

[1] R.K. Ahuja, et al., *Network Flows*, Prentice Hall, 1993.

[2] T.H. Cormen, et al., *Intro. to Algorithms*, McGraw Hill, 1990.

[3] F. Hanchek and S. Dutt, "Node-covering based defect and fault tolerance methods for increased yield in FPGAs," *Proc. Int. Conf. VLSI Design*, pp. 225–229, Jan., 1996.

[4] F. Hanchek and S. Dutt, "Methodologies for tolerating logic and interconnect faults in FPGAs," *IEEE Trans. Comp.*, special Issue on Dependable Computing, Jan. 1998, pp. 15-33.

[5] N. Hastie and R. Cliff, "The implementation of hardware subroutines on FPGAs," *Proc. CICC*, pp. 31.4.1–31.4.4, 1990.

[6] F. Hatori, *et al.*, "Introducing redundancy in FPGAs," *Proc. IEEE CICC*, pp. 7.1.1–7.1.4, 1993.

[7] S. Khuller and J. Naor, "Flow in planar graphs with vertex capacities," Technical report UMIACS-TR-91-102, CS-TR-2715, University of Maryland, College Park, MD, June 1991.

[8] V. Kumar, et al., "An approach for yield enhancement of program. gate arrays," *Proc. ICCAD*, pp. 226–229, Nov. 1989.

[9] J. McDonald, et al., "A fine grained, highly fault tolerant system based on WSI and FPGA technology," *FPGAs*, W. Moore and W. Luk (eds.), Abingdon EE & CS Books, Abingdon, England, pp. 114–126, 1991.

[10] J. Narasimhan, et al., "Yield enhancement of program. ASIC arrays by reconfiguration of circuit placements," *IEEE T-CAD of ICs and Syst.*, Vol. 13, No. 8, pp. 976–986, Aug. 1994.

[11] K. Roy and S. Nag, "On routability for FPGAs under faulty conditions," *IEEE TC*, Vol. 44, pp. 1296–1305, Nov. 1995.

[12] V.P. Roychowdhury, et al., "Efficient algorithms for reconfiguration in VLSI/WSI arrays," *IEEE Trans. Comp.*, Vol.39, No.4, pp.480-489, Apr. 1990.

[13] V. Shanmugavel, "Low overhead fault reconfiguration techniques for FPGAs," *M.S. Thesis*, Dept. of EECS, Univ. of Illinois at Chicago, IL, 1998.

[14] A. Tucker, *Applied Combinatorics*, John Wiley & Sons, New York, pp. 307, 1984.