Quality-Time Tradeoffs in Component-Specific Mapping:

How to Train Your Dynamically Reconfigurable Array of Gates with Outrageous Network-delays

Hans Giesen
giesen@seas.upenn.edu

Raphael Rubin
rafi@seas.upenn.edu

Benjamin Gojman
bgojman@acm.org

André DeHon
andre@acm.org

Department of Electrical and Systems Engineering
University of Pennsylvania, 200 S. 33rd St., Philadelphia, PA 19104

ABSTRACT

How should we perform component-specific adaptation for FPGAs? Prior work has demonstrated that the negative effects of variation can be largely mitigated using complete knowledge of device characteristics and full per-FPGA CAD flow. However, the cost of per-FPGA characterization and mapping could be prohibitively expensive. We explore lightweight options for per-FPGA mapping that avoid the need for *a priori* device characterization and perform less expensive per FPGA customization work. We characterize the tradeoff between Quality-of-Results (energy, delay) and per-device mapping costs for 7 design points ranging from complete mapping based on knowledge to no per-device mapping. We show that it is possible to get 48–77% of the component-specific mapping delay benefit or 57% of the energy benefit with a mapping that takes less than 20 seconds per FPGA. An incremental solution can start execution after a 21 ms bitstream load and converge to 77% delay benefit after 18 seconds of runtime.

Keywords

FPGA; Variation; Component-Specific Mapping

1. INTRODUCTION

Process variation is large in today's CMOS technology and continues to grow as feature sizes scale down. At minimum feature size, this results in nominally identical devices that have a large range of threshold voltages and hence operating delays. As a result, we are forced to use large, non-minimum feature sizes, at the expense of higher capacitance, and to use high operating voltages that lead to greater dynamic and leakage energy to accommodate the worst-case fabricated devices on today's multi-billion transistor integrated circuits. Consequently, we pay a large energy penalty for variation that threatens to increase the energy used per LUT evaluation as we scale to smaller feature sizes and undermine the traditional benefits of feature-size scaling.

FPGAs, unlike ASICs, can mitigate the impact of variation by assigning functions to resources **after** fabrication, when process variation has already occurred. Resources that use transistors at the extreme tails of the device characteristic distribution can be avoided. Slow resources can be assigned off the critical path. This allows resources to use smaller transistors and operate at lower voltages. Full mapping benefits can reduce energy by 2-3$\times$ and allow the continued reduction of energy at smaller feature sizes [26].

Full component-specific mapping requires both an extensive per-chip resource characterization phase [14] and per-chip mapping phase [26] in contrast to the conventional one-mapping-fits-all (OMFA) model that performs mapping only once to be used across any number of specific FPGA components. When full characterization may take days and mapping times run into hours, this cost can be prohibitive.

Lightweight repair schemes that precompute alternate resources and select among them [19, 16, 28] provide more practical schemes for pure defect tolerance. By performing a single precomputation of alternative resources, this reduces the per-FPGA mapping time down to seconds. This basic idea can be applied to variation: *identify the slow paths that limit operation and replace them with faster, precomputed alternatives.* In this paper, we identify and explore a range of algorithms for timing repair exploiting these precomputed alternatives and characterize their costs and benefits.

We first review variation and component-specific mapping (Sec. 2) and precomputed alternatives (Sec. 3). We tune the routing architecture (Sec. 4) to the variation mitigation problem, and then describe the mapping algorithms (Sec. 5). Sec. 6 describes our methodology, Sec. 7 present our experimental results, and Sec. 8 discusses their implications.

Our novel contributions include:

- Show how to adapt lightweight, load-time route alternative selection to address variation.
- Show how to adapt incremental, in-system repair to address variation.
- Characterize time (runtime, measure) and quality (delay, energy) achievable across 7 design points between OMFA and full CAD, perfect-knowledge mapping.

FPGA '17, February 22 - 24, 2017, Monterey, CA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4354-1/17/02...$15.00
DOI: http://dx.doi.org/10.1145/3020078.3026124

2. BACKGROUND

2.1 Process Variation

Even identically drawn transistors in a modern VLSI technology will differ from each other [4, 30]. Nominal critical dimensions now measure in single or double-digit nanometers, meaning the presence or absence of individual atoms has a significant impact on performance. Phase-shift masking, etching, and lensing effects result in approximate feature definition [3]. Differences in local oxide thickness [2], random dopant fluctuation [1], and stochastic dopant placement provide a strong random component to the composition of each transistor. As a result, key characteristics, such as the threshold voltage (V_{th}), vary widely from chip-to-chip and from device-to-device within a single chip.

These effects directly impact the delay of each resource in the FPGA [32]. In modern, short-channel, velocity-saturated transistors operating above threshold, delay scales linearly in $V_{dd} - V_{th}$, such that devices slow down as V_{th} increases. On a 65 nm FPGA, Gojman measures a spread of over 100 ps for nominally identical paths within a LAB [14] and nominally identical interconnect segments [13], and these spreads increase as features sizes shrink.

Variation also limits our ability to scale down operating voltage, resulting in higher dynamic energy [7]. Since delay above threshold is proportional to $V_{dd} - V_{th}$, we are forced to increase V_{dd} to offset high V_{th} values, increasing the dynamic operating energy that scales as $C\,(V_{dd})^2$. Design functionality depends on all used devices being able to switch, and design delay is determined by the slowest of numerous parallel paths. With random V_{th} variation, this means we sample the V_{th}-distribution millions-to-billions of times on today's FPGAs, and the worst device among those sampled will be the limiting V_{th} for operation.

One way to combat variation is to increase device size, which increases energy through increased switching capacitance. For random dopant variation, variance scales roughly as $\frac{1}{\sqrt{L \cdot W}}$, where L and W are the length and width of the transistor [18]. By scaling up device widths, we reduce the relative effects of variation. However, this also directly increases the device capacitance and indirectly increases the wire capacitance by making the chip larger and hence wires longer. This reverse scaling of device size increases the C in the dynamic operating energy $C\,(V_{dd})^2$.

2.2 Component-Specific Mapping

The idea of component-specific mapping for defects has a history that predates FPGAs. We have long accepted that hard disks will not be manufactured perfectly and use defect maps to avoid the small fraction of sectors that cannot reliably store data. The TERAMAC computer provided the first large-scale use of defect identification and component-specific mapping for FPGA-like architectures [9]. This required a complete run of the placement and routing tools that were aware of the defects in the machine. Wong and Gojman extended the characterization of resources from defects to variation, showing how a modern FPGA with adjustable clocks can self-characterize the delay of individual resources [33, 14]. Using a Configured Test Circuit (CTC) on the FPGA fabric, they measure the delay of specific resource sets. Mehta extended the component-specific mapping concept to variation mitigation, building upon the data that Wong and Gojman showed how to extract [26].

The TERAMAC and Mehta model accepts that we must first measure devices and bring that information into the routing and perhaps placement phase. This means we must measure and store gigabytes of information for each chip, and we must run the CAD tools uniquely for each chip—a contrast from the component-independent model where we only need to generate a single mapping that can be reused across any number of chips.

To avoid the cost of full chip mapping, we can design the architecture or architectural mapping to allow small edits to the bitstream to exchange bad resources for good ones. Lach first showed how to strategically reserve spare LUTs and precompute alternate mappings that allow any defective LUT to be replaced locally [19]. Rubin showed how to reserve spare wiring tracks and precompute alternate routes to locally avoid interconnect defects [28]. These solutions accept a loss of efficiency from the full mapping approach in order to avoid the cost of computing a completely unique mapping for each chip. They also invest additional up-front costs to precompute alternatives to defects in order to minimize the per-chip mapping costs.

2.3 Temperature, Activity, and Aging

Beyond manufacturing variation, environmental and usage effects can also impact the delay of individual resources, and aging can change their delay over time [31]. Component-specific mapping allows us to reduce the manufacturing margins, but may still need margins for these other environment and usage effects. To the extent environment effects impact the die uniformly, such as ambient temperature, dynamic voltage scaling [8] used on top of component-specific mapping can reduce the level of margining. Local variation in chip activity [36] can cause resource delays to diverge from the delays captured by isolated CTC tests [22]. The online-monitoring techniques we adapt from COSMIC TRIP [12] (Sec. 5.5) capture these effects, while the algorithms that use CTC testing will not. Periodic re-characterization of the delays can reduce the necessary aging margins, which becomes more viable with the lightweight load-time techniques we describe here or can be eliminated with COSMIC TRIP.

3. CHOOSE-YOUR-OWN-ADVENTURE

All of our lightweight algorithms build on Choose-Your-own-Adventure (CYA) precomputed alternatives [28]. The CYA bitstream is organized by 2-point nets and contains multiple alternative paths for each 2-point net route. A 2-point net links a single source to a single destination; a full net with fanout to multiple destinations is represented by a collection of 2-point nets. At load time, the bitstream loader configures each 2-point net, then performs a simple test to validate that the configured route successfully transmits the intended signal. If it works, the loader keeps the path and proceeds to load the next 2-point net. If it fails, it tries one of the alternatives for the 2-point net stored in the bitstream. The loader does not require significant state or decision making; it simply loads bit patterns and branches on success failure indications from tests. The original CYA formulation suggested it could be an extension of the existing FSM that controlled bitstream loading. We expect the supervisory processors on the Stratix-10 [17] could be programmed to perform CYA bitstream loading.

The standard CYA approach to bitstream construction is to split the FPGA routing resources into two sets–a *base* set for normal routing and a *reserved* set to use for repairs. A normal Pathfinder [25] route is used within the base set to produce a base route. Every net in the application netlist has a route within the base resources, and the base resources for a net are reserved exclusively for that net. Alternate routes for each 2-point net are then identified from reserved resources and the unused base resources. These alternatives are allocated non-exclusively. Since most 2-point nets will use their base route, the alternatives will only be lightly used. Identifying multiple alternative routes for each 2-point net deals both with the cases where other nets do use resources that conflict with some alternative routes and the cases where the alternatives themselves are unusable.

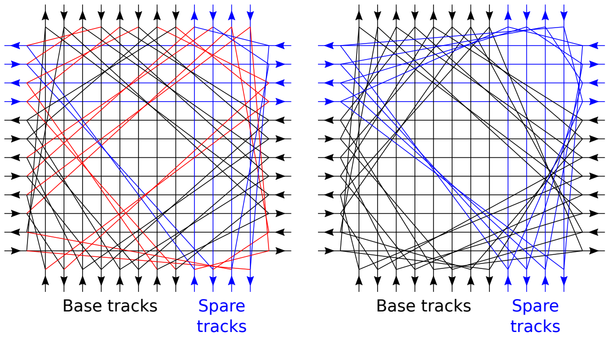
The key goal of Rubin's CYA alternative generation was to maximize diversity in order to minimize the chance that the alternative set would be unable to provide a defect-free path [28]. For timing-repair, we also care about the delay of the routes. Consequently, we tuned the cost function to prioritize alternative path generation by path delay with care to avoid duplicate paths.

4. ARCHITECTURE

As noted, CYA bitstream generation splits the channels in the architecture into two domains: base tracks and reserved spare tracks that are only available to alternative routes. Channels are interconnected via a modified Wilton S-box [24]. Fig. 1a represents a traditional Wilton S-box in a segment length 1 architecture. Tracks and connections entirely in the base domain are shown in black. Blue identifies the spare domain. A significant fraction of the connections is depicted in red, indicating cross-domain connections. These pose two problems: First, the spare tracks are reserved during base routing, so tracks that attempt to cross the boundary become dead-ends. Second, although base tracks are not reserved during alternative generation, they are often occupied, meaning these cross-domain connections also result in blocked paths for the alternatives. Fig. 2 illustrates how these dead-ends (red connections) increase the delay from a CLB to different channels. We eliminated the dead-ends by modifying the Wilton S-box such that base tracks never cross over to reserved tracks and vice versa. The modified S-box (Fig. 1b) can be regarded as two separate Wilton S-boxes, one of which switches among the base tracks, and the other among reserved tracks.

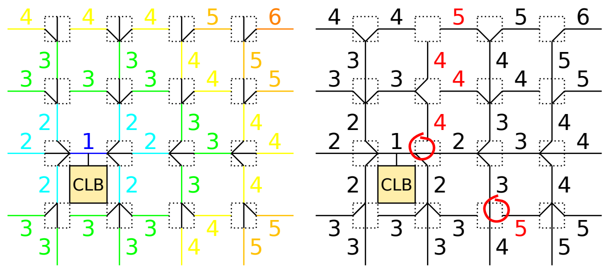
We also split the C-boxes so that extra CLB pins are connected only to spare tracks, and base CLB pins to base tracks. This saves area by omitting switches that would almost never be used. A drawback is that base tracks cannot be utilized by alternatives, even in situations where not all base tracks are occupied. The population of switches connecting spare tracks to extra CLB pins is controlled by $F_{C_{in,extra}}$ and $F_{C_{out,extra}}$. Their counterparts $F_{C_{in}}$ and $F_{C_{out}}$ are limited to the base tracks and regular CLB pins.

When Rubin used CYA for defect-tolerance, he only addressed the presence of spare paths. Considering delay and energy, a good alternative should also maintain or reduce delay. Even before considering variation, delay depends on the number of segments that a path traverses, the segment lengths, the fan-in of S-box multiplexers and CLB input pins. Sparse C-box population results in segments that are not connected to all CLB outputs and inputs, and segment staggering means that not all tracks can get from the source to destination using the same number of segments in a path.



(a) Original Wilton S-Box (b) Modified Wilton S-Box
Blue links show Reserved Tracks; Red links show switch connections that become unusable when we partition Base and Reserved Tracks.

Figure 1: S-Box Optimized for Reserved Tracks



(a) No dead-ends (b) With dead-ends
Red numbers in the (b) figure highlight delays that are larger than in the (a) figure due to the dead-end connections highlighted with red circles.

Figure 2: Effect of Dead-Ends in S-Box on Delay

To guarantee that the architecture will have good timing alternatives for every 2-point net, we derived a formula relating the number of distinct good timing alternatives, N_{alts}, of a 2-point net to the spare architectural resources:

$$N_{alts} = \left\lfloor \frac{O_s}{4} \right\rfloor \lfloor Tracks/output \rfloor \lfloor Inputs/track \rfloor \qquad (1)$$

Here, $\frac{O_s}{4}$ is the number of extra CLB output pins per CLB side. These pins are only available for alternative routes. The formula guarantees that there will at least N_{alts} connections from every CLB to a spare track that connects to the destination at the optimal stagger offset. The formula considers alternatives different as long as at least one section of the path (CLB pin or track) differs. The number of spare tracks connected to an output is

$$Tracks/output = \min\left(Round\left(F_{C_{out,extra}} \frac{W_s}{2} \right), \left\lfloor \frac{W_s}{2L_{seg}} \right\rfloor \right),$$

where $\frac{W_s}{2}$ is the number of spare tracks in one direction. The min operator limits $Tracks/output$ to the number of output multiplexers in a channel. The inputs per track is

$$Inputs/track = \frac{\left\lfloor \frac{I_s}{4} \right\rfloor Tracks/input}{\frac{W_s}{2}} \qquad (2)$$

with I_s as number of extra CLB input pins. Switches connecting to these pins are equally distributed among the tracks, so we computed $Inputs/track$ by dividing the number of

Table 1: Fast Alternatives Architectures Parameters

Guaranteed fast alternatives	1	2
Regular input pins	27	27
Regular output pins	8	8
$F_{C_{in}}$	0.15	0.15
$F_{C_{out}}$	0.2	0.3
Segment length	4	4
Extra input pins (I_s)	16	16
Extra output pins (O_s)	4	8
$F_{C_{in,extra}}$	0.25	0.25
$F_{C_{out,extra}}$	0.1	0.1
Spare tracks (W_s)	16	16
Overhead Area Sparing (% base)	20.8	21.7

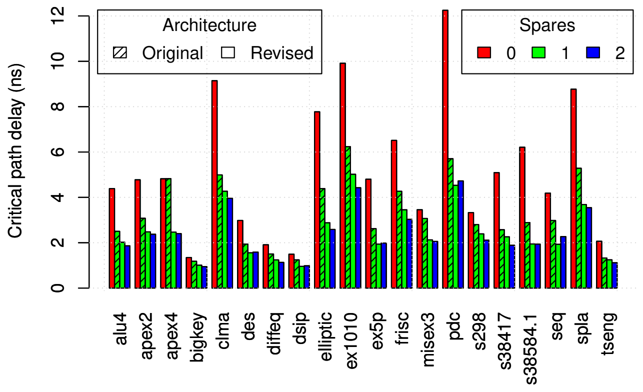


Figure 3: Architecture and Spare Provisioning Impact for Full-Path Pathfinder Selection at V_{dd} = 0.60V, 64 Alternatives

switches by the number of tracks in one direction. The number of tracks per input is provided by

$$Tracks/input = Round\left(F_{C_{in,extra}}\frac{W_s}{2}\right). \qquad (3)$$

To guarantee that alternatives can use segments with the same length, we modified the S-box connectivity at the edge of the FPGA, and we restricted the number of reserved tracks to multiples of $2L_{seg}$. We also ensure that the fan-in of multiplexers in the reserved domain is never higher than the fan-in in the base domain. Tab. 1 summarizes the architecture parameters selected for our experiments.

Fig. 3 shows the impact of the switchbox rewiring and spare allocation on the Pathfinder Selection algorithm. For the same resources (1 guaranteed fast alternative), the split-domain Wilton increases the potential gains from repair by 45% and reduces minimum channel width 10%. Using one guaranteed alternative saves 50% (geomean) delay, and a second provides marginal additional benefit.

5. MAPPING ALGORITHMS

In this section, we describe the algorithms that we characterize. Tab. 2 summarizes the key characteristics of the algorithms to highlight their differences.

5.1 OMFA

As a baseline, One-Mapping-Fits-All (OMFA) is the standard component-independent mapping. Working on nominal delay estimates for routing resources, VPR-Pathfinder routing [25, 23] is performed once, and the same mapping of nets to resources is used for all chips. The only time required to map a design with OMFA is the load time, which is configuration bandwidth dominated.

$$T_{omfa} = N_{bits} \times T_{bit} \qquad (4)$$

We use N_{bits} estimates from [28], using VPR to supply detail switch counts, and take T_{bit} =1b/ns after the Virtex-5 [34].

5.2 Full Knowledge

For the highest-quality mapping, we perform a normal VPR-Pathfinder-style placement and routing based on a routing graph where the delay over every link in the network is set to match the specific FPGA component. We assume CTC-style measurement of basic resources to obtain the link delays. Resources are decomposed into Discrete Units of Knowledge or DUKs from [14]. Pathfinder routing [25] is already designed to find shortest paths in this routing graph with irregular, heterogeneous delays, and techniques from [26] allow us to represent per-switch delays in VPR [23].

Full Knowledge routing requires a processor capable of running a full Pathfinder router to produce the component-specific bitstream. In typical operation, we imagine this would be performed once, before the FPGA platform is deployed. The bitstream would then be stored in configuration ROM or flash memory on the FPGA platform.

To estimate Full Knowledge mapping time, we include both the time to run VPR routing on the component delays, T_{vpr}, and the time to measure all the paths, $N_{dukpaths}$, necessary to compute all DUKs in the FPGA.

$$T_{full} = T_{vpr} + T_{dmeas} \times N_{dukpaths} \qquad (5)$$

We estimate the number of DUKs and paths based on [13]:

$$N_{dukpaths} = 2 \cdot N_{segments} \cdot (2L_{seg} + 1) + \qquad (6)$$
$$\left(1 + N_{ch} \cdot F_{cout} \cdot L_{seg} \cdot F_{cin} \cdot \frac{N_{ins}}{2}\right) 5K \cdot N_{luts} \cdot N_{outs}$$

We estimate T_{dmeas} as 2 seconds, based on observations that 4 DUKs can be measured on average per configuration in less than 8 seconds. We expect these numbers are conservative and could be significantly reduced with appropriate tuning. From prior work (e.g., [12]), we know algorithm runtime and testing are the dominant time components. We keep the models simple for illustration, omitting lower-order contributors such as load and reconfiguration time for algorithms with large testing and algorithm time.

5.3 CYA Defect-Only

CYA performs greedy, load-time selection among the alternatives. We adapt CYA for timing optimization by testing each 2-point net alternative, not just for functionality, but also for operation at a specified delay. We use a CTC measurement technique like the one from Wong or Gojman [33, 14] to test if a path will run at a specified delay. Load-time selection simply tests for a performance threshold and takes the first alternative that meets the specified performance; it does not characterize the performance of alternatives or try to select the highest-performance alternative.

The simplest load-time selection is a defect-only case where the timing test is set to some large threshold value (*e.g.*, 10 ns). Defect-only CYA tests at this large threshold value

Table 2: Key Algorithm Characteristics

	Measure			Base	Alter-	Greedy?	Delay	State
Algorithm	When	How	What	Rsrv	nates		Type	Bytes
Full Knowledge	Mfg.	CTC	All DUKs	N	N/A	No	Static	100M
Pathfinder Repair	Load	CTC	Paths + Repair DUKs	Y	Route	No	Static	10M
Pathfinder Selection	Load	Binary CTC	Paths + Repair Paths	Y	Precomp.	No	Static	10M
Incr. CYA	Operation	DDFFL	Delay at LUT (MD)	Y	Precomp.	By Slack	Observ.	1M
CYA Slack-Budget	Load	Binary CTC	Paths	Y	Precomp.	By Net	Static	10
CYA Defect-Only	Load	Binary CTC	Paths	Y	Precomp.	By Net	Static	10
OMFA	Never	N/A	N/A	N	N/A	N/A	Static	10

to filter out resources that are slow enough to be considered defective. Tab. 2 marks algorithms that only use CTC measurements to decide whether a path meets a threshold as "Binary CTC" to distinguish them from cases where the algorithm uses a series of CTC measurements to estimate the delay of a path or resource.

The Defect-Only CYA load time is dominated by testing alternatives for the threshold cutoff, T_{thresh}:

$$T_{defect-CYA} = N_{atry} \times N_{tmeas} \times T_{thresh} \qquad (7)$$

N_{atry} is the total number of alternatives tried during the load. Threshold measurement count, N_{tmeas}, is set to 1000.

5.4 CYA Slack-Budget

A more sophisticated option tests each path against a required time (RT). We could set the required time to the nominal delay for the path for each 2-point net. However, we can achieve the nominal delay for the circuit even when off-critical path 2-point net links do not make their nominal delay. That is, there is slack on these paths, and we can allow 2-point nets to use some of that slack. As a result, we budget the slack along the 2-point nets in a path from inputs to outputs. For this work, we use a very simple slack-budgeting scheme where each 2-point net in a path gets its delay-proportional share of the total path slack ($Slack(2pt_i) = Slack(Path(i)) \times \frac{Delay(2pt_i)}{Delay(CriticalPath) - Slack(Path(i))}$). Each 2-point net may be part of multiple paths that have differing initial slack, which results in unclaimed path slack after this formula is applied. Therefore, we distribute slack by repeatedly applying this formula until all the residual slacks are negligible or entirely distributed. More sophisticated slack-budgeting schemes are known in the literature (*e.g.*, [11]), but we leave those for future work. Finally, we scale the delay budgets to match the timing target for each load, resulting in a required time:
$RT_i = \frac{DelayTarget}{Delay(CriticalPath)} \times (Delay(2pt_i) + Slack(2pt_i))$.
Slack-Budget CYA performs a binary search to determine the minimum $DelayTarget$ achievable.

Slack-Budget CYA loading is the same as Defect-Only CYA, except that it uses the per-net timing target delay, T_{targ}, determined from slack budgeting. For simplicity in estimation, we conservatively use the target circuit delay:

$$T_{sbudget-CYA} = N_{atry} \times N_{cmeas} \times DelayTarget \qquad (8)$$

N_{atry} includes all alternatives tried across all delay targets in the binary search.

5.5 Incremental CYA

The CYA Slack-Budget greedy selection of the first "good-enough" alternative may not allocate fast resources where they are most needed. In the Incremental CYA algorithm, we postpone delay measurement and circuit customization to runtime, where they are performed in parallel with the main circuit operation. This effectively reduces the initial preparation time to the time needed for the defect CYA algorithm. Furthermore, testing is performed in the final environment with the full circuit configured, meaning temperature and activity effects are included in the characterization. During operation the algorithm performs measurements to locate the slowest resource and replaces the slowest path with an unused and non-conflicting CYA alternative path. As customization takes place incrementally, a circuit can already take advantage of delay improvements before the algorithm completes. Since repairs are made in order of need, the slowest paths get the first chance to select from available alternatives, providing a form of *list scheduling* [15]. Computations that can tolerate variable delay, such as best-effort and streaming dataflow computation, can start performing useful work immediately during this initial tuning phase. Tasks with real-time requirements may not meet their full-speed operation goals until a number of repairs have occurred. As we show in Sec. 7, timing repair can typically be achieved in tens of seconds.

Our algorithm is an adaptation of COSMIC TRIP [12], which was originally devised to deal with circuit slowdown caused by aging. As in COSMIC TRIP, we assume an FPGA equipped with Difference Detectors with First-Fail Latches (DDFFL) [20] connected to every LUT output to establish whether signals attain their final value at a time instant that precedes the end of the clock period by a configurable amount of time. The time between the start of the clock period and the latest arrival time for errorless operation is called the maximum delay MD_i of a LUT i. From the MD_i estimates, the algorithm computes the relative lateness, RL_i of every LUT. RL_i indicates the additional time that a LUT needs to produce an output value compared to its predecessors and its nominal delay. The LUT with the relative lateness that most exceeds its slack is selected for repair.

In COSMIC TRIP, the slack, $Slack_i$, was derived from the MD_i estimates before the circuit was affected by aging. When incremental repair is applied to reduce a timing margin, there is no equivalent to delays before aging to use as basis of the slack computation. Therefore, the search algorithm must be revised to accommodate variation. We constrain the slowest-LUT search to the critical path because any delay improvement in the remaining circuitry will not affect the minimum clock period of the circuit as a whole. We identify the critical-path LUTs by determining the LUTs that minimize the slack computed from the current timing errors. Every repair potentially affects the trajectory of the critical path, so we incrementally recompute the slack during every search, increasing the per-repair costs over COSMIC TRIP. When a LUT has been repaired, we must update the MD_i's and $Slack_i$'s in the network. However, only the MD_i intervals of the repaired LUT and its recursive fanout cone need to be reset and remeasured. We can reuse the MD_i's outside of the cone, reducing the time to updated the MD_i estimates compared to the initial estimation.

The incremental repair algorithm has much higher complexity than the CYA loader and requires megabytes of memory. We imagine it running on an attached processor such as the embedded ARM core on modern Zynq and Arria SoCs.

For Incremental CYA, we first run defect-only CYA, and then perform incremental repair attempts during operation.

$$T_{incr-CYA} = T_{defect-CYA} + T_{incr} \tag{9}$$

The incremental repairs require both algorithm time to compute the next measurement or repair attempt and operational cycles during which the DDFFL collects samples.

$$T_{incr} = T_{incr-select} + N_{tot-eval-cyc} \times T_{cycle} \tag{10}$$

However, the circuit may be performing useful work during the algorithm time and the computation, just not at the final rate of operation. An alternate indication of the cost of the algorithm is the lost time compared to running at the final operating speed.

$$\begin{aligned} T'_{incr} &= \left(\frac{T_{incr-select}}{T_{cycle}} + N_{tot-eval-cyc}\right) \\ &\quad \times (T_{cycle} - T_{final-cycle}) \end{aligned} \tag{11}$$

5.5.1 *Observed and Worst-Case Delays*

Conventional vendor CAD maps designs for worst-case delays with large margins. Component-specific mapping can map to the specific delays of a particular chip. However, the worst-case delays calculated with static timing analysis may still be larger than the delay paths typically seen in the chip. This may be in part due to false paths in the netlist graph that are not sensitizible [10, Ch. 8] or due to real paths that are, nonetheless, activated very rarely [27]. VPR timing estimates do not eliminate false paths, and even the best false-path estimates are necessarily conservative.

One fundamental difference of Incremental CYA is that it actively optimizes *observed* delays rather than worst-case delays. A path that is never sensitized does not contribute to the delay (MD) and lateness (RL) calculations. This can allow Incremental CYA to operate faster than a static timing analysis might predict. It also means that Incremental CYA will not repair a path before it is sensitized; consequently, it will never spend resources repairing a false path.

5.6 Pathfinder Repair

To make loading simple and fast, CYA makes several simplifications relative to a Full Knowledge route. Most notably, it splits base and reserved tracks, it uses a limited number of full LUT-to-LUT paths, and it performs alternative selection in a greedy fashion rather than using Pathfinder-style negotiated congestion. To characterize the effects of these limitations, we create two intermediates points between Full Knowledge routing and CYA. These algorithms can be viewed as limit studies providing insight into how much quality we are compromising by each of the individual simplifications.

In Pathfinder Repair, we look specifically at the impact of the base and reserved track split where we only reroute nets that fail to meet their timing, and we reroute these using only the routing resources available to CYA—the reserved tracks. The entire design is routed using the base resources with their nominal delay, just as for a CYA design. However, rather than pre-computing alternatives, the algorithm performs full-knowledge characterization of the reserved resources, and full Pathfinder negotiated-congestion routing for the two-point nets whose base routes do not meet the timing target. We implement this modification inside VPR by identifying only the failed two-point nets as the logical graph to route and marking only the reserved resources as available for routing. As a result, the Pathfinder Repair route has the highest quality possible for a design with the base/reserved track split. It sacrifices quality by not ripping up good routes in the base to reuse their resources, but, as a result, it saves time by only performing component-specific routing on nets that fail to meet timing in the base route.

Pathfinder Repair will require a processor and memory with the full capabilities to represent the detail FPGA routing graph for the reserved tracks and run full VPR-style routing. It must also be able to perform DUK measurements and DUK computations for the reserved track resources.

Pathfinder Repair requires DUK measurement time, repair time, $T_{vpr-repair}$, and time to measure each 2-point net to see if it meets its required timing target:

$$\begin{aligned} T_{path-repair} &= T_{vpr-repair} + T_{dmeas} \times N_{dukpaths} \\ &\quad + N_{2pt} \times N_{cmeas} \times T_{cycle} \end{aligned} \tag{12}$$

$N_{dukpaths}$ here uses Eq. 6 with $N_{ch} = W_s$, since Pathfinder Repair only needs to characterize the reserved resources. Since repair time only needs to route the failing nets on the smaller set of reserved tracks, routing time is lower than Full Knowledge routing ($T_{vpr-repair} < T_{vpr}$). We set N_{cmeas} to 2^{15} to match the Full Knowledge measurements.

5.7 Pathfinder Selection

Pathfinder Selection is designed as a mid-point between Pathfinder Repair and CYA to characterize the impact of using a limited set of LUT-to-LUT paths rather than performing full-knowledge route selection. That is, a key simplification in CYA is that it keeps only a small number of LUT-to-LUT path alternatives rather than performing a search on the route resource graph to explore available paths. This is what allows the CYA loader to be simple and avoid representing the route graph. CYA also allocates these LUT-to-LUT paths in a greedy manner; this also simplifies state and decision making. Pathfinder Selection keeps the limited LUT-to-LUT paths used by CYA, but performs Pathfinder-style negotiated congestion among the nets that fail to meet timing in the base route and their alternative paths in order to perform the repair. Compared to Pathfinder Repair, this characterizes the impact of only using a limited number of LUT-to-LUT paths. Compared to CYA, this characterizes the impact of greedy route selection. Pathfinder Selection is implemented in a modified VPR Pathfinder router where we restrict path expansion for a net to the set of precomputed alternatives that meet its slack budget target. As with CYA and Pathfinder Repair, we limit the set of 2-point nets to route to the set that fails to meet timing in the base route.

Pathfinder Selection has the same processor and memory needs as Pathfinder Repair since it must represent the individual routing resources to detect conflicts. Pathfinder Selection does not need to measure or compute DUKs since it operates entirely on LUT-to-LUT paths.

Pathfinder Selection replaces the runtime of full VPR with the runtime of Pathfinder negotiation on precomputed alternatives paths, $T_{vpr-select}$, and DUK characterization with 2-point alternative target delay filtering.

$$\begin{aligned} T_{path-select} &= T_{vpr-select} + N_{2pt} \times N_{cmeas} \times T_{targ} \\ &\quad + N_{2pt} \times N_{alt} \times N_{cmeas} \times T_{targ} \end{aligned} \quad (13)$$

6. METHODOLOGY

We compare the various algorithms on the Toronto 20 benchmark [5] set. We use VPR 5.0.2 [23] for placement and extend it with timing-target routing [29]. Our target architecture is an Island-style architecture [6] with 6-input LUTs ($K = 6$) and 8 base LUTs per cluster ($N = 8$) and a segment length of 4 using the split Wilton switchbox (Sec. 4), making it similar to the Stratix-IV [21]. Routes are performed with a base track allocation set at the minimum number of routing channels for the design. We add sparing to guarantee at least one alternative as fast as the base routes (*i.e.*, 16 spare inputs, 4 spare outputs, W_s =16 reserved tracks, with $F_{C_{out,extra}}$ =0.20 and $F_{C_{in,extra}}$ =0.15) as described in Sec. 4. Except for the OMFA and Full Knowledge mapping, all cases use separated base and reserved tracks. Full knowledge mapping performs its single route on the full set of base and reserved resources.

We use a 22-nm CMOS process modeled by the Predictive Technology Model (PTM) [35] with a typical operating V_{dd} =0.8V and Gaussian distributed threshold voltage with $\mu_{V_{th}}$=400mV, $\sigma_{V_{th}}$=36mV. We construct an FPGA "chip" by independently sampling each transistor's threshold voltage from this distribution and use the same set of 20 "chips" across all 7 algorithms. For most results, rather than presenting the characteristics of individuals, we present the 95% yield point—the delay or energy achieved by the second slowest or second highest energy chip in the batch of 20.

For Full Knowledge mapping and repair, routing starts with a complete delay map for the resources in the network. For load-time CYA, we simulate the CYA loader algorithm. For iterative repair CYA, we simulate both the DDFFL data collection and the repair algorithm.

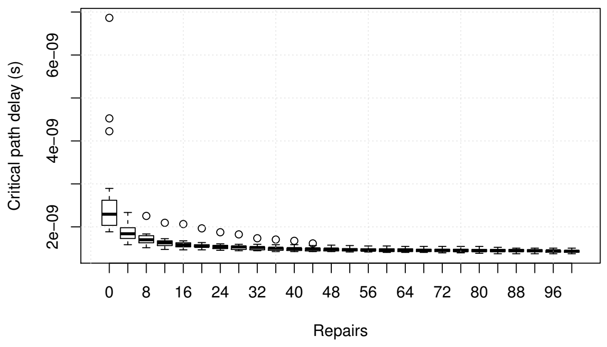
The algorithm time for mapping can be directly converted to energy assuming constant operating power for the mapping processor during the computation.

$$E_{alg} = T_{alg} \times P_{proc} \quad (14)$$

We run our algorithms on a laptop-class Intel Core i7-5600U processor at 2.6 GHz, monitor power consumption with PowerTop, and estimate P_{proc} ≈0.19 W. To run the full set of experiments, we also run jobs on a cluster of 2.7 GHz Intel Xeon processors and scale the runtime to match the laptop-class processor used for timing and power estimates.

7. EXPERIMENTS

Fig. 4 shows an illustrative Incremental CYA repair sequence using the `des` benchmark. We use boxplots to characterize the distribution of the 20 chips, 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. Before repair, the design has a large range of potential delays,



For each repair point, a boxplot characterizes the delay achieved for the set of 20 "chips" used in the experiment. Circles represent the outlier data points.

Figure 4: Incremental CYA Delay vs. Repairs for `des` for One Fast Alternative Sparing at V_{dd} =0.60V, 64 Alternatives

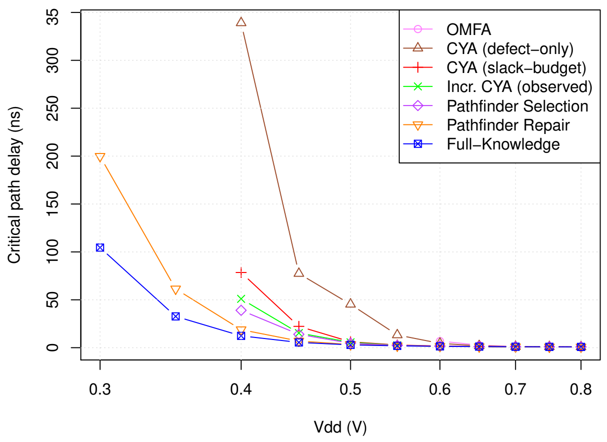


Figure 7: Delay vs. Voltage for `des` with One Fast Alternative Sparing, 64 Alternatives

spread over 5 ns, depending on how the randomly sampled slow resources happen to align with the critical path. If we had to guarantee 95% yield, we would be forced to treat this design as operating at 4.5 ns. However, after the first few repairs, the median drops below 2 ns and the worst-case chip is under 2.3 ns. As repairs continue, the distribution tightens. By 100 repairs, the delay is 1.4 ns, and the entire spread in the distribution is less than 0.13 ns. These 100 repairs occur over a period of one second.

Fig. 5 shows how Incremental CYA makes use of alternatives. We see the largest gains come from having one fast alternative, with some additional gains going to 4 alternatives. Only a couple of designs show additional improvement with 16 alternatives, and the 64 alternatives provides no significant gains. The guarantee of only one or a few fast alternatives coupled with the cost function prioritization that makes sure the most promising alternatives are selected first, and hence kept in the smaller alternative sets, means that the algorithm can generally satisfy the design without going very deep into the alternative set.

Fig. 6 shows how the algorithms compare. Two designs (`clma`, `ex1010`) do not achieve 95% yield at 0.6V for OMFA.

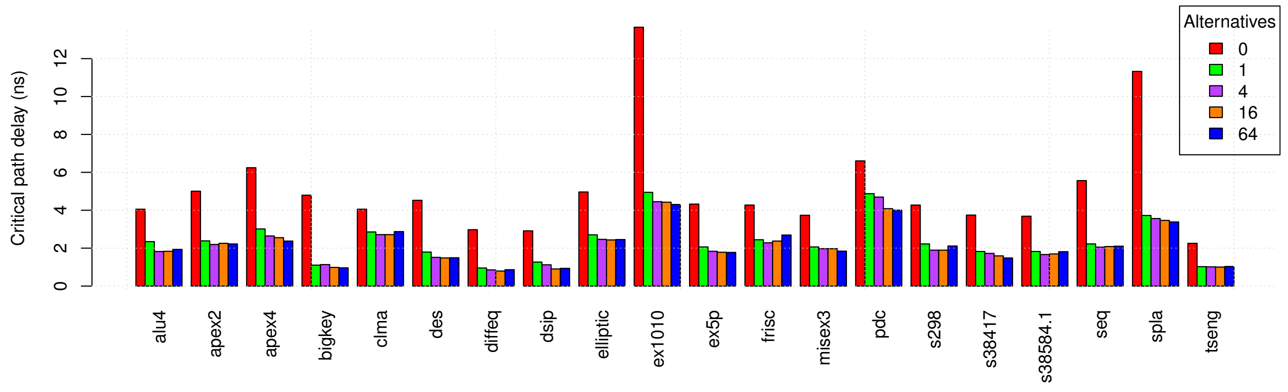


Figure 5: Delay vs. Alternatives for Incremental CYA with One Fast Alternative Sparing at $V_{dd} = 0.60\text{V}$

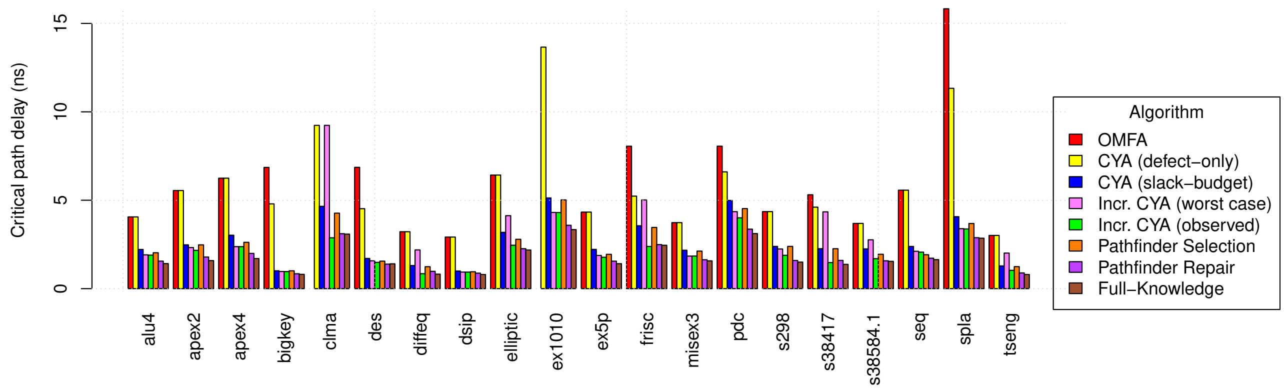


Figure 6: Delay for All Algorithms for One Fast Alternative Sparing at $V_{dd} = 0.60\text{V}$ and 64 Alternatives

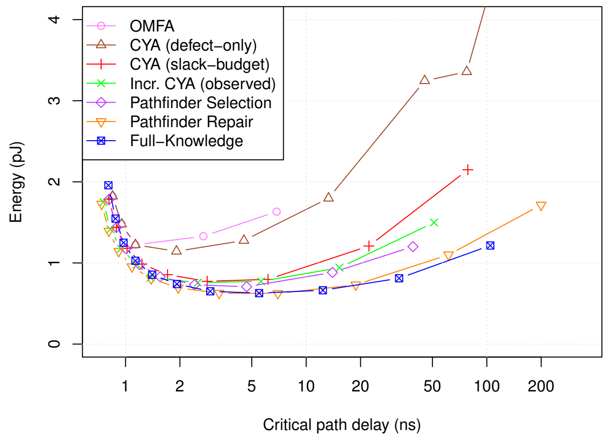


Figure 8: Energy vs. Delay for des with One Fast Alternative Sparing, 64 Alternatives

Slack-budget CYA is always able to improve over OMFA, with the improvement often being substantial. For a few designs, the worst-case, Incremental CYA shows little or no improvement over the defect-CYA that is run as a prefix to the incremental improvement. Nonetheless, the observed delay for Incremental CYA always achieves delays below Slack-budget CYA, showing that the incremental repair is effective in practice. Pathfinder Selection is only slightly better than Slack-budget CYA, suggesting that the greedy path selection in CYA has a modest effect on solution quality. Pathfinder Selection can be worse than observed delays for Incremental CYA since it is optimizing for static timing analysis. Full Knowledge achieves the lowest delays, as expected, but clearly shows that the slack-budget and Incremental CYA are closer to it than to the OMFA delays. Pathfinder Repair is only moderately worse than Full Knowledge, suggesting that algorithms do not sacrifice much quality for the simplification of only repairing slow paths. The larger gap between Pathfinder Selection and Pathfinder Repair, more evident in Fig. 9 and 10, shows that the limited set of full-path alternatives does have a quality impact. This suggests that additional tuning to generate a better or larger set of alternatives might be able to improve CYA quality.

As we lower the voltage, the delay increases and variation has a larger impact on chip delay. Fig. 7 shows how the algorithms stack up on delay for specific voltages. OMFA cannot guarantee 95% operational yield below 0.60 V, while Full Knowledge and Pathfinder Repair can scale down to 0.30 V. Various CYA alternatives and Pathfinder Selection can scale to 0.40 V. At 0.80 V the delay improvement among algorithms is small and undifferentiated. As voltage drops, we see larger separation among the algorithms.

Up to the point where leakage dominates, the lower voltage of operation turns into reduced energy (see Fig. 8). Component-specific repair allows the design to operate to and past the minimum energy point. The ability to reduce the delay at lower voltages, reduces the leakage penalty, allowing the component-specific repairs to shift the energy minimum down to lower energy points at greater delays.

In Figs. 9 and 10, we plot the quality resulting from the algorithms against the time required for mapping and loading. Raw bitstream loads can occur in hundredths of a second, while Full Knowledge mappings take 10^7–10^8 seconds. Defect-only, Slack-budget, and Incremental CYA map

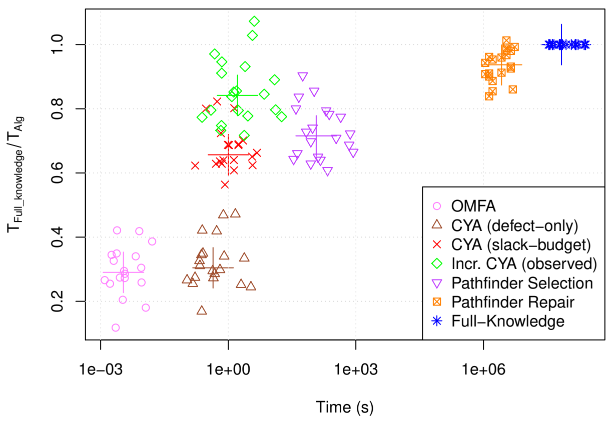


Figure 9: Delay vs. Mapping Time with One Fast Alternative Sparing at $V_{dd} = 0.60$V, 64 Alternatives

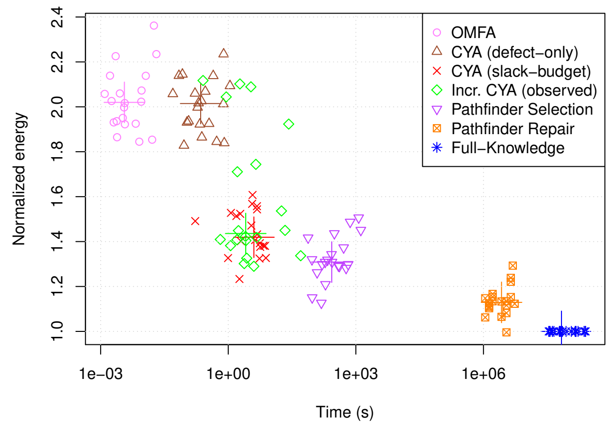


Figure 10: Energy vs. Mapping Time with One Fast Alternative Sparing, 64 Alternatives

in 1–10 seconds. Incremental CYA delay results are within $\frac{0.16}{1.0-0.29}$≈23%, and energy results within $\frac{0.44}{2.02-1}$≈43% of the Full Knowledge mapping.

Tab. 3 summarizes how the algorithms fare when we use them to minimize energy while achieving a delay only 20% larger than the nominal delay. Knowledge mapping schemes spend their dominant time characterizing the chip. Tab. 3 separates measurement time from mapping time, so we can also reason about their delay to get the design running on the FPGA assuming we already have a delay map. Furthermore, we expect the characterization times can be reduced by tuning, including simply running fewer measurement samples, perhaps at the expense of less accurate characterization. From the breakdowns in the table, we can see that the mapping time alone can cost two orders of magnitude more time than the CYA algorithms.

8. DISCUSSION

Slack-budget CYA gets half-way to the delay benefits of full-knowledge mapping, with under ten seconds of measurement and mapping time. At the expense of more sophisticated on-chip measurement and algorithms, Incremental CYA maps just as fast and closes over half of the remaining gap. These show that it is possible to achieve much of the potential benefits of component-specific with lightweight schemes that run quickly.

Table 3: Quality vs. Mapping Costs with One Fast Alternative Sparing Targeting $1.2 \cdot Delay_{nominal}$

Algorithm	T_{FK}/T_{alg}	E/E_{FK}	T_{meas} (s)	T_{alg} (s)	E_{cust} (J)
OMFA	0.98	1.4	0.0	0.0045	0.00086
CYA (def. only)	0.98	1.4	0.021	0.0	0.021
CYA (Sl. budg.)	1.02	1.3	0.063	0.0	0.057
Incr. CYA	1.01	1.1	0.050	0.84	0.18
Pathfinder Sel.	1.02	1.3	14	13	15
Pathfinder Rep.	1.03	1.0	$1.4 \cdot 10^6$	125	$4.6 \cdot 10^5$
Full Knowledge	1.00	1.0	$3.4 \cdot 10^7$	396	$1.6 \cdot 10^7$
	geomean aggregates				

Note that the Incremental CYA achieves break-even energy within 20 minutes of operation. As we see in Tab. 3, customization for Incremental CYA costs around 0.18 J and typically reduces it by $0.3E_{FK}$. Assuming that a clock cycle using the Full Knowledge algorithm costs around 1.1 pJ, the savings would be around 0.33 pJ, meaning the cost of customization is repaid after 5.5×10^{11} operations, or, assuming a 2 ns typical cycle time, around 1090 seconds (18 minutes).

Only Incremental CYA fully deals with in-system timing variation and aging. As such, the gap between Incremental CYA and a margined Full Knowledge is likely to be smaller in practice than illustrated here. Alternately, using Incrementally CYA on top of a Full Knowledge routed base route could achieve the high quality of Full Knowledge without needing additional margins.

9. CONCLUSION

Component-specific mitigation of delay variation can be quite tractable. While Full Knowledge characterization and mapping can take megaseconds (days), Slack-budget CYA typically achieves over 50% of the potential delay recovery and over 50% of the potential energy recovery, with under twenty seconds of load-time mapping. Incremental CYA requires similar tuning time and achieves comparable energy recovery (57% average) while achieving over 70% (77% average) of the delay recovery. With these lightweight schemes, energy break-even occurs in hours.

Acknowledgements

H. Giesen was supported by the Leggett Family Fellowship. This research was funded in part by DARPA/CMO contract HR0011-13-C-0005. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

10. REFERENCES

[1] A. Asenov. Random dopant induced threshold voltage lowering and fluctuations in sub-0.1 μm MOSFET's: A 3-D "atomistic" simulation study. *IEEE Trans. Electron Devices*, 45(12):2505–2513, December 1998.

[2] A. Asenov. Intrinsic threshold voltage fluctuations in decanano MOSFETs due to local oxide thickness variation. *IEEE Trans. Electron Devices*, 49(1):112–119, January 2002.

[3] A. Asenov, S. Kaya, and A. R. Brown. Intrinsic parameter fluctuations in decananometer MOSFETs introduced by gate line edge roughness. *IEEE Trans. Electron Devices*, 50(5):1254–1260, May 2003.

[4] K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer. High-performance CMOS variability in the 65-nm regime and beyond. *IBM J. Res. and Dev.*, 50(4/5):433–449, July/September 2006.

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

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

[7] D. Bol, R. Ambroise, D. Flandre, and J.-D. Legat. Interests and limitations of technology scaling for subthreshold logic. *IEEE Trans. VLSI Syst.*, 17(10):1508–1519, 2009.

[8] C. T. Chow, L. S. M. Tsui, P. H. W. Leong, W. Luk, and S. J. E. Wilton. Dynamic voltage scaling for commercial FPGAs. In *ICFPT*, pages 173–180, 2005.

[9] W. B. Culbertson, R. Amerson, R. Carter, P. Kuekes, and G. Snider. Defect tolerance on the TERAMAC custom computer. In *FCCM*, pages 116–123, April 1997.

[10] S. Devadas, A. Ghosh, and K. Keutzer. *Logic Synthesis*. McGraw-Hill, New York, 1994.

[11] S. Ghiasi, E. Bozorgzadeh, S. Choudhuri, and M. Sarrafzadeh. A unified theory of timing budget management. In *ICCAD*, pages 653–659, 2004.

[12] H. Giesen, B. Gojman, R. Rubin, and A. DeHon. Continuous online self-monitoring introspection circuitry for timing repair by incremental partial-reconfiguration (COSMIC TRIP). In *FCCM*, pages 111–118, 2016.

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

[14] 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.*, 7(4):5:1–5:23, Dec. 2014.

[15] R. Graham. Bounds on multiprocessor timing anomalies. *SIAM J. Appl. Math*, 7:416–429, 1969.

[16] C. He, M. F. Jacome, and G. de Veciana. A reconfiguration-based defect-tolerant design paradigm for nanotechnologies. *IEEE Design and Test of Computers*, 22(4):316–326, July-August 2005.

[17] D. L. How and S. Atsatt. Sectors: Divide conquer and softwarization in the design and validation of the Stratix 10 FPGA. In *FCCM*, pages 119–126, May 2016.

[18] K. J. Kuhn. Reducing variation in advanced logic technologies: Approaches to process and design for manufacturability of nanoscale cmos. In *IEDM*, pages 471–474, 2007.

[19] J. Lach, W. H. Mangione-Smith, and M. Potkonjak. Low overhead fault-tolerant FPGA systems. *IEEE Trans. VLSI Syst.*, 6(2):212–221, June 1998.

[20] 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*, pages 109–116, 2012.

[21] 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*, pages 33–42, 2009.

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

[23] 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*, pages 133–142, 2009.

[24] M. I. Masud and S. Wilton. A new switch block for segmented FPGAs. In *FPL*, pages 274–281, 1999.

[25] L. McMurchie and C. Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *FPGA*, pages 111–117, 1995.

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

[27] K. Minkovich and J. Cong. Mapping for better than worst-case delays in LUT-based FPGA designs. In *FPGA*, pages 56–64, 2008.

[28] R. Rubin and A. DeHon. Choose-Your-Own-Adventure Routing: Lightweight Load-Time Defect Avoidance. *ACM Tr. Reconfig. Tech. and Sys.*, 4(4), December 2011.

[29] R. Rubin and A. DeHon. Timing-Driven Pathfinder Pathology and Remediation: Quantifying and Reducing Delay Noise in VPR-Pathfinder. In *FPGA*, pages 173–176, 2011.

[30] P. Sedcole and P. Y. K. Cheung. Parametric yield modeling and simulations of FPGA circuits considering within-die delay variations. *ACM Tr. Reconfig. Tech. and Sys.*, 1(2), June 2008.

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

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

[33] J. S. Wong, P. Sedcole, and P. Y. K. Cheung. Self-measurement of combinatorial circuit delays in FPGAs. *ACM Tr. Reconfig. Tech. and Sys.*, 2(2):1–22, June 2009.

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

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

[36] K. M. Zick and J. P. Hayes. On-line sensing for healthier FPGA systems. In *FPGA*, pages 239–248, 2010.