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

HANS GIESEN, BENJAMIN GOJMAN, RAPHAEL RUBIN, JI KIM, and ANDRÉ DEHON,
University of Pennsylvania

We show that continuously monitoring on-chip delays at the LUT-to-LUT link level during operation allows a field-programmable gate array to detect and self-adapt to aging and environmental timing effects. 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 190ms for the toronto20 benchmarks. The result is a step toward systems where adaptive reconfiguration on the time-scale of seconds is viable and beneficial.

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**; **Online test and diagnostics**; **Process variations**;

Additional Key Words and Phrases: Measurement, algorithms, performance, reliability, aging, self-measure, component-specific mapping

ACM Reference format:
Hans Giesen, Benjamin Gojman, Raphael Rubin, Ji Kim, and André DeHon. 2018. Continuous Online Self-Monitoring Introspection Circuitry for Timing Repair by Incremental Partial-Reconguration (COSMIC TRIP). *ACM Trans. Reconfigurable Technol. Syst.* 11, 1, Article 3 (January 2018), 23 pages.
https://doi.org/10.1145/3158229

1 INTRODUCTION

The delay of individual circuit elements changes over time due to aging [43] and is also affected by environmental factors, such as local circuit temperature and supply voltage [49]. 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 [34]. For instance, Gojman measured sub-2ns

H. Giesen was supported in part by the Leggett Family Fellowship. This research was funded in part by DARPA/CMO Contract No. HR0011-13-C-0005. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

Authors' addresses: H. Giesen, R. Rubin, and A. DeHon, 200 S. 33rd Street, Philadelphia, PA 19104; emails: {giesen, rafi}@seas.upenn.edu, andre@acm.org; J. Kim, 364 Upson Hall, Ithaca, NY 14853, USA; email: ji@csl.cornell.edu; B. Goman, 111 8th Ave., Google, New York, NY 10011; email: bgojman@gmail.com.

ACM 1936-7406/2018/01-ART3 $15.00
https://doi.org/10.1145/3158229

delays on a 65nm FPGA for paths that Quartus estimated at over 3ns when accounting for composite worst-case conditions [17]. These high margins undermine the nominal benefits of scaled technology nodes [6].

Field-programmable gate arrays (FPGAs) can mitigate some of these effects using post-fabrication mapping [33]. By adapting the mapping to the fabricated delays of elements, we can largely eliminate margins for process variation. While recent work has demonstrated how to measure the delays of individual circuit elements [17], it is still necessary to margin for environmental factors (e.g., local self-heating) 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. Furthermore, 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 [22], to identify signal transition timing (Section 3) 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 (Section 4). 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 [38] to provide concrete timing repair options for our algorithm, responding to an aging event that slows down a circuit element (Section 10) 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) Architectural refinements to improve the effectiveness of repairs using CYA reserved tracks on modern Wilton-style switchbox wiring
(5) Characterization of the speed of repair identifying and replacing slow circuit elements that exceed their slack

A preliminary version of this work was presented as Reference [13]. The architectural enhancements described in Section 8 and the experiments reported in Section 10.3 are new for this expanded article. The repair algorithms were refined. All the result figures and tables have been revised to reflect the improvements from algorithms and architecture.

2 BACKGROUND

2.1 Aging

The small feature sizes of advanced process nodes mean more pronounced wear and aging effects that also change the delay of resources [43]. Time-dependent-dielectric-breakdown (TDDB) [37], Negative Bias Temperature Instability (NBTI) [39], Hot-Carrier Injection (HCI) [35], and electromigration [1] 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 performance or short lifetimes [9]. 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 [42], but many of the aging effects are stochastic, such that open-loop wear-leveling must still margin for worst-case effects.

2.2 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 [7, slide 44]? Accumulating 10^{10} errors over 10 years (3×10^8s) has a mean-time-between device failures on a chip of 30ms. Failures will not be evenly distributed in time. Nonetheless, this example 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 and substantial progress along that path. The CYA alternative selection we build upon has been demonstrated to handle 0.1–1% random defects [38], while full-knowledge mapping has been able to tolerate 1–10% [11, 33]. 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.

2.3 Timing Extraction

It is possible to perform on-chip measurements of the path delays on an FPGA with precisions down to 1–2ps using the programmable clock generators on modern FPGAs [44, 45]. Using a carefully chosen set of path measurements, Gojman was able to calculate the delays at the level of individual LUTs and interconnect segments in an FPGA down to <7ps [16, 17]. These can potentially be used with component-specific mapping [33] 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 Section 2.2.

2.4 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 [27, 49]. Consequently, even component-specific mapping based on timing extraction must margin for environmental, short-term aging, self-heating, and local voltage fluctuations.

2.5 DVFS

Dynamic voltage and frequency scaling (DVFS) can be used to compensate for the environment and aging at a coarse granularity. That is, with a suitable control loop that measures the actual delays for a circuit during operation and adjusts the frequency, a chip can run at the maximum frequency allowed by the particular chip at that particular point during its lifetime [3, 8, 21].

However, DVFS cannot change the resources used in the critical path to reduce its delay. This work goes beyond DVFS by replacing or repairing critical path circuitry.

3 DIFFERENCE DETECTOR WITH FIRST-FAIL LATCH

If we could put every signal on the chip on an oscilloscope, then we could determine their timing, including identifying which paths limit the operating clock frequency and which paths are operating slower than expected. Ideally, we would watch every transition of a signal to identify when the signal finally settled to its final value. 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. If the chain were long enough to cover the clock period, then we could sample the signal down to the resolution of the delay between successive registers (e.g., 17ps for one implemented on top of a Virtex-5 [12]). We identify the last time the signal transitions to its final value to identify its settling time.

Such a long sample register would be expensive, requiring a chain of 100 registers just to sample a single signal at 40ps intervals over a 4ns clock; the area of the 100 registers alone would be larger than the LUT and interconnect path that it monitored. We can approximate this monitor using a single register that samples at a single delay offset from the clock [22].[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. As long as we capture the final value of the signal at the end of the clock cycle, we can compare this to the sampled value for each delay (Difference Detector, DD) to 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 [22] 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 (Figure 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 Figure 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, then 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. If an FFL holds a zero, then none of the transitions failed for that offset during that set of test samples. If there were any failures, then the FFL will hold a one. By sweeping the delay across different offsets, we can identify the latest time that the signal transitions to its final value.

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 [40]. The Xilinx Ultrascale series has delay control on individual input pins down to 5–15ps [46]. Levine shows how to use the programmable clock controls on an Altera Cyclone III

[1] We use the early sample clock (M_CLK in Reference [22]) 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.

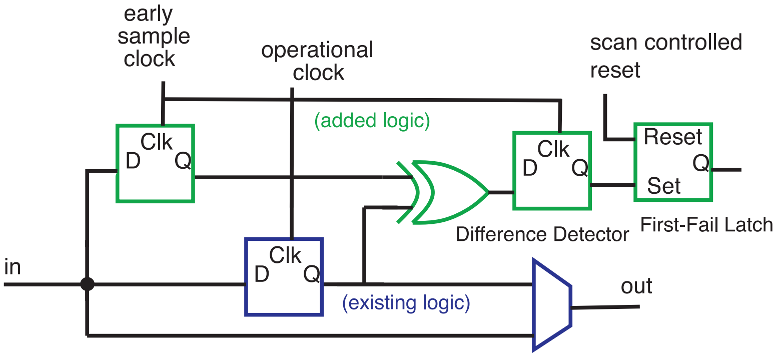


Fig. 1. Difference detector with first-fail latch.

to implement a programmably tunable delay line with 96ps of resolution on top of a conventional 65nm FPGA [22]. 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 40nm process) [15] and the VTR 7.0 [28] 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 Configurable Logic Block (CLB) [23] 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. It is possible that this solution could be approximated with lower resolution using an overlay technique such as the one from Levine [22] on newer register-rich FPGAs (e.g., Reference [25]), but that introduces additional challenges that we leave for future work.

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

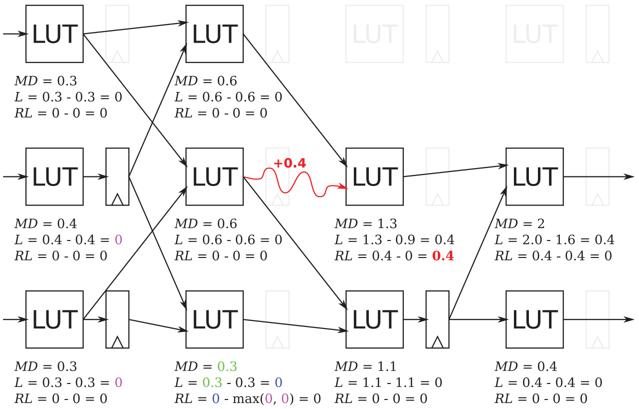
4 LINK TIMING

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 what resources are responsible for the lateness of the output. Any LUT or interconnect hop in the fan-in cone of the output could be 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, then that makes the link

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



The wavy line is the slow link.
Colored numbers illustrate how values are used in the calculation.

Fig. 2. Illustration of lateness calculation.

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.

5 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 use the circuit netlist and required times to 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 process for calculating lateness is illustrated in Figure 2. 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:

$$L_i = MD_i - RT_i. \tag{1}$$

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:

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

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

5.1 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.,

$$Slack_i = ALAP_i - ASAP_i. \tag{3}$$

If we compute ASAP and ALAP values entirely from nominal delay, then 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

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

where $Outputs(i)$ 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 j can be caused by both lateness of LUT i and increased propagation delay through LUT j. Hence, the lateness of LUT j is probably caused by the predecessor i with the highest $MD_i + ND_{i,j}$, 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, then

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

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

6 IDENTIFYING THE LATEST LUT

6.1 Brute-force Algorithm

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 (Algorithm 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 $RL_i - Slack_i$
```

6.2 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, reducing the number of offsets at which we must sample. Typically, this means high resolution measurements are only needed around the repair target, and other regions of the delay range will suffice with lower resolution measurements. 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 $[MD_i^l, MD_i^h]$:

$$L_i \in \left[L_i^l, L_i^h\right] = \left[MD_i^l - RT_i, MD_i^h - RT_i\right]. \tag{6}$$

The relative lateness in turn satisfies

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

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 (Algorithm 2). In the adaptive algorithm, we do not need to know the required precision to differentiate the latest LUT in advance; the algorithm effectively refines the precision as needed to disambiguate the latest LUT.

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 Figure 7, we normalize the performance of the adaptive algorithm to the brute-force algorithm, showing that the adaptive algorithm converges up to 4× faster than the brute-force algorithm.

6.3 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; in fact, there may be no bound on the number of cycles that guarantees an uncommon slow path is activated. If *Cycles* is too low, then 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, then the algorithm converges more slowly (which, for our experiments, leads to impractically long simulation times). If some paths are truly infrequent, or even never activated for some data sets, then 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, then 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 [32]. 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. If the register is loaded from a random primary input every cycle, then the registered input will look like any circuit input. However, with logic in front of the register, the data-processing inequality says that the register value will not be more random than the input [10]. Furthermore, if the register is only loaded on select cycles, its output will not change on every cycle, meaning the logic paths following the register may not be activated with a unique input value on every cycle.

6.4 Unexpected Transitions

With the adaptive algorithm, we can use internal self-consistency of the measurements to detect some cases where 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 Figure 3. We selected *Cycles* to use with each design based on Figure 3.

Table 1 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 identified a few features that might have an impact on cycles and used regression to attempt to fit a model for the cycles per offset. We might expect designs with more LUTs to require more cycles simply because there are more LUTs to sensitize. As noted above, designs with registers may result in portions of the design being activated or seeing key input combinations less frequently than others, demanding more cycles to provide sufficient samples. Deeper circuits have more opportunities for masking and varying path-length combinations that may lead to a need for more samples to sensitize all the delay paths. Also as noted above, to get a full set of transition cases, we may need to sample all of the input combinations to a logic cone, so we might expect the necessary samples to grow exponentially with the inputs in the recursive fanin of a gate or output. We separate out both total fan-in and registered fan-in, in case registered inputs present a larger challenge. We try to make a prediction $\hat{y}_i$ of the response, y_i, the number of cycles per offset for design i, with the formula

$$\log \hat{y}_i = LOG(\vec{x}_i) \cdot \vec{w}_{log} + \vec{x}_i \cdot \vec{w}_{lin} + EXP(\vec{x}_i) \cdot \vec{w}_{exp}. \tag{7}$$

Here, $\vec{x}_i$ is a vector with the five metrics of Table 1 for design i, each normalized to a standard deviation of 1. We introduced the logarithm on the left-hand side to ensure positive responses, and we compensate for that on the right-hand side by including the logarithm of each feature, $LOG(\vec{x}_i)$. To avoid logarithms of zero, we replace them with 1 before normalization. We conjectured the cycles

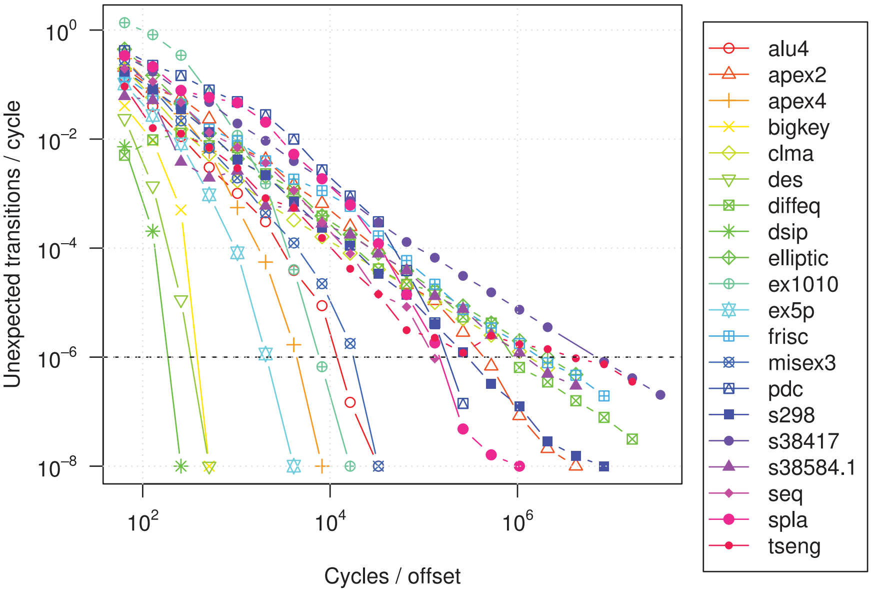


Fig. 3. Correlation between false-positive rate and cycles per offset.

Table 1. Toronto20 [4] Benchmark Design Characteristics

Design	LUTs	Flip-flops	Logic depth	Reg. fan-in	Total fan-in	Cycles / offset
alu4	824	0	6	0	14	15,578
apex2	971	0	7	0	36	486,091
apex4	793	0	6	0	9	5,763
bigkey	579	224	4	2	10	511
clma	3,223	33	11	274	282	1,482,013
des	557	0	4	0	19	489
diffeq	666	377	8	84	85	996,719
dsip	689	224	4	2	10	255
elliptic	1,816	1,122	10	325	326	2,077,654
ex1010	2,589	0	7	0	10	8,157
ex5p	578	0	5	0	8	2,301
frisc	1,744	886	12	121	124	1,871,608
misex3	768	0	6	0	14	38,330
pdc	2,205	0	7	0	16	234,751
s298	653	8	9	278	281	327,490
s38417	2,606	1,463	6	64	64	7,505,369
s38584.1	2,325	1,260	7	101	109	1,350,172
seq	867	0	6	0	38	130,544
spla	1,853	0	7	0	16	191,806
tseng	647	385	7	50	51	3,407,117
w_{log}	0.179	0.193	9.908	0.320	2.310	
w_{lin}	−0.590	−1.748	0.248	−3.068	−2.621	
w_{exp}	0.038	0.389	−0.017	0.133	0.227	

might have an exponential dependence on some of the design features, such as fan-in or depth, so we included linear terms, too. The exponential terms, $EXP(\vec{x}_i)$ further improve the prediction. We computed the regression coefficients, which we will collectively refer to as $\vec{w}$, using

$$\hat{w} = \arg\min_{\vec{w}} \sum_i |\vec{x}_i \cdot \vec{w}_{lin} + EXP(\vec{x}_i) \cdot \vec{w}_{exp} + LOG(\vec{x}_i) \cdot \vec{w}_{log} + w_0 - \log y_i|^2 + \lambda|\vec{w}|^2. \tag{8}$$

w_0 is a constant determining the intercept. This form of regression is known as Ridge regression [18]. The last term penalizes large $\vec{w}$ to prevent overfit. Overfitting is the phenomenon that regression coefficients accurately predict the provided features and responses (training data) but perform poorly on new input. The λ is obtained via cross validation, which estimates the true prediction error by splitting the available data into a set that is used solely for determining $\vec{w}$ and another set that for calculating the error. The average error obtained by repeating this process a number of times on different data set divisions provides a good estimate of the true prediction error. The obtained weights from performing our Ridge regression best fit are summarized at the bottom of Table 1 ($w_0 = 6.955$). Each column shows the associated term in the regression coefficient vectors. Logic depth has the largest coefficient, indicating that more cycles are needed when the circuit paths are long. Together, our linear regression achieves a correlation coefficient of 0.91, predicting the largest *Cycles* values within 70%. Given the 4-order of magnitude range for cycle requirements, this estimate is useful to understand roughly how many cycles to expect. More importantly, however, the regression coefficients provide insight into the dominant circuit characteristics that drive some circuits to need more sample cycles to get adequate timing estimates.

7 TIMING REPAIR BY INCREMENTAL PARTIAL-RECONFIGURATION (TRIP)

COSMIC TRIP can be used to reduce critical path delay to deal with timing faults.

7.1 Choose-Your-own-Adventure Routing

Timing Repair by Incremental Partial-reconfiguration (TRIP) employs CYA precomputed alternatives [38] 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. The CYA router runs once for a design. It generates a large number of alternate 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.

7.2 Algorithm

COSMIC TRIP'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 (Algorithm 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, then this will repair it relatively quickly. It is possible that multiple links will need to be repaired to actually accelerate the LUT. That is, multiple inputs may arrive at the LUT at the same time, within the resolution of the measurement timing interval; in this case, repairing one of the LUT inputs may deliver no speedup, since another LUT input may be equally slow. It is necessary to repair all of the simultaneously arriving late inputs to reduce the lateness of the LUT. 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. Since CYA stores and loads alternatives as bitstream edits to install or remove the alternative, the bitstream is only a factor of N larger than a standard bitstream [38], and CYA can compose the N^K variants for each LUT on the fly from the N alternate paths in the bitstream. The TRIP algorithm runs on a processor embedded in the FPGA fabric, such as an ARM core on a Zynq or Arria SOC-FPGA.

ALGORITHM 3: TRIP Algorithm

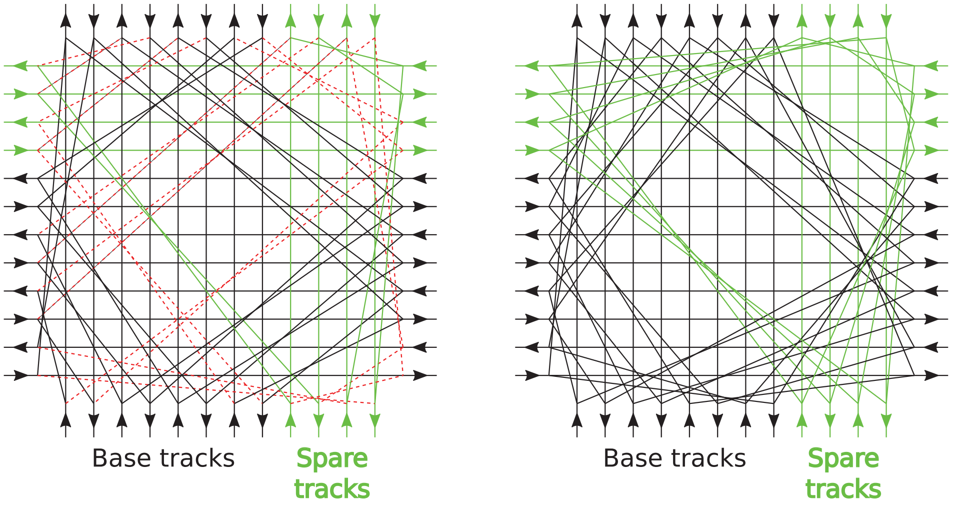
```
Locate slowest LUT$_j$ (Algorithm 2)
$t_l \leftarrow MD_j^l$
$BackOff \leftarrow 1.0$
Store current alternatives for inputs of LUT$_j$
repeat
    $T \leftarrow RT_j \times BackOff$
    $Attempt \leftarrow 0$
    repeat
        Replace random input of LUT$_j$ with alternative
        Reset $MD$ of LUTs affected by repair
        Run $Cycles$ with $EarlySampleOffset = T$
        Update $MD$ intervals.
        $t \leftarrow MD_j^h$
        $Attempt \leftarrow Attempt + 1$
        Restore alternatives of LUT$_j$ if $t \geq t_l$
    until $t \leq T$ or $Attempt > Max\_attempts$
    if $t > T$ then
        $BackOff = BackOff \times BackOffFactor$
until $t \leq T$
```

7.3 Memory Requirements for Repair

We must represent the circuit netlist (Section 5) and the delay state of the LUTs within the graph to support the TRIP algorithm. We represent the netlist by specifying each of the K predecessors to each LUT. For an N_{lut}-node design, this means $K \cdot N_{lut}$ numbers. If we use a 32b pointer (4B) for each predecessor, then 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 1ps resolution for up to 64ns of delay; and 1B would support up to 256ps. If we use 1B for D and ND and 2B for the rest, then 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.

8 ARCHITECTURE REFINEMENT

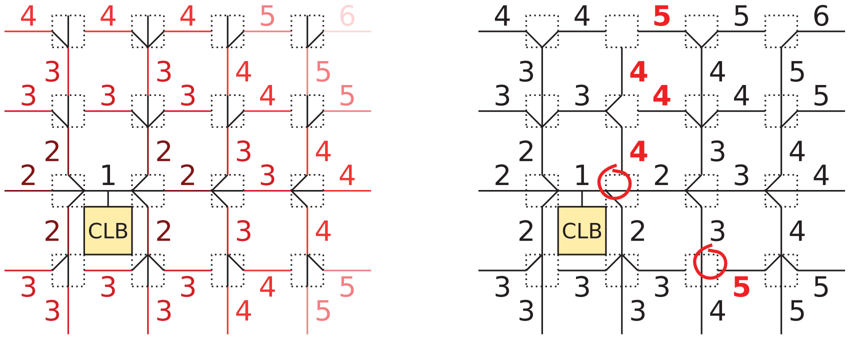
The original CYA design was developed for a bidirectional architecture in the classic Island-Style FPGA model [5, 36] using a subset switch [38]. On the *subset* switch, wiring tracks live in disjoint domains. Consequently, it was reasonable to simply reserve tracks (Section 7.1) for use as



(a) Original Wilton S-Box (b) Revised Wilton S-Box

Green links show Reserved Spare Tracks; Red, dashed links show switch connections that become unusable, and, hence, represent dead ends, when we partition Base and Reserved Tracks.

Fig. 4. Revised switchbox topology.

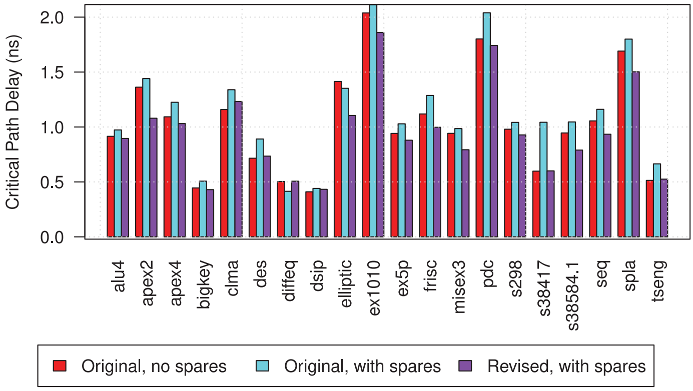


(a) No dead-ends (b) With dead-ends

Bold red numbers in the (b) figure highlight delays that are larger than in the (a) figure due to the circled dead-end connections.

Fig. 5. Effect of dead-ends in S-box on delay.

alternatives without interfering with the connectivity of the base tracks. Bidirectional routing architectures allowed a track to route signals in any direction, originally using simple pass transistors for switch points. However, modern architectures buffer routing signals and now use single-driver, directional-drive routing [20, 24], where wires are driven in a single direction by a dedicated drive buffer. Modern architectures also mostly use the Wilton-style switchboxes [31], where domains are not disjoint. The initial routing cannot traverse reserved tracks. Moreover, base tracks are often occupied when alternatives are routed. Hence, switches crossing over between base and reserved tracks become effectively unusable (see Figure 4(a)). The result of these unusable switches is that many paths run into *dead-ends* (see Figure 5(b)). That is, even when there is no congestion, at some switchboxes, there are many cases where there are no switch connections that will allow a route to



(a) Critical Path Delay

Design	Channel Width Change
apex2	-4
apex4	-8
clma	4
dsip	-4
elliptic	-4
ex5p	-4
misex3	-4
s38417	-2
Others	0

(b) Minimum Channel Width

Fig. 6. Impact of split switchbox revision on base routes at $V_{dd} = 0.8$V.

be extended in a particular direction, such that the switchbox becomes a dead-end for a particular route. This reduction in connectivity leads to paths that must traverse more segments compared to architectures without CYA (see Figure 5). This means the original CYA separation strategy does not work as well on the Wilton-switchbox designs.

Working with the Wilton-switchbox, it is more useful to architecturally split the base route tracks from the reserved route tracks. Here, we provide Wilton-switchbox connectivity, but only within each of the two disjoint sets of tracks (see Figure 4(b)). This allows all connections to remain within each set, avoiding the dead ends (see Figure 5).

Expanding on the original CYA, we add spare inputs and spare LUTs, with their own pins, to each CLB to tolerate failures or slowdowns in inputs, outputs, and LUTs. We connect the spare pins to the reserved track domain and the original pins to the base track domain. With the base and reserved track split, the cross-connections would mostly go unused. Splitting these CLB connections saves area and avoids adding excessive capacitance to the original or spare resources.

As a result of removing the dead-end connections, we are able to both route the base routes in fewer channels (Figure 6(b)) and reduce the delay for the base routes (see Revised versus Original comparison for the cases with spares in Figure 6(a)). The comparison to the original case without spares shows that adding spares to the original case does slow down the design due to the dead-end connections and the additional area for the spares. However, the revised architecture with spares but no dead-ends generally results in lower net delays than the original design without spares.

If we want to have spare tracks that can provide connections that are as short as the base paths, then we must arrange for the reserved track set to include paths with the minimum number of segments. For example, segment staggering means that if we only add a directional pair of reserved tracks, then they will be at a single, specific staggered offset and not aligned to provide connectivity with the minimum number of segments. Consequently, we will need more reserved tracks to provide all spare tracks at all stagger offsets. Similarly, connection-box depopulation and the number of sides each input or output appears on will also impact the minimum number of reserved tracks necessary to guarantee that segment-minimizing routes on reserved tracks between all source and destination CLBs will exist. Typically, the connection-box depopulation is characterized by F_C, the fraction of wires in each channel to which each CLB input or output is connected [36]. This can be further refined to specify input connectivity, $F_{C_{in}}$, independently from output connectivity, $F_{C_{out}}$.

Table 2. Fast Alternatives Architectures Parameters

Guaranteed fast alternatives	1
Regular input pins	27
Regular output pins	8
$F_{C_{in}}$	0.15
$F_{C_{out}}$	0.1
Segment length	4
Extra input pins (I_s)	16
Extra output pins (O_s)	4
$F_{C_{in,extra}}$	0.25
$F_{C_{out,extra}}$	0.1
Spare tracks (W_s)	16
Overhead Area Sparing (% base)	18.8

In either case, the number of connections to a channel with W tracks for a given input or output is $W \times F_C$. Using $F_{C_{out}} = 0.1$ and $F_{C_{in}} = 0.15$, 16 spare tracks are sufficient to guarantee at least one route in the reserved tracks has minimum segment length if we use 16 spare inputs, 4 spare outputs, and connection-box connectivity for the spare inputs of $F_{C_{in}} = 0.25$ and the spare outputs of $F_{C_{out}} = 0.1$. This level of sparing spends about 20% area overhead for the benchmark set (see Table 2).

9 METHODOLOGY

We augmented a version of VPR 5.0.2 [29] to model independent transistor variation and CYA alternatives. We use a Predictive Technology Mode (PTM) [48] for the base technology. We assume a 22-nm CMOS process with $\mu_{V_{th}} = 400\text{mV}$, $\sigma_{V_{th}} = 36\text{mV}$, and typical operating $V_{dd} = 0.8\text{V}$. 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 (Section 2.1). HSPICE simulations provide the delay model for each buffer in the interconnect.

9.1 Experimental Architecture

We use an island-style architecture [5] with 6-input LUTs ($K = 6$) and 12 LUTs per cluster ($N = 12$) and a segment length of 4, similar to a Stratix-IV [23]. Routing is directional with a revised Wilton-style S-box (Section 8) and $F_{C_{in}} = 0.15$, $F_{C_{out}} = 0.1$ C-box connectivity for base tracks and $F_{C_{in}} = 0.25$, $F_{C_{out}} = 0.1$ for reserved tracks.

Clusters have 43 inputs. To support CYA, we reserve 4 LUTs, 16 inputs, and 16 tracks 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 [30]. Base routes are mapped with full-knowledge of delays [33], providing a high-quality mapping as our repair target. Alternatives are generated based on nominal delays for the reserved resources. We generate 64 alternate routes for each two-point net in the original design.

9.2 Dynamic Timing Simulation

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

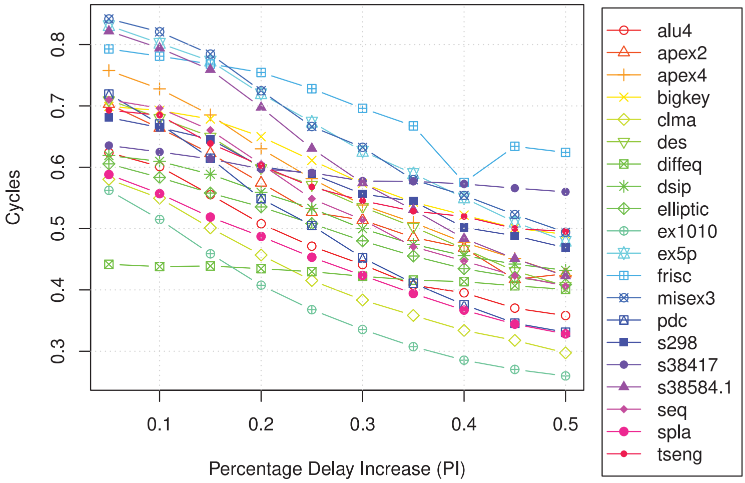


Fig. 7. Aging experiment: cycles dependency on delay increase.

detector, we developed a custom simulator that tracked 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 worked on path delays from our modified VPR and had the ability to revise the path delays based on the selected CYA alternatives.

9.3 Aging Delays

The key issue in aging is delays that exceeded 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 added delay to a link, we added the sum of the slack for the link, $Slack_{i,j}$, and the added delay, d_{add}, for a total of $Slack_{i,j} + d_{add}$.

10 AGING REPAIR

We performed three sets of experiments, one to specifically show how time-to-repair depends on the magnitude of the aging slowdown (Section 10.1), one to simulate a potential distribution of aging events (Section 10.2), and one to characterize the impact of the revised switchbox architecture (Section 10.3).

10.1 Delay Magnitude Experiment

In the first set, we added 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 allowed 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 picked 100 random nodes and injected 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 Figure 7, normalized to cycles required with a brute-force algorithm for $PI = 20\%$. Figure 7 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.

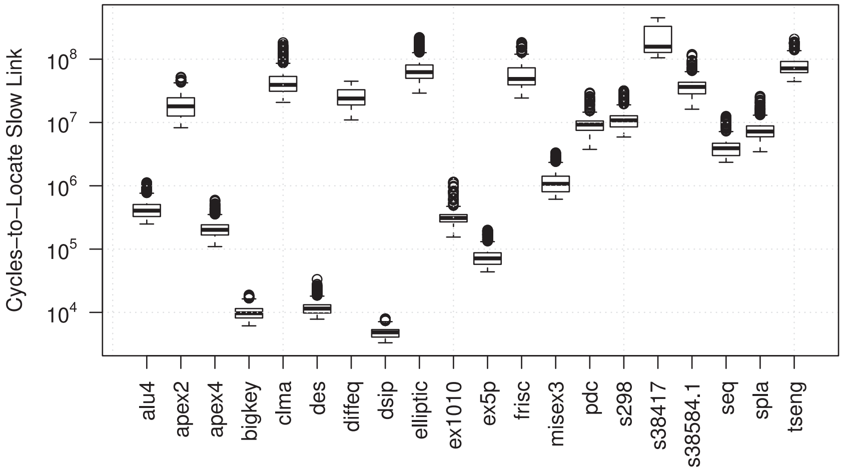


Fig. 8. Aging experiment: cycles-to-locate slowest resource.

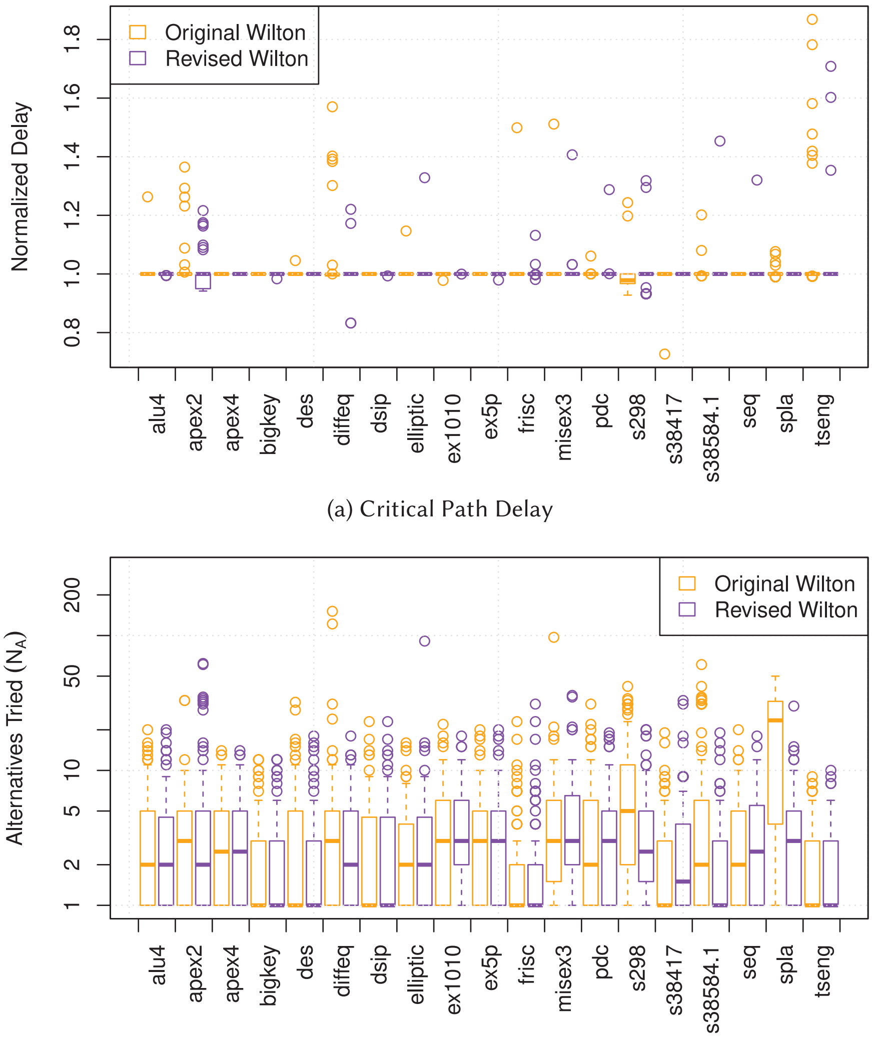
10.2 Random Delay Insertion Experiment

In the second experiment, we also inserted delays at random LUT-to-LUT links in the circuit. Here, we both picked a random node and a random delay percentage for d_{add}. The absolute value for d_{add} was sampled from a Gaussian distribution with $\mu = d_{nominal}$ and $\sigma = 0.3 \cdot d_{nominal}$. Figure 8 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. We use boxplots to characterize the distribution of the 5 × 100 aged-delay insertion experiments; the thick line marks the median, and the box captures the two quartiles on either side of the median, with the circles denoting the outliers.

Since the repair algorithm is unaware of the speed of input links and alternatives, it may need to try many alternatives (Section 7) to repair the aged link. The average number of repairs, N_a, required to restore timing after the slow LUT is identified is shown in Table 4 with 5 × 100 injected delays. The designs required on average 4.8 repair trials, suggesting that 0.81 alternatives are evaluated for each of the 6 inputs before finding an available and adequate performance alternative.

10.3 Architecture Impact

To characterize the impact of the architectural refinements from Section 8 on aging repair, we reran the random delay insertion experiment from Section 10.2 with both the original and revised architecture for 100 delay injections for a single "chip." Figure 9(a) shows the impact of the architecture refinement on the critical path delay after repair, normalized to the delay before injection. In the vast majority of cases, both architectures restore the delay to its original value. Most random injections are off the original critical path. Since the aged links have slack, even repairs that are slower than the original are typically good enough to reduce the delay below the original critical path. The outliers typically arise when the delays are injected into the critical path. Their lack of slack demands higher quality alternatives to restore the circuit delay to its original value. We see that our algorithm typically reduces those delays more effectively with the new architecture. The number of alternatives that is installed to resolve each aging event is shown in Figure 9(b). Both the medians and most extreme outliers were reduced slightly with the revised architecture.



(a) Critical Path Delay

(b) Alternatives Tried

Fig. 9. Impact of switchbox revision on repair quality and alternatives explored.

Using 64 alternatives and many trials, both the original and revised architecture were able to repair almost all delay injection cases to their original delay. Figure 10 shows how the revised architecture is able to perform equally well with fewer alternatives and fewer repair trials.

10.4 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:

$$T_{repair} = (N_{off} + N_a)(T_{coff} + Cycles \cdot T_{age} + N_{lut}T_{bit})N_{off} \cdot T_{nxt} + N_a \cdot 2L_pT_{frm}. \tag{9}$$

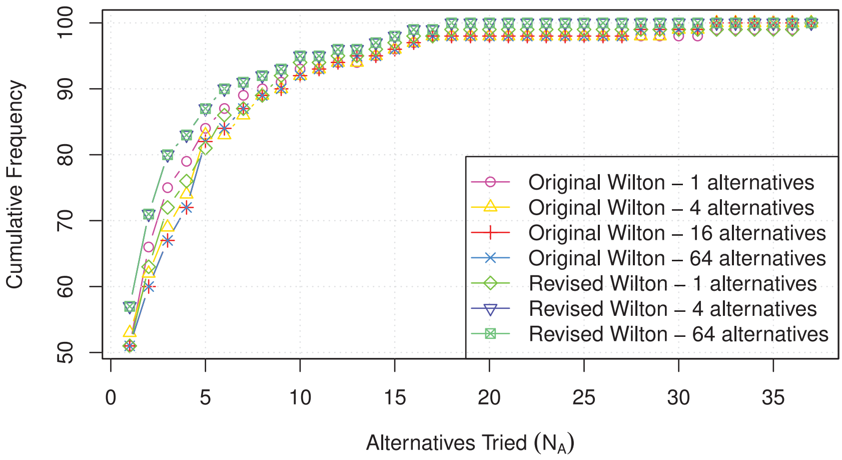


Fig. 10. Impact of switchbox revision on alternative usage for des.

Table 3. 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 [2]	20 μs
Cycles	Cycles per iteration	Table 1
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 [38]	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 [38]	1.3 μs

Table 3 describes the variables and the technology parameters we use in estimation. This assumes a frame-oriented configuration model similar to the Virtex series [47], 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. Table 4 shows the time to repair. From Figure 11, it is clear that the time running samples (next to last column in Table 4, $(N_{off} + N_a) \cdot Cycles \cdot T_{age})$ and the time computing the next offset $(N_{off} \cdot T_{nxt})$ dominate the other components of the total repair time. For designs with small numbers for *Cycles*, the time for configuring the sample clock can also be significant.

We capture T_{nxt} from a Java implementation of Algorithm 1 and Algorithm 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 saw that the total algorithm computation time $(N_{off} \cdot T_{nxt})$ is less than 12ms and only dominates in cases where $Cycles < 10^5$. Even if it ran 30× slower, total repair time would still be less than 400ms in the worst-case.

Table 4 also illustrates the benefits of the revised switchbox architecture and spare channel provisioning. The average repair path length, L_p, is roughly half of the value reported in Reference [13], and the worst-case repair time drops from 300 to 190ms.

Table 4. Comparison of Time-to-repair

						Time (ms)	
Design	N_{off}	$T_{age}(ns)$	T_{nxt} (μs)	N_a	L_p	**Samples**	T_{repair}
alu4	29	1.16	26	11.1	6.2	0.52	2.5
apex2	41	1.35	40	13.5	7.2	26.78	38.8
apex4	38	1.33	27	3.7	6.1	0.29	2.3
bigkey	20	0.62	41	2.7	5.7	0.01	1.3
clma	34	1.49	80	3.7	6.8	73.79	85.6
des	25	0.98	30	3.3	6.2	0.01	1.4
diffeq	25	0.67	60	3.7	5.8	16.56	21.2
dsip	19	0.63	41	3.4	6.0	0.00	1.3
elliptic	35	1.45	329	3.4	6.5	105.70	128.5
ex1010	40	2.30	41	4.2	7.3	0.76	3.6
ex5p	35	1.07	21	3.7	5.9	0.09	1.7
frisc	36	1.24	158	2.7	6.5	81.35	93.6
misex3	32	1.08	30	4.4	6.1	1.33	3.3
pdc	41	2.23	73	4.5	7.3	21.22	27.7
s298	36	1.23	32	5.9	6.2	14.43	19.0
s38417	30	0.71	590	2.9	6.6	150.07	182.9
s38584.1	29	1.04	227	3.8	6.6	40.34	52.9
seq	33	1.20	33	4.0	6.0	5.13	7.7
spla	42	1.86	55	8.5	7.1	14.88	21.5
tseng	24	0.73	60	2.4	5.2	59.98	67.9

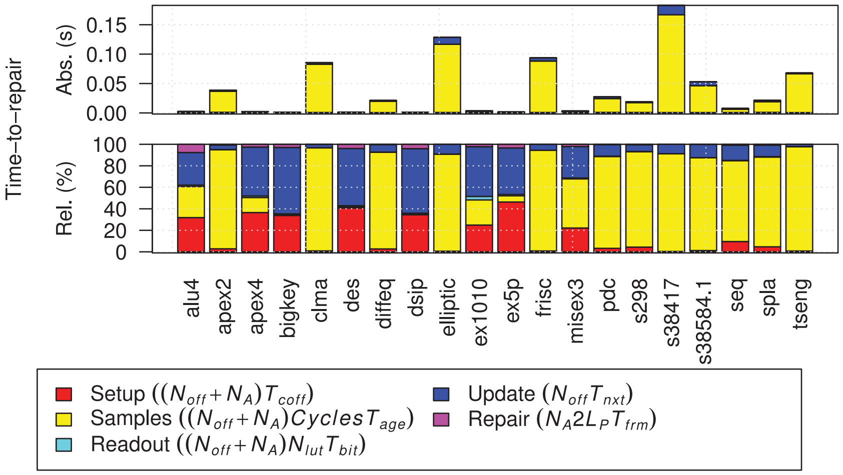


Fig. 11. Breakdown of execution time.

While these are small designs and the repair time is still large for the Borkar challenge (Section 2.2), 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.

11 DISCUSSION

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. Implementations will need to employ a technique for detecting timing-delay faults (e.g., Reference [3]) or circuit logic errors (e.g., Reference [19]) 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.

12 FUTURE WORK

Our primary assumption for this article has been that we do not get to control the input values to the circuit—the circuit is simply exercised as a side-effect of normal operation. If we were able to use a test sequence designed for delay fault testing [26], then we could remove the uncertainty about whether or not the slow paths were sensitized. In general, we should be able to run much shorter tests and more quickly diagnose and repair slow links. This could be natural for applications with periodic downtime, or, simply with real-time constraints that the FPGA can exceed (e.g., between image frames when performing real-time image capture or playback) [41].

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 [27]. 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 [14]. We expect that there is room to optimize beyond the simple algorithms we employ, accelerating localization and repair and consuming alternatives more judiciously.

13 CONCLUSIONS

Fine-grained, continuous monitoring of the delay of signals on an FPGA during operation can be very lightweight. This monitoring provides information that 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] Syed M. Alam, Gan Chee Lip, Carl V. Thompson, and Donald E. Troxel. 2004. Circuit level reliability analysis of Cu interconnects. In *Proceedings of the International Symposium on Quality Electronic Design*. 238–243.

[2] Altera. 2005. Implementing PLL Reconfiguration in Stratix & Stratix GX Devices (AN282). Retrieved from https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/an/an282.pdf.

[3] Todd Austin, David Blaauw, Trevor Mudge, and Krisztián Flautner. 2004. Making typical silicon matter with razor. *IEEE Comput.* 37, 3 (March 2004), 57–65.

[4] Vaughn Betz and Jonathan Rose. 1999. FPGA Place-and-Route Challenge. Retrieved from http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html.

[5] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. 1999. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA.

[6] David Bol, Renaud Ambroise, Denis Flandre, and Jean-Didier Legat. 2009. Interests and limitations of technology scaling for subthreshold logic. *IEEE Trans. VLSI Syst.* 17, 10 (2009), 1508–1519.

[7] Shekhar Borkar. 2004. Microarchitecture and Design Challenges for Gigascale Integration. Retrieved from http://www.microarch.org/micro37/presentations/MICRO37%20Sborkar.pdf. Keynote talk of the *International Symposium on Microarchitecture*.

[8] C. T. Chow, L. S. M. Tsui, Philip H. W. Leong, Wayne Luk, and Steve J. E. Wilton. 2005. Dynamic voltage scaling for commercial FPGAs. In *Proceedings of the International Conference on Field-Programmable Technology*. 173–180.

[9] Lloyd Condra, J. Qin, and Joseph B. Bernstein. 2007. State of the art semiconductor devices in future aerospace systems. In *Proceedings of the FAA/NASA/DoD Joint Council on Aging Aircraft Conf.*

[10] Thomas Cover and Joy Thomas. 1991. *Elements of Information Theory*. John Wiley and Sons, Inc., New York.

[11] André DeHon and Nikil Mehta. 2013. Exploiting partially defective LUTs: Why you don't need perfect fabrication. In *Proceedings of the International Conference on Field-Programmable Technology*. 12–19

[12] Claudio Favi and Edoardo Charbon. 2009. A 17ps time-to-digital converter implemented in 65nm FPGA technology. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 113–120. Retrieved from DOI: https://doi.org/10.1145/1508128.1508145

[13] Hans Giesen, Benjamin Gojman, Raphael Rubin, and André DeHon. 2016. Continuous online self-monitoring introspection circuitry for timing repair by incremental partial-reconfiguration (COSMIC TRIP). In *Proceedings of the FCCM*. 111–118.

[14] Hans Giesen, Raphael Rubin, Benjamin Gojman, and André DeHon. 2017. Quality-time tradeoffs in component-specific mapping: How to train your dynamically reconfigurable array of gates with outrageous network-delays. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 85–94. DOI: https://doi.org/10.1145/3020078.3026124

[15] Jeffrey B. Goeders and Steven J. E. Wilton. 2012. VersaPower: Power estimation for diverse FPGA architectures. In *Proceedings of the International Conference on Field-Programmable Technology*. 229–234. DOI: https://doi.org/10.1109/FPT.2012.6412139

[16] Benjamin Gojman and André DeHon. 2014. GROK-INT: Generating real on-chip knowledge for interconnect delays using timing extraction. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. 88–95.

[17] Benjamin Gojman, Sirisha Nalmela, Nikil Mehta, Nicholas Howarth, and André DeHon. 2014. GROK-LAB: Generating real on-chip knowledge for intra-cluster delays using timing extraction. *ACM Trans. Reconfig. Tech. Syst.* 7, 4, Article 5 (Dec. 2014), 23 pages. DOI: https://doi.org/10.1145/2597889

[18] Arthur E. Hoerl and Robert W. Kennard. 1970. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics* 12, 1 (Feb. 1970), 55–67.

[19] Edin Kadric, Kunal Mahajan, and André DeHon. 2014. Energy reduction through differential reliability and lightweight checking. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. 243–250

[20] Guy Lemieux, Edmund Lee, Marvin Tom, and Anthony Yu. 2004. Directional and single-driver wires in FPGA interconnect. In *Proceedings of the International Conference on Field-Programmable Technology*. 41–48. DOI: https://doi.org/10.1109/FPT.2004.1393249

[21] Joshua M. Levine, Edward Stott, and Peter Y. K. Cheung. 2014. Dynamic voltage & frequency scaling with online slack measurement. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 65–74. DOI: https://doi.org/10.1145/2554688.2554784

[22] Joshua M. Levine, Edward Stott, George A. Constantinides, and Peter Y. K. Cheung. 2012. Online measurement of timing in circuits: For health monitoring and dynamic voltage & frequency scaling. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. 109–116.

[23] David Lewis, Elias Ahmed, David Cashman, Tim Vanderhoek, Chris Lane, Andy Lee, and Philip Pan. 2009. Architectural enhancements in Stratix-III and Stratix-IV. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 33–42. DOI: https://doi.org/10.1145/1508128.1508135

[24] David Lewis, Vaughn Betz, David Jefferson, Andy Lee, Chris Lane, Paul Leventis, Sandy Marquardt, Cameron McClintock, Bruce Pedersen, Giles Powell, Srinivas Reddy, Chris Wysocki, Richard Cliff, and Jonathan Rose. 2003. The stratix routing and logic architecture. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 12–20. DOI: https://doi.org/10.1145/611817.611821

[25] David Lewis, David Cashman, Mark Chan, Jeffery Chromczak, Gary Lai, Andy Lee, Tim Vanderhoek, and Haiming Yu. 2013. Architectural enhancements in Stratix V. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 147–156. DOI: https://doi.org/10.1145/2435264.2435292

[26] Chin Jen Lin and S. M. Reddy. 1987. On delay fault testing in logic circuits. *IEEE Trans. Comput.-Aid. Des.* 6, 5 (September 1987), 694–703. DOI: https://doi.org/10.1109/TCAD.1987.1270315

[27] Timothy A. Linscott, Benjamin Gojman, Raphael Rubin, and André DeHon. 2016. Pitfalls and tradeoffs in simultaneous, on-chip FPGA delay measurement. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 100–104. DOI: https://doi.org/10.1145/2847263.2847334

[28] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. 2014. VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Trans. Reconfig. Tech. Syst.* 7, 2 (July 2014), 6:1–6:30. DOI: https://doi.org/10.1145/2617593

[29] Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, Wei Mark Fang, and Jonathan Rose. 2009. VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling. In *Proceedings of the FPGA*. 133–142. DOI: https://doi.org/10.1145/1508128.1508150

[30] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. 2000. Timing-driven placement for FPGAs. In *Proceedings of the FPGA*. 203–213.

[31] M. Imran Masud and Steve Wilton. 1999. A new switch block for segmented FPGAs. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*. 274–281.

[32] F. G. Maunsell. 1937. A problem in cartophily. *Math. Gazette* 22 (1937), 328–331.

[33] Nikil Mehta, Raphael Rubin, and André DeHon. 2012. Limit Study of Energy & Delay Benefits of Component-Specific Routing. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 97–106. DOI: https://doi.org/10.1145/2145694.2145710

[34] Evelyn Mintarno, Joëlle Skaf, Rui Zheng, Jyothi Velamela, Yu Cao, Stephen Boyd, Robert Dutton, and Subhasish Mitra. 2011. Self-tuning for maximized lifetime energy-efficiency in the presence of circuit aging. *IEEE Trans. Comput.-Aid. Des.* 30, 5 (May 2011), 760–773.

[35] Shing-Hwa Renn, Christine Raynaud, Jean-Luc Pelloie, and Francis Balestra. 1998. 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. Dev.* 45, 10 (October 1998), 2146–2152.

[36] Jonathan Rose and Stephen Brown. 1991. Flexibility of interconnection structures for field-programmable gate arrays. *IEEE J. Solid-State Circ.* 26, 3 (March 1991), 277–282.

[37] Elyse Rosenbaum, Peter M. Lee, Reza Moazzami, P. K. Ko, and Chenming Hu. 1989. Circuit reliability simulator-oxide breakdown module. In *Technical Digest of the IEEE International Electron Device Meeting*. 331–334.

[38] Raphael Rubin and André DeHon. 2011. Choose-Your-Own-Adventure Routing: Lightweight Load-Time Defect Avoidance. *ACM Trans. Reconfig. Tech. Syst.* 4, 4 (December 2011), 33:1–33:24. Retrieved from DOI: https://doi.org/10.1145/2068716.2068719

[39] Dieter K. Schroder and Jeff A. Babcock. 2003. Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing. *J. Appl. Phys.* 94, 1 (July 2003), 1–18.

[40] Stefanos Sidiropoulos and Mark A. Horowitz. 1997. A semidigital dual delay-locked loop. *IEEE J. Solid-State Circ.* 32, 11 (Nov 1997), 1683–1692. DOI: https://doi.org/10.1109/4.641688

[41] Steven K. Sinha, Peter M. Kamarchik, and Seth Copen Goldstein. 2000. Tunable fault tolerance for runtime reconfigurable architectures. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. 185–192.

[42] Suresh Srinivasan, Ramakrishnan Krishnan, Prasanth Mangalagiri, Yuan Xie, Vijaykrishnan Narayanan, Mary Jane Irwin, and Karthik Sarpatwari. 2008. Toward increasing FPGA lifetime. *IEEE Trans. Dep. Secure Comput.* 5, 2 (April 2008), 115–127. DOI: https://doi.org/10.1109/TDSC.2007.70235

[43] Edward A. Stott, Justin S. J. Wong, Pete Sedcole, and Peter Y. K. Cheung. 2010. Degradation in FPGAs: Measurement and modelling. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 229–238.

[44] Tim Tuan, Austin Lesea, Chris Kingsley, and Steven Trimberger. 2011. Analysis of within-die process variation in 65nm FPGAs. In *Proceedings of the International Symposium on Quality Electronic Design*. 1–5. Retrieved from DOI: https://doi.org/10.1109/ISQED.2011.5770808

[45] Justin S. Wong, Pete Sedcole, and Peter Y. K. Cheung. 2009. Self-measurement of combinatorial circuit delays in FPGAs. *ACM Trans. Reconfig. Tech. Syst.* 2, 2 (June 2009), 1–22. DOI: http://doi.acm.org/10.1145/1534916.1534920

[46] Xilinx. 2015. UltraScale Architecture and Product Overview (DS890). Retrieved from http://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.

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

[48] Wei Zhao and Yu Cao. 2006. New generation of predictive technology model for sub-45 nm early design exploration. *IEEE Trans. Electron. Dev.* 53, 11 (2006), 2816–2823.

[49] Ken M. Zick and John P. Hayes. 2010. On-line sensing for healthier FPGA systems. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 239–248.

Received November 2016; accepted October 2017