Appearing in *IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM), May 1–3, 2016

Continuous Online Self-Monitoring Introspection Circuitry for Timing Repair by Incremental Partial-reconfiguration (COSMIC TRIP)

Hans Giesen, Benjamin Gojman*, Raphael Rubin, Ji Kim†, André DeHon
Department of Electrical and Systems Engineering,
University of Pennsylvania, 200 S. 33rd Street, Philadelphia, PA 19104
{giesen,bgojman,rafi}@seas.upenn.edu, ji@csl.cornell.edu, andre@ieee.org
*Now affiliated with Google, Inc., †Now affiliated with Cornell University

Abstract—**We show that continuously monitoring on-chip delays at the LUT-to-LUT link level during operation allows an FPGA to detect and self-adapt to aging and environmental effects on timing. Using a lightweight (<4% added area) mechanism for monitoring transition timing, a Difference Detector with First-Fail Latch, we can estimate the timing margin on circuits and identify the individual links that have degraded and whose delay is determining the worst-case circuit delay. Combined with Choose-Your-own-Adventure precomputed, fine-grained repair alternatives, we introduce a strategy for rapid, in-system incremental repair of links with degraded timing. We show that these techniques allow us to respond to a single aging event in less than 300 ms for the toronto20 benchmarks. The result is a step toward systems where adaptive reconfiguration on the time-scale of seconds is viable and beneficial.**

I. Introduction

The delay of individual circuit elements changes over time due to aging [1] and is also affected by environmental factors such as local circuit temperature and supply voltage [2]. As feature sizes shrink, the impact of aging increases. Conventional solutions that margin for worst-case environment, worst-case data sequences, and worst-case accumulated aging after multi-year lifetimes impose large timing and energy margins on circuits [3]. For instance, Gojman measured sub-2 ns delays on a 65 nm FPGA for paths that Quartus estimated at over 3 ns when accounting for composite worst-case conditions [4].

FPGAs can potentially mitigate some of these effects using post-fabrication mapping [5]. By adapting the mapping to the fabricated delays of elements, we can largely eliminate margins for process variation. Nonetheless, it is still necessary to margin for environmental factors and aging.

To address these margins it is necessary to measure and repair timing in the final, operational circuit. We show how to perform lightweight, in-system continuous monitoring to drive online adaptive mappings. As a result, the circuit can run as fast as the fabricated FPGA allows, detect when environment or aging slows a component down, and rapidly identify and repair the failing element. This exploits a unique feature of FPGAs compared to ASICs—the ability to assign resources to functions at a fine granularity after fabrication. Our solution shows how to exploit partial reconfiguration to manage rapid repair, potentially one repair every few seconds, as the FPGA operates. This is a step toward the vision of reconfigurable circuits that optimize and self-heal throughout their lifetime.

We use the Difference Detector with First-Fail Latch (DDFFL), a lightweight structure that operates on skewed clocks [6], to identify signal transition timing (Sec. III) and exploit the fact that FPGAs have flip-flops on the output of every LUT to capture fine-grained timing information at the level of interconnect paths between pairs of LUTs (Sec. IV). This allows us to identify the LUT-to-LUT link whose current delay deviates most from expectations and most impacts the critical path. Because all delays are measured during complete operation of the circuit, environmental, data, coupling effects, and clock skew are already factored into the measured delays. We use Choose-Your-own-Adventure (CYA) precomputed, fine-grained alternate paths [7] to provide concrete timing repair options for our algorithm, responding to an aging event that slows down a circuit element (Sec. IX) in tens to hundreds of milliseconds.

Our contributions include:

1) Lateness calculus that works with a Difference Detector with First-Fail Latch to assign lateness blame to individual LUT-to-LUT path links
2) Adaptive algorithm to rapidly identify the components with the largest slack violation for repair
3) Timing repair algorithm based on DDFFL, the lateness calculus, and CYA
4) Characterization of the speed of repair identifying and replacing slow circuit elements that exceed their slack.

II. Background

The small feature sizes of advanced process nodes means more pronounced wear and aging effects that also change the delay of resources [1]. Time-dependent-dielectric-breakdown (TDDB) [8], Negative Bias Temperature Instability (NBTI) [9], Hot-Carrier Injection (HCI) [10], and electromigration [11] cause FPGA resources to slow down or fail over their lifetime. The traditional approach has been to margin for worst-case degradation over the expected lifetime, sacrificing energy and performance. This can mean unreasonably poor

© 2016 IEEE

performance or short lifetimes [12]. Wear-leveling, which exploits FPGA configurability to load-balance potentially aging portions of circuits across the FPGA resources, can partially mitigate the worst-case lifetime effects [13], but many of the aging effects are stochastic, such that open-loop wear-leveling must still margin for worst-case effects.

A. Challenge

In 2004, Borkar suggested an extreme challenge: How can we handle 100 billion transistors chips where 10% of the transistors fail throughout the operational lifetime of the chip [14]? Accumulating 10^{11} errors over 10 years (3×10^8 hours) has a mean-time-between device failures on a chip of 30 ms. Failures will not be evenly distributed in time. Nonetheless, this suggests the need to develop repair strategies that work in a seconds to milliseconds time frame—far beyond the capabilities of traditional solutions.

While the work in this article does not completely solve this extreme version of the challenge, we demonstrate a potential path to addressing this problem. The CYA alternative selection we build upon has been demonstrated to handle 0.1–1% random defects [7], while full-knowledge mapping has been able to tolerate 1–10% [5], [15]. While this previous work demonstrated that the volume of defects is approachable, they have not demonstrated the necessary speed of diagnosis and repair, which we begin to demonstrate in this work.

B. Timing Extraction

It is possible to perform on-chip measurements of the path delays on an FPGA with precisions down to 1–2 picoseconds using the programmable clock generators on modern FPGAs [16], [17] and calculate the delays at the level of individual LUTs and interconnect segments in an FPGA down to $<$7 ps [4], [18]. These can potentially be used with component-specific mapping [5] to mitigate the impact of delay variation on FPGAs. Periodic recharacterization could address aging on a coarse time scale. However, this demands a long characterization period (days for even a small FPGA), management of per-FPGA resource delay maps, and a full placement-and-routing for each circuit we map to each FPGA. As such, it cannot supply the rapid characterization and repair required to address the challenge of Sec. II-A.

C. Self-heating and Local Voltage Fluctuation

Furthermore, timing extraction does not address in-situ environmental timing effects. The delay of the circuit will change with ambient temperature and the supplied voltage. Activity within the circuit will also impact the local temperature and voltage seen by resources, in turn impacting their delays [2], [19]. Consequently, even component-specific mapping based on timing extraction must margin for environmental, short-term aging, self-heating, and local voltage fluctuations.

III. Difference Detector with First-Fail Latch

If we could put every signal on the chip on an oscilloscope, we could determine their timing, including identifying which paths limit the operating clock frequency and which paths are operating slower than expected. We could integrate a digital transition monitor on a chip by building a long chain of registers, each of which handles a slightly delayed version of the clock. We identify the last time the signal transitions to its final value to identify its settling time.

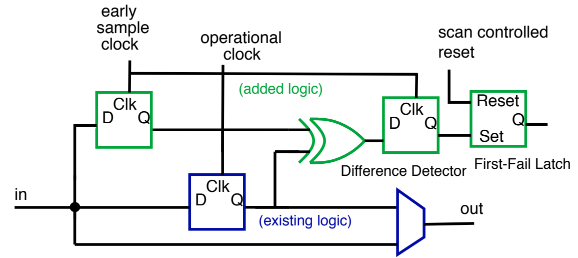

Fig. 1. Difference Detector with First-Fail Latch

Such a long sample register would be expensive, requiring a chain of 100 registers just to sample a single signal at 40 ps intervals over a 4 ns clock. We can approximate this monitor using a single register that samples at a single delay offset from the clock [6].[1] That is, by setting the delay for the sample register, we can capture the value at that delay. By changing the delay and repeating the signal transitions, we can approximate the sample chain over a number of clock cycles. By comparing this value to the final value at the end of the clock cycle (Difference Detector, DD), we can determine the shortest delay period for which the final output settled.

The delay of the circuit depends on the circuit state and the inputs that are propagating through it. The transition observed on a single sample may not be the slowest path that determines the clock cycle. However, if we run the experiment for a large number of cycles, sampling the delays of a large number of input vectors, and determine if the sampled value fails to match the final value on *any* of the cycles, we can better approximate whether or not the signal has settled to its final value at the end of the delay period. Consequently, Levine [6] adds a *first-fail latch* (FFL) after the DD flip-flop that will be set if the signal failed on any of the cycles within an experiment. This yields the Difference Detector with First-Fail Latch (DDFFL).

This setup (Fig. 1) adds only a single XOR gate, a pair of flip-flops and a scannable set-reset flip-flop to the flip-flop that already exists on the output of every LUT in a typical FPGA. The new flip-flop connected to the input gets a configurably delayed clock (early sample clock in Fig. 1) to sequentially explore the various delay offsets. The XOR computes whether or not the early sampled value matches the final value on the operational flip-flop. If they differ, it sets the difference flip-flop. The FFL effectively OR's the error over a number of test samples. At the end of the sample period, we can serially scan out the FFL values using JTAG or a data readback path. By scanning the delay across different offsets, we can identify the latest time that the signal transitions to its final value.

[1] We use the early sample clock (M_CLK in [6]) to register the difference signal, placing the "blind-spot" just before the delay offset becomes as large as the cycle—a region we do not intend to use in this application.

The DD also needs a programmable delay for the early sample clock. We can drive all the early sample clocks with the same delayed clock, so we only need one programmable delay line on the chip. Programmable delay lines are commonly used for Delay-Locked-Loops (DLLs) with a high resolution [20]. The Xilinx Ultrascale series has delay control on individual input pins down to 5–15 ps [21]. Levine shows how to use the programmable clock controls on an Altera Cyclone III to implement a programmably-tunable delay line with 96 ps of resolution on top of a conventional 65 nm FPGA [6]. Modern FPGAs already distribute tens of clocks across the chip.

The DDFFL can be implemented as a modest change to the base FPGA fabric with high timing resolution. Using the estimates for flip-flop sizes (54 minimum width transistor equivalents for a width 4 flip flop in a 40 nm process) [22] and the VTR 7.0 [23] estimates for the tile area of a Stratix-IV (84,375 minimum transistor width equivalents),[2] we estimate that adding three flip-flops and an XOR gate to each of the 20 flip-flops in the 10 logic elements in a Stratix-IV CLB [24] would increase the tile area by about 4%. By integrating the DDFFL into the FPGA fabric, we can keep the monitor signals short, minimize their variation effects, control their timing, and make sure that the monitoring circuitry does not consume application logic or congest the programmable paths used for application routing.

The DDFFL monitoring and early sample clock scanning can run continuously, concurrent with the application. It is exercised by the actual in-field data and monitors the delay of the circuit during deployed operation.

IV. Link Timing

The most obvious use for the DDFFL is to measure the register-to-register paths on an FPGA circuit, as in previous work. This will allow us to determine the delay to each output register. Then, we can identify which output is the latest to arrive. This output sets a lower bound on the clock frequency. However, once we know which output is late, we still have little idea about which LUT or interconnect hop in the fan-in cone of the output is responsible for making the output late.

We can do better by observing the timing not just at the registered output of the FPGA circuit, but at *every* LUT in the circuit. In particular, FPGAs have a flip-flop attached to the output of every LUT, and that flip-flop is there whether or not it is used. As a result, we can still use the DDFFL-augmented flip-flop to capture the delay of the LUT. Once we have the maximum delay to each LUT in the circuit, we can estimate the lateness of every LUT-to-LUT link in the netlist.

Note that variation and configuration skew in the clock network is factored into the sampled timing. If the clock arrives earlier (or later) at a LUT than its predecessor, that makes the link look slower (or faster). Since the scheme is already expecting to see delay variation in links, this just adds the clock distribution variation in with the link delay variation.

[2] Specifically, the model for the `k6_frac_N10_mem32K_40nm.xml`.

V. Lateness Calculus

The DDFFL allows us to find the maximum delay value, MD_i, for each LUT output i; that is, the latest time that the LUT i changes to its final value. Starting from this maximum delay value, we identify how late each node is. Then, we can look at the slack on the node to identify links that must be repaired to restore timing.

The lateness, L_i, of a node describes the delay difference between the actual signal arrival at the LUT's output and when we need or expect the signal to arrive, the required time RT_i:

$$L_i = MD_i - RT_i. \quad (1)$$

For aging, the required time constraint is based on the original circuit operation delay before aging. Consequently, this timing incorporates any clock skew variation into the delay expectations at each node. This raw lateness does not directly tell us the delay of a single LUT-to-LUT link because it also includes the delays accumulated in preceding LUTs along the path. To assign lateness to a single LUT-to-LUT link, we compute a relative lateness, RL_i, that cancels out the prior LUT delays:

$$RL_i = L_i - \max_{j \in Inputs(i)} L_j. \quad (2)$$

$Inputs(i)$ is the set of LUTs immediately preceding LUT_i.

A. Component-Specific Slack

A late LUT that is not on the critical path may not impact the circuit delay. Therefore, we are primarily concerned with identifying late signals that exceed their slack budget. The slack is the amount of delay that can be added to a node before exceeding its latest possible arrival time ($ALAP_i$) delay and will impact the critical path, *i.e.*:

$$Slack_i = ALAP_i - ASAP_i. \quad (3)$$

If we compute ASAP and ALAP values entirely from nominal delay, we account neither for the fact that some elements actually run faster than nominal, nor for the potential clock skew variation. As a result, we get a more accurate and useful value by defining ASAP/ALAP delays using the actual measured delays. $ASAP_i$ is simply the maximum delay MD_i, while $ALAP_i$ can be derived using

$$ALAP_j = \min_{i \in Outputs(j)} (ALAP_i - D_{i,j}), \quad (4)$$

where $Outputs(j)$ is the set of immediate successors of LUT_i and $D_{i,j}$ is the delay between the outputs of LUTs i and j. We cannot exactly determine $D_{i,j}$ based on the MD_i's, but we can approximate it. The lateness of LUT i can be caused by both lateness of LUT_j and increased propagation delay through LUT_i. Hence the lateness of LUT i is probably caused by the predecessor j with the highest $MD_j + ND_{i,j}$, where $ND_{i,j}$ is the nominal value of $D_{i,j}$. If we assume that the delay increase on all paths through LUT_i are the same,

$$D_{i,j} = MD_i + ND_{i,j} - \max_{k \in Inputs(i)} (MD_k + ND_{i,k}). \quad (5)$$

If we determine and store the MD_i values during operation before an aging event, we can precompute the slack associated with each node for use in prioritizing repairs.

VI. Identifying the Latest LUT

The simplest approach to determining timing, is to sweep the early sample clock at regular precision steps, $Precision$, across the entire clock cycle period, noting the latest time at which the difference detector detects an erroneous value on the early sampled output (Alg. 1).

Algorithm 1 Brute-force algorithm

> **for** each LUT$_j$ **do**
> $MD_j.delay \leftarrow 0$; $MD_j.fail \leftarrow$ **false**
> **for** $i \leftarrow Clock_Period$ **downto** 0 **by** $Precision$ **do**
>   Reset difference detectors
> $EarlySampleOffset \leftarrow i$
>   **for** $j \leftarrow 1 \ldots Cycles$ **do**
>   Run circuit
>   **for** each LUT$_j$ **do**
>   **if** error(LUT$_j$) **and** $\overline{MD_j.fail}$ **then**
> $MD_j.delay \leftarrow i$; $MD_j.fail \leftarrow$ **true**
> Slowest LUT $\leftarrow$ LUT with highest $RL_i - Slack_i$

A. Adaptive Refinement Algorithm

The smaller the difference in relative lateness between LUTs, the higher the $Precision$ a search algorithm requires to distinguish them. A major drawback of the brute-force algorithm is that the entire delay range is measured at the same precision. However, if our goal is to simply identify the best repair candidate, we do not need that fine of a resolution throughout the entire range. Typically, this means high resolution measurements are only needed around the repair target. To speed up late LUT identification, we develop an adaptive algorithm that effectively reduces the sample resolution outside areas of interest.

The adaptive algorithm replaces the fixed quantization steps of the brute-force algorithm with variable intervals. Assuming that MD_i is located in the interval $[MD_i^l, MD_i^h]$:

$$L_i \in [L_i^l, L_i^h] = [MD_i^l - RT_i, MD_i^h - RT_i]. \qquad (6)$$

The relative lateness in turn satisfies

$$RL_i \in [RL_i^l, RL_i^h] = [L_i^l - \max_{j \in Inputs(i)} L_j^h, L_i^h - \max_{j \in Inputs(i)} L_j^l].$$

As a result, at any point in our adaptive refinement, we have an estimate on the delay and relative lateness of each signal. Our goal is to tighten the RL intervals until we can identify a slow LUT that most exceeds its available slack ($\overline{RL_i} - Slack_i > \overline{RL_j} - Slack_j$ for all $j \neq i$, with $\overline{RL}$ indicating the center of interval RL). To refine our estimates, we pick a candidate LUT with the largest $RL_i^h - Slack_i$. As long as its $RL_i^l - Slack_i$ is less than some other LUT's $RL_i^h - Slack_i$, we do not know if this really is the LUT that most exceeds its slack bound. By performing a measurement at $EarlySampleOffset$, T, within a delay interval $[MD_i^l, MD_i^h]$, we can tighten the delay by updating either MD_i^h, if it fails, or MD_i^l if it succeeds. As a consequence, we will either tighten the upper bound for our candidate, perhaps such that it no longer has the largest RL_i^h, in which case we have a new candidate with maximum delay to refine, or we will tighten its lower bound, reducing the set of LUTs whose RL intervals overlap with it. A measurement at a particular T will sample the outputs of all the LUTs and allow us to update all the MD intervals that enclose T and, consequently, all their associated RL intervals. We continue refining until we are left with a LUT that unambiguously most exceeds its slack (Alg. 2).

Algorithm 2 Adaptive Refinement Algorithm

> Let m be the LUT that maximizes $RL_m^h - Slack_m$
> $Candidates \leftarrow$ all LUTs whose intervals overlap RL_m
> **while** $\#Candidates > 1$ **or** no refinement possible **do**
>   Pick $EarlySampleOffset$
>   based on MD_m and $\{MD_j \mid j \in \text{pred}(m)\}$
>   Run $Cycles$ clock cycles
>   Update MD, RL intervals of all LUTs
>   Let m be the LUT that maximizes $RL_m^h - Slack_m$
> $Candidates \leftarrow$ all LUTs whose intervals overlap $RL_m - Slack_m$
> **return** LUT with largest $\overline{RL_m} - Slack_m$

In Fig. 3 we normalize the performance of the adaptive algorithm to the brute-force algorithm, showing that the adaptive algorithm converges 2–5$\times$ faster than Alg. 1.

B. Cycles per Offset

A key question in the timing algorithm is determining the number of *Cycles* to sample and observe at each delay offset. Since path sensitization can be data dependent, the slowest path may not be sensitized after a small number of cycles. If $Cycles$ is too low, the worst-case delay paths may remain unsensitized leaving the algorithm with too low of a maximum delay estimate. If the value is too high, the algorithm converges more slowly. If some paths are truly infrequent, or even never activated for some data sets, it may be desirable to optimize ignoring these infrequent slow paths and spend additional time recovering in the rare case that they do occur.

If the inputs were random and the design were purely combinational, we could treat this as a Coupon-Collector Problem and see that it would take $i \times 2^i$ random input vectors to have a 50% chance of seeing all 2^i potential input vectors [25]. As we increase the number of sample cycles, we increase the chance of having a full set. For combinational inputs, two factors cause our real logic to behave differently from random: (1) many input vectors sensitize the same path, so we do not need all input vectors; (2) inputs do not occur with equal frequency. Registered inputs to internal logic cones complicate the issue further. When there is logic in front of the register, the data-processing inequality says that the register value will not be more random than the input [26]. Furthermore, if the register is only loaded on select cycles, the logic paths following the register may not be activated with a unique input value on every cycle.

With the adaptive algorithm, we can use internal self-consistency of the measurements to detect some cases where

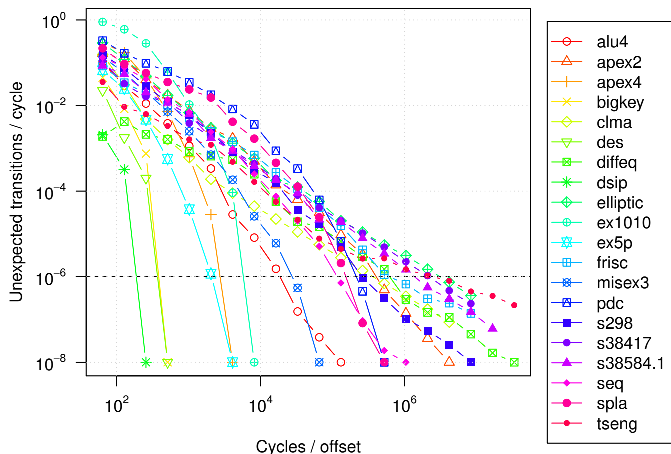

Fig. 2. Correlation between False Positive Rate and Cycles per Offset

the MD_i^h estimate is too low; that is, if we run at an $EarlySampleOffset$ larger than MD_i^h and see a timing failure, we know the MD_i^h estimate is too low. We call such transitions *unexpected* and use their rate to help identify the appropriate number of $Cycles$ to use in the algorithm.

We find that the algorithm makes reasonable forward progress when the unexpected transition rate is below 10^{-6}. The $Cycles$ setting to achieve this unexpected transition rate differs from design to design. The probability of observing rare transitions within one iteration decreases when the number of cycles per offset increases as shown in Fig. 2.

Tab. I presents the number of cycles per offset for an unexpected transition probability of 10^{-6} as used for all simulation results. To gain some insight into the dependence of cycles on design features, we used linear regression to attempt to fit a model for the cycles per offset:

$$\hat{w} = \arg\min_{w} \sum_{i} |x_i w_{lin} + EXP(x_i) w_{exp} + w_0 - y_i|^2 \quad (7)$$

Here, x_i is a vector with the five metrics of Tab. I for design i, each normalized to a standard deviation of 1. $EXP(x_i)$ is similarly a vector with the exponentials of each of the five metrics, and w_0 is a constant determining the intercept. The response, y_i, is the number of cycles per offset from our method. The obtained weights are summarized at the bottom of Tab. I ($w_0 = -2236672$), indicating a strong influence of the flip-flops. Together, this achieves a correlation coefficient of 0.93, predicting the largest $Cycles$ values within 45%.

VII. Timing Repair by Incremental Partial-reconfiguration (TRIP)

COSMIC TRIP can be used to reduce critical path delay to deal with timing faults. Timing Repair by Incremental Partial-reconfiguration (TRIP) employs CYA precomputed alternatives [7] for repair. CYA reserves a set of FPGA resources (*e.g.*, LUTs, wiring tracks) for use during repair. Normal routing is performed on a set of non-reserved base resources, while alternate routes are allowed to use these reserved repair resources as well as unused base resources. The CYA router runs once for a design. It generates a large number of alternate

TABLE I
Toronto20 [27] Benchmark design characteristics

Design	LUTs	Flip-flops	Logic depth	Reg. fan-in	Total fan-in	Cycles / offset
alu4	824	0	6	0	14	22691
apex2	971	0	7	0	36	476919
apex4	793	0	6	0	9	4023
bigkey	579	224	4	2	10	512
clma	3223	33	11	274	282	409723
des	557	0	4	0	19	511
diffeq	666	377	8	84	85	741103
dsip	689	224	4	2	10	256
elliptic	1816	1122	10	325	326	3475941
ex1010	2589	0	7	0	10	8147
ex5p	578	0	5	0	8	2351
frisc	1744	886	12	121	124	666631
misex3	768	0	6	0	14	31421
pdc	2205	0	7	0	16	250880
s298	653	8	9	278	281	260465
s38417	2606	1463	6	64	64	2297882
s38584.1	2325	1260	7	101	109	1586519
seq	867	0	6	0	38	126826
spla	1853	0	7	0	16	202004
tseng	647	385	7	50	51	2609885
w_{lin_j}	-1077149	1634715	1214510	-1877499	144876	
w_{exp_j}	97629	-94782	-15221	712832	-393838	

LUT-to-LUT paths for every two-point net in the design and stores those in an expanded bitstream. As a result, there is a single bitstream generated for every design and used across all chips. A simple FPGA bitstream loader FSM embedded in the FPGA logic can test each LUT-to-LUT link as it is installed and replace it if it is defective. Rubin's CYA loader will only detect defects, not timing faults. TRIP shows how to use the CYA alternatives to repair timing faults.

A. Algorithm

COSMIC's DDFFL and the lateness calculus are only able to tell us which LUT is slow, not which input link to the LUT is latest, making the LUT slow. However, the alternatives that CYA offers are individual LUT-to-LUT links. Since we do not know which input link is slow, we need to try repairing each of them. Consequently, TRIP (Alg. 3) randomly selects one of the K LUT inputs from the slowest LUT to repair. It replaces the present alternative with the next one from a cyclically ordered set with N alternatives. This method ensures that each of the $K \cdot N$ alternatives is tried in a short period. If a single input link causes the LUT to be late, this will repair it relatively quickly. It also deals with repairing multiple input links. The strategy of successively, randomly selecting and installing alternatives in a cyclic fashion will eventually sample all potential N^K combinations to deal with this case where multiple inputs must be repaired. The TRIP algorithm runs on a processor embedded in the FPGA fabric, such as an ARM core on a Zynq or Arria SOC-FPGA.

B. Memory for Repair

We must represent the graph and the delay state of the LUTs within the graph to support the TRIP algorithm. We capture the graph by representing each of the K predecessors to each LUT. For an N_{lut}-node design, this means $K \cdot N_{lut}$ numbers.

Algorithm 3 TRIP Algorithm

Locate slowest LUT$_j$ (Alg. 2)
$t_l \leftarrow \max_{k \in LUTs} MD_k^l$; $t_h \leftarrow \max_{k \in LUTs} MD_k^h$
Store current alternatives for inputs of LUT$_j$
$Attempt \leftarrow 0$
repeat
  Replace random input of LUT$_j$ with alternative
  Reset MD of LUTs affected by repair
  Run $Cycles$ with $EarlySampleOffset = t_l$
  Update MD intervals.
  $t \leftarrow \max_{k \in LUTs} MD_k^h$
  $Attempt \leftarrow Attempt + 1$
  Restore alternatives of LUT$_j$ if $t \geq t_l$
until $t < t_l$ or $Attempt > Max_attempts$

If we use a 32b pointer (4B) for each predecessor, that means $4K \cdot N_{lut}$ bytes. For state, we need to keep the intervals of MD, RT, L, and $Slack$ for each node, as well as D and ND for each link. This gives us $(8+4K)\,N_{lut}$ numbers to store. A 16b (2B) delay value would allow us to represent 1 ps resolution for up to 64 ns of delay; and 1B would support up to 256 ps. If we use 1B for D and ND and 2B for the rest, this means we need $(16+4K)\,N_{lut}$ bytes to store timing state. For $K = 6$, the 40B per LUT for timing is about twice the 24B per LUT to store the graph. Together, this means 64MB to support a one million 6-LUT design.

VIII. Methodology

We augmented a version of VPR 5.0.2 [28] to model independent transistor variation and CYA alternatives. We use a Predictive Technology Mode (PTM) [29] for the base technology. We assume a 22-nm CMOS process with $\mu_{V_{th}}$=400mV, $\sigma_{V_{th}}$=36mV, and typical operating V_{dd} =0.8V. The V_{th} of individual transistors is sampled randomly and independently from a Gaussian distribution with this $\sigma_{V_{th}}$. This models both variation and the accumulated effects of random aging events that may have preceded the point aging experiments we perform. HSPICE simulations provide the delay model for each buffer in the interconnect.

We use an island-style architecture [30] with 6-input LUTs ($K = 6$) and 10 LUTs per cluster ($N = 10$) and a segment length of 4, similar to a Stratix-IV [24]. Routing is directional with a Wilton-style S-box and F_{cin}=0.15, F_{cout}=0.1 C-box connectivity. Clusters have 33 inputs. To support CYA, we reserve 2 LUTs, 6 inputs, and 20% channel width beyond what is needed for base low-stress route for repairs. This means the initial mapping is to an $N = 8$ cluster with 27 inputs targeting a low-stress route (20% channels over minimum for the $N = 8$ design) consistent with previous work [31]. Base routes are mapped with full-knowledge of delays [5], providing a high-quality mapping as our repair target. We generate 64 alternate routes for each two-point net in the original design.

Standard static timing analysis as in VPR does not capture the delays as a function of data. Consequently, to model data-dependent transition timing and the samples captured by the difference detector, we developed a custom simulator that tracks all transitions and their timing through the mapped circuit netlist. Since the Toronto 20 benchmarks do not come with representative test vectors, we used random data for the inputs, with care to treat clocks and resets appropriately. The simulator works on path delays from our modified VPR and has the ability to revise the path delays based on the selected CYA alternatives.

The key issue in aging is delays that exceed the available slack. Delays below the slack for a node or link will not impact the operational frequency for the circuit. Consequently, for our experiments, when we add delay to a link, we add the sum of the slack for the link, $Slack_{i,j}$, and the added delay, d_{add}, for a total of $Slack_{i,j} + d_{add}$.

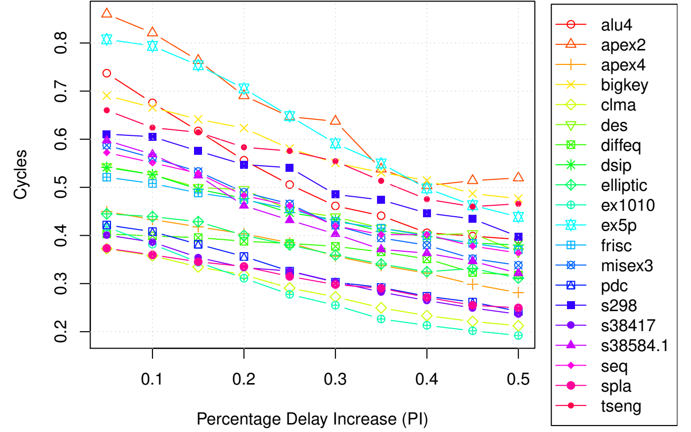

Fig. 3. Aging Experiment: Cycles Dependency on Delay Increase

IX. Aging Repair

We perform two sets of experiments. In the first set, we add delays that increase the delay of a link above the slack by a fixed percentage, PI, of the link delay ($d_{add_{i,j}} = PI \times D_{i,j}$). This allows us to characterize how the time to localize and repair a link varies with the amount by which the delay exceeds the previous critical path delay. We expect that localizing delays becomes easier as the delay increases. We pick 100 random nodes and inject a delay for the specified PI. The number of cycles spent for PI values from 5% to 50% in increments of 5% is shown in Fig. 3, normalized to cycles required with a brute-force algorithm for $PI = 20\%$. Fig. 3 shows that the time to locate the injected delay typically does decrease with the magnitude of the delay increase, but the decrease is less than a factor of two across this range.

In the second experiment, we also inserted delays at random LUT-to-LUT links in the circuit. Here, we both pick a random node and a random delay percentage for d_{add}. The absolute value for d_{add} is sampled from a Gaussian distribution with $\mu = d_{nominal}$ and $\sigma = 0.3 \cdot d_{nominal}$. Fig. 4 shows the average time-to-locate the slowest resource across a series of 100 aged-delay-insertion experiments on each of 5 chips, where each "chip" has an independent set of transistor V_{th} delays.

Since the repair algorithm is unaware of the speed of input links and alternatives, it may need to try many alternatives (Sec. VII) to repair the aged link. The average number of repairs, N_a, required to restore timing is shown in Tab. III with

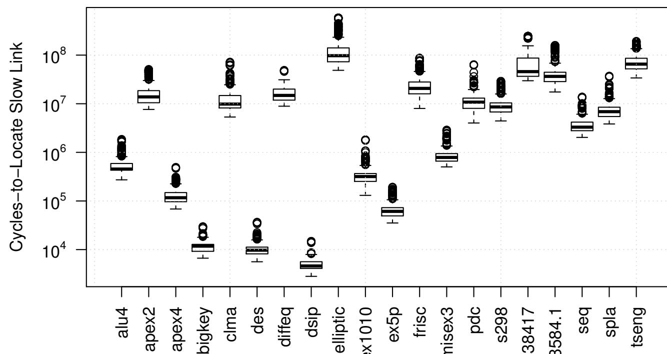

Fig. 4. Aging Experiment: Cycles-to-Locate Slowest Resource

TABLE II
ABSOLUTE REPAIR TIME MODEL PARAMETERS

Var.	Description	Value
N_{off}	Number of values needed for $EarlySampleOffset$ to find slowest LUT	
T_{coff}	Time to configure the sample clock offset [32]	20 μs
$Cycles$	Cycles per iteration	Tab. I
T_{age}	Clock cycle time for unrepaired logic	
N_{lut}	Number of LUTs on the FPGA (assume smallest square that encloses design)	
T_{bit}	Rate to scan bits into device [7]	1b/ns
T_{nxt}	Compute time to update ranges and decide next $EarlySampleOffset$ (Intel Xeon, 2.7 GHz)	
N_a	Number of alternatives installed to recover from aging event	
L_p	Number of segments in repaired path	
T_{frm}	Time to load a frame [7]	1.3 μs

5×100 injected delays. The designs require on average 5.2 repair trials, suggesting that 0.87 alternatives are evaluated for each of the 6 inputs before finding an available and adequate performance alternative.

A. Time-to-Repair

In addition to time to run experiments, it will also cost time to (a) set the $EarlySampleOffset$ and clear difference detectors, (b) read out a set of detector values, (c) decide which sample offset to use next, and (d) reconfigure from one CYA alternative to another. To get an estimate of these effects, we use the following model for total repair time:

$$T_{repair} = (N_{off} + N_a)\,(T_{coff} + Cycles \cdot T_{age} + N_{lut}T_{bit}) + N_{off} \cdot T_{nxt} + N_a \cdot 2L_pT_{frm}$$

Tab. II describes the variables and the technology parameters we use in estimation. This assumes a frame-oriented configuration model similar to the Virtex series [33], and we make the conservative assumption that every segment is in a distinct frame (L_pT_{frm}); the multiplier of two accounts for the fact that an old path must be removed along with the mapping of a new path. Tab. III shows the time to repair. From Fig. 5, it is clear that the time running samples (next to last column in Tab. III, $(N_{off} + N_a) \cdot Cycles \cdot T_{age}$) and the time computing the next offset ($N_{off} \cdot T_{nxt}$) dominate total repair time.

We capture T_{nxt} from a Java implementation of Alg. 1 and Alg. 3 on our Intel Xeon simulation machines. In practice, we envision the repair algorithm running in tightly coded C on an embedded processor on the FPGA, such as the ARM on a Zynq or Arria. We see that the total algorithm computation time ($N_{off} \cdot T_{nxt}$) is less than 28 ms and only dominates in cases where $Cycles < 10^5$. Even if it ran 30× slower, total repair time would still be less than a second in the worst-case.

TABLE III
COMPARISON OF TIME-TO-REPAIR

Design	N_{off}	T_{age}	T_{nxt} (μs)	N_a	L_p	**Time** (ms) Samples	T_{repair}
alu4	23	1.03 ns	95	4.6	11.7	0.55	3.6
apex2	35	1.22 ns	94	13.6	15.4	20.38	33.2
apex4	33	1.61 ns	64	4.5	15.6	0.21	3.3
bigkey	22	0.64 ns	101	3.8	15.2	0.01	2.9
clma	30	2.07 ns	193	3.4	15.6	25.17	34.7
des	20	0.84 ns	101	4.9	17.0	0.01	2.8
diffeq	22	0.86 ns	139	4.3	15.5	13.97	20.6
dsip	18	0.68 ns	123	3.2	12.5	0.00	2.8
elliptic	36	1.94 ns	608	5.0	15.7	241.12	297.8
ex1010	40	2.71 ns	82	3.9	16.9	0.89	5.4
ex5p	27	0.99 ns	79	4.0	13.2	0.06	3.0
frisc	37	1.39 ns	278	8.7	5.0	34.85	53.7
misex3	27	1.10 ns	81	4.1	12.5	0.93	4.0
pdc	45	2.89 ns	103	14.2	19.8	32.91	49.8
s298	37	1.41 ns	76	6.1	12.9	13.64	19.8
s38417	25	1.43 ns	1101	4.3	17.2	80.59	123.1
s38584.1	25	1.08 ns	467	4.9	13.4	42.62	63.4
seq	29	1.34 ns	66	4.2	13.6	4.98	8.5
spla	38	2.12 ns	62	4.8	15.9	16.10	21.6
tseng	29	0.85 ns	75	3.6	12.6	64.78	75.8

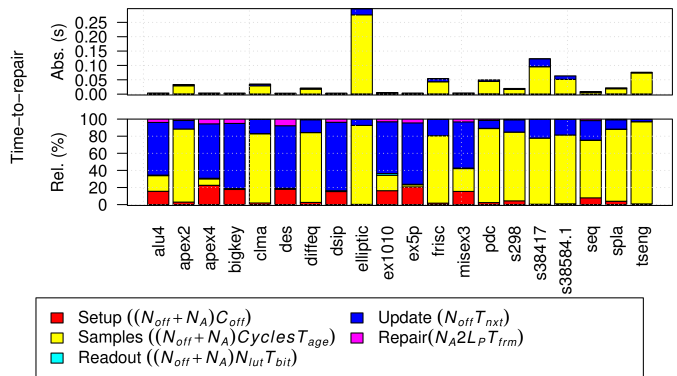

Fig. 5. Breakdown of Execution Time

While these are small designs and the repair time is still large for the Borkar challenge (Sec. II-A), this shows how we can bring repair time down to the right magnitude. We believe this is enabling and can be paired with a few additional techniques to fully address the challenge.

X. DISCUSSION AND FUTURE WORK

COSMIC TRIP implementations must be prepared to capture aging failures that exceed the operational clock cycle. As we have already noted, using only the data inputs to the system to drive timing measurements, there is no guarantee the system will see all the delay paths in any bounded number of cycles. This means it will also need to catch operational timing failures when these paths are sensitized. As a result, implementations will need to employ a technique

for detecting timing-delay faults (*e.g.*, [34]) or circuit logic errors (*e.g.*, [35]) alongside our continuous monitoring and repair. This technique will work most naturally in a system and application environment that can tolerate variable frequencies and potential stalls, such as a best-effort accelerator, streaming operators with data presence, and latency-insensitive systems.

COSMIC TRIP monitoring and repair could be used to address many problems beyond aging. Incremental repair could address variation and the on-chip coupling impacts such as self-heating and local V_{dd} drop [19], and it will be useful to evaluate its effectiveness in these contexts. The ability to improve timing in system could also be translated into the ability to reduce voltage, and hence energy, to meet fixed timing requirements. We expect that there is room to optimize beyond the simple algorithms we employ, accelerating localization and repair and consuming alternatives more judiciously.

XI. Conclusions

Fine-grained, continuous monitoring of the delay of signals on an FPGA during operation can be very lightweight. This monitoring can be used to identify the resources impacted by an aging event to prioritize them for repair. This localization and repair allows us to reduce the timing and energy margins that must be applied when the FPGA is run open-loop with no knowledge of aging or environmental effects. Coupled with a source of spare resources that can be installed in milliseconds, this allows in-system repairs in the sub-second range.

References

[1] E. A. Stott, J. S. J. Wong, P. Sedcole, and P. Y. K. Cheung, "Degradation in FPGAs: measurement and modelling," in *FPGA*, 2010, p. 229.

[2] K. M. Zick and J. P. Hayes, "On-line sensing for healthier FPGA systems," in *FPGA*, 2010, pp. 239–248.

[3] E. Mintarno, J. Skaf, R. Zheng, J. Velamela, Y. Cao, S. Boyd, R. Dutton, and S. Mitra, "Self-tuning for maximized lifetime energy-efficiency in the presence of circuit aging," *IEEE Trans. Computer-Aided Design*, vol. 30, no. 5, pp. 760–773, May 2011.

[4] B. Gojman, S. Nalmela, N. Mehta, N. Howarth, and A. DeHon, "GROK-LAB: Generating real on-chip knowledge for intra-cluster delays using timing extraction," *ACM Tr. Reconfig. Tech. and Sys.*, vol. 7, no. 4, pp. 5:1–5:23, Dec. 2014. [Online]. Available: http://doi.acm.org/10.1145/2597889

[5] N. Mehta, R. Rubin, and A. DeHon, "Limit Study of Energy & Delay Benefits of Component-Specific Routing," in *FPGA*, 2012, pp. 97–106.

[6] J. M. Levine, E. Stott, G. A. Constantinides, and P. Y. Cheung, "Online measurement of timing in circuits: for health monitoring and dynamic voltage & frequency scaling," in *FCCM*, 2012, pp. 109–116.

[7] R. Rubin and A. DeHon, "Choose-Your-Own-Adventure Routing: Lightweight Load-Time Defect Avoidance," *ACM Tr. Reconfig. Tech. and Sys.*, vol. 4, no. 4, December 2011.

[8] E. Rosenbaum, P. Lee, R. Moazzami, P. Ko, and C. Hu, "Circuit reliability simulator-oxide breakdown module," in *IEDM*, December 1989, pp. 331–334.

[9] D. K. Schroder and J. A. Babcock, "Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing," *J. App. Phys.*, vol. 94, no. 1, pp. 1–18, July 2003.

[10] S.-H. Renn, C. Raynaud, J.-L. Pelloie, and F. Balestra, "A thorough investigation of the degradation induced by hot-carrier injection in deep submicron n- and p-channel partially and fully depleted unibond and SIMOX MOSFETs," *IEEE Trans. Electron Devices*, vol. 45, no. 10, pp. 2146–2152, October 1998.

[11] S. Alam, G. C. Lip, C. Thompson, and D. Troxel, "Circuit level reliability analysis of Cu interconnects," in *ISQED*, 2004, pp. 238–243.

[12] L. Condra, J. Qin, and J. B. Bernstein, "State of the art semiconductor devices in future aerospace systems," in *Proc. FAA/NASA/DoD Joint Council on Aging Aircraft Conf.*, April 2007.

[13] S. Srinivasan, R. Krishnan, P. Mangalagiri, Y. Xie, V. Narayanan, M. Irwin, and K. Sarpatwari, "Toward increasing FPGA lifetime," *IEEE Trans. on Dep. and Secure Comput.*, vol. 5, no. 2, pp. 115–127, April 2008.

[14] S. Borkar, "Microarchitecture and design challenges for gigascale integration," http://www.microarch.org/micro37/presentations/MICRO37%20Sborkar.pdf, December 2004, keynote talk *Int. Symp. on Microarchitecture*.

[15] A. DeHon and N. Mehta, "Exploiting partially defective LUTs: Why you don't need perfect fabrication," in *ICFPT*, December 2013.

[16] J. S. Wong, P. Sedcole, and P. Y. K. Cheung, "Self-measurement of combinatorial circuit delays in FPGAs," *ACM Tr. Reconfig. Tech. and Sys.*, vol. 2, no. 2, pp. 1–22, June 2009. [Online]. Available: http://doi.acm.org/10.1145/1534916.1534920

[17] T. Tuan, A. Lesea, C. Kingsley, and S. Trimberger, "Analysis of within-die process variation in 65nm FPGAs," in *ISQED*, March 2011, pp. 1–5.

[18] B. Gojman and A. DeHon, "GROK-INT: Generating real on-chip knowledge for interconnect delays using timing extraction," in *FCCM*, 2014, pp. 88–95.

[19] T. A. Linscott, B. Gojman, R. Rubin, and A. DeHon, "Pitfalls and tradeoffs in simultaneous, on-chip FPGA delay measurement," in *FPGA*, February 2016, pp. 100–104.

[20] S. Sidiropoulos and M. Horowitz, "A semidigital dual delay-locked loop," *IEEE J. Solid-State Circuits*, vol. 32, no. 11, pp. 1683–1692, Nov 1997.

[21] Xilinx, "Ultrascale architecture and product overview (ds890)," http://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf, August 2015.

[22] J. Goeders and S. J. Wilton, "VersaPower: Power estimation for diverse FPGA architectures," in *ICFPT*, 2012, pp. 229–234.

[23] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz, "VTR 7.0: Next generation architecture and CAD system for FPGAs," *ACM Tr. Reconfig. Tech. and Sys.*, vol. 7, no. 2, pp. 6:1–6:30, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2617593

[24] D. Lewis, E. Ahmed, D. Cashman, T. Vanderhoek, C. Lane, A. Lee, and P. Pan, "Architectural enhancements in Stratix-III and Stratix-IV," in *FPGA*, 2009, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/1508128.1508135

[25] F. G. Maunsell, "A problem in cartophily," *The Mathematical Gazette*, vol. 22, pp. 328–331, 1937.

[26] T. Cover and J. Thomas, *Elements of Information Theory*. New York: John Wiley and Sons, Inc., 1991.

[27] V. Betz and J. Rose, "FPGA Place-and-Route Challenge," <http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html>, 1999.

[28] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, and J. Rose, "VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling," in *FPGA*, 2009, pp. 133–142. [Online]. Available: http://doi.acm.org/10.1145/1508128.1508150

[29] W. Zhao and Y. Cao, "New generation of predictive technology model for sub-45 nm early design exploration," *IEEE Trans. Electron Dev.*, vol. 53, no. 11, pp. 2816–2823, 2006.

[30] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Norwell, Massachusetts, 02061 USA: Kluwer Academic Publishers, 1999.

[31] A. Marquardt, V. Betz, and J. Rose, "Timing-driven placement for FPGAs," in *FPGA*, 2000, pp. 203–213.

[32] Altera, "Implementing PLL reconfiguration in stratix & stratix GX devices (an282)," https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/an/an282.pdf, 2005.

[33] *Virtex-5 FPGA Configuration User Guide*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, September 2008, UG191 <http://www.xilinx.com/bvdocs/userguides/ug191.pdf>.

[34] T. Austin, D. Blaauw, T. Mudge, and K. Flautner, "Making typical silicon matter with Razor," *IEEE Computer*, vol. 37, no. 3, pp. 57–65, March 2004.

[35] E. Kadric, K. Mahajan, and A. DeHon, "Energy reduction through differential reliability and lightweight checking," in *FCCM*, 2014.