DEFECT-TOLERANT FPGA SWITCH BLOCK AND CONNECTION BLOCK WITH FINE-GRAIN REDUNDANCY FOR YIELD ENHANCEMENT

Anthony J. Yu *Guy G. F. Lemieux*

Department of Electrical and Computer Engineering
University of British Columbia, Vancouver, BC, Canada
email: { anthonyy | lemieux } @ ece.ubc.ca

ABSTRACT

Future process nodes have such small feature sizes that there will be an increase in the number of manufacturing defects per die. For large FPGAs, it will be critical to tolerate multiple defects [6]. We propose a number of changes to the detailed routing architecture of island-style FPGAs to tolerate multiple random, distributed interconnect defects *without re-routing* and with *minimal impact* on signal timing. Our scheme is a *user option* prebuilt into an architecture, requiring +11% area for additional multiplexers. Unused (spare) wiring tracks are also needed, bringing total overhead to 24% to tolerate stuck-at or open faults, or 34% to include bridging. User circuits that do not fully stress the routing network already have these tracks freely available. The delay penalty is programmable: 5–10% if defect rates are expected to be sufficiently low, but can be as high as 25% if defect rates are high. Our schemes can tolerate more than 10 interconnect defects for large array sizes of 128 × 128. Unlike row/column redundancy schemes, our schemes are scalable: they naturally tolerate more defects as the FPGA array size increases. This work is the first *detailed* analysis of fine-grained defect-tolerant schemes in FPGAs.

1. INTRODUCTION

Field programmable gate arrays (FPGAs) are large integrated circuits comprised of programmable logic blocks and programmable routing. Their size, density and regular layout makes them attractive for aggressive tuning in the latest technology processes. As such, they are also prone to manufacturing defects [6].

Since FPGAs are routing dominated, defects are more likely found in the interconnect than in the logic blocks. This makes the ability to tolerate defects in the interconnection network extremely important. In this paper, the interconnection network encompasses the physical wiring, the switch elements, and the configuration bits found in both the switch block and the connection block.

Traditional defect-tolerant schemes involve the use of entire spare rows and/or columns in the array of logic blocks. This method is capable of tolerating clusters of defects, however the consolidation of spare resources severely restricts its ability to tolerate multiple distributed defects. Our approach addresses this problem by embodying a fine-grain approach to redundancy: spare resources are distributed in every routing channel.

This paper presents a new switch and connection block architecture that is capable of tolerating multiple distributed defects within the interconnection network. Our approach is based on defect avoidance by shifting. As shown in Figure 1, shifting allows signals to route around a defect. We show that this can be done in a controlled, localized fashion without re-routing.

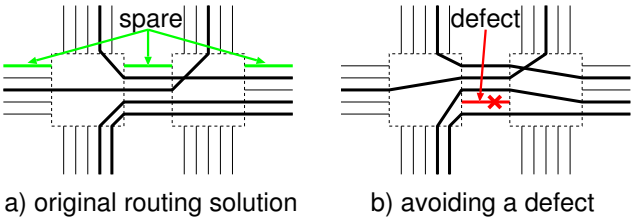

Fig. 1. Shifting signals to avoid defects.

To successfully correct a defect, we need to know where the defects are. One such way to provide this information is through the use of a list of defective resources, called a *defect map*. This map can be stored on-chip in non-volatile storage, or in an off-chip database indexed using a unique on-chip ID. When an FPGA is being programmed, the defect map specific to that FPGA is called up and the correction is applied. Our architecture is designed so that correction can be applied through bitstream manipulation alone. The correction can be applied during programming or bitstream generation. It can even be applied by means of an embedded processor within the FPGA. With this latter method, defect correction can be completely hidden from the user.

Our architecture has some disadvantages. First, like any spare-based redundancy scheme, it requires some amount of overhead which cannot be avoided. We evaluate a number of detailed variations to our scheme and precisely quantify the impact on area, delay and yield. Next, our architecture does not tolerate transient faults (single event upsets), clusters of defects, or defects in the logic blocks. To address these latter two issues, our architecture can be complemented with spare row/column techniques, although there are other possibilities as well.

Despite these disadvantages, there are also notable advantages:

- Tolerates multiple distributed random defects,
- Scalable: more defects are tolerated in larger FPGA arrays,
- Defect corrections do not change signal timing,
- Defect corrections can be applied very quickly, and
- Crosstalk defects [19] can be repaired by separating nets.

2. PREVIOUS WORK

Fault redundancy can be loosely classified into three groups: software redundancy, hardware redundancy and run-time redundancy.

Each of these approaches have their advantages, and typically trade off between time (critical path delay and processing/application time) and resources (silicon area, external storage, etc). Our approach is a combination of software and hardware redundancy.

2.1. Software-based Redundancy

In a software redundancy approach, CAD tools are used to map around faulty resources. This method typically has no hardware overhead. The effectiveness and efficiency of correction is dependent on the abilities of the CAD tools. Furthermore, this method is impractical in a production environment because: 1) generating a unique placement and routing solution for each FPGA is time-consuming, and 2) verifying timing of each solution is impossible.

Xilinx solves these problems with their EasyPath [20] technology. Rather than forcing the configuration bitstream to avoid the defects, Xilinx forces the defects to avoid the bitstream. They do this by obtaining the customer's final bitstream and selecting chips which contain defects only in the unused portions of the chip.

Two other approaches have been proposed to solve these problems. The first method is to precompute a number of placement and routing solutions for a particular design. Each precomputed solution differs by its resource usage. When programming a defective chip, defect correction simply involves selecting the appropriate solution (one that does not use the defective resource(s)) [12, 13].

The second method requires the reservation of spare resources. By carefully avoiding the use of certain resources, it is possible to avoid defects by "shifting" the *entire* design [9] by one row or column in the array. Design shifting can be applied in a relatively short amount of time. Without special hardware support, however, shifting results in a slight variance in IO timing. It can also be complicated by heterogeneous (memory or DSP) blocks in the array. Furthermore, to support multiple defects, they must be perfectly aligned to the spare locations.

2.2. Hardware-based Redundancy

Hardware redundancy involves the addition of extra or spare resources. The spare resources allow defective parts to be swapped with empty spare ones. This exchange reduces correction time since the time required to swap is typically less than the time needed to generate a new placement and routing solution.

The spare row and column technique is one of the first hardware redundancy approaches [11] and has been successfully applied in industry [3]. This method adds one spare row and one spare column to the layout. It also requires the routing network to be modified. In the event of a defect, the row or column containing the defect is bypassed, and the spare row or column is utilized. The ability to bypass entire rows and columns gives this approach the ability to tolerate defect clusters. Unfortunately, published research does not present the delicate circuit details needed to perform the bypass. Altera patents provide some insight [4] and indicate that additional circuitry is required for bypassing.

Redundancy can be implemented at a finer level. For example, additional connections can be added inside the switch block to tolerate one transistor defect per switch block [8]. Unfortunately, this approach is impractical because it significantly alters delay.

2.3. Run-time Fault Tolerance

Fault tolerance can also be addressed during run-time. As transistor sizes shrink, FPGAs become susceptible to transient faults such as single event upsets [5, 10]. To alleviate this problem, techniques have been developed to detect and correct transient errors through reprogramming or bit scrubbing [2, 7]. However, it is not clear whether these techniques can be extended to correct permanent manufacturing defects; simply reprogramming is insufficient.

2.4. New Approach

The proposed approach is a combination of software and hardware redundancy. Additional routing resources are added to facilitate and simplify defect correction. A new switch block design allows defects to be bypassed by computing a new configuration for a small, localized part of the FPGA. This ensures that areas outside of the neighbourhood of the first defect can still tolerate other defects. The affected neighbourhood is so small that defect correction can be achieved by modifying the configuration bitstream alone. The defect correction also introduces minimal timing disturbances.

3. ARCHITECTURE AND IMPLEMENTATION DETAILS

This section describes the architecture and implementation details of our approach, the type of defects we consider, how defect correction is applied, and the limitations of our design.

3.1. Switch Block Changes for Defect Tolerance

The proposed defect redundant switch block builds upon the directional switch block described in [16], which is not defect tolerant. Figure 2 shows both a detailed and high-level representation of the directional switch for length 1 wires. In the high-level representation, individual wires and buffers are replaced by arrows.

To make this switch block defect tolerant, we wrapped two layers of multiplexers around the directional switch. This is shown as the two outer layers in Figure 3. The outer-most layer represents the shift-avoid layer of multiplexers (*omux*), and the middle layer represents the shift-restore layer of multiplexers (*imux*). Clearly, this extra multiplexing costs in area (11%+) and delay (15%+).

The *omux* allows signals to "steer" away from a downstream defect. By means of these multiplexers, signals routed on track t can be shifted up to tracks t+1 or t+2. When there is a defect on track t, the defect is avoided by shifting up all signals routed on tracks $\geq$ t. Signals on tracks $<$ t remain in place. Clearly, the shifting requires that there be spare routing tracks. These spares incur about 10% area overhead for each spare set, but this amount diminishes as device channel widths increase.

The *imux* is used to reverse or restore the shift-avoid action in an upstream switch block. These multiplexers allow a signal on track t+1 or t+2 to shift down to track t, thereby nullifying any upstream shifting action. To keep the effects of track shifting localized, we have designed the switch block such that any signal leaving a perturbed neighbourhood can be restored to the original track number. This localization allows our architecture to tolerate multiple distributed defects.

To reduce the delay of long nets, a bypass path similar to [17] is introduced into the switch block. This bypass path connects a straight-through wire endpoint directly with the corresponding *omux* on the opposite side of the switch block. Bypassing the *imux* and the directional switch creates an alternate, reduced delay path for signals travelling across a channel.

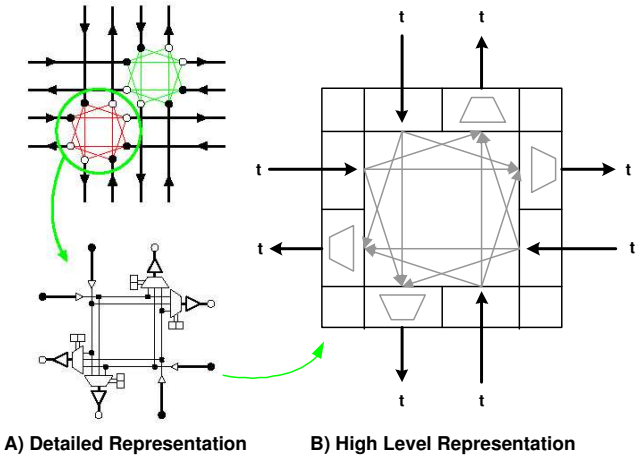

Fig. 2. Directional switch block.

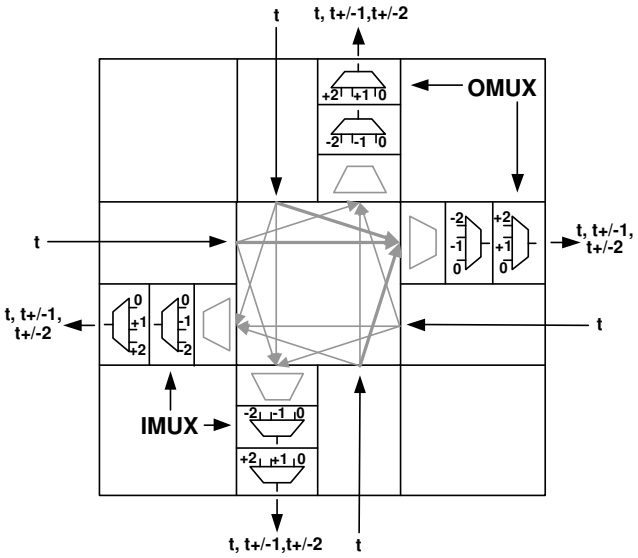

Fig. 3. High-level view of defect-tolerant switch block.

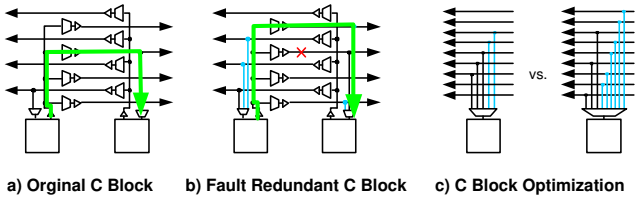

Fig. 4. Connection block modifications.

3.2. Switch Block Changes for Area Reduction

In an attempt to reduce area and delay overhead, we considered reducing the flexibility of the directional switch, F_s. F_s is the number of other wires connected to a given switch block wire [18]. By decreasing F_s, we can shrink the size of the multiplexers in the switch, hence improving both area and delay. With long wires, the flexibility at the end switch blocks or *endpoints* is different than at the middle switch blocks or *midpoints*. We considered the following switch flexibilities:

1. The *E3M2* switch is the directional switch described in [16]. It has $F_s = 3$ for endpoints and $F_s = 2$ for midpoints.
2. The *E3M1* switch also uses $F_s = 3$ at endpoints. However, midpoints are reduced to $F_s = 1$, meaning they can only turn either left or right (not both). The turn direction alternates along the length of a wire.
3. The *E2M1* switch has $F_s = 2$ for endpoints and $F_s = 1$ for midpoints. Endpoints include only straight-through connections. Turns are handled in the same manner as *E3M1*.

3.3. Connection Block Changes

As a consequence of track shifting, signals that were once routed on track t can now reside on tracks t+1 or t+2. To accommodate for this variability, the connection block must also be modified. In our architecture, the CLB outputs do not need to be modified because they are already fully connected to all of the tracks. However, the CLB input connectivity is increased by adding the additional required connections to the shifted tracks. This modification is shown in Figure 4ab.

Initially, the CLB inputs are connected to half of the routing tracks. To adjust for the track shifting, for every track t that is connected to a CLB input, we ensure that tracks t+1 and t+2 are also connected to the input. Thus, if a signal gets shifted up by 1 or 2, the CLB can still extract the correct signal. Clearly, we can reduce this overhead by maximizing the number of consecutive tracks that are connected to a particular CLB input, as shown in Figure 4c. However, this optimization is left for future work.

3.4. Supported Defects

With our schemes, interconnect defects can be categorized into three disjoint classes: single-length, double-length and intolerable defects. Examples of the first two classes are shown in Figure 5.

For example, if an open or stuck-at fault occurs on the wire, or there is a stuck-at fault in the wire driver or the output of the *omux*, the defect is a *single-length defect*. In this case, one switch block avoids the defect and all adjacent "downstream" switch blocks do the restore. This kind of defect is isolated to one wire length. Figure 5a demonstrates how a single-length defect is corrected using two parallel straight-through signals. With single-length defects, the change is purely localized in the channel to a group of wires with common start and ending points in the array. Such a group of wires is called a ***trackgroup***.

If a defect is found in any of the multiplexers (aside from the output of the *omux*), the defect is categorized as a *double-length defect*. Due to their location, these defects actually impair the defect-correcting ability of the current switch block. To fix this, the switch block of the adjacent "upstream" trackgroup is used to avoid the defect, and the downstream switch blocks do the restore. Hence, this kind of defect requires two wire lengths to correct.

Figure 5b indicates how a double-length defect spans two adjacent trackgroups: the upstream trackgroup on the left, and the defective one on the right. In fact, for this example there are addi-

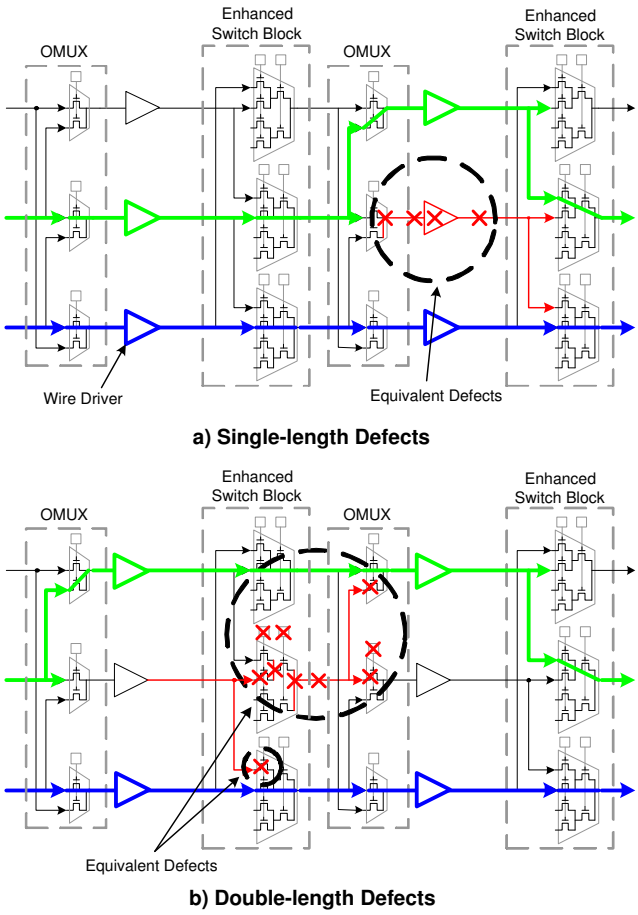

Fig. 5. Two classes of tolerable defects.

tional upstream switch blocks (above and below) that reside in the vertical channel. As shown in Figure 6a, contention arises when a straight-through signal is shifted up onto a track that is expected to be available for turning signals. To avoid contention, signals on tracks $\geq$t in the vertical channel must be shifted before arriving.

All of the switch blocks in the vertical channel within a distance of one wire length must participate in correcting a double-length defect. Hence, they must all be defect-free (or contain a defect in exactly the same track). This results in a fairly large (but still localized) neighbourhood which must be defect-free.

The upstream pre-shifting just described is one way to solve the conflict problem with double-length defects. A more robust solution is shown in Figure 6b. Here, the *imux* is *embedded* within the switch block and the internal switch block multiplexers are duplicated. This shrinks the requisite defect-free area to just the two adjacent trackgroups. We find this increases total area by about 4%, but due to mux sizes this also improves delay by a similar amount. Multiplexer embedding also produces better yield results.

Wire bridges and certain source-drain shorts in the multiplexer are usually categorized as double-length defects. These defects have the potential to render two adjacent tracks as unusable. To avoid such a defect, the upstream switch block(s) must shift tracks up by 2 and the downstream switch blocks must shift down by 2.

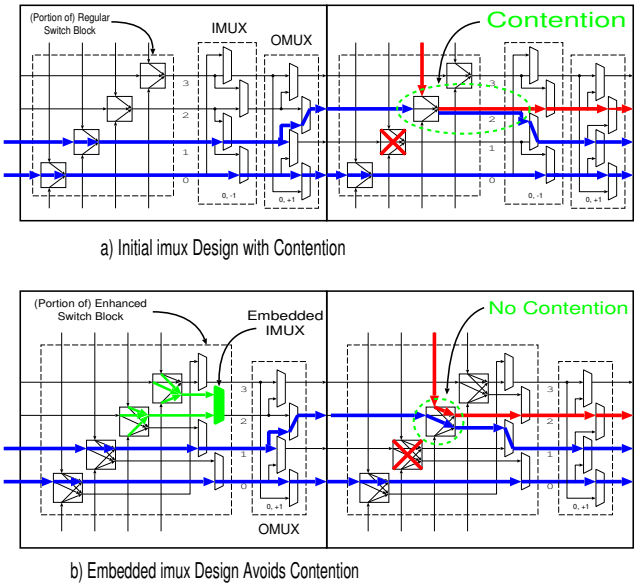

Fig. 6. Embedding *imux* to avoid contention.

If this must be implemented by two +/-1 shifts, the surrounding neighbourhood that must be defect-free is considerably larger than the double-length defect situation and this also harms yield.

There exists a class of intolerable defects that is not shown in Figure 5. This includes power/ground shorts and clusters of defects. The first type of defect cannot be tolerated. However, it is possible to tolerate the latter by complementing our architecture with a spare row/column technique, e.g. [11].

3.5. Modes of Operation and True Overhead

Our architecture makes defect redundancy an option to the user. This means we expect FPGA devices to operate in two modes: *normal defect-tolerant mode* and *recovery mode*.

The *normal* mode assumes the customer will buy imperfect, low-cost devices by applying defect correction. In this mode, the routing software reserves a spare routing track in each trackgroup.[1] This reduces the number of routing tracks available to the application, but the spares are needed for defect correction. For many applications which do not stress the routing network, this is an easy way to lower device cost.

The *recovery* mode assumes the customer will buy perfect devices at a price premium. In this mode, the routing software uses the additional *imux* and *omux* routing multiplexers to increase the flexibility of the interconnect. In essence, the router is using the redundant resources to recover some area/delay efficiency that was sacrificed when they were added. This mode is used for those few applications that have high interconnect demands where the resulting increase in interconnect flexibility is even more helpful. However, in this mode, there is no natural ability to tolerate defects.

When recovery mode is used, our results show an increase of 11% in area and 6% in delay. **Recovery mode is the true over-**

[1]Two spares are needed for devices with bridging defects, which may be sold at even lower cost.

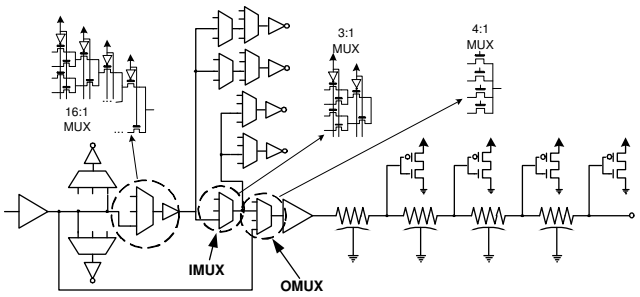

Fig. 7. HSPICE model for delay characterization.

head cost of our redundancy scheme. It is the additional cost imposed on aggressive applications with high interconnect demands.

When normal mode is used, the overhead appears to be higher because spare tracks are also counted as overhead. This is misleading! Applications with low interconnect demands already have an abundance of unused routing tracks, so this extra capacity is already built-in. The customer has already paid for these unused routing tracks, so there is no real end-user cost to supplying them. Our redundancy scheme merely finds a use for these free tracks by calling them spare tracks.

3.6. Detailed Transistor-level Design

The transistor circuit model used for HSPICE simulations is shown in Figure 7. The components in the circuit (from left-to-right) are: input buffer, directional multiplexer, strengthening buffer, shift-restore multiplexer (*imux*), shift-avoid multiplexer (*omux*), tapered driver and the wire model with loads.

For area considerations, the directional multiplexer is implemented using a tree of minimum-sized transistors. This allows us to use encoded control lines and reduce SRAM usage. We also assumed both true and complemented outputs are available from a 6-transistor SRAM cell.

The *omux* is implemented using a decoded one-level multiplexer of minimum-width pass transistors. Each pass transistor is controlled by an independent SRAM cell. The motivation for this is to reduce delay.

We explored 3 different implementations of the *imux*: decoded, encoded and embedded. The decoded multiplexer is identical in implementation as the *omux*. This kind of multiplexer trades area for delay. The encoded *imux* is built like the directional multiplexer. It trades delay for area.

As mentioned earlier, it is also possible to embed the *imux* into the directional multiplexer. This enhanced multiplexer is built by duplicating the inputs of the directional multiplexer for track t+1 and t+2, and connecting them to the directional multiplexer for track t. An embedded *imux* allows signals to turn and shift at the same time. This improves yield with double-length defects at the expense of some area.

By varying the implementation of the *imux* and shifting ability of the multiplexers, we obtained 7 different defect-tolerant implementations. The +/-2 shifts use additional area to improve yield of bridging defects. The attributes and differences between the switch implementations are summarized in Table 1.

The area and delay performance of the implementations are also sensitive to the precise transistor-level circuit design of the multiplexers and buffers. We used the procedures described in [15, 16] to determine the best transistor sizes for lowest area-delay product. Delay results are computed from HSPICE simulations of TSMC's 180nm technology.

Arch.	imux implementation	imux "-2" shift	omux "+2" shift
EM22	embedded	Y	Y
EM12	embedded	N	Y
EM11	embedded	N	N
FL22	flat	Y	Y
EN22	encoded	Y	Y
EN12	encoded	N	Y
EN11	encoded	N	N

Table 1. Defect-tolerant switch implementations.

3.7. Limitations

In order to implement our redundancy scheme, we assume that FPGA and VLSI testing strategies can identify the defect locations to produce a defect map. This map may be provided by the vendor or even generated by the user. Furthermore, the defect map does not need to be overly detailed. For each defect, it must identify the wire segment location in the array (x, y and track numbers) and type (single- or double-length). Bridges are adjacent defect pairs.

Our method of dealing with bridging defects assumes that routing tracks within the same channel are laid out beside one another. This may not be a realistic assumption since there are many factors that influence the layout of an FPGA. Our solution should only be viewed as a general approach. To fully protect an FPGA from bridging defects, the final FPGA layout must be considered.

As described earlier, defects must be surrounded by some defect-free resources for successful repair. As a result, our approach cannot tolerate clusters or closely-spaced defects. To reconcile this shortcoming, it is possible to complement our architecture with a spare row/column technique [11].

Finally, defects in the logic block have been ignored in this paper. This issue has been addressed in the past [12, 13, 14]. We feel these techniques, or a spare row/column technique, can be used to complement our interconnect-based schemes.

3.8. Trade-offs

In our architecture, area and delay trade-offs can be made in two places. First, the implementation of multiplexers can be varied: reducing multiplexer levels by flattening increases area but improves delay. Second, the addition of a bypass path also increases area but improves delay.

Next, there are trade-offs between the amount of defect tolerance (yield) and area or delay. First, we can eliminate the ability to shift by two. This reduces the size of the shifting multiplexers. To continue to tolerate bridging and source-drain shorts, we require two +1 shifts followed by two -1 shifts. In fact, any combination of shifts, a +2 shift followed by two -1, two +1 followed by a -2 is acceptable. However, changing the shifting nature of the architecture increases the number of defect classes and the repair length. An increase in repair length negatively affects the number of defects we can tolerate. Second, usage of the bypass path improves delay, but this increases the repair length and consequently lowers yield.

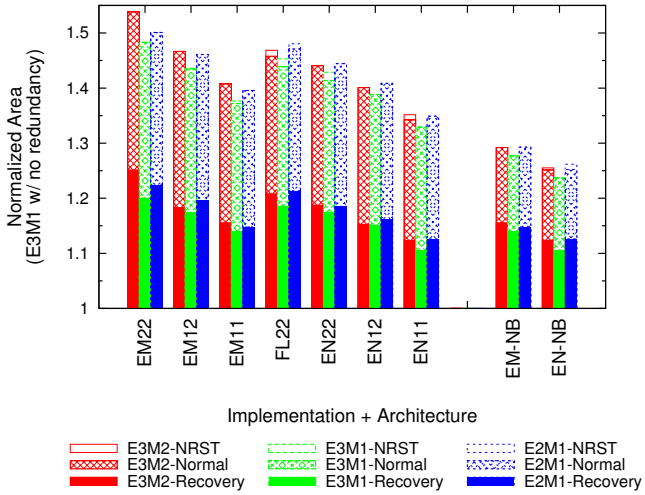

Fig. 8. Area of defect-tolerant implementations.

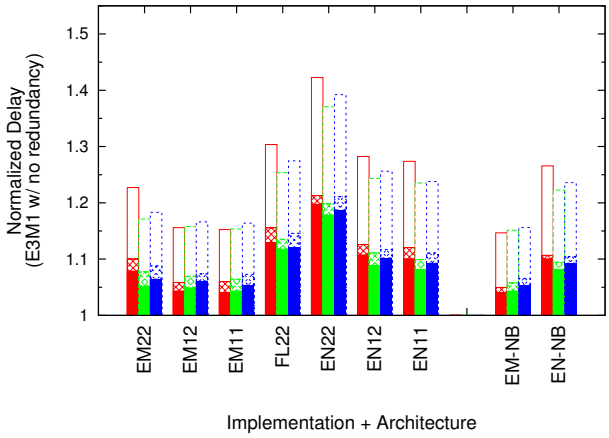

Fig. 9. Delay of defect-tolerant implementations.

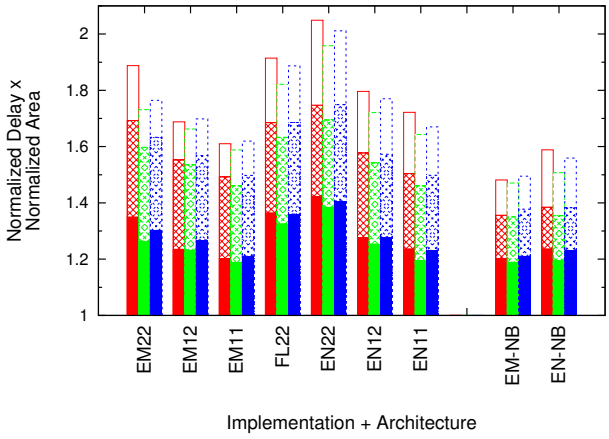

Fig. 10. Area-delay product comparison.

4. RESULTS

The new architectural features were incorporated into an enhanced version of the VPR place and route tool, VPRx [16], which now supports directional wires [15]. VPRx was then used to map the 20 largest MCNC benchmark circuits [1] into an island-style FPGA consisting of directional length 4 wires and CLBs containing eight 4-input LUTs. The area and critical path delay results we present are the geometric averages of all benchmarks as reported by VPRx. The yield results were obtained from Monte Carlo simulations.

4.1. Area

Routing experiments with non-defect tolerant switch blocks indicated that the directional switch *E3M1* uses 1.5% less area than *E3M2* and *E3M1*. The average critical path delay for *E3M1* was also 4.5% lower than the other two architectures. In comparison, the average channel width increased by 3.2% and 8.9% for *E3M1* and *E2M1*, respectively. Hence, we selected the non-defect tolerant *E3M1* to be the basis for all area and delay normalization (=1.0).

Figure 8 presents the average area overhead for the defect-tolerant switch blocks. The results have been normalized to the non-defect tolerant *E3M1*, the best alternative without defect tolerance. When routing the design in *normal* mode, two spare sets of wires were added in the channel for the 7 architectures that tolerate bridging defects. Only one spare set of wires is inserted for the architectures that do not tolerate bridging defects (2 architectures with -NB). These spare wires were not used during routing. The EN11-E3M1 architecture was the most area-efficient, having an area overhead of 24% for non-bridging defects and 34% for bridging defects. The difference in area cost (10%) is one set of spare wires. Notice that the second-best area architecture, EM11-E3M1, needs +4% to embed the *imux* but it tolerates more defects.

We should note that we do not compare our area results to the traditional spare row/column approach. Although the area overhead of spare rows/columns is very clear, the additional circuitry required within each CLB to bypass a faulty row/column is non-trivial and is not reported in previous work.

4.2. Delay

The average critical path delay for each architecture is shown in Figure 9. These numbers were obtained by rerouting the 20 benchmark circuits using a channel width equal to the minimum channel width obtained from the defect-tolerant area investigation plus one additional set of wires. Unlike the spare wires that are held in reserve, the router was allowed to use this new set of wires to relieve delay increases caused by congestion. Our experiment indicated that the EM11-E3M2 architecture gave the lowest average critical path delay overhead of 15%. Overhead for the non-embedded version, EN11-E3M1 was 24%.

Figure 9 also shows the importance of the bypass path. Results without the bypass are labelled -NRST (no route on straight through) and experience higher delay. However, section 4.4 below explains why using the bypass negatively impacts yield.

4.3. Area and Delay Recovery

Next, we explored the true area and delay overhead by using the routing tool in recovery mode. In general, we observed that the

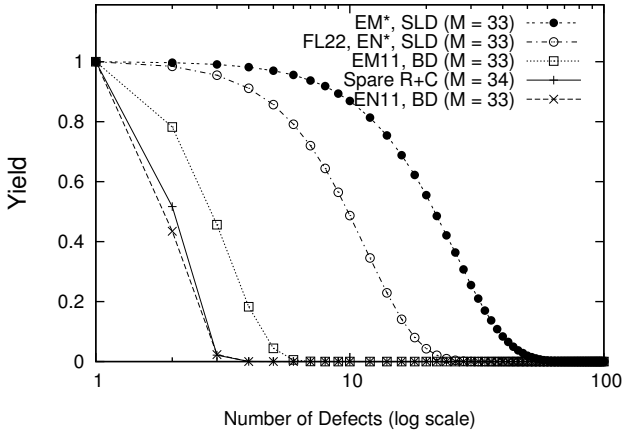

Fig. 11. Yield comparison of fault classes.

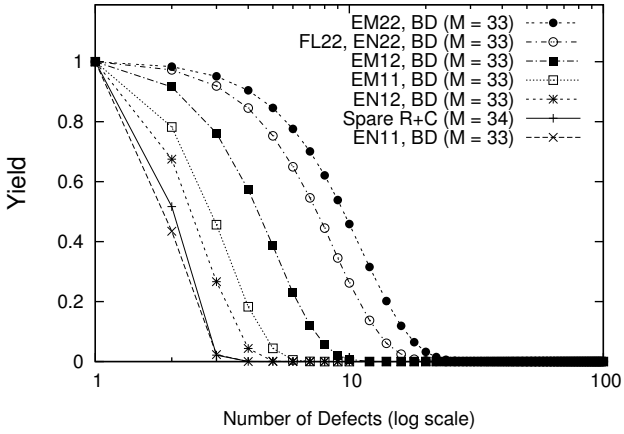

Fig. 12. Yield comparison of defect-tolerant implementations.

router needs lower channel widths for defect-tolerant architectures in recovery mode than non-defect-tolerant architectures. Hence, the additional multiplexers do help improve interconnect flexibility.

Figure 8 shows the true area overhead for our defect-tolerant implementations in recovery mode. The EN11-E3M1 architecture demonstrated the lowest area overhead of 11%.

The critical path delay overhead in recovery mode is shown in Figure 9. The EM11-E3M2 architecture demonstrated the lowest true delay overhead of 5%.

Using the delay and area results obtained from the previous two experiments, we computed the area-delay product for each architecture in recovery mode. Figure 10 shows that the EM11-E3M1 architecture produced the lowest area-delay product.

4.4. Failure Analysis

In addition to comparing the defect tolerance of the different implementations of our architecture, we also compared the expected yield of our approach to an architecture containing both one spare row and one spare column. The ability to swap a row and a column allows the latter architecture to tolerate multiple defects within the same channel. However, as the array size grows, it becomes increasingly unlikely that a second defect lies in the same row/column. Hence, this architecture fails when there are defects in two (or more) different rows or columns.

Yield estimates were obtained through Monte Carlo simulations. For a given number of defects, random faults were injected into the interconnects for 100,000 different FPGA dies. When a defect is injected into the FPGA, its trackgroup was marked. Neighbouring trackgroups that must also be perfect to ensure successful defect correction were also marked. The first defect was always tolerable, but yield failure occurs when a new defect lands in a location that was previously marked.

To simplify our results, we fixed the switch block flexibility at *E3M1* since this generally produced the best area, delay and area-delay product results. We also fixed the channel width at 80 tracks, and initially fixed the FPGA array size $M \times M$ to $M = 33$.

Figure 11 shows the result of our yield analysis for single-length defects (SLD, best case) and double-length bridging defects (BD, worst case). BD are worst-case because implementations without +/-2 shifting ability need larger regions to be defect-free.

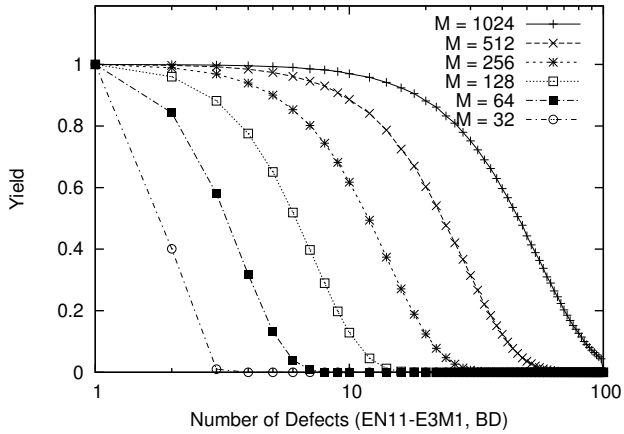

Fig. 13. Yield as die size scales.

In practice, the actual yield will lie somewhere between the SLD and BD curves. This could be determined using real manufacturing data.

Figure 12 shows a yield comparison between our 7 switch implementations for double-length bridging defects.

Figure 13 shows that our least-favourable architecture can tolerate an increasing number of the worst-case defects as the array size scales (M is increased). The spare row/column approach does not scale beyond 2 defects at all. This is important because it is anticipated that devices manufactured in future process generations will have multiple interconnect faults [6].

In these yield estimates, we are overly optimistic in two ways. First, our results assume all implementations use the same area. We anticipate that tolerating multiple defects significantly offsets the area overhead and ultimately leads to an increase in 'good die per wafer'. Second, our yield results assume the 'delay bypass' path of the *omux* is not being utilized. If it is utilized, then we must either: (a) perturb timing when correcting a defect, or (b) limit device defects to one-per-channel (instead of per-trackgroup) and use the embedded *imux*.

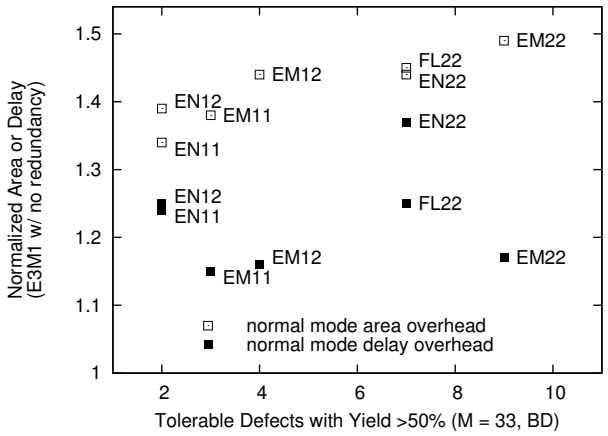

Fig. 14. Summary of area/delay overhead vs defect tolerance.

Arch	Area	Delay	Area Recovery	Delay Recovery	Yield
EM22	7	3	7	3	1
EN22	4	7	4	7	2
FL22	6	6	5	6	2
EM12	5	2	6	1	4
EM11	2	1	2	2	5
EN12	3	5	3	5	6
EN11	1	4	1	4	7

Table 2. Summary ranking of defect-tolerant schemes w/ E3M1.

5. CONCLUSIONS

This paper is the first detailed study of the true area and delay overheads required for a hardware-based fine-grained defect-tolerant interconnect scheme in FPGAs.

We presented a new defect-tolerant switch block and connection block architecture that can tolerate an increasing number of permanent manufacturing defects as the FPGA array size scales up. Our proposed scheme handles tens of distributed random defects. Previous approaches do not scale beyond 2 distributed defects. However, previous approaches are much better for clustered defects.

Our approach has a true area overhead of approximately 11% and delay overhead of 4% on aggressive applications that do not wish to be defect-tolerant.

When defect-tolerance is desired, it is tempting to include the cost of reserving a spare track. This increases area overhead to 25–40% and delay overhead to 15–25%. However, we note that less aggressive applications will already have these spare (unused) routing tracks available for free, so the actual cost is much closer to the true area overhead.

We have presented a range of implementation options that have a range of area and delay costs. Of these options, EN11-E3M1 has the lowest area, EM11-E3M2 has the lowest delay, and EM22-E3M1 has the highest yield.

Table 2 ranks the defect-tolerant switch implementations in terms of area, delay, area recovery, delay recovery, and yield. Figure 14 gives another view of the area/delay overhead and yield performance of the implementations.

6. REFERENCES

[1] Lgsynth93 benchmark set: Version 4.0. Technical report, Collaborative Benchmarking Laboratory, 1993.

[2] M. Abramovici, J. M. Emmert, and C. E. Stroud. Roving stars: An integrated approach to on-line testing, diagnosis, and fault tolerance for FPGAs. In *NASA/DoD Workshop on Evolvable Hardware*, 2001.

[3] Altera Corp. Altera's patented redundancy technology dramatically increases yields on high-density APEX 20KE devices. In *Press Release*, Nov. 27 2000.

[4] Altera Corp. In *United States patents 6,034,536, 6,166,559, 6,337,578, 6,344,755, 6,600,337 and 6,759,871*, 2000–2004.

[5] G. Asadi and M. B. Tahoori. Soft error rate estimation and mitigation for SRAM-based FPGAs. In *FPGA*, pages 149–160. ACM Press, 2005.

[6] N. Campregher et al. Analysis of yield loss due to random photolithographic defects in the interconnect structure of FPGAs. In *FPGA*, pages 138–148, February 2005.

[7] C. Carmichael, M. Caffrey, and A. Salazar. Correcting single-event upsets through Virtex partial configuration. In *Xilinx Application Notes, XAPP216 (v1.0)*, 2000.

[8] A. Doumar and H. Ito. Design of switching blocks tolerating defects/faults in FPGA interconnection resources. In *IEEE Symp. on Defect and Fault-Tolerance*, pages 134–142, 2000.

[9] A. Doumar, S. Kaneko, and H. Ito. Defect and fault tolerance FPGAs by shifting the configuration data. In *IEEE Symp. on Defect and Fault-Tolerance*, pages 377–385, 1999.

[10] S. Hareland et al. Impact of CMOS process scaling and SOI on the soft error rates of logic processes. In *IEEE Nuclear and Space Radiation Effects Conference*, pages 73–74, 2001.

[11] F. Hatori, T. Sakurai, et al. Introducing redundancy in FPGAs. In *Custom Integrated Circuits Conference*, 1993.

[12] W.-J. Huang and E. McCluskey. Column-based precompiled configuration technique for FPGA fault tolerance. In *Field Programmable Custom Computing Machines*, 2001.

[13] J. Lach, W. H. Mangione-Smith, and M. Potkonjak. Efficiently supporting fault-tolerance in FPGAs. In *FPGA*, pages 105–115. ACM Press, 1998.

[14] V. Lakamraju and R. Tessier. Tolerating operational faults in cluster-based FPGAs. In *FPGA*, pages 187–194, 2000.

[15] G. Lemieux, E. Lee, M. Tom, and A. Yu. Directional and single-driver wires in FPGA interconnect. In *Field-Programmable Technology*, 2004.

[16] G. Lemieux and D. Lewis. *Design of Interconnection Networks for Programmable Logic*. Kluwer, 2004.

[17] D. Lewis, E. Ahmed, et al. The Stratix II logic and routing architecture. In *FPGA*, pages 14–20, February 2005.

[18] J. Rose and S. Brown. Flexibility of interconnection structures in FPGAs. *J. of Solid State Circuits*.

[19] S. J. E. Wilton. A crosstalk-aware timing-driven router for FPGAs. In *FPGA*, pages 21–28. ACM Press, 2001.

[20] Xilinx, San Jose, CA. *EasyPath Solutions*, 2005. `http://www.xilinx.com/products/easypath/`.