Efficient Incremental Rerouting for Fault Reconfiguration in Field Programmable Gate Arrays[*]

Shantanu Dutt,[1] Vimalvel Shanmugavel[1] and Steve Trimberger[2]
[1]{dutt,vshanmug}@eecs.uic.edu [2]steve.trimberger@xilinx.com
[1]Dept. of EECS, U. of Illinois-Chicago [2]Xilinx Inc.

Abstract *The ability to reconfigure around manufacturing defects and operational faults increases FPGA chip yield, reduces system downtime and maintenance in field operation, and increases reliabilities of mission- and life-critical systems. The fault reconfiguration technique discussed in this work use the principle of node covering in which reconfiguration is achieved by constructing replacement chains of cells from faulty cells to spare/unused ones. A key issue in such reconfiguration is efficient incremental rerouting in the FPGA. Previous methods for node-covering based reconfiguration are "static" in the sense that extra interconnects are added a-priori as part of the initial circuit routing so that a specific fault pattern (e.g., one fault per row) can be tolerated [1]. This, however, results in worst-case track overheads and also in an inflexibility to tolerate other realistic fault patterns. In this paper, we develop dynamic reconfiguration and incremental rerouting techniques that are fault specific. In this approach, the FPGA is initially routed without any extra interconnects for reconfiguration. When faults occur, the routed nets have to be minimally perturbed to allow these interconnects to be inserted "on-the-fly" for reconfiguration. These requirements are addressed in our minimally incremental rerouting technique* `Conv_T-DAG`*, which uses a cost-directed depth-first search strategy. We prove several results that establishes the near-optimality of* `Conv_T-DAG` *in terms of track overhead. To the best of our knowledge, this is the first time that an incremental rerouting technique has been developed for FPGAs. For several benchmark circuits, the static approach to tolerating one fault per row resulted in a 43% to 34% track overhead. Using the dynamic reconfiguration approach and* `Conv_T-DAG` *results in an average overhead of only 16%—an improvement of more than 50%. Over all circuits, the reconfiguration time per fault ranges from 16.8 to 72.9 secs. Simulation of smaller fault sets of one to four faults show very small track overheads ranging from 1.75% to 4.49%.* `Conv_T-DAG` *can also be used for interconnect fault tolerance.*

Keywords: dynamic fault reconfiguration, FPGA defect/fault tolerance, incremental circuit rerouting, reconfiguration time, track overhead.

1 Introduction

Fault tolerance (FT) is the ability to retain full or partial functionality of the chip after occurrence of faults. FT increases the fabrication yield of chips, and raises reliability and availability of an FPGA-based platform for the user. Since FPGAs are composed of a large number of identical cells in a regular array, a software approach to FT is readily apparent. This is essentially based on noting the locations of faulty cells and completely rerouting the circuit in the FPGA or a subarray to avoid them using spares or unused cells instead [2, 3, 6, 7]. However, requiring the layout tools to perform a new routing of a circuit for each new faulty cell or wiring location encountered puts a heavy burden on the user, who must also keep track of all of the different routings for a given circuit design. Furthermore, in a mission/life-critical scenario, it is desirable to perform fault reconfiguration quickly so that system function is suspended for only seconds. However, complete rerouting of the entire circuit on an FPGA generally takes in the order of hours.

Of the previous work in FT FPGAs, the method of interest to this research is the *node covering* technique [1], which has been developed for segmented-array architectures like those of Xilinx, ORCA and Altera Flex FPGAs. This method is also a software approach, but has several distinguishing features from other methods: it requires neither the factory nor the user to generate new routing maps to reconfigure around faulty cells or wiring; instead, the original configuration data can be reused. This FT method involves a routing strategy that uses additional wiring segments, termed *reserved segments (RSs)*, at all branch points of the circuit nets. This results in retaining total functionality with somewhat reduced routability. The RS locations are independent of actual fault locations and hence this technique is *static.*

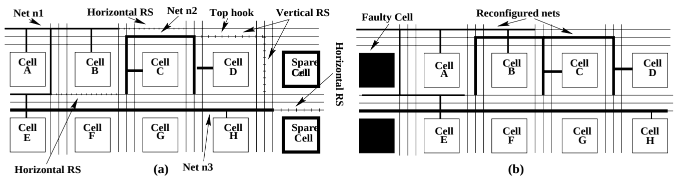

Figure 1: *(a) Fault tolerant routing of nets using reserved segments. (b) Reconfiguration around faulty cells.*

The motivation for this research is to retain total functionality with as little track and reconfiguration-time overhead as possible when faults occur, and to increase FT and reconfiguration flexibility compared to the static method. The new FT technique uses a similar node-covering concept, but makes RS insertions only specific to faults wherever and whenever required. It is thus a *dynamic* technique; it also reduces the number of RSs required for FT. However, this shift from a static method to a dynamic one introduces additional algorithmic complexities, since techniques need to be devised that can perturb the initial routing efficiently (in terms of both track and time overheads) to accommodate the RSs needed for reconfiguration. It is also desirable to constrain the perturbation space so that timing delays on all or the most critical nets do not increase. In this paper, we develop efficient algorithms for dynamic FT that address these issues. In the sequel, we will describe our techniques in the context of track segments of length one in the FPGA, though the methods are readily extendible to multiple-length segments.

2 The Static Node Covering Method

In the node-covering approach, each logic cell u in the FPGA is assigned a *cover cell* which can be reconfigured to replace it in the event that u becomes faulty. The FT FPGA designs contain one spare cell at the end of each row, and so are able to tolerate one faulty cell per row. In order for a cell to cover another cell (the *dependent cell*), the cover cell must be able to duplicate the functionality of the dependent cell. 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 track segment also includes the corresponding track segment—a *cover segment*—bordering the cover cell. 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. Otherwise, additional segments, termed *reserved segments (RSs)*, should be attached to the net to provide covers; see Fig. 1. In the static method, RSs are included at every branch-point and every rightmost end-point of nets so that single faults at any location in each row can be tolerated.

Across a number of benchmark circuits, the track overhead incurred in a FT circuit using the static method was found to be an average of 43% for the best routing heuristic and 34% for the best combination of heuristics [1].

[*]This work was funded partly by Darpa Contract # F33615-98-C-1318 and partly by a grant from Xilinx Corp.

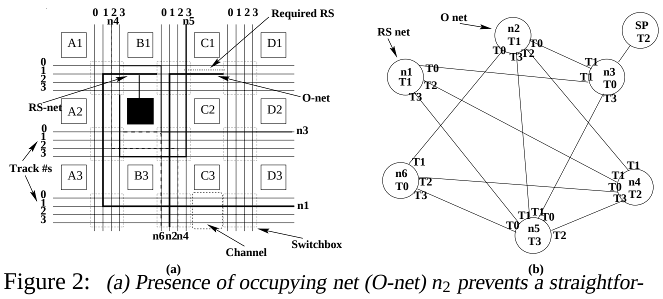

Figure 2: *(a) Presence of occupying net (O-net) n_2 prevents a straightforward insertion of an RS. Thus the O-net must be moved to another track, possibly bumping other nets. (b) Overlap graph representation for the given routing. The track number at an end-point of an edge indicates the track the corresponding net would move to if this edge is traversed.*

3 The Dynamic Fault Tolerance Method

Once a fault location is known (using well-known detection and diagnosis techniques), a reconfiguration path from it to a spare cell can be determined using, for example, network flow techniques [5]. The spare cells can either be those that are statically allocated at the periphery of the FPGA, those that are unused by the working circuit at various intermediate positions of the FPGA, or those that are become dynamically available due to the on-line roving of a tester in the FPGA [8]. All RSs needed along this path can also be determined as described in Sec. 2. For each RS, if the required wire segment where the RS is to be inserted is vacant, the insertion is accomplished by including this segment as part of the corresponding net. If the required wire segment is occupied by another net, then the RS insertion will cause a displacement or "bumping" of this net. As shown in Fig. 2(a), 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 RS insertion problem can now be stated as follows. The RS-net has to be extended by one segment towards the direction of the cover cell, and this segment is currently occupied by the O-net. Thus the O-net needs to be moved 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, and denoted by $n_i^{T_j \rightarrow T_k}$. This transition may result in 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 transition sequence is shown in Fig. 3b by dark arcs, where net n_2 initiates a set of transitions which finally terminate in "spare" nodes, which are vacant segments of appropriate total lengths in which a bumped net can move in without bumping any other net. As seen in the figure, the set of transitions take on a directed-acyclic graph (DAG) structure, termed a *transition DAG (T-DAG)*, with the spares forming the leaf nodes. The RS insertion is successful if a T-DAG rooted at the corresponding O-net can be found whose leaves are spare nodes; such a T-DAG is termed a *converging T-DAG*. Note that we are performing a minimal incremental rerouting of the FPGA by this process of RS insertions and net bumpings. Further, since net bumpings only involve nets changing their track positions but not their topologies, the timing properties of these nets remain intact. Since RS insertions affect RS-net delays minimally (by about 7% [1]), this incremental rerouting approach only marginally affects FPGA performance.

3.1 The Overlap Graph

The *overlap graph* (*OG*) is a graph representation of the circuit routing on the FPGA. In the overlap graph $OG(V,E)$, the set of nodes $V = S \cup \{n_1, n_2, \ldots, n_m\}$, where each n_i is a routed net of the circuit and S is the set of "spare" track segments as described above. 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[1] in the FPGA. Fig. 2(b) shows nets n_2 and n_6 having an edge between them in the OG since they are routed through a common vertical channel on the right of the faulty cell. We define $adj(n_i)$ as the set of all nets adjacent to n_i in the OG. Let n_i be on track T_j; then for a track $T_k \neq T_j$, we define $adj^{T_k}(n_i)$ as set of all nets on track T_k that are adjacent to n_i in the OG.

[1]A *channel* is the set of all track segments between two adjacent switchboxes of the FPGA; see Fig. 2.

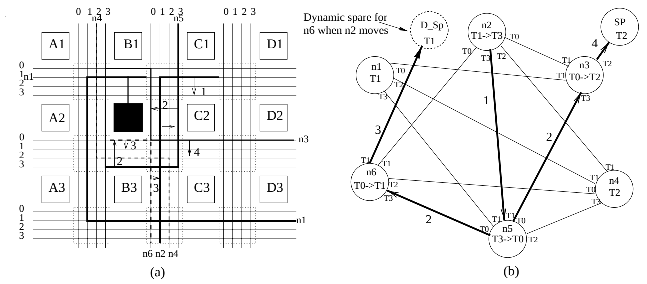

Figure 3: *Searching the OG for a converging transition DAG to determine feasibility of RS insertion.*

The OG can be used as an effective model in the evaluation of the required T-DAG. Since $OG(V,E)$ represents the circuit routing on the FPGA, a T-DAG is a DAG embedded in the OG (the undirected edges of the OG become directed arcs in the direction of the transitions; see Fig. 3b).

3.2 Determining a Converging Transition DAG

A converging T-DAG rooted at an O-net can be determined by performing a search on the DAG until all leaf nodes of the search DAG are spare nodes. The search is, however, not straightforward, since the OG changes as we traverse it—its labels indicating net occupancies of tracks (see Fig. 2) change with each net transition.

This process is illustrated in Fig. 3 for a small circuit and for a single RS insertion for extending net n_1, the RS-net. The corresponding O-net n_2 transits from T_1 to T_3 and bumps into n_5. The movement of n_2 from T_1 creates a "dynamic" spare node (labeled by D_Sp in the figure) for net n_6, which information is added to the OG. The bumped net n_5 then transits from T_3 to T_0 where it bumps n_6 and n_3. n_6 then transits to the above dynamically created spare on T_1, while n_3 transits to its spare node on T_2. Thus a converging T-DAG (here it is a tree) is determined in the OG. The transition arcs are shown dark in Fig. 3b and numbered chronologically in the order in which they are traversed in the search process. After every transition, the track labels of the nodes and those of their edges need to be updated.

Algorithmic Issues: A number of questions need to be answered for developing an algorithm for searching a dynamically changing OG for converging T-DAGs for all required RSs for the given set of faults.

1. For a single RS insertion, each transition of a node, starting with the root node (the O-net), can generate multiple branches in the search (when the transition causes multiple nets to be bumped). Is the order in which the branches are searched determinant of the success or failure of finding a converging T-DAG?
2. What are the terminating conditions of the search process of a particular branch?
3. Are there ways of guiding the search process so that much fewer backtracks are needed compared to randomly making transition choices?
4. For multiple RS insertions typically required for reconfiguring around a single fault, is the order in which they are processed determinant of the success or failure of finding converging T-DAGs for all of them? Similarly for multiple faults.

The questions posed are answered in the following results.

Order Independence in Branch Traversal:

Theorem 1 *If there exists a converging T-DAG for a net $n_i^{T_j}$, then it will be found by a search process irrespective of the order in which the branches of the DAG are searched.*

The above result is useful since it eliminates the need for any order in which the branches of the transition DAG have to be evaluated for convergence. In particular, if a depth-first order of searching the branches it used, the algorithm becomes space-efficient and thus also time-efficient in a practical sense. The following corollary to Theorem 1 is also relevant.

Corollary 1 *If there exists a converging T-DAG for a net $n_i^{T_j}$, then it will be found by a search process irrespective of the order in which its possible transitions are investigated.*

Terminating Condition: An obvious termination of the search process at a branch in the current DAG occurs when the node at the branch transits to a spare node; this is a successful termination. However, in order to keep the search process tractable, we also need to prune certain transitions that are not helpful in finding a solution, and backtrack up the branch to try alternative transitions. The following theorem establishes that when evaluating a transition, if we form a cycle in the transition graph created so far, i.e., we revisit an ancestor of the current node in the current DAG, then we can reject the transition that formed the cycle and try an alternate transition for the node without sacrificing optimality.

Theorem 2 *If there exists in the OG a converging directed cyclic transition graph rooted at node n_i, then there exists a converging T-DAG rooted at n_i.*

Heuristic Transition Costs: Corollary 1 establishes that if a converging T-DAG exists, then it will be found irrespective of the order in which various possible transitions of a node n_i is investigated. If a transition is unsuccessful according to the condition established in Theorem 2, then an alternative transition is investigated. However, it will be time-efficient if an appropriate heuristic measure can be used to determine which transitions are more likely to be successful so that fewer T-DAGs are searched and backtracked. A good heuristic cost will consider both the "magnitude" of bumpings (total length of bumped nets) and the likelihood of convergence of these bumpings. Three different transition cost heuristics have been evaluated; two are reported here:

1. $sum(n_i^{T_j \to T_k}) = \sum_{n_j \in adj^{T_k}(n_i)} l(n_j)$ where $l(n_j)$ is the total length of n_j in terms of the track segments (each of length 1) that it occupies. This heuristic is reasonable, but only considers the bumping magnitude. For example, according to it, it is equally costly to bump a net of length 9 as it is to bump 3 nets each of length 3. However, the latter case has a higher likelihood of convergence since there is greater flexibility in moving 3 bumped nets than a single net of the same total length. This leads to the next cost function.
2. $sqrt(n_i^{T_j \to T_k}) = [\sum_{n_j \in adj^{T_k}(n_i)} l(n_j)] / \sqrt{|adj^{T_k}(n_i)|}$.

Section 4 presents empirical data that demonstrates the benefit of these transition-cost (TC) functions in providing time-efficient searches, by comparing the search time to that of a partially random selection process.

Insensitivity or Sensitivity to RS Insertion Order? A net connected in a vertical channel requires two RSs, one horizontal and one vertical as shown in Fig. 1. The horizontal RS is called a *hook*, and it can be placed on the channel above the faulty cell (*top-hook*) as shown in Fig. 1, or in the channel below the faulty cell (*bottom-hook*). The choice of top-hook or bottom-hook for a vertical RS is made based on the minimum TCs of the set of corresponding O-nets in the two channels.

It is clear that the TCs of possible top and bottom hooks of a vertical RS can be affected by the top/bottom hook choices for previously inserted vertical RSs. Thus the choice of hook position is potentially sensitive to their insertion order, and thus so is solution feasibility (finding a solution in the number of available tracks) and track overhead. However, horizontal RSs and the vertical portions of vertical RSs have to be inserted in a particular channel; there is no choice in the matter. We thus have the following result.

Theorem 3 *The order in which insertions of horizontal RSs and the vertical segments of vertical RSs are made does not affect the track overhead if all net retractions have been made before the RSs are inserted.*

This result brings brings forth the advantage of making all net retractions (vacating terminal net segments of reconfigured nets connected to replaced cells) first. We thus use the following strategy for inserting RSs for reconfiguring single or multiple faults.

Algorithm `Reconfigure`
1. Make all horizontal and vertical RS-net retractions.
2. Insert all horizontal RSs and vertical portions of vertical RSs in any order (solution feasibility and track overhead is insensitive to this order by Thm.3) by calling the converging T-DAG finding algorithm `Conv_T-DAG` (Fig. 4) for each RS.

```
Algorithm Conv_T-DAG($OG, n_i^{T_j}$) /* Find a converging T-DAG rooted at $n_i^{T_j}$ */
Begin
    $m = 0$; /* Number of transitions tried */
    while ($m < t_o$) do begin /* $t_o$ is the number of occupied tracks */
        m=m+1;
        $k$ = Get_next_best_transition($n_i^{T_j}, m$); /* $k$ is the track number of the $m$th best transition according to the heuristic cost function used */
        if ($adj^{T_k}(n_i) == \emptyset$) then /* there exists a spare for $n_i$ on $T_k$ */
            return(1) /* a converging T-DAG was found for $n_i^{T_j}$ */
        else begin
            for each $n_r \in adj^{T_k}(n_i)$ do begin
            if ($n_r$ marked "visited") then break; /* this transition to $T_k$ results in a cycle; try the next best transition */
            Update OG to reflect that $n_i$ has moved to track $T_k$;
            for each $n_r \in adj^{T_k}(n_i)$ do
                Update OG to reflect that $n_r$ is no longer on track $T_k$;
            $numb\_succ = 0$; /* this records # of $n_r$s for which converging T-DAGs are found */
            for each $n_r \in adj^{T_k}(n_i)$ do begin
                Mark $n_r$ as "visited";
                $success$ = Conv_T-DAG($OG, n_r^{T_k}$); /* find converging T-DAGs rooted at each $n_r$ in depth-first order */
                Mark $n_r$ as "unvisited"; /* $n_r$ can be visited again without forming a cycle */
                if not($success$) then break; /* we need to find converging T-DAGs for all $n_r \in adj^{T_k}(n_i)$ in order for $n_i$ to make a successful transition to $T_k$ */
                else if ($success$) then $numb\_succ = numb\_succ + 1$;
            endfor
            if ($numb\_succ == |adj^{T_k}(n_i)|$) return(1); /* converging T-DAGs were found for all nets in $adj^{T_k}(n_i)$ */
        endelse
    endwhile
    return(0). /* no transition of $n_i$ was successful */
End. /* Conv_T-DAG( ) */
```

Figure 4: *The optimal depth-first search algorithm for finding a converging T-DAG.*

3. In some order evaluate hook position choice and insert them by calling `Conv_T-DAG` for each such RS; unfortunately solution feasibility and track ovhd is not impervious to this order.

End /* `Reconfigure` */

Incremental Rerouting Algorithm: The optimal depth-first search based converging T-DAG algorithm `Conv_T-DAG` for performing minimal incremental rerouting is presented in Fig. 4. The algorithm is based on the concepts and results established above and is self-explanatory.

Theorem 4 *Algorithm* `Conv_T-DAG` *is optimal with respect to finding a set a net transitions for inserting an RS in a given channel with the given number of tracks, i.e., it will find a solution within the given number of tracks if it exists.*

Theorem 5 *Algorithm* `Reconfigure` *will find a set of net transitions for inserting all horizontal RSs and the vertical portions of vertical RSs with minimal track overhead.*

Note that we do not claim optimality in terms of minimal track overhead for the entire reconfiguration process, since for vertical RSs it includes inserting hooks whose channel choice is based on a heuristic cost function. The cost functions however, seem to work well in practice. Also, since the hooks are about one-third of all RSs to be inserted, it will not be amiss to state that `Reconfigure` is near-optimal.

4 Experimental Results and Discussion

The dynamic fault tolerant techniques were tested on a number of benchmark circuits (obtained from the U. Toronto FPGA group) ranging from about 80 cells and nets to about 500 cells and nets. The input to our software is the mapped and routed circuits produced by their SEGA tools [4]. The metrics of interest for circuit reconfiguration around faults are FPGA reliability & yield increases, increase in circuit delay, track overhead and reconfiguration time. Since fewer RSs are inserted for the same fault patterns in the dynamic rerouting method compared to the static technique [1], the first and second metric for the static method are lower and upper

Name	FPGA size		Non-FT	Static method		Dynamic method			Overhead		Reconf_time/fault (secs)	
	Nets	size	tracks	tracks	ovhd%	worst	average	std_dev	worst%	average%	worst	average
apex7	124	11x11	12	18	50	15	14.24	.91	25	18.66	3.729	2.335
alu4	236	18x18	12	17	42	15	13.64	.68	25	13.6	22.287	8.307
term1	88	9 x 9	11	15	36	14	12.88	.816	17	13.66	3.324	1.252
alu2	138	14x14	8	12	50	11	10.88	1.0	38	36.00	22.007	1.349
example2	197	13x13	14	18	29	16	14.44	.40	14	3.14	23.837	10.773
too_large	177	14x14	11	16	45	15	13.26	.72	36	20.72	15.870	7.238
C499	142	12x12	11	17	54	14	13.32	.54	27	21.09	13.176	6.343
9symml	71	12x12	7	10	43	9	8.56	.63	29	22.28	.501	.350
busCntrl	144	24x24	9	13	44	12	10.76	.70	33	19.55	2.203	2.061
CbnrD4	454	23x23	14	17	21	15	14.84	.54	7	6.00	96.086	45.171
DbnrD4	489	21x21	13	17	30	14	13.80	.40	8	6.15	143.242	60.197
EbnrD4	326	17x17	10	14	40	14	11.72	.49	40	17.2	43.058	14.386
dmaD4	195	22x22	9	14	56	12	11.68	.46	33	9.77	7.82	4.627
dram_fsm	391	10x10	9	11	22	11	9.92	.72	22	10.22	58.233	21.60
C1335	141	12x12	13	17	31	15	13.9	.40	15	7.0	48.986	19.942
C880	173	13x13	14	20	43	18	16.00	0.84	29	14.28	21.623	10.502
vda	216	16x16	14	25	79	19	17.20	.74	36	22.85	35.132	17.415
k2	388	21x21	17	27	59	20	19.16	.61	18	12.70	206.10	98.460
Total			208	298	774	262	240.2				767.214	332.308
Average			11.55	16.55	43	14.55	13.35		25.96	15.48	42.623	18.4615

Table 1: *Track overheads and reconfiguration times for worst- and average-case fault patterns using the sum TC heuristic.*

bounds, respectively, for the dynamic technique. These two metrics were shown to be very good for the static method [1], and these will thus be even better for the dynamic one. Hence we only focus on the last two metrics, track overhead & reconf. time.

Fault simulations were carried out for three cases: (1) The worst case fault pattern in which one fault per row is simulated with the leftmost cell in each row being faulty. In terms of the number of RSs, this worst-case pattern will match the static method; thus RSs are required at all net branch-points and right end-points. (2) One random fault per row; average results are collected over 25 runs. (3) Only 1-4 faults are simulated.

The results of the first two simulations for the three different TC heuristics have been tabulated, though only the results for the sum heuristic is shown in Table 1; the overall results of the other heuristic, and for the limited faults case are stated in the discussion below.

- The dynamic method shows considerable improvement over the static method for all the different TCs. The static method has a track overhead of 43% with its best combination routing heuristics producing an overhead of 34%. The dynamic method's worst case simulations produce the following overheads: sum TC = 25.96%; square root TC = 25.96%.
- Average case Monte-Carlo simulations show even more significant improvements in track overheads over the static method: sum TC = 16.68%; square root TC = 15.58%.
- The sum TC heuristic is the most time-efficient for both the worst- and average-case simulations with average reconfiguration times over all circuits of about 43 and 18 secs, respectively. In the future we will improve our data structures to decrease these times by factors of two to four. By noting the reconfiguration times per fault with increasing number of circuit nets we can conclude that the growth is that of a low-degree polynomial (most likely quadratic)[2].
- The suitability of heuristic TCs for time-efficiency was demonstrated by an experiment in which we randomly chose transitions only for the O-net (root of the T-DAG); the transitions for the other nodes were chosen based on the sum TC[3]. On the average, the partial-random case shows an increase in reconfiguration time over that of the full sum TC case by 65% for the worst-case fault-pattern and by 126% for the average-case pattern. The track overheads remain the same as that of the sum TC used. These results also serve as empirical validation of Theorem 1 and Corollary 1.
- The last simulation considers a limited number of faults; this case occurs mostly in fabrication defects and short-life missions. The track overheads for all cases are very small: 1.75% for 1 fault to 4.49% for 4 faults. Except for one circuit, all other circuits require an absolute of 1 extra track to tolerate 4 faults; a few do not require any extra tracks!

[2]The RS insertion problem is probably NP-complete (graph coloring could be reducible to it–we will investigate this in future work), though its average complexity is probably tractable as demonstrated by our experimental results for our near-optimal algorithm `Reconfigure`.

[3]The reason for having partial randomness as opposed to randomizing all transition choices was to keep the simulation time tractable.

5 Conclusions and Future Work

The dynamic FT approach was proposed as an alternative to the static approach to increase reliability and flexibility in tolerating fault patterns, and to reduce track overheads. The algorithmic issues for the dynamic approach, and in particular for incremental rerouting, were exposed, and addressed by establishing formal results. Based on these results, near-optimal and time-efficient algorithms were developed and tested. The track overheads improved over those of the static method by more than 50%, and the reconfiguration times per fault were shown to be in the order of 16 to 73 secs. It was encouraging to note that for tolerating a limited number of faults (one to four), either zero or one extra tracks were required for all but one of the circuits. These results augur well for the application of the dynamic FT technique to significant yield, availability and reliability improvements.

Our algorithms can also naturally tolerate interconnect faults by finding converging T-DAGs for nets that occupy faulty interconnect segments. Our OG-based model and T-DAG search algorithm `Conv_T-DAG` may also be extendible to perform incremental rerouting for VLSI to support incremental VLSI re-design. Further, it appears that `Conv_T-DAG` can be augmented to develop a minimal-track FPGA detailed routing algorithm. These issues will be investigated in future research.

References

[1] F. Hanchek and S. Dutt, 'Methodologies for Tolerating Logic and Interconnect Faults in FPGAs", *IEEE Trans. Computers*, Special Issue on Dependable Computing, Jan. 1998, pp. 15-33.

[2] V. Kumar, A. Dahbura, F. Fisher, and P. Juola, "An approach for the yield enhancement of programmable gate arrays," *Proc. IEEE ICCAD*, pp. 226–229, Nov. 1989.

[3] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Low Overhead Fault-Tolerant FPGA Systems," *IEEE Transactions on VLSI Systems*, Vol. 6, No. 2, 1998.

[4] G. Lemieux and S. Brown, "A Detailed Router for Allocating Wire Segments in FPGAs", *ACM Physical Design Workshop*, CA, pp. 215 - 226, April 1993.

[5] N. R. Mahapatra and S. Dutt, "Efficient Network-Flow Based Techniques for Dynamic Fault Reconfiguration in FPGAs", *Proc. 29th Fault-Tolerant Computing Symposium (FTCS-29)*, pp. 122-129, June 1999.

[6] J. Narasimhan, K. Nakajima, C. Rim, and A. Dahbura, "Yield enhancement of programmable ASIC arrays by reconfiguration of circuit placements," *IEEE Trans. Comput.-Aided Design of Integrated Circuits and Syst.*, Vol. 13, No. 8, pp. 976–986, August 1994.

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

[8] Vinay Verma, "Reconfiguration Issues in On-Line BISTer Roving in Fault-Tolerant FPGAs", *Digest of Student Papers—29th Fault-Tolerant Computing Symposium*, pp. 48-50, June 1999.