Tolerating Operational Faults in Cluster-based FPGAs*

Vijay Lakamraju and Russell Tessier
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003
{vlakamra, tessier}@ecs.umass.edu

Abstract

In recent years the application space of reconfigurable devices has grown to include many platforms with a strong need for fault tolerance. While these systems frequently contain hardware redundancy to allow for continued operation in the presence of operational faults, the need to recover faulty hardware and return it to full functionality quickly and efficiently is great. In addition to providing functional density, FPGAs provide a level of fault tolerance generally not found in mask-programmable devices by including the capability to reconfigure around operational faults in the field. In this paper, incremental CAD techniques are described that allow functional recovery of FPGA design configurations in the presence of single or multiple operational faults. Our preferred approach to fault recovery takes advantage of device routing hierarchy in architectural families such as Xilinx Virtex [2] and Altera Apex [3] to quickly swap unused logic and routing resources in place of faulty ones within logic clusters. These algorithms allow for straightforward implementation within a local fault-tolerant system without the need to access a remote processing location. If initial recovery attempts through localized swapping fail, an incremental router based on the widely-used PathFinder maze routing algorithm [10] can be applied remotely in an attempt to form connections between newly-allocated logic and interconnect based on the history of the initial design route.

1 Introduction

As reconfigurable devices grow in capacity to include millions of logic gates their role as computing devices becomes increasingly diverse. By virtue of their propensity for functional specialization and reduced power consumption, FPGAs have recently become important design components in computing platforms which demand significant operational fault tolerance [5]. While the demonstrated benefits of FPGAs have broadened their appeal to this computing sector, increased die sizes and lower operating voltages have increased concern about potential in-field fault risks due to gamma radiation and metal stress [19]. This concern is particularly acute given the less-than ideal operating environment in which these systems are frequently deployed. While the programmable nature of reconfigurable devices would seem to make them ideal platforms to promote run-time fault recovery, few practical recovery techniques have been implemented to date. This has been due in part to the fact that until recently the large majority of device transistors have been isolated in the global routing matrix of the FPGA, necessitating time-consuming net re-route procedures for almost all potential device faults. Additionally, early FPGA architectures generally contained relatively small amounts of logic and routing resources compared to today's devices making the fault recovery problem in years past nearly as complex as initial design layout.

Recent trends in FPGA device architecture have modified this viewpoint considerably. Most contemporary programmable logic devices now support multiple levels of architectural hierarchy containing both tightly-connected *clusters* of multiple LUT/FF pairs and a less-populated global interconnect grid containing channels of wire segments and switch matrices. Since a sizable percentage of potential fault points are currently located inside coarse-grained clusters, faults in these areas can often be addressed without the need for incremental rip-up and re-try. For these existing cluster-based architectures, intra-cluster interconnect structures facilitate straightforward substitution of unused cluster interconnect, LUTs, and flip-flops for faulty resources using low-complexity placement algorithms.

While intra-cluster resource exchange is effective for many faults, routing grid and switch matrix failures generally require at least partial net re-route in order to reconnect routed signals. To support a fully integrated solution in one system, an incremental router has been developed based on the PathFinder [10] routing algorithm. Unlike previous incremental routers [20] [12], our approach leverages route history from the *initial* route to better complete net connections. The router is shown to be highly-effective in successfully recovering from hundreds of interconnect faults in a fraction of the time that would be required to re-route the circuit from scratch. The demonstrated effectiveness of the router makes it appropriate not only for fault recovery but also in the implementation of dynamically-reconfigurable computing circuits which have communication patterns that may change frequently.

Our system is designed to provide multiple levels of system recovery for FPGA devices that have been diagnosed as faulty by fault detection methodologies [14] [21]. Intra-cluster resource swapping is structured to limit the need for compute power and memory facilitating its implementation on remaining functional resources of a fault tolerant system. Incremental routing is designed to be performed off-line by a remote system with superior computational resources and storage capacity.

The organization of this paper is as follows. In Section 2 a description of the issues involved in providing FPGA fault tolerance

* V. Lakamraju is supported by Darpa Order E349: Contract F30602-96-1-0341

is presented. Section 3 describes previous work in FPGA fault tolerance and overviews cluster-based FPGAs. In Section 4, our CAD system is described. Experimental results obtained by applying the system to a collection of FPGA benchmarks is presented in Section 5. Finally, Section 6 summarizes our work and outlines directions for future work.

2 Problem Definition

2.1 Operational Faults

Like all discrete semiconductor devices, a field programmable gate array can be adversely affected by faults at various stages of component lifetime. While most defects appear immediately following fabrication, occasionally, after extended periods of device use, *operational* faults can affect in-service programmable components. Common operational faults include open/short metal (4-17% of faults) and transistor stuck-at faults (25-75% of faults) [22] with manifestation rates that vary based on system environmental conditions such as exposure to gamma radiation and extreme temperature. In general, recent trends in FPGA architecture increase the vulnerability of devices to faults. As VLSI feature sizes shrink and threshold voltages are reduced, the likelihood of oxide breakdown and electromigration grows. Depending on their duration, faults can be either transient or permanent. In general, it is possible to recover from transient faults by reprogramming the FPGA with the original configuration bit stream. In case reprogramming the device is unsuccessful, the fault is considered to be permanent.

Unlike manufacturing defects, which can often be overcome via the use of spare routing wires and programmable fuses set at the factory [17], operational failures must be addressed by generating a new programming configuration for a circuit with the same functionality as the original. In this paper stuck-at 1, stuck-at 0, wire open, and wire short operational faults are considered. Faulty resources are avoided by determining new design configurations that avoid the use of faulty resources such as look-up tables, flip-flops, multiplexers, pass transistors, and wire segments.

2.2 Fault Recovery System Model

In order to support fault recovery, our CAD tools make specific assumptions about the environment in which an FPGA is deployed. These assumptions are generally consistent with existing computing systems [5] [25] that benefit from fault recovery.

- **Hardware Redundancy** - For systems with hard real-time constraints, hardware redundancy allows for periodic functional hardware test and fault recovery of system components with no system down-time. This often means complete hardware redundancy of critical subsystems, including FPGAs and associated microprocessors and memories that might be used to configure them. For systems that can tolerate system down-time, hardware redundancy is not needed to recover a faulty FPGA if the microprocessor and associated memories can continue functioning normally during the recovery effort.
- **External Interface** - With the advent of the internet, many real-time and fault tolerant systems have the capability to communicate with computationally-powerful remote systems through standard protocols. For special-purpose platforms, such as space and avionics systems, this communication may take place through a dedicated link. In our system it is assumed that if device recovery cannot be performed by the local system, remote computing resources can be accessed to aid in the recovery effort.

For many computing platforms and especially for time-critical systems, it is highly desirable to perform fault recovery in seconds rather than minutes or hours. This constraint generally precludes the option of re-placing and re-routing an FPGA from scratch. In general, it is assumed that the local fault tolerant system has a modicum of compute power in the form of a microprocessor or microcontroller and a small amount of memory necessary to perform basic intra-cluster resource swapping but not more intensive incremental routing. For interconnect faults, the fault tolerant system has the capability to pass the location of the fault to a remote system which can quickly perform re-route in seconds using routing graph cost information derived during initial device routing. While five years ago the notion of remote, automated access to vendor place-and-route tools would have been far-fetched, new approaches to web-based FPGA CAD make this approach a much more likely scenario [15]. Additionally, the practice of fault tolerant systems contacting remote locations for service without user intervention has been in place for over ten years [25].

2.3 Cluster-based Architectures

While early FPGA architectures typically contained simple logic blocks containing one or two LUT/flip-flop pairs, more recent devices [2] [1] [3] have clustered multiple LUT/FF pairs together into a single *cluster* to take advantage of design locality and to reduce FPGA place-and-route time. A key action in designing these architectural families has been determining the granularity of the logic cluster. As previously described by Betz and Rose [6], if logic clusters contain insufficient logic resources, the amount of inter-cluster routing resources needed for routing will be great and if clusters contain excessive amounts of logic, much of these resources will be wasted. Figure 1 shows a generalized model of a cluster-based FPGA device. Each cluster contains N basic logic elements (BLEs), each possessing a single look-up table/flip-flop pair. The cluster has a total of I inputs and O outputs which connect cluster logic to the surrounding interconnection matrix. In [6] it was determined that the appropriate relationship between N and I is $I = 2N + 2$. Unless otherwise noted, this architectural relationship is followed in subsequent experiments.

In this paper we extend the Betz model to provide a more accurate view of a range of commercial FPGA architectures by providing an output multiplexer (OMUX) for cluster outputs and allowing for reduced fanout between cluster inputs and input multiplexers (IMUX). In general, cluster-based architectures (e.g. Virtex, Apex) contain output multiplexers to allow for flexible interconnection between BLE outputs and channel wiring. While output multiplexers are frequently fully populated (e.g. every BLE can drive each cluster output), input multiplexers may be either fully (Altera Flex10K, Apex) or partially (Xilinx Virtex, XC5200) populated. In this paper we indicate that the fraction of cluster inputs that can be connected to each BLE input by a parameter F_i that ranges between 0 and 1.

The relatively tight interconnect structure found inside clusters offers opportunities for rapid, localized fault recovery. Intra-cluster fault tolerance can be achieved both by attempting to leverage unused inputs on look-up tables for fault substitution and by swapping faulty BLEs with unused functional ones. It will be shown that the assignment of cluster inputs to BLE inputs plays an impor-

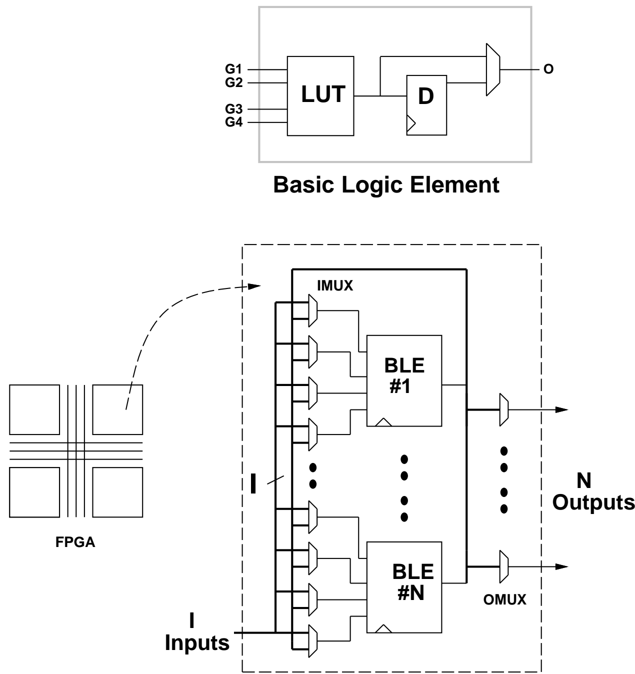

Figure 1: Basic Logic Element and Logic Cluster

tant role in determining the feasibility of exchanging one BLE for another during fault recovery.

3 Related Work

Our research extends previously-reported CAD techniques for overcoming operational FPGA faults. The large majority of previous CAD approaches in this area have focused on recovery from logic cluster (block) faults rather than interconnect faults. In general, these approaches require the functionality of an *entire* cluster be reimplemented in an unused cluster if a fault is detected. In [13], a fault recovery approach for logic block defects was described that reserved spare rows and columns of logic blocks to overcome individual block failures. While this approach allowed for recovery with no required on-line device re-route, track width penalties as high as 35% were reported. In [21], FPGA arrays were divided into a collection of tiles, each of which could be implemented in one of many pre-compiled layouts. If a logic block fault within a tile occurred, a new tile configuration which left the affected block unused could be substituted. Recently, an incremental placement approach was described that uses on-line min-max positioning to quickly move faulty logic blocks to unused device blocks [11]. Not only did this technique and most other block movement approaches require incremental re-route following placement, its applicability to coarse-grained cluster-based architectures is limited. Effectively an entire cluster would have to be removed from use even if a fault affected only an isolated LUT or FF of a cluster.

The presence of interconnect faults during device operation generally leaves incremental net re-route as the only viable recovery alternative. While rip-up and retry based routing for designs routed from scratch has been explored for nearly forty years [9]

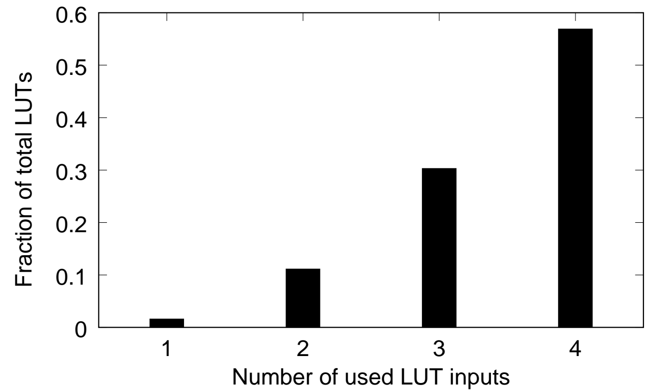

Figure 2: 4-Input Look-Up Table Input Utilization

[16], only recently has significant interest been given to recovering previously-working route configurations. In [12], a re-route approach for FPGAs was described that re-routes nets following logic block movement. This approach did not consider the removal of previously-routed nets that were unaffected by the fault but formed a blockage of a wiring resource required by the faulty net, a potential impediment to successful route completion.

4 Fault Recovery Approaches

Our prototype system has been designed to consider a series of recovery actions following detection of a defect by a fault diagnosis system. In this section recovery approaches are described in the context of FPGA architectural limitations. The recovery system has been automated to invoke the correct individual recovery approach or combination of approaches depending on the location of the fault (logic cluster or global interconnect) and the availability of processing resources capable of participating in the recovery effort (local system or remote site). Stuck-at, open, and short faults are considered as potential device defects. Specific faults can be diagnosed within the logic and interconnect of a local cluster, at the interface between clusters (channel-cluster I/O switches, wires), and in switch matrices and wire segments of routing channels.

A goal of both intra-cluster and interconnect/cluster interface recovery is to use heuristics that have reduced algorithmic complexity in an effort to support implementation locally on a microprocessor of the fault tolerant system. Since these heuristics, targetted to internal cluster faults, are the preferred recovery mechanisms, they are described first.

4.1 Overcoming Intra-Cluster Faults

The first set of recovery approaches addresses wire and transistor failures in cluster input multiplexers, look-up tables, and flip-flops. Recovery techniques for faults in these locations attempt to take advantage of logical redundancy by replacing faulty LUT inputs and BLEs with cluster resources that were unused by original design mapping.

Look-up Table Input Exchange

Single faults in BLE input multiplexers, LUT input wires, and LUT SRAM bits can lead to incorrect results being generated by an individual look-up table. While most FPGA look-up tables support a maximum of four logic inputs, in many cases, following technology mapping, not all four are used. This result is illustrated in Figure 2 by the graph of LUT input usage for the ten benchmarks listed in Table 1. These benchmarks were technology mapped to 4-input look-up tables using FlowMap [8] configured for area minimization. In the figure it is apparent that over 40% of LUTs, on average, contain spare inputs that are unused. As a result, it may be possible to overcome faults in these LUTs by permuting LUT input pin assignments to effectively eliminate damaged resources from the active compute set.

For FPGA devices that contain full input multiplexer connectivity (e.g. Altera Flex10K) the existence of a spare LUT input is sufficient to ensure that the replacement LUT input pin will allow connection to the same *cluster* inputs as the original. In the case of fractional input multiplexer population (e.g. $F_i \neq 1$), LUT input swapping requires checking to determine if the replacement LUT input has the capability to attach to the same cluster input as the original. Given a total of at most $n \leq 3$ remaining functional inputs, a total of 3! = 6 possible LUT input (and hence LUT programming bit configurations) permutations can be considered to maintain LUT functionality while avoiding the fault. As an example of an input multiplexer configuration that supports fault tolerance, consider BLE 1 and associated input signals in Figure 3. In this figure the cluster input signals that drive the input multiplexer for a BLE input can be identified by the squares at the intersection of the cluster inputs and BLE inputs. In the case of a BLE input failure for BLE 1, the remaining three inputs can be configured to cover any permutation of cluster inputs that drive the LUT. Since the number of cluster input to BLE input permutations is small, a quick enumeration of possible covering patterns can be performed.

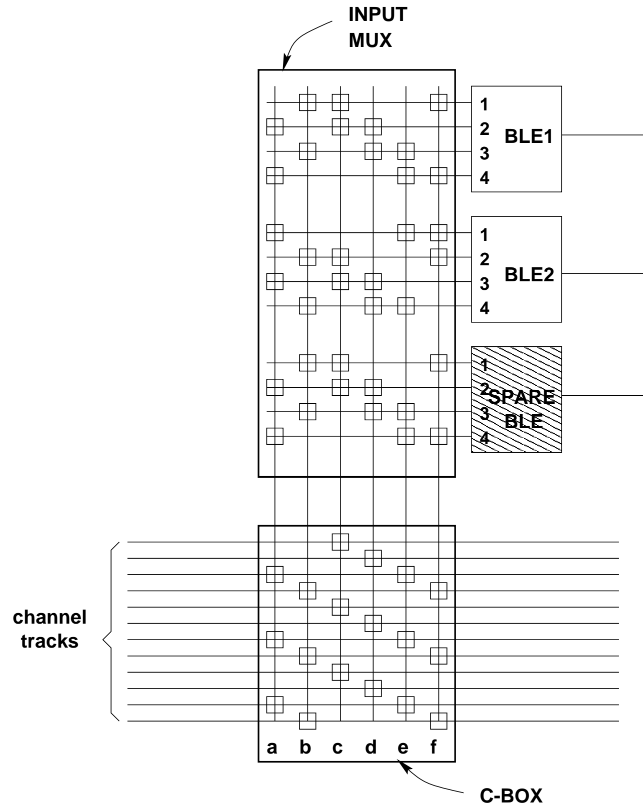

Figure 3: Fault Tolerant Cluster Input Pattern

Basic Logic Element Exchange

While effective for some LUT failures, LUT input swapping only provides a limited fix for many logic cluster faults. In cases of LUT output failure, BLE flip-flop failure, or full usage of LUT inputs, logic cluster functionality can be preserved only by swapping an entire BLE with an unused one exhibiting similar intra-cluster connectivity. As with LUT input swapping, the feasibility of BLE swapping is dependent on the connectivity between cluster inputs and BLE inputs. In general, two BLEs may be swapped if the inputs of the faulty BLE and the spare BLE connect to the same set of cluster input signals. While this clearly is the case for fully-connected input multiplexers, this criterion requires attention to cluster-to-BLE connection patterns in the case of partially-populated input multiplexers. An example configuration which allows BLE swapping is shown in Figure 3. By examining the figure it can be seen that BLE 1 has the same LUT input connection pattern as the spare and therefore could be swapped in as a replacement. The inputs of BLE 2 must be examined more closely, however, to make this determination. In fact, the functionality of BLE 2 can be reconstructed in the spare BLE if LUT inputs are permuted (e.g. BLE 2/input 1 replaced by spare BLE/input 4, BLE 2/input 2 replaced by spare BLE/input 1, etc.) and LUT programming is modified to take the permutation into account. If it is not known a priori if a spare BLE can cover a faulty one, a full enumeration of cluster input to BLE input coverage can be performed prior to swapping. It is assumed here that the spare BLEs are fault-free as determined by the previous run of the fault detection system.

In many cases it may not be feasible or desirable to save a spare BLE during design mapping. For a given number of cluster BLEs, N, and associated cluster inputs, I, it is usually possible to achieve full BLE utilization through cluster packing [6]. In Section 5, we consider the incremental cost of adding a spare BLE to a cluster and leaving this resource unused during initial mapping as a way of facilitating BLE swapping.

Cluster Input/Output Exchange

While LUT input and BLE swapping can be used to overcome individual faults inside clusters, these procedures are ineffective for cluster input wire, cluster output wire, or cluster output multiplexer failures. Using techniques similar to those described for LUT input and BLE exchange, it is possible to use spare cluster inputs or outputs as replacements for faulty wire resources. In general, the exchange must be made so that channel track to cluster input/output connectivity is preserved, thus eliminating the need for incremental net re-route. As an example, in Figure 3, input b could be exchanged for input f since they both connect to the same channel tracks.

4.2 Incremental Routing Approaches

In the event of interconnect segment failure or if logic cluster exchange approaches are ineffective, incremental routing techniques are needed to re-connect nets affected by faults. To maximize net

R: Design nets to be re-routed.
MaxIter: Maximum re-route iterations.
Iter: Current iteration number.

Remove faulty nodes from routing graph
Initialize H_n values to those from initial route.
Add nets affected by faults to **R**.
While nets in R > 0 **and Iter** $<$ **MaxIter**
  Order nets by bounding box size.
  For each unrouted net
    Maze-route net using node values H_n, P_n.
    Update P_n values.
  Endfor
  Update H_n values
  Remove nets with no overused nodes from **R**.
  Add nets with congested nodes to **R**.
EndWhile

Figure 4: Incremental Routing Algorithm

routability, we have based our router on the PathFinder [10] negotiated congestion algorithm, a widely-used maze-routing algorithm. PathFinder is a multi-iteration maze router that re-routes each net in sequence for each iteration. The routing search for each net evaluates a series of routing nodes (cluster pins and wire segments) each of which has been assigned a node cost value, c_n, based on the following equation [7] [10]:

$$c_n = (1 + H_n) * (1 + P_n) \quad (1)$$

where P_n is the *present* cost of the node, based on the number of nets currently assigned to the node, and H_n is a *history* cost value indicating that a node, while perhaps uncongested presently, was overused during one or more previous iterations. By performing multiple iterations with a non-decreasing history value, shortest-path net routes can be guided away from congested device areas to areas with available routing resources.

While several incremental re-route approaches have been developed to recover from interconnect faults [12] [20], none consider the routing cost parameters of the *initial* route in making re-route decisions. Our router extends the original PathFinder approach by using history values from the initial route to guide incremental re-route. Additionally, nets unaffected by operational faults may be ripped up to remove blockages that may inhibit routing for fault-affected nets.

The outer-loop of the routing algorithm used to re-route nets over multiple iterations is shown in Figure 4. Unlike traditional PathFinder formulations, in our formulation only a subset of nets are re-routed in each iteration. Following each iteration, nets associated with overused nodes are designated for rip-up, history values are updated, and ripped-up nets are re-routed in the following iteration. We have found a maximum iteration count of about 30 to be appropriate in determining success or failure of re-routing. To achieve accelerated routing speed, an A* search parameter was added to the PathFinder cost function as in [23] [24] to promote depth-first search behavior without loss of routing quality. The route sequence for sources and sinks of individual nets, indicated as **Maze-route** in Figure 4, is described in detail in [24].

Design	Source	# BLEs	# Clusters		
			N=1	N=4	N=8
beast10k	GEN	9800	9800	2456	1227
bubble sort	RAW	12293	12293	3074	1537
clma	MCNC	8383	8383	2121	1056
elliptic	MCNC	3604	3604	903	453
ex1010	MCNC	4598	4598	1191	595
frisc	MCNC	3556	3556	892	448
pdc	MCNC	4575	4575	1194	593
s38417	MCNC	6406	6406	1604	803
s38584.1	MCNC	6447	6447	1612	806
spla	MCNC	3690	3690	953	476

Table 1: Benchmark Statistics

5 Results

To judge the performance of our fault recovery system, ten benchmark circuits, listed in Table 1, were used. These benchmarks are from the MCNC suite [26], the RAW benchmark suite [4], and a benchmark circuit generator [18]. All experiments were run on a 366MHz Celeron-based PC with 256MB of memory.

To determine the relative area taken up by inter and intra-cluster transistors, designs were packed, placed, and routed by VPACK and VPR [7] using the minimum number of logic clusters and routing tracks needed to successfully implement the design. Device channel routing segments consisted of long-lines, hex-lines, and single-length lines in the same proportional distribution as found in the Xilinx Virtex [2] architecture. An accurate measurement of cluster and interconnect transistors was determined using FPGA area measurement approaches detailed in [6] by using the **trans_count** tool developed at the University of Toronto. The results in Figure 5 indicate an increasing fraction of device area taken up by intra-cluster transistors as cluster sizes increase. This finding motivates our development of intra-cluster recovery techniques.

In general, incremental re-route could easily overcome single segment faults, even using FPGAs that contained the minimum number of tracks per channel needed to successfully route the circuit. In Figure 6, the importance of using history values during incremental re-route is shown for designs containing multiple design faults and input multiplexer flexibility of $F_i = 1$. The curves represent an average percentage of failures across 50 trials for each benchmark at specified fault counts. In many cases, the non-history version would fail to complete successfully for even a small numbers of faults.

Figure 7 shows the average time needed to re-route a circuit given a specific number of interconnect faults. The nearly horizontal line corresponds to the amount needed to re-route the device from scratch following a fault. In cases where incremental re-route with history failed, the incremental re-route time was added to the from-scratch route time in determining the average. For greater than 475 faults this leads to higher average route times than if only from-scratch re-route is performed. Not only is the finding important for fault tolerance but it also is directly applicable to the support of dynamic reconfiguration of circuits. Following placement modification, the incremental re-route approach can be applied even if the number of nets affected is in the hundreds.

As a final step, all incremental recovery approaches were combined into an automated CAD system. The following multi-step

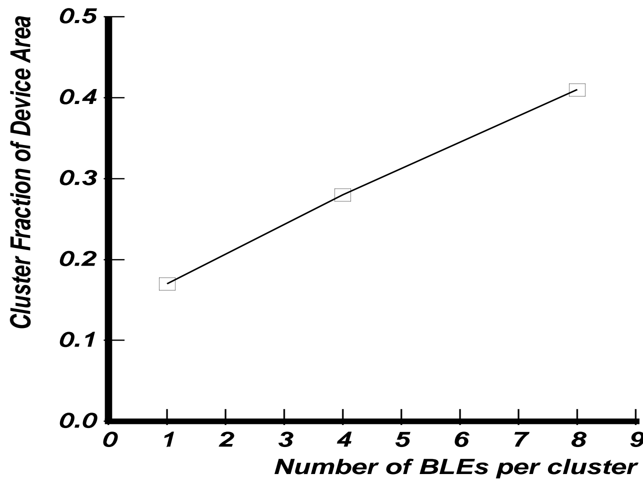

Figure 5: Fraction of Transistors in FPGA Clusters

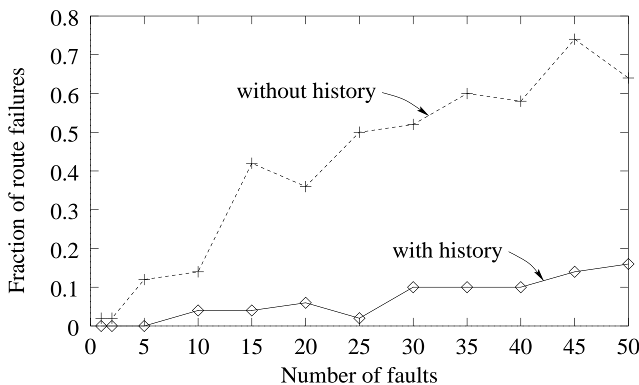

Figure 6: The Effect of Initial-route History on Re-route

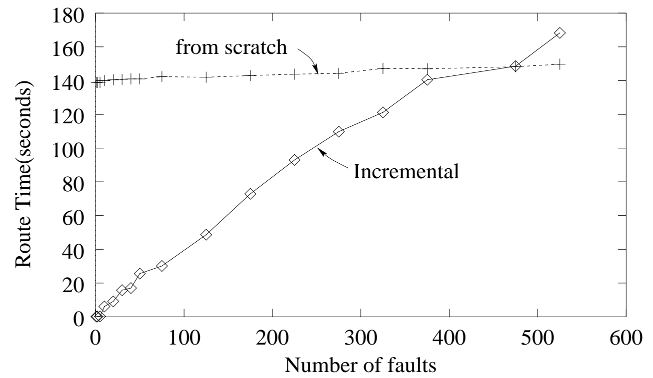

Figure 7: Average Re-route Time Comparison

Action	No Spare BLE	One Spare BLE
	% success	% success
No action	36.4	34.6
LUT input swap	9.5	11.2
BLE swap	0.1	22.4
Cluster I/O swap	3.7	7.0
Incremental re-route	48.1	24.4
Re-route from scratch	2.6	0.3

Table 2: Fault Recovery Effectiveness

procedure was applied 10,000 times to a randomized selection of the placed-and-routed benchmarks that had been previously mapped to FPGAs. Target FPGAs for the initial and incremental mappings contained the minimum number of tracks per channel necessary to complete routing successfully.

Recovery steps were as follows:

1. Each transistor and wire in the target device is assigned a specific number for a total of M potential single-fault points.

2. A random number number generator selects one of the M locations as faulty.

3. The fault recovery system determines if the fault has affected the mapped circuit. If the fault does not affect circuit functionality, no further action is taken.

4. Depending on fault location, the appropriate recovery action is determined. For logic cluster faults, a progression of action is performed if early actions fail (e.g. LUT swap, followed by BLE swap, followed by cluster input swap, etc.)

Experiments showed that it was possible to recover from almost all 10,000 cases of randomized single-fault insertion without rerouting from scratch. These 10,000 cases did not consider faults that affect wires corresponding to global signals such as power and clock nets. Table 2 shows the percentage of time each recovery approach was used successfully to overcome the single fault. Targetted devices contained 4 BLEs per cluster ($N = 4$) and were set to an F_i value of 1. For results shown in the second column of the table (i.e. for "No spare BLE"), it was found that design clusters were generally fully filled by initial packing so that BLE exchange was largely ineffective. Even without this recovery action, incremental re-route was needed only about 50% of the time, limiting the need for the fault tolerant system to access remote processing resources. The numbers improve significantly if *reserved* cluster resources in the form of two cluster inputs (in addition to the 10 original) and a spare (fifth) BLE are added to each cluster. These resources remain unused during initial design mapping and can be deployed during fault recovery to take the place of faulty resources. The most dramatic benefit of this addition can be seen in the *incremental re-route* and *BLE swap* rows of the third column. Since the spare BLE can be used to overcome logic cluster faults, the number of cases in which incremental re-route is needed drops nearly in half. On average, the cost of adding the spare resources to the clusters increased overall device area by about 20% (5223 transistors per cluster vs. 6251 transistors per cluster) for $N = 4$ and by about 8% (12297 transistors per cluster vs. 13248 transistors per cluster) for $N = 8$. These results indicate that the overhead is reduced as the cluster size is increased.

It was mentioned in Section 2.3 that not all FPGA devices (e.g. Virtex, XC5200) contain fully-connected input multiplexers. For these devices, each BLE input is driven by a fraction of cluster inputs (defined in Section 2.3 to be F_i). Often the assignment of cluster inputs to BLE inputs is randomized to increase the number of possible routing choices, thus enhancing routability. However, as shown in Figure 3, for specific, periodic input switch patterns,

LUT input, BLE, and cluster I/O swapping can be used to guarantee successful exchange in the presence of single faults. To determine the cost of using switches assembled in a swappable versus randomized pattern, a number of experiments on the benchmark circuits were performed. After performing placement and routing for designs mapped to devices with $N = 4$ BLEs per cluster, $I = 10$ inputs per cluster, and $F_i = 0.5$ it was determined a 5-10% area penalty due to increased required track count exists for the patterned versus the non-patterned switch cases.

6 Future Work

Several fault recovery issues remain for future investigation. New algorithms that can quickly combine cluster input, LUT input, and basic logic element exchange are needed to more thoroughly explore the cluster re-implementation. Additionally, for practical applications, the incremental router based on PathFinder must be enhanced for critical-path delay improvement to limit the effects of post-recovery performance degradation.

The work outlined in this paper, in conjunction with new fault-detection algorithms for cluster-based architectures described in [14], form the basis of an automated fault diagnosis and recovery system for commercial FPGAs that is currently under development at the University of Massachusetts. Our end goal is to be able to detect and recover from faults in Xilinx Virtex devices automatically using local intra-cluster resource swapping, remote web-based processing for incremental routing, and configuration bitstream generation using Xilinx JBits. This complete system is currently under development.

References

[1] *Flex10K Data Sheet*. Altera Corporation, 1998.

[2] *Virtex Data Sheet*. Xilinx Corporation, 1998.

[3] *Apex Data Sheet*. Altera Corporation, 1999.

[4] J. Babb, M. Frank, V. Lee, E. Waingold, and R. Barua. The RAW Benchmark Suite: Computation Structures for General Purpose Computing. In *Proceedings, IEEE Workshop on FPGA-based Custom Computing Machines*, Napa, Ca, Apr. 1997.

[5] K. Bernhardt. Advanced Technologies for a Command and Data Handling Subsystem in a "Better, Faster, Cheaper" Environment. In *14th Digital Avionics Systems Conference*, Cambridge, Ma, Nov. 1995.

[6] V. Betz and J. Rose. Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size. In *Proceedings, IEEE Custom Integrated Circuits Conference*, pages 551–554, 1997.

[7] V. Betz and J. Rose. VPR: A New Packing, Placement, and Routing Tool for FPGA Research. In *Proceedings, Field Programmable Logic, Seventh International Workshop*, Oxford, UK, Sept. 1997.

[8] J. Cong and Y. Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Design. *IEEE Transactions on Computer-Aided Design*, pages 1–12, Jan. 1994.

[9] W. Dees and R. Smith. Performance of Interconnection Rip-up and Reroute Strategies. In *Proceedings, ACM/IEEE 18st Design Automation Conference*, 1981.

[10] C. Ebeling, L. McMurchie, S. Hauck, and S. Burns. Placement and Routing Tools for the Tryptych FPGA. *IEEE Transactions on VLSI Systems*, pages 473–482, Dec. 1995.

[11] J. M. Emmert and D. Bhatia. Reconfiguring FPGA Mapped Designs with Applications to Fault Tolerance and Reconfigurable Computing. In *Field Programmable Logic Workshop (FPL'97)*, Oxford, England, Sept. 1997.

[12] J. M. Emmert and D. Bhatia. Incremental Routing in FPGAs. In *International ASIC Conference (ASIC'98)*, 1998.

[13] F. Hanchek and S. Dutt. Methodologoies for Tolerating Cell and Interconnect Faults in FPGAs. *IEEE Transactions on Computers*, 47(1):15–32, Jan. 1998.

[14] I. G. Harris and R. Tessier. Testing Approaches for Cluster-based FPGAs. In *submitted to 37th Design Automation Conference*, June 2000.

[15] S. Hauck. Data Security for Web-based CAD. In *Proceedings, ACM/IEEE 35th Design Automation Conference*, 1998.

[16] D. Hill. A CAD System for the Design of Field Programmable Gate Arrays. In *in 28th Design Automation Conference*, June 1991.

[17] N. Howard, A. Tyrrell, and N. Allinson. The Yield Enhancement of Field-Programmable Gate Arrays. *IEEE Transactions on VLSI Systems*, pages 115–123, Mar. 1994.

[18] M. Hutton, J. Rose, and D. Corneil. Generation of Synthetic Sequential Benchmark Circuits. In *International Symposium on Field Programmable Gate Arrays*, Monterey, Ca., Feb. 1997.

[19] R. Katz, K. LaBel, J. J. Wang, B. Cronquist, R. Koga, S. Penzin, and G. Swift. Radiation Effects on Current Field Programmable Technologies. *IEEE Transactions on Nuclear Science*, 44(6):1945–1956, Dec. 1997.

[20] A. Mathur and C. L. Liu. Timing-Driven Placement Reconfiguration for Fault Tolerance and Yield Enhancement in FPGAs. In *Proceedings Ed&TC96*, 1996.

[21] N. Shnidman, W. H. Mangione-Smith, and M. Potkonjak. On-Line Fault Detection for Bus-Based Field Programmable Gate Arrays. *IEEE Transactions on Very Large Scale Integration*, 6(4):656–665, Dec. 1998.

[22] D. P. Siewiorek and R. Swarz. *Reliable Computer Systems: Design and Evaluation*. Digital Press, Burlington, Ma, 1992.

[23] J. Swartz, V. Betz, and J. Rose. A Fast Routability-Driven Router for FPGAs. In *6th International Workshop on Field-Programmable Gate Arrays*, Monterey, Ca, Feb. 1998.

[24] R. Tessier. Negotiated A* Routing for FPGAs. In *Proceedings: Fifth Canadian Workshop on Field-Programmable Devices*, Montreal, Quebec, June 1998.

[25] S. Webber and J. Beirne. The Stratus Architecture. In *Proceedings: 21st International Symposium on Fault-Tolerant Computing*, 1991.

[26] S. Yang. Logic Synthesis and Optimization Benchmarks. *Microelectronics Centre of North Carolina Tech. Report*, 1991.