Proactive Runtime Detection of Aging-Related Silent Data Corruptions: A Bottom-Up Approach

Jiacheng Ma
University of Michigan
USA

Majd Ganaiem
Technion - Israel Institute of Technology
Israel

Madeline Burbage
University of Washington
USA

Theo Gregersen
University of Washington
USA

Rachel McAmis
University of Washington
USA

Freddy Gabbay
The Hebrew University of Jerusalem
Israel

Baris Kasikci
University of Washington and Google
USA

Abstract

Recent advancements in semiconductor process technologies have unveiled the susceptibility of hardware circuits to reliability issues, especially those related to transistor aging. Transistor aging gradually degrades gate performance, eventually causing hardware to behave incorrectly. Such misbehaving hardware can result in silent data corruptions (SDCs) in software—a type of failure that comes without logs or exceptions, but causes miscomputing instructions, bitflips, and broken cache coherency. Alas, while design efforts can be made to mitigate transistor aging, complete elimination of this problem during design and fabrication cannot be guaranteed. This emerging challenge calls for a mechanism that not only detects potentially aged hardware in the field, but also triggers software mitigations at application runtime.

We propose Vega, a novel workflow that allows efficient detection of aging-related failures at software runtime. Vega leverages the well-studied gate-level modeling of aging effects to identify susceptible signal propagation paths that could fail due to transistor aging. It then utilizes formal verification techniques to generate short test cases that activate these paths and detect any failure within them. Vega integrates the test cases into a user application by directly fusing them together, or by packaging the test cases into a library that the application can invoke. We demonstrate our proposed techniques on the arithmetic logic unit and floating-point unit of a RISC-V CPU. We show that Vega generates effective test cases and integrates them into applications with an average of 0.8% performance overhead.

ACM Reference Format:
Jiacheng Ma, Majd Ganaiem, Madeline Burbage, Theo Gregersen, Rachel McAmis, Freddy Gabbay, and Baris Kasikci. 2024. Proactive Runtime Detection of Aging-Related Silent Data Corruptions: A Bottom-Up Approach. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3622781.3674182

1 Introduction

In recent decades, the semiconductor industry has made remarkable technological progress. Continuous advancements in process nodes have ensured a consistent downsizing of transistors to nanoscale dimensions, yielding improvements in performance and reductions in energy consumption. However, these advances have also heightened circuit reliability challenges [9, 57], notably evidenced by the emergence of silent data corruptions in data centers [23, 36–38, 52, 69, 75].

Silent data corruptions, or SDCs, are a form of undetected failure that occurs without generating logs, exceptions, or immediate program crashes. Instead, SDCs silently introduce incorrect data into applications. As a result, an error can spread far from its point of origin, potentially leading to failures that are difficult to predict, prevent, and troubleshoot. Recently, cloud providers have identified CPUs with such SDCs in their data centers [37, 52, 75].

SDCs are risky, as they challenge the fundamental "fail-stop" assumption of hardware failures that software developers have been accustomed to for decades. Most software applications in data centers and personal computers assume that a circuit, having undergone correct design, fabrication, and testing, will either function correctly or not work at all.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0391-1/24/04
https://doi.org/10.1145/3622781.3674182

Unfortunately, SDCs usually involve hardware malfunctions like miscomputing instructions and broken cache coherency, which are typically not considered by applications. Worse, these faults may be transient or persistent, which increases the difficulty of monitoring and mitigating SDCs.

Transistor aging is believed to be one of the causes of SDCs [12, 17, 19, 20]. This gradual performance degradation of transistors over time steadily increases signal propagation delays. Eventually, this aging results in timing violations inside a circuit, thus causing certain components to malfunction. Alibaba observed that a significant portion of SDCs in their CPUs appear only after a period of usage [75], suggesting that these SDCs may be attributed to transistor aging.

A straightforward strategy for mitigating the risks associated with SDCs is to conduct frequent and proactive testing using well-designed test cases, enabling quick identification and removal of malfunctioning hardware. Following this strategy, previous research has explored the formulation of test cases specifically for the detection of SDCs, as well as the development of frameworks for test management and scheduling. However, in order to stress individual components inside the circuit, these works tend to create complex tests with a long execution time, preventing them from being frequently scheduled. For example, in Alibaba, such tests are only scheduled once every three months [75].

The detection of SDCs—especially these attributed to aging—would be markedly improved by increasing testing frequency. Aging accumulates progressively and a circuit may exhibit SDCs at any stage of its lifecycle, so more frequent testing helps ensure more timely detection. Ideally, test cases should be selectively integrated within applications, guaranteeing routine execution and enabling immediate error-handling. This approach is traditionally impractical because of the long execution times associated with existing SDC tests. For example, Google's SiliFuzz [69] generates around 500,000 test cases, and a full execution of DCDiag [10]—Intel's official CPU diagnosis tool—takes 45 minutes.

Previous research tends to yield tests with a long execution time because of their adoption of a "top-down" approach. These works treat the hardware as a black box and generate test cases atop an abstract model of it. For example, Google's SiliFuzz [69] generates test cases by fuzzing the instruction set architecture (ISA) of a CPU, while Intel designed OpenDCDiag [1]—an open-source variation of DCDiag [10]—on top of popular libraries such as zlib [2] and eigen [48]. Due to the lack of implementation details in the abstract hardware model, these methods must generate a set of complex tests to ensure that all components inside the hardware are stressed.

However, hardware is not a black box. Circuit design is detailed in hardware description languages (HDLs) and subsequently synthesized into a netlist, which comprises a complex placement of gates and wires. This netlist serves as a blueprint during chip fabrication. Unsurprisingly, as a dominant factor in circuit reliability, transistor aging has been extensively studied at the level of gates and netlists [12, 22, 51, 64, 65]. Particularly, prior research has identified key electrical effects that contribute to transistor aging and developed comprehensive models to estimate these effects [15, 43].

In this paper, we present Vega, a novel "bottom-up" workflow designed to bridge the gap between the gate-level understanding of transistor aging and the proactive detection of aging-related SDCs in software. Vega empowers frequent and routine detection at application runtime, thereby improving the effectiveness of transistor aging failure detection. Specifically, Vega is comprised of three phases:

1– Aging Analysis identifies susceptible signal propagation paths that can potentially fail due to transistor aging. This is achieved through the use of aging-aware static timing analysis, supplemented by the well-studied gate-level models for transistor aging [15, 43].

2– Error Lifting transforms aging-prone paths into short test cases that are executable in a software environment and integrable into an application. This conversion leverages a combination of formal methods, logical modeling for timing errors, and heuristics based on the hardware's microarchitecture to ensure precise test case generation. As a byproduct, this phase additionally yields a number of failure models for the analyzed hardware, which can be valuable for future research in circuit and software reliability.

3– Test Integration combines test cases with an application. We showcase two approaches for such integration: a profile-guided method for automated test instrumentation, and a manual method for a more controlled integration.

We demonstrate Vega on the arithmetic logic unit (ALU) and the floating-point unit (FPU) of a RISC-V CPU, synthesized into a 28nm cell library. We show that Vega can identify aging-susceptible signal paths and generate effective test cases to target faults arising from them: these test cases incur negligible runtime performance overhead while ensuring routine aging detection.

Overall, we make the following contributions:

- We design Vega, a novel workflow that bridges the gap between the physical understanding of transistor aging and the proactive detection of aging-related SDCs in software.
- We evaluate Vega with a circuit synthesized into a real-world cell library, demonstrating the capability of frequent aging-related failure detection with negligible runtime overhead.
- We provide a set of circuit-level failure models for the analyzed hardware to facilitate future research into silent data corruptions.

2 Background and Motivation

This section begins with context about recent observations of silent data corruptions (SDCs) on data center-deployed hardware circuits (§2.1). Next, we summarize the development

process of such circuits (§2.2), and explore how transistor aging—a common cause of SDCs—can impact the performance and reliability of hardware circuits (§2.3). After each subsection, we present key takeaways that motivate Vega.

2.1 Silent Data Corruptions

Silent data corruptions (SDCs) are a form of undetected failure that silently introduces incorrect data into applications. These occur without generating logs, triggering exceptions, or causing immediate program crashes. Consequently, SDCs can propagate beyond their point of origin, leading to issues that are challenging to prevent, predict, or troubleshoot.

Recently, major data center operators, including Meta, Google, and Alibaba, have reported incidents of SDCs within their clusters [37, 52, 75]. Investigations into these SDCs uncovered that they stem from faults within computational circuits (i.e., CPUs), rather than the more typical suspect, memory devices. Occasionally, a CPU, despite having been correctly designed, fabricated, and tested, may still consistently produce incorrect results during certain operations, leading to various misbehaviors including miscomputing instructions and disruptions in cache coherency.

SDCs may manifest at any point during the lifecycle of a circuit, but only a limited subset of them can be detected during factory testing. Alibaba's data indicates that a significant 73.5% of the SDCs they identified occur in CPUs that have already been in use, either during system re-installations (63.9%) or while in production (9.6%) [75].

Currently, data center operators detect SDCs through extensive, long-running test suites that proactively stress the underlying hardware. However, given the low probability of SDC occurrence, it is impractical to run these tests frequently. For instance, in Alibaba's data centers, these tests are conducted only once every three months [75]. Consequently, there is a growing need for a more efficient and practical mechanism to identify SDCs.

Takeaway #1. Increasing the frequency of SDC testing can lead to more timely detection of SDCs. Reducing test execution time can make frequent testing more practical.

2.2 Hardware Development

The development of digital circuits involves a complex, multi-stage process, employing a diverse range of simulation and automated design tools at each stage.

2.2.1 Digital Circuits Design Flow. Digital circuits such as CPUs and GPUs are initially developed in hardware description languages (HDLs) like Verilog, SystemVerilog, and VHDL. HDLs empower developers to precisely describe the functionality of each component in a hardware design. Functionalities can then be tested through circuit simulations, allowing developers to verify and debug a design before it progresses to the physical fabrication stage.

Subsequently, in a process akin to software compilation, a hardware synthesizer transforms the circuit's functional description from an HDL into a netlist. The netlist is structured as a directed graph comprised of a large number of *cells* from a standard cell library, with wires that describe the electrical connections between the cells. These libraries, provided by chip manufacturers, describe the functionality and timing behavior of predefined circuit components such as logical gates and flip-flops, allowing the synthesized design to be practically implemented in hardware.

Following circuit synthesis, the hardware design progresses to a stage known as place-and-route. This stage involves strategically positioning cells into designated locations on a silicon die, and creating the wires that interconnect these cells. Moreover, it also ensures the clock signal reaches all parts of the chip in a timely and synchronized manner, which is crucial for the proper functioning of synchronous logic. Additionally, static timing analysis (STA) is employed to evaluate the compliance of digital signals with timing constraints, thereby determining the circuit's maximum operating frequency and ensuring its reliable operation. As an industry standard, STA assumes a set of conservative conditions for factors like temperature, voltage drop, and process variations, and calculates signal propagation delays for the worst case scenarios. While this approach may lead to some false positives by flagging non-critical timing issues, it prioritizes design robustness by ensuring functionality even under worst-case conditions.

2.2.2 Automated Design Tools. Automated design tools play a crucial role in almost every phase of hardware development, making it possible for large and complex circuits to be tested before the costly chip fabrication (a.k.a., tape-out), thus saving time and resources by identifying potential issues early in the design cycle [40, 60, 61, 73]. These tools rely on well-validated physical models and simulations to perform critical analyses such as timing, power consumption, and circuit reliability. To ensure that these models capture the real-world behavior of fundamental circuit components (e.g., transistors and wires), leading semiconductor foundries routinely perform a rigorous validation process that compares the simulation outputs against actual fabricated silicon. Such validation guarantees the high fidelity of the physical models in predicting device behavior under various conditions with recent technology nodes [49], thereby making them trusted resources in the semiconductor industry.

Takeaway #2. Hardware is not a black box: its implementation details can provide insights that guide SDC detection. Well-validated physical models can power a series of analyses that help identify SDCs.

2.3 Transistor Aging

Transistor aging, which is believed to be a significant cause of SDCs [12, 17, 19, 20], refers to the gradual degradation

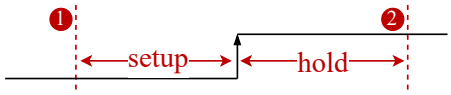

Figure 1. The setup and hold windows of a clock edge. Signals should arrive at its destination flip-flop before ❶ and hold stable until ❷.

of the performance and reliability of transistors over time. This process leads to an increase of a transistor's threshold voltage, which in turn causes a higher switching delay. As a result, signal propagation through an aged circuit may take longer than anticipated, potentially violating the circuit's timing constraints and resulting in malfunctions.

In modern circuits, this aging process predominantly stems from a physical phenomena called *bias temperature instability* (BTI), occurring when a static voltage is applied to a transistor for a long period of time [15, 55]. In other words, when a transistor remains in a constant state without regular switching, it is more likely to experience aging.

2.3.1 The Nonuniform Nature of Transistor Aging.

Transistor aging in a circuit is a nonuniform process [43], and several factors vary degradation rates. A key factor is the different BTI stress each transistor experiences during operation. Rarely-used circuit components tend to have more transistors idling in a fixed state, increasing their vulnerability to BTI effects. This variation is heightened in CMOS-based technologies due to their inherent design and operational characteristics. Specifically, as p-type transistors are more susceptible to BTI effects than n-type transistors, logical gates that consistently idle in a "0" state tend to age faster than those that idle in a "1" state or that switch regularly.

There are several additional causes of non-uniform transistor aging [44]. For example, clock gating, a standard power-saving technique, has been identified as a primary cause of uneven transistor aging. Clock gating inadvertently introduces varying levels of BTI stress across different areas of the clock network. Therefore, it leads to different aging rates in different regions of the network.

2.3.2 Timing Violations Caused by Transistor Aging.

For a circuit to operate correctly, it is crucial that signals comply with their timing constraints, reach their intended destinations, and remain stable within a critical time interval, as shown in Figure 1. Unfortunately, an aged circuit may potentially breach these requirements, leading to two types of timing violations: *setup violations* and *hold violations*.

A setup violation happens when a signal arrives at a flip-flop too late (i.e., after ❶), failing to meet the required setup time before the clock edge. In contrast, a hold violation happens when a signal changes too soon (i.e., before ❷), failing to hold stable for the required window after the clock edge. Both setup violations and hold violations can result in incorrect data being captured by the flip-flop, thereby causing circuit malfunctions. When this arises due to transistor aging in a previously-working circuit, this can result in SDCs or other application-level misbehavior. While setup violations can be addressed by lowering the clock frequency, this approach is ineffective for hold violations, as the clock frequency does not affect the required hold time. Consequently, hold violations are considered more severe than setup violations, as they necessitate chip repair.

2.3.3 The Physical Model for Transistor Aging.

The reaction-diffusion model, widely accepted for transistor aging, effectively describes the increase in a transistor's threshold voltage under BTI stress [13, 14, 26, 29]. With this model, the threshold voltage increase of a transistor, denoted as ΔV_{th}, can be determined via the following equation:

$$\Delta V_{\text{th}} \propto e^{\frac{E_a}{kT}} (t - t_0)^{1/6}, \tag{1}$$

where E_a is a constant related to process technology, T is the operating temperature, k is Boltzmann's constant, and $t - t_0$ represents the duration for which the transistor undergoes stress due to BTI effects. The reaction-diffusion model shows that the most significant V_{th} degradation happens early in the circuit's lifetime. For example, approximately 70% of the V_{th} degradation that will occur within a 10-year time frame occurs within the first year.

Using this equation, we can calculate the changes in a transistor's threshold voltage based on the duration of its exposure to BTI stress. Once the stress is removed, some of the degradation can be reversed, and a similar equation can be employed to quantify the recovery process.

Like other physical models, the reaction-diffusion model has also been validated against fabricated silicon with recent technology nodes, making it trustworthy and reliable as a tool for predicting transistor aging [13, 25, 26].

2.3.4 Profiling BTI Stresses with Signal Probability.

A common method for profiling the BTI stress of a logical element is to use *signal probability* (SP). SP calculates the probability of a signal being in the logical "1" state, calculated as the ratio of time spent in that state over total time. For example, an SP of 0.25 means the signal is in the logical "1" state 25% of the time (and in the "0" state 75% of the time). Usually, an SP profile is gathered by conducting functional simulations for the circuit, using a set of representative workloads that the circuit is expected to process.

With a given SP profile, we can calculate the ΔV_{th} for each transistor within a given cell (e.g., a `NOT` gate) using the reaction-diffusion model. Subsequently, analog simulation techniques, e.g., SPICE [63], can be employed to determine the change in the cell's switching delay. This change can then be considered in static timing analysis to identify timing violations that may occur after the impact of transistor aging.

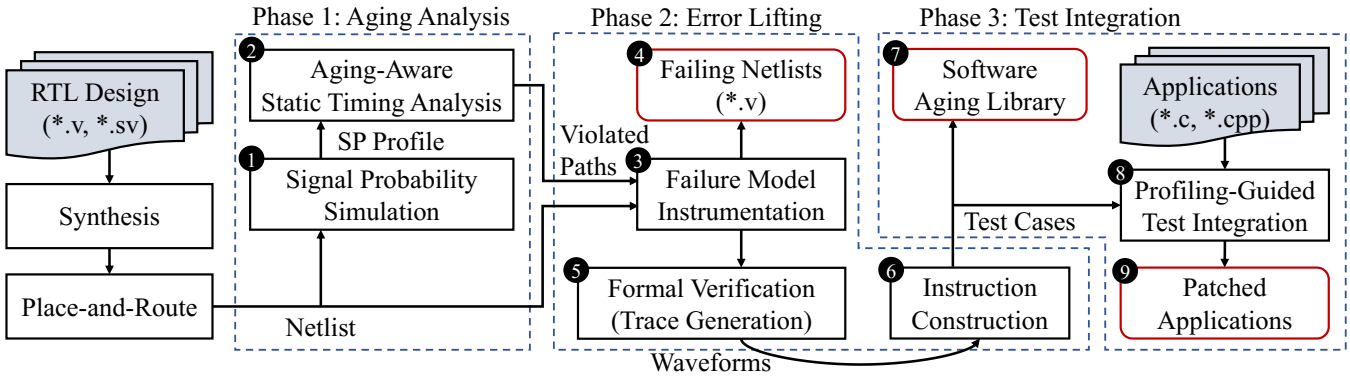

Figure 2. Overview of Vega's workflow, comprising three key phases: Aging Analysis, Error Lifting, and Test Integration. Each step in the workflow is outlined with a black box, with the inputs enclosed in gray boxes and the outputs in red boxes.

Takeaway #3. Signal probability profiles, along with aging-aware static timing analysis, can help identify logical elements prone to timing violations and potential SDCs, thereby producing effective targets for test cases.

3 Design of the Vega Workflow

Vega leverages these three takeaways to *enable frequent and proactive detection of aging-related SDCs at application runtime*. The workflow produces a set of software-executable instructions for testing a CPU's critical functional units, such as the ALU or FPU. Vega adopts a bottom-up approach, analyzing a CPU's detailed implementation to craft precise test cases for likely aging-related faults. The test suite is compact enough to be seamlessly integrated into an application's runtime. This targeted approach makes it practical to conduct frequent and timely detection of aging-related SDCs.

Figure 2 illustrates the three-phase workflow of Vega. In the first phase, *Aging Analysis*, Vega identifies locations within a circuit that are the most vulnerable to transistor aging. These are potential origins of SDCs that Vega aims to detect. To precisely identify the locations, Vega simulates the circuit with a set of representative workloads, and uses gate-level modeling of transistor aging to determine each components' timing degradation.

In the second phase, *Error Lifting*, Vega generates tiny software-executable test cases for each potential aging-related timing violation pinpointed in Phase One. First, Vega employs formal verification techniques to create a cycle-accurate trace of module-level input that could trigger a specific timing violation inside the CPU. Then, Vega translates the trace into assembly instructions using heuristics based on the CPU's microarchitecture. As a byproduct, this phase also yields a set of circuit-level failure models. These models, formatted as gate-level netlists, simulate failures and describe the possible misbehavior of the CPU as it ages.

In the third and final phase, *Test Integration*, Vega performs application-level integration of the previously-generated test cases. Vega supports two methods of integration, allowing test integration without code modification or enabling greater developer control. One approach produces a software library supporting different strategies of transistor aging detection and response, and wrappers compatible with various programming languages. Another approach minimizes a developer's integration effort by employing profile-guided techniques to embed test cases directly into an application while incurring minimal performance overhead.

In the rest of this section, we explain the design details of these three phases using an example circuit.

```
1  module adder (clk, a, b, o);
2    input wire clk; input wire [1:0] a, b;
3    output reg [1:0] o; reg [1:0] aq, bq;
4    always @(posedge clk) begin
5      aq <= a; bq <= b;
6      o <= aq + bq;
7    end
8  endmodule
```

Listing 1. An example hardware module.

3.1 Preparation for the Workflow

Consider the hardware module presented in Listing 1, written in System Verilog, as a representative example. This module implements a pipelined 2-bit adder that calculates the sum of two 2-bit integers, denoted as `a` and `b`. The computation is segmented into two cycles. In the first, `a` and `b` are sampled in `aq` and `bq`, respectively (Line 5). In the second, `aq` and `bq` are summed, with the result stored in `o` (Line 6).

As we described in Section 2.2, this module will proceed through a sequence of processes during development, including circuit synthesis and place-and-route, which eventually transform the circuit into the netlist illustrated in Figure 3. For simplicity, we exclude components used solely for timing correction, such as clock buffers, and employ a minimal standard cell library. This library consists of three cell types: `AND` and `XOR` cells, which respectively perform the "and" and

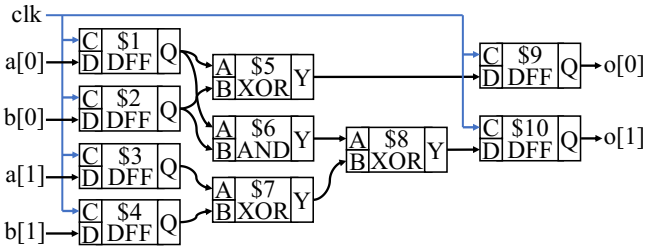

Figure 3. The netlist associated with Listing 1. Components used for timing correction (e.g., clock buffers) are excluded.

"xor" operations on their inputs, and the DFF cell, a D-type flip-flop that registers its input D for one clock cycle. For our example, we also assume the maximum propagation delay of the AND, XOR and DFF cells is 0.3ns, and the minimum delay is 0.1ns. The DFF cell requires a setup time of 0.06ns and a hold time of 0.03ns. Additionally, the module targets an operation frequency of 1GHz; therefore, each cycle spans 1ns.

Our example netlist satisfies the cells' timing constraints: The longest path ($4 → $7 → $8 → $10) accumulates a maximum delay of 0.9ns, indicating signal arrival at $10 more than 0.06ns (setup time) before the next clock edge. Conversely, the shortest path ($1 → $5 → $9) has a total minimum delay of 0.2ns, ensuring $9's input is stable for greater than 0.03ns (hold time) after the clock edge. However, transistor aging may disrupt these constraints. To evaluate aging's impact on the circuit, the netlist is forwarded to the Aging Analysis phase for further examination.

3.2 Aging Analysis

In the first phase, Aging Analysis, Vega identifies signal propagation paths that are likeliest to violate timing constraints after experiencing realistic transistor aging. First, Vega instruments the circuit's netlist and simulates a set of representative workloads on it (❶ in Figure 2). We record the signal probability for each cell in the design, which correlates with its susceptibility to BTI. This profile is then consumed by an *Aging-Aware Static Timing Analysis* to determine the most likely locations of future timing violations (❷).

3.2.1 Signal Probability Simulation. To determine which of a circuit's cells will experience the largest effect from transistor aging, Vega needs to estimate the likelihood that a cell will rest at different signals during its lifetime. Vega does this through simulation of an instrumented circuit. It attaches a counter to the output port of each cell in the circuit's netlist, recording the incidence of logical "0" and "1" states from that cell. For our example circuit, these counters attach to the Q port for DFF cells and the Y port for AND cells and XOR cells. Notably, these counters are driven by a separate free running clock generated by Vega. Vega assures this clock continues toggling even if the clock within the circuit is paused.

Signal	SP	Signal	SP	Signal	SP
DFF$1.Q	0.85	DFF$2.Q	0.54	DFF$3.Q	0.38
DFF$4.Q	0.27	XOR$5.Y	0.46	AND$6.Y	0.48
XOR$7.Y	0.13	XOR$8.Y	0.52	DFF$9.Q	0.44
DFF$10.Y	0.54				

Table 1. An SP profile associated with the netlist in Figure 3.

Vega then simulates the instrumented circuit, post-place and route, with an HDL simulator and a set of representative workloads. After the simulation is completed, Vega aggregates the values of each cell's counters to determine what fraction of the time the cell remained at logical "1": together, this forms a signal probability (SP) profile for the circuit. Table 1 shows an example SP profile corresponding to a simulation of the netlist in Figure 3. Notably, the XOR cell $7 has a particularly extreme SP value; therefore, it is under the highest BTI pressure and more susceptible to transistor aging. The clk signal is omitted from this example, because clock distribution in a placed and routed design involves the use of numerous clock buffers. In a real-world SP profile, each of these clock buffers is profiled individually.

3.2.2 Aging-Aware Static Timing Analysis. Vega can now identify timing violations that may emerge in the circuit due to transistor aging. By harnessing the SP profile generated in the last step along with an *aging-aware timing library*, Vega can quantify the performance degradation of each logical cell in the circuit. This timing library characterizes how signal probability affects each cell's timing characteristics, such as maximum and minimum propagation delay, over a period of time. Vega generates this library by conducting analog simulation with SPICE [63] for each cell in the standard cell library, determining how changes in a cell's physical property correspond to changes in its timing characteristics. Since multiple circuit designs may use the same standard cell library, this work is pre-computed to accelerate Aging-Aware Static Timing Analysis (STA). Figure 4 shows an example entry from a pre-computed timing library, showcasing the speed degradation of a typical AND cell.

Once the aging-aware timing library is generated, Vega looks up cell data by their signal probabilities and updates the timing characteristics of the netlist under test. Vega performs static timing analysis to identify signal propagation paths that exhibit timing violations. These paths are considered aging-prone, with a higher risk of experiencing timing violations due to the impact of transistor aging. The Aging-Aware STA is based on a 10-year assumed lifetime that is commonly adopted by mission critical systems [32]. During the Aging-Aware STA, Vega also analyzes the effect of aging on the clock distribution network. This analysis can reveal phase shifts of the clock signals in different locations, which could potentially lead to hold violations.

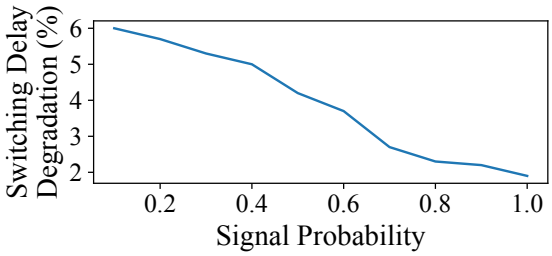

Figure 4. The switching delay degradation of a 28nm XOR cell under different levels of SP over a 10-year period.

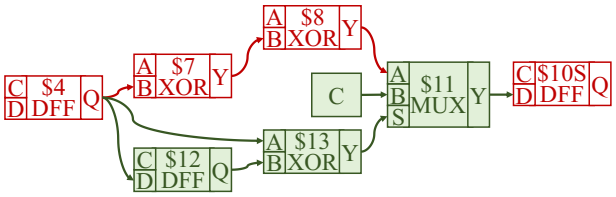

Figure 5. Failure model for a setup violation in the path $4 → $7 → $8 → $10, with red highlighting the failing path and green indicating the instrumented cells. Unrelated signals are omitted for clarity.

Notably, the Aging-Aware STA runs while taking into account the most pessimistic conditions for voltage drop, process variation, on-chip variations, and temperature. These conditions are provided as part of the requirements specified by foundries [27], and are aligned with the industry standard in delivering statistically robust analysis results. Consequently, while paths identified by Aging-Aware STA may not manifest failures in the real world due to less critical voltage and temperature, the real world's failing paths would be captured using these conservative conditions.

Based on the SP profile in Table 1 and the timing library demonstrated in Figure 4, Vega finds that the propagation delay of the path $4 → $7 → $8 → $10 accumulates to 0.946ns after considering transistor aging. Therefore, it violates the required setup window (0.946ns > 1ns - 0.06ns) and incurs a violation. For demonstration purpose, we also assume that a phase shift is detected between the clock signals connected to DFF $1 and DFF $9, causing a hold violation in path $1 → $5 → $9. These violating paths are provided for the next phase, Error Lifting, to help test case formulation.

3.3 Error Lifting

In the Error Lifting phase, Vega formulates test cases for aging-related hardware faults, targeting these tests to the aging-sensitive signal paths identified in the previous phase. Vega crafts test cases in two steps. First, it instruments the hardware module's netlist with a failure model propagating the effect of a previously-identified timing violation (Figure 2, ❸). Then Vega uses a hardware formal verification tool to produce a sequence of cycle-accurate, module-level

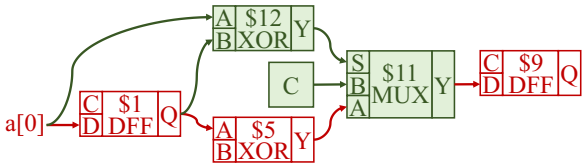

Figure 6. Failure model for hold violation path $1 ⇝ $9.

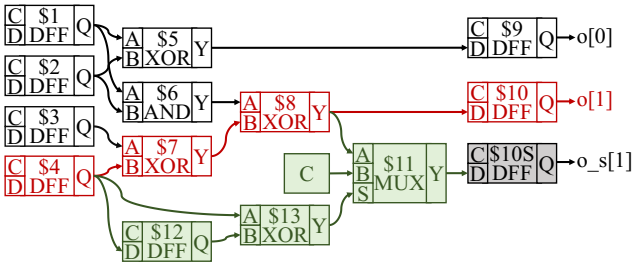

Figure 7. The netlist instrumented with the shadow replica (in gray) and the failure model (in green).

inputs that provably provoke the current hardware fault (❺). These inputs, represented in a hardware waveform, are constrained to ensure an incorrect value will be generated in the module's output. Finally, Vega processes the input sequence to generate a series of software-executable instructions that activates the exact waveform inside the circuit (❻).

We opt for formal verification in test case generation because it provides a systematic way to explore infrequently used components inside the circuit. Formal verification enables us to create a concise set of test cases that target specific potential failures inside the circuit, therefore avoiding the long execution time associated with tests crafted from top-down approaches like fuzzing. Furthermore, formal methods allow us to prove a given hardware fault cannot provoke logical inconsistencies in the tested module, a possibility that fuzzing cannot conclusively rule out.

Formal verification only operates within the logical domain and does not consider timing violations, so we need to introduce a model that logically describes the misbehaviors associated with these timing violations. Moreover, hardware formal verification via model checking proves more practical at the scale of individual hardware modules, rather than an entire hardware design: a single FPU is much easier to check than a full CPU with cache, memory, and complex software running on top of it. Consequently, we choose to only apply formal verification on specific hardware modules, and then construct the corresponding instructions by leveraging our understanding of the hardware design's microarchitecture.

3.3.1 Logical Models for Timing Violations. As we described in Section 2.3, transistor aging may cause setup violations and hold timing violations. During a setup violation, the signal reaches the flip-flop too late, failing to meet the

setup window. As a result, the flip-flop may sample an incorrect value during the clock tick. However, it is important to note that even if a signal path fails to meet its required setup time, the flip-flop may still sample the correct value in some cycles, provided the values previously held in the path remain unchanged [67]. As a result, for a signal path between a pair of DFFs $X \rightsquigarrow Y$ that violates its required setup time, Y's output at cycle $t+1$ is logically modeled as:

$$Y(t+1) = \begin{cases} Y_{original}(t+1) & \text{if } X(t) = X(t-1) \\ C & \text{otherwise} \end{cases}, \quad (2)$$

where $Y_{original}$ represents the value of Y assuming no violation occurs, and C denotes the wrong value sampled by Y when the timing violation occurs. For formal verification, C is set to a constant value—either 0 or 1—to limit the search space the formal verification tool is required to explore. Nonetheless, the tool can conduct separate rounds of verification for each case of C, allowing Vega to generate test cases for different scenarios that could occur in an actual circuit.

Similarly, in a hold violation, the flip-flop may still sample a correct value by chance when the path's value remains unchanged in the next cycle. Therefore, for a signal path $X \rightsquigarrow Y$ that violates its required hold time, Y's output is modeled as:

$$Y(t+1) = \begin{cases} Y_{original}(t+1) & \text{if } X(t) = X(t+1) \\ C & \text{otherwise} \end{cases}. \quad (3)$$

In the special case where the path starts and ends at the same flip-flop, we consider Y to always produce the value C. This approach is adopted because, in these situations, the value captured by Y relies on its own value in the same cycle. As a result, Y will consistently be in a meta-stable state [47, 67].

3.3.2 Failure Model Instrumentation. To integrate this logical model of timing violations into the circuit's netlist, we introduce a MUX cell. MUX functions as a selector, outputting either of its inputs, A or B, depending on the value of a select signal, S. Figure 5 shows the instrumented failure model for the setup violation occurring in path `$4` (X) $\rightsquigarrow$ `$10` (Y). In this instrumentation, DFF `$12` is used to retain the output value of `$4` for a cycle, thereby allowing $X(t) = X(t-1)$ to be calculated. Similarly, Figure 6 shows the failure model for the hold violation in path `$1` (X) $\rightsquigarrow$ `$9` (Y). In this instrumentation, $X(t+1)$ is derived from the input of `$1`, since `$1` is a DFF and its input value will be output in the next cycle.

Failure Model Instrumentation can function in one of two modes. In one mode, the instrumentation phase generates a *failing netlist*, a Verilog file that describes the behavior of the circuit component after the impact of transistor aging (❹). This file can be synthesized for a range of targets, including simulation environments and FPGAs, rendering it useful as a circuit-level failure model in future reliability research. Currently, this circuit-level failure model enables us to validate the effectiveness of test cases constructed by Vega, as detailed in Section 5.2.2.

Alternately, the instrumentation phase can prepare the netlist for trace generation to support the crafting of targeted test cases for the modeled failure. Instead of directly integrating the failure model into the netlist, Vega first generates a shadow replica for a portion of the netlist. Specifically, for an aging-prone path $X \rightsquigarrow Y$, the instrumentation conducts a static analysis of the circuit and identifies all cells that can potentially be influenced by Y—this includes Y itself. Based on this analysis, it creates copies of these identified cells, with the same interconnections between copied cells that the original cells have. The failure model is then integrated into this shadow replica, so the module-wide effects of a targeted timing violation can be tracked.

Figure 7 shows the instrumentation for the setup violation path `$4` $\rightsquigarrow$ `$10`. As shown in the figure, a shadow cell, `$10S`, is created, with its input linked to the failure model and its output to a shadow wire named `o_s[1]`. In the subsequent step, this shadow wire will be used by the formal verification tool to guide the generation of module-level inputs.

Hold violation paths use similar instrumentation to setup violation paths. In the case of the hold violation path `$1` $\rightsquigarrow$ `$9` from Figure 6, a shadow replica will be created for cell `$9`, with its output linked to a shadow wire.

3.3.3 Trace Generation using Formal Methods. After the shadow replica is created and connected to the failure model, Vega incorporates a formal verification tool to produce a sequence of module-level inputs that provokes the instrumented failure. Specifically, it formulates a *cover property*—a System Verilog primitive—that requires that the value in the shadow replica differs from the values in its corresponding original copy. For example, Vega generates the below property for the instrumented netlist in Figure 7:

```
1  cover property (@(posedge clk) o[1] != o_s[1]);
```

Hardware formal verification tools are designed to interpret these properties, combining a variety of formal methods to attempt to prove whether the properties remain true for a given design. To aid debugging, these tools often provide a trace of design inputs to illustrate the result of this proof. In the case of cover properties, the tool will attempt to find a trace causing the expression in the property to evaluate to true for at least one cycle.

The traces that show a shadow replica's output differs from its original copy can be used to generate software tests for hardware faults. In the example circuit from Figure 7 with C—the current fault's constant output—set to 1, the hardware formal verification tool finds the trace described in Table 2. When the circuit's input signals align with the trace, the expression in the cover property will evaluate to true at cycle 3. The trace is then captured and saved as a waveform, which proceeds to step ❻ for instruction generation.

In some scenarios, it is necessary to apply extra restrictions on the module's input to prevent unrealistic traces from being generated. These restrictions are described using

Cycle	1	2	3
a[1:0]	'b01	'b11	'b11
b[1:0]	'b11	'b00	'b01
o[1]	'b0	'b0	'b0
o_s[1]	'b0	'b0	'b1

Table 2. An example trace that provokes the instrumented failure in Figure 7. o[1] and o_s[1] mismatch at cycle 3.

the *assume property* primitive of System Verilog, and writing useful restrictions requires that developers have some knowledge of the target microarchitecture's behavior. For instance, when analyzing a hardware module like an ALU, we may restrict the range of input to include only valid operations.

3.3.4 Mitigation for Initial Value Dependency. In some instances, the traces produced by the formal verification tool may not reliably trigger failures in a real-world execution. This issue occurs because the tool assumes that the circuit's initial state has been perfectly reset. Specifically, the tool first simulates the circuit's reset behavior to obtain the initial values for each signal within the circuit, before beginning its symbolic exploration of the design. Consequently, it may generate traces that are effective only under these specific initial values. However, in a real-world execution, these initial values may be modified by a previous instruction, potentially making the generated trace ineffective.

To mitigate this issue, Vega allows configuring the failure model to activate only when detecting a rising or falling edge in the value of X (i.e., the starting point of the violated path). For example, in Figure 7, it may replace cell $13 with logics that determines ¬$12.Q ∧ $4.Q (i.e., a rising edge) or $12.Q ∧ ¬$4.Q (i.e., a falling edge).

3.3.5 Instruction Construction. Reliable activation of aging-related failures is raised to the software level in this step, which translates a trace of module-level I/O signals into assembly code that could generate them. Vega leverages expert knowledge of the CPU's microarchitecture here. This step is the most labor-intensive part of Vega, but only has to be done once: For each CPU microarchitecture and hardware component under analysis, developers must write a script to facilitate instruction construction. To create this script, developers use their knowledge of how each instruction activates signals in the analyzed hardware component; therefore, they can create a look-up table that links the activation of signals to an instruction. In some cases, additional steps like mapping constant values to specific registers are necessary.

During instruction construction, Vega determines the values of the input registers and the expected value of the output registers. However, the allocation of these registers is deferred to the next phase (i.e., Test Integration), to allow a more seamless integration of test cases with applications.

3.4 Test Integration

In this phase, Vega crafts the constructed instruction sequences into test cases that can be run from applications. There are two methods of integration, allowing flexibility in how SDCs can be monitored. First, Vega can create a software aging library for detecting aging-related SDCs (❼). Second, Vega provides a profile-guided method to automatically integrate the test cases into an application (❽).

3.4.1 Generation of Software Aging Library. In this approach, Vega combines the generated test cases together in a C file, using a set of pre-defined templates. In this C file, each test case is specified with the standard inline assembly format, while the registers are designated as variables for clarity. Furthermore, Vega generates support files for the compilation, as well as a set of helper functions and language-specific wrappers. These helper functions are designed to support different scheduling methods for the test cases, allowing them to be executed either sequentially or in a random order. Additionally, for programming languages that support exceptions, this library can be configured to trigger an exception when failing a test case. This allows detected hardware faults to be handled by an exception handler (e.g., a catch block) within the call stack.

3.4.2 Profile-Guided Test Integration. To enable test integration without the need to modify the application's source code, Vega employs a profile-guided approach for embedding test cases. Specifically, Vega first instruments the application with a series of counters, which track and record the invocations of application code (i.e., at the granularity of basic blocks) throughout the application runtime. Vega then executes the application with representative inputs to collect a *profile* that reflects the characteristics of the application's execution. Using this profile, Vega identifies a location in the program that, while not frequently invoked, is still routinely accessed. This location is chosen to be the point of test integration.

Subsequently, Vega integrates its generated SDC test cases into the chosen location. Vega estimates the expected performance overhead of the test cases by comparing the number of intermediate representation (IR) instructions before and after adding the test cases. If the estimated overhead is above a user-defined threshold, Vega then restricts the invocation of test cases so that they trigger with a certain probability, controlling SDC testing frequency at a finer granularity. Thus, Vega ensures that the overall performance overhead remains within manageable limits.

4 Implementation

We prototype and demonstrate Vega for the arithmetic logic unit (ALU) and floating-point unit (FPU) of the CV32E40P [33, 46, 59]—an open-source, 32-bit, in-order RISC-V CPU—and a real-world 28nm process technology. However, Vega's design

can be applied to other instruction sets, microarchitectures, and process technologies.

During Aging Analysis, we use embench's floating point matrix inversion test, "minver", as a representative workload for the ALU and FPU under test. To construct the aging-aware timing library, SPICE [63] is used to conduct the analog simulation which determines the gate delay degradation of cells with varying SP profiles. Cadence Innovus [3] is then used to perform the timing analysis. Subsequently, we employ a set of TCL scripts to post-process the timing report and update the cells' timing characteristics to those affected by aging. Error Lifting is primarily implemented atop of Yosys [76], adding ~3,700 lines of C++ for Failure Model Instrumentation and ~400 lines of Python for Instruction Construction. JasperGold [4] is used to conduct the formal verification. Profile-Guided Test Integration is implemented as a set of LLVM [56] passes with ~800 lines of C++.

5 Evaluation

We evaluate Vega along the following dimensions:

Effectiveness. Can Vega identify signal propagation paths prone to transistor aging (§5.2.1)? Can Vega generate test cases that are executable in a software environment (§5.2.2)? Can these test cases detect aging-related SDCs, and do they outperform randomly generated test cases (§5.2.3)?

Efficiency. How much performance overhead do these test cases incur when integrated into an application (§5.3)?

5.1 Experimental Setup

Hardware. We evaluate Vega on the ALU and FPU of the CV32E40P. These components are synthesized for a 28 nm process technology using Cadence Genus [5] and Synopsys Design Compiler [6], and placed-and-routed using Cadence Innovus [3]. The ALU targets an operating frequency of 167 MHz and the FPU targets a frequency of 250 MHz.

Software. We evaluate Vega's performance impact with embench benchmarks [7], a benchmark set for embedded CPUs like the CV32E40P. These benchmarks are compiled using OpenHWGroup's clang fork [8] with an `-O2` flag. This set of benchmarks are also used as representative workloads during Signal Probability Simulation (§3.2.1).

Failing Netlists. To evaluate Vega's effectiveness in identifying aging-related SDCs, we use the failing netlists generated during failure model instrumentation (Section 3.3.2). We configure these failing netlists to operate in three distinct modes by setting C, the value sampled by the endpoint of the failing path that we describe in §3.3.2. C is either held at 0, 1, or allowed to take a random value in each cycle.

Simulation Environment. All experiments are carried out using the official simulation framework of the CV32E40P with Verilator [70]. To speed up simulation and focus on the primary evaluation targets, i.e., the ALU and FPU, only these components are replaced with the placed-and-routed netlist. The remainder of the CPU is simulated in System Verilog.

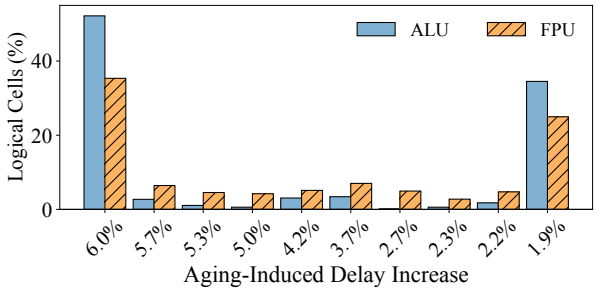

Figure 8. The distribution of aging-induced delay increase for the logical cells in FPU and ALU.

Unit	WNS / # of Violated Paths	
	Setup	Hold
ALU	-76ps / 11	— / 0
FPU	-157ps / 1,363	-1ps / 3

Table 3. STA Result with Aging-Aware Timing Libraries.

5.2 Effectiveness of Vega

5.2.1 Potential Aging Identification. Despite the ALU and FPU being correctly placed-and-routed, initially meeting required timing constraints, Vega reveals that over an extended period of usage (10 years in our analysis), transistor aging has the potential to break these constraints. Figure 8 shows a histogram of the distribution of transistor aging delay increases for both the FPU and ALU. The figure illustrates that aging degradation is not uniformly distributed across logical cells. Specifically, 52% and 35% of the logical cells in the ALU and FPU, respectively, experience a 6% delay increase, and 35% and 25% of the cells exhibit a delay increase of 1.9%. The remaining cells experience a delay increase ranging from 2.2% to 5.7%. Table 3 summarizes the worst negative slack (WNS) and the number of identified timing violations within the ALU and FPU after 10 years of aging. In summary, Vega identifies 11 aging-prone paths in the ALU, and 1,366 such paths in the FPU.

Many of these aging-prone paths share the same pairs of starting and ending points, indicating that these paths would exhibit the same misbehavior under the failure model we employ (§3.3.1). Filtering these paths, Vega recognizes 6 unique pairs of starting and ending points for the ALU and 41 pairs for the FPU. Therefore, for the rest of our analysis and test case generation, we only use one representative failing path for each unique pair of starting and ending points.

5.2.2 Test Case Construction. For each unique pair of starting and ending points, Vega invokes formal verification

Unit	w/o Mitigation (%)				w/ Mitigation (%)			
	S	UR	FF	FC	S	UR	FF	FC
ALU	66.7	33.3	0	0	33.3	66.7	0	0
FPU	51.2	43.9	4.9	0	40.2	43.9	8.5	7.3

Table 4. Result of Test Case Construction. "S" denotes the successful construction of a test case; "UR" indicates that the formal verification tool proves that the failing path cannot cause an actual error; "FF" indicates a timeout occurred in the formal verification tool; "FC" indicates a waveform is generated while Vega fails to convert it to a test case.

to produce a set of waveforms that activate this failing path in an observable manner, and then converts these waveforms into a few test cases. Depending on configuration, different numbers of test cases may be generated. When the mitigation for initial value dependency (Section 3.3.4) is disabled, Vega produces a maximum of 2 test cases for each pair, attributable to the failure model's constant, C, which can be either 0 or 1. With the mitigation enabled, Vega generates a maximum of 4 test cases per pair, in order to account for different signal transitions (i.e., rising or falling) in the starting point.

Table 4 presents the effectiveness of this process. Without the mitigation, Vega can construct the test cases for 66.7% and 51.2% of these unique pairs of endpoints identified in the ALU and FPU respectively. Additionally, it formally proves that 33.3% of the pairs in the ALU and 43.9% of the pairs in the FPU will not cause an actual error—no allowable set of inputs to the module can trigger the timing violation for these paths. Enabling the mitigation reduces the proportion of test cases that can be successfully generated. However, because it generates up to twice as many test cases, it can produce a more robust test suite.

In some instances, we observed that Vega may produce a waveform that is not convertible into a practical test case. These instances are indicated as "FC" in Table 4. All such instances occur with the FPU. This situation happens because certain failures require multiple instructions to propagate an error to the output; moreover, the only detectable erroneous output is a status flag (e.g., a flag indicating an overflow), which is already altered by a prior instruction. As a result, Vega cannot compare this output against a correct value, making the conversion impossible.

Table 5 shows the total number of test cases generated by Vega and the corresponding CPU cycles required for their execution, both in scenarios with and without the mitigation for initial value dependency. Notably, a complete execution of these test cases consumes only a few hundred to a couple thousand cycles, thereby making frequent testing practical.

5.2.3 Quality of Test Cases. We evaluate the quality of these test cases by simulating them against the failing netlists produced by failure model instrumentation (Section 3.3.2). For each failing netlist associated with one of the generated

Unit	w/o Mitigation		w/ Mitigation	
	Test Cases	Cycles	Test Cases	Cycles
ALU	8	124	8	134
FPU	42	685	66	1202

Table 5. The quantity of test cases generated and the number of CPU cycles required for their execution.

Unit	FM	w/o Mitigation (%)				w/ Mitigation (%)			
		Det.	B	L	S	Det.	B	L	S
ALU	0	100.0	50.0	0	0	100.0	50.0	0	0
	1	100.0	75.0	0	0	100.0	50.0	0	0
	R	100.0	75.0	0	0	100.0	50.0	0	0
FPU	0	95.4	72.7	4.5	9.1	100.0	72.7	0	9.1
	1	95.4	81.8	9.1	0	100.0	81.8	4.5	0
	R	95.4	72.7	4.5	9.1	95.4	72.7	4.5	9.1

Table 6. The quality of the generated test cases measured by their ability to detect failures. "FM" refers to the failure mode used in the experiment; "Det." indicates the failures that are detectable by one of the test cases; "B" represents the failures detected by a test case that executed before the test case designed to detect it; "L" represents the failures that are not detected by their corresponding test case, but are identified by later test case; "S" indicates cases where the failure results in the CPU stalling.

test cases, we run the entire suite of test cases to see whether the suite can detect the failure. As mentioned in experimental setup, we configure each failing netlist to fail in three modes: with C held to 0, 1, or taking a random value at each cycle.

Table 6 shows the result of this experiment. In summary, the test cases generated by Vega are generally effective in detecting their intended failures. Interestingly, in many cases, a failure is identified by a test case designed for another failure, before its own corresponding test case is scheduled to execute. In rare cases, a failure may be missed by its own test case; however, it is very likely to be identified by a subsequent test case. In two instances, the failure occurs on a set of ready/valid handshake signals, causing the CPU to stall as it waits for data that cannot be transmitted. From the software's perspective, the application does not progress or respond to interrupts, thus making the failure detectable. In one particular instance, a failure remained undetected after the execution of the entire set of test cases. This occurs because the test cases generated for this failure depend on certain initial signal values to be effective. Unfortunately, these values are modified by a prior instruction, thereby preventing the failure from being detected. However, using the mitigation technique described in Section 3.3.4, Vega can generate a test suite that successfully detects this failure when C is held to a fixed value. When C's value changes randomly at each cycle, the test suite fails to detect one

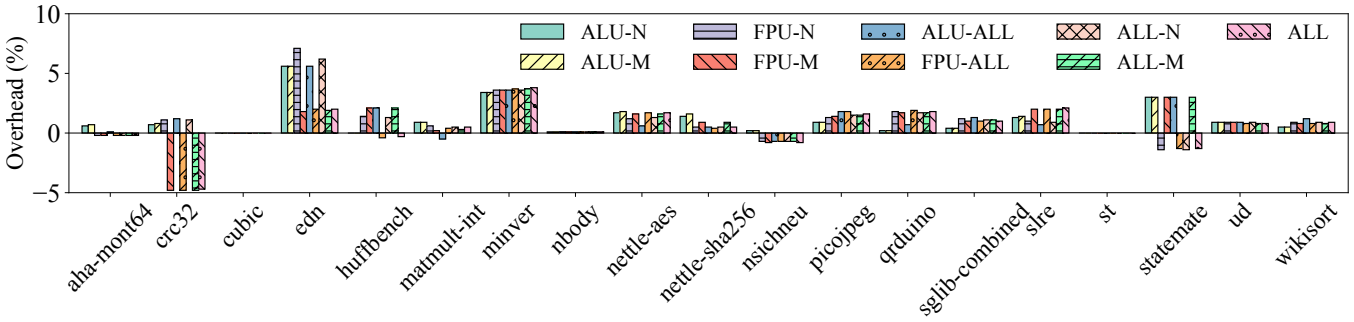

Figure 9. Performance overhead of the embench benchmark set with Vega's Profile-Guided Test Integration. The "-M" and "-N" labels indicate that only the test cases generated with and without the mitigation technique are enabled, respectively.

Unit	FM	Vega	Random
ALU	0	100%	50.0%
	1	100%	50.0%
	R	100%	45.0%
FPU	0	100%	35.3%
	1	100%	95.4%
	R	95.4%	97.2%

Table 7. Effectiveness of Vega-generated vs. random test cases, measured by their ability to detect failures.

failure due to the resulting randomness in fault visibility for a tested instruction's outputs.

To establish a fair baseline for comparison, we implemented a random test suite generator that produces test cases in the style and quantity of Vega's trace-generated test cases: each case verifies the functional correctness of a single random instruction from the current module's instruction set, using random inputs. For each experiment, we generate a test suite of these random tests, and we execute the suite in the same environment used to evaluate Vega. We run 10 experiments against each module and failure model (where C was set to different values). The average number of faults detected across these experiments is shown in Table 7. In most scenarios, Vega outperforms the random test case generator in producing effective test cases. Surprisingly, when C is held to 1 or takes random values, randomly generated test cases can also be effective for the FPU, detecting up to 97.2% of failures. However, this approach lacks Vega's unique benefit of formally proving certain failures would never occur.

5.3 Efficiency of Vega

We evaluate the performance overhead of Vega's Profile-Guided Test Integration by comparing the execution time of the benchmarks instrumented with the test cases against their baseline. We adopt a variety of configurations, with different configurations enabling different sets of test cases.

Figure 9 presents the overhead. On average, Vega's Profile-Guided Test Integration introduces 0.8% overhead. In many instances, this overhead is so negligible that it becomes indistinguishable with environmental noise (e.g., compiler optimizations accidentally triggered as a side effect of the instrumentation), resulting in a negative overhead. Therefore, we conclude that Vega's Profile-Guided Test Integration can effectively manage and minimize its performance overhead.

6 Discussion

We now discuss the design decisions and trade-offs of Vega and other frameworks for SDC detection (§6.1), the impact of environmental noises (§6.2), and future directions (§6.3).

6.1 Vega Versus Top-Down Frameworks

Current frameworks for SDC detection, such as SiliFuzz [69] and OpenDCDiag [1], utilize a top-down approach. These frameworks treat the underlying hardware as a black box, relying on fuzzing or analysis of popular software libraries and applications to generate test cases. Consequently, they often produce a large volume of tests aimed at uncovering a broad spectrum of potential hardware failures.

Vega's novelty lies in its targeted, bottom-up approach, focusing solely on identifying aging-related hardware failures. To achieve this, it uses low-level implementation details to guide test generation. This focus allows Vega to produce a significantly smaller test suite. As a result, these tests can be executed very frequently (e.g., per second), enabling proactive and real-time monitoring of hardware health.

We view Vega and these top-down frameworks as orthogonal tools: their different approaches and targeting provide different strengths. Although these differences hinder a direct comparison of framework utility, with the difficulty of porting architecture-specific test frameworks to our RISC-V CPU an additional burden, we can envision scenarios where multiple frameworks can be used in tandem to target different testing needs. For example, a data center operator could leverage Vega to embed targeted tests within their software for continuous monitoring, while long-running test suites generated by top-down frameworks could be scheduled during regular maintenance cycles for broader coverage.

6.2 Impact of Environmental Noises

Environmental noises, such as voltage and temperature, may influence the effectiveness of Vega-generated test cases. During Aging Analysis, Vega relies on worst-case temperature and voltage assumptions when determining signal propagation delays, which may result in an overestimation of likely hardware faults. A more detailed analysis that considers real-world power delivery and temperature variations within the circuit could potentially help eliminate these false positives.

Current test suite deployment in data centers uses a few approaches to mitigate environmental noises, including test repetition, frequent execution, and execution while carefully varying relevant physical conditions [36, 75]. These techniques are applicable to Vega, and Vega's short test suite length is conducive to timely tests that reduce environmental variance between a period of testing and code execution.

6.3 Future Research Directions

While we present Vega as a complete, end-to-end prototype for the bottom-up workflow, each of its three phases holds the potential for future improvement.

The Aging Analysis phase can be expanded to analyze further circuit reliability issues, such as dynamic IR drop and electromigration. Similar to transistor aging, these issues have also been well-studied at the transistor and gate level.

Future research could also explore alternative test case generation methods during Error Lifting. One avenue involves fast exploration of useful test cases via random and fuzzing-based methods. Harnessing insights from Aging Analysis, we could also develop efficient filtering techniques to identify the most effective test cases for SDC testing.

Moreover, Instruction Construction could be simplified by automating test case generation using program synthesis and machine learning. This would reduce the manual effort involved in adapting Vega's workflow to a new module.

Finally, future research could investigate configuring Vega for a commercial setting, where chip manufacturers generate test suites for data center operators. Concurrently, data center operators could collect valuable traces and statistics within their own environment. Chip manufacturers could then harness this data to refine Aging Analysis and generate a test suite tailored for specific data center workloads.

7 Related Work

Analyses of SDCs. Previous research studied radiation-induced SDCs, including in storage systems [24], memory devices [58], FPGAs [68], HPC systems [39, 41, 62], and satellite processors [77]. Recently, hyperscalers have reported SDCs caused by malfunctioning CPUs in both case studies and comprehensive analyses at the scale of data centers [23, 36–38, 52, 69, 75]. Vega is inspired by these studies, and focuses on the detection of aging-related SDCs inside CPUs.

Detection of SDCs. Data center operators identify SDCs in their CPU populations by conducting infrequent or low priority tests [69, 75]. In certain systems, SDCs are detected through checksums or by verifying whether the computation result is reasonable [23, 50]. Additionally, SDCs can be detected by introducing redundancies in software or hardware designs [21, 28, 34, 74]. Vega focuses on generating compact test cases for frequent, at-scale SDC testing in data centers.

Test Case Construction for SDCs. OpenDCDiag [1] detects SDCs by using popular software libraries. Google's SiliFuzz [69] synthesizes test cases by fuzzing a functional model of the CPU. Unlike these works, Vega constructs test cases by analyzing the CPU's low-level implementation.

Analyses of Transistor Aging. Previous work has analyzed the effect of transistor aging on various hardware designs, such as CPUs [43], GPUs [45], and FPGAs [16], as well as a variety of hardware building blocks, such as standard cell libraries [18, 53] and SRAMs [35]. In this paper, we focus on transistor aging within CPUs; however, Vega's design and insights can be applied to other hardware designs.

Hardware Mitigations for Transistor Aging. Prior work has explored transistor aging mitigation techniques that operate across multiple phases of the hardware design process. These works include the design of aging-resistant microprocessors [43] and SRAM caches [30], as well as EDA tools for simulating, analyzing, and mitigating the impact of transistor aging [31, 54, 66, 71, 72]. Unlike these works, Vega analyzes a hardware design for the sake of building better application-level detection techniques for transistor aging.

Software Mitigations for Transistor Aging. Previous research on software mitigations for transistor aging includes inserting NOP instructions [42] and scheduling anti-aging programs during processors' idle periods [11]. Vega is orthogonal to these works, and provides a systematic way to provoke specific aging-related failures within a CPU design.

8 Conclusions

In this paper, we presented Vega, a bottom-up workflow that bridges the gap between the physical understanding of transistor aging and the software detection of aging-related SDCs. Vega targets the most aging-prone components within a CPU and generates a compact test suite that only takes hundreds to thousands of cycles to execute, therefore enabling frequent detection of aging-related SDCs at application runtime.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Iulian Neamtiu, for their valuable feedback. This work is supported by Intel TSA Center and PRISM Research Center, a JUMP Center cosponsored by SRC and DARPA.

References

[1] https://github.com/opendcdiag/opendcdiag.

[2] https://www.zlib.net/.

[3] https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html.

[4] https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html.

[5] https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html.

[6] https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[7] https://github.com/embench/embench-iot.

[8] https://github.com/openhwgroup/corev-llvm-project.

[9] Aging problems at 5nm and below. https://semiengineering.com/aging-problems-at-5nm-and-below/.

[10] Intel® data center diagnostic tool for intel® xeon® processors. https://www.intel.com/content/www/us/en/support/articles/000058107/processors/intel-xeon-processors.html.

[11] Haider Muhi Abbas, Mark Zwolinski, and Basel Halak. Aging mitigation techniques for microprocessors using anti-aging software. *Ageing of Integrated Circuits: Causes, Effects and Mitigation Techniques*, pages 67–89, 2020.

[12] Mridul Agarwal, Bipul C Paul, Ming Zhang, and Subhasish Mitra. Circuit failure prediction and its application to transistor aging. In *25th IEEE VLSI Test Symposium (VTS'07)*, pages 277–286. IEEE, 2007.

[13] M. A. Alam and C. Augustine K. Roy. Reliability- and process-variation aware design of integrated circuits—a broader perspective. In *2011 International Reliability Physics Symposium, Monterey, CA, USA*, pages 4A.1.1–4A.1.11, 2011. doi: 10.1109/IRPS.2011.5784500.

[14] M. A. Alam and S. Mahapatra. A comprehensive model of pmos nbti degradation. *Microelectronics Reliability*, 45:71–81, 2005. https://doi.org/10.1016/j.microrel.2004.03.019.

[15] M.A. Alam, H. Kufluoglu, D. Varghese, and S. Mahapatra. A comprehensive model for pmos nbti degradation: Recent progress. *Microelectronics Reliability*, 47(6):853–862, 2007. Modelling the Negative Bias Temperature Instability.

[16] Abdulazim Amouri, Florent Bruguier, Saman Kiamehr, Pascal Benoit, Lionel Torres, and Mehdi Tahoori. Aging effects in fpgas: An experimental analysis. In *2014 24th international conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2014.

[17] Hussam Amrouch. *Techniques for aging, soft errors and temperature to increase the reliability of embedded on-chip systems*. PhD thesis, Karlsruhe, Karlsruher Institut für Technologie (KIT), Diss., 2015, 2015.

[18] Hussam Amrouch, Behnam Khaleghi, Andreas Gerstlauer, and Jörg Henkel. Reliability-aware design to suppress aging. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[19] Hussam Amrouch, Javier Martin-Martinez, Victor M van Santen, Miquel Moras, Rosana Rodriguez, Montserrat Nafria, and Jörg Henkel. Connecting the physical and application level towards grasping aging effects. In *2015 IEEE International Reliability Physics Symposium*, pages 3D–1. IEEE, 2015.

[20] Md Toufiq Hasan Anik, Sylvain Guilley, Jean-Luc Danger, and Naghmeh Karimi. On the effect of aging on digital sensors. In *2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID)*, pages 189–194. IEEE, 2020.

[21] Sanem Arslan and Osman Unsal. Efficient selective replication of critical code regions for sdc mitigation leveraging redundant multithreading. *The Journal of Supercomputing*, 77(12):14130–14160, 2021.

[22] Altug Hakan Baba and Subhasish Mitra. Testing for transistor aging. In *2009 27th IEEE VLSI Test Symposium*, pages 215–220. IEEE, 2009.

[23] David F Bacon. Detection and prevention of silent data corruption in an exabyte-scale database system. 2022.

[24] Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Garth R Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS)*, 4(3):1–28, 2008.

[25] 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 Journal of Research and Development*, 50(4.5):433–449, 2006.

[26] S. Bhardwaj, W. Wang, R. Vattikonda, Y. Cao, and S. Vrudhula. Predictive modeling of the nbti effect for reliable design. In *Proceedings of the Custom Integrated Circuits Conference*, pages 189–192, 2006. https://doi.org/10.1109/CICC.2006.320885.

[27] David Blaauw, Kaviraj Chopra, Ashish Srivastava, and Lou Scheffer. Statistical timing analysis: From basic principles to state of the art. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(4):589–607, 2008.

[28] Yuriy Brun, George Edwards, Jae Young Bang, and Nenad Medvidovic. Smart redundancy for distributed computation. In *2011 31st International Conference on Distributed Computing Systems*, pages 665–676. IEEE, 2011.

[29] A. Calimera, M. Loghi, E. MacIi, and M. Poncino. Aging effects of leakage optimizations for caches. In *Proceedings of the ACM Great Lakes Symposium on VLSI, GLSVLSI*, pages 95–98, 2010. https://doi.org/10.1145/1785481.1785504.

[30] Andrea Calimera, Mirko Loghi, Enrico Macii, and Massimo Poncino. Dynamic indexing: Leakage-aging co-optimization for caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(2):251–264, 2014.

[31] Kueing-Long Chen, Stephen A Saller, Imelda A Groves, and David B Scott. Reliability effects on mos transistors due to hot-carrier injection. *IEEE Transactions on Electron Devices*, 32(2):386–393, 1985.

[32] Automotive Electronics Council. *AEC - Q100-REV-G: Failure mechanism based stress test qualification for integrated circuit*. AAEC – Q100 – REV-G standard.

[33] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, 2017.

[34] Ádria Barros de Oliveira, Lucas Antunes Tambara, and Fernanda Lima Kastensmidt. Applying lockstep in dual-core arm cortex-a9 to mitigate radiation-induced soft errors. In *2017 IEEE 8th Latin American Symposium on Circuits & Systems (LASCAS)*, pages 1–4, 2017.

[35] Jie Ding, Dave Reid, Plamen Asenov, Campbell Millar, and Asen Asenov. Influence of transistors with bti-induced aging on sram write performance. *IEEE Transactions on Electron Devices*, 62(10):3133–3138, 2015.

[36] Harish Dattatraya Dixit, Laura Boyle, Gautham Vunnam, Sneha Pendharkar, Matt Beadon, and Sriram Sankar. Detecting silent data corruptions in the wild. *arXiv preprint arXiv:2203.08989*, 2022.

[37] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245*, 2021.

[38] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The {rocksdb} experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49, 2021.

[39] Jack Dongarra, Thomas Herault, and Yves Robert. *Fault tolerance techniques for high-performance computing*. Springer, 2015.

[40] Mohamed A Elgamel and Magdy A Bayoumi. Eda industry tools: State of the art. *Interconnect Noise Optimization in Nanometer Technologies*, pages 107–123, 2006.

[41] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC'12:*

Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pages 1–12. IEEE, 2012.

[42] Farshad Firouzi, Saman Kiamehr, and Mehdi B Tahoori. Nbti mitigation by optimized nop assignment and insertion. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 218–223. IEEE, 2012.

[43] Freddy Gabbay and Avi Mendelson. Asymmetric aging effect on modern microprocessors. *Microelectronics Reliability*, 119:114090, 2021.

[44] Freddy Gabbay, Firas Ramadan, and Majd Ganaiem. Clock tree design considerations in the prescence of asymmetric transistor aging. In *20203 10th Design and Verification Conference - Europe (DVCON)*, 2023.

[45] Freddy Gabbay, Firas Ramadan, Majd Ganaiem, Ofrie Rosenthal, and Lior Bashari. Effect of Asymmetric Transistor Aging on GPGPUs. In *Proceedings of the 5th International Conference on Microelectronic Devices and Technologies (MicDAT '2023)*, pages 52–56, 2023.

[46] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[47] Ran Ginosar. Metastability and synchronizers: A tutorial. *IEEE Design & Test of Computers*, 28(5):23–35, 2011.

[48] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[49] Wenwen He and Kelly Yang. Quality control of subcontractor management in wafer foundry. In *2013 IEEE International Conference on Industrial Engineering and Engineering Management*, pages 645–649, 2013.

[50] Yi He, Mike Hutton, Steven Chan, Robert De Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. Understanding and mitigating hardware failures in deep learning training systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–16, 2023.

[51] Jeffrey Hicks, Daniel Bergstrom, Mike Hattendorf, Jason Jopling, Jose Maiz, Sangwoo Pae, Chetan Prasad, and Jami Wiedemer. 45nm transistor reliability. *Intel Technology Journal*, 12(2), 2008.

[52] Peter H Hochschild, Paul Turner, Jeffrey C Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 9–16, 2021.

[53] Saman Kiamehr, Farshad Firouzi, Mojtaba Ebrahimi, and Mehdi B Tahoori. Aging-aware standard cell library design. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–4. IEEE, 2014.

[54] Anand T Krishnan, Frank Cano, Cathy Chancellor, Vijay Reddy, Zhangfen Qi, Palkesh Jain, John Carulli, Jonathan Masin, Steve Zuhoski, Srikanth Krishnan, et al. Product drift from nbti: Guardbanding, circuit and statistical effects. In *2010 International Electron Devices Meeting*, pages 4–3. IEEE, 2010.

[55] S. V. Kumar, C. H. Kim, and S. S. Sapatnekar. An analytical model for negative bias temperature instability. In *2006 IEEE/ACM International Conference on Computer Aided Design, San Jose, CA, USA*, pages 493–496, 2006. doi: 10.1109/ICCAD.2006.320163.

[56] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[57] Changze Liu, Yongsheng Sun, Pengpeng Ren, Dan Gao, Weichun Luo, Zanfeng Chen, and Yu Xia. New challenges of design for reliability in advanced technology node. In *2020 4th IEEE Electron Devices Technology & Manufacturing Conference (EDTM)*, pages 1–4. IEEE, 2020.

[58] Lucas Matana Luza, Daniel Söderström, Helmut Puchner, Rubén García Alía, Manon Letiche, Alberto Bosio, and Luigi Dilillo. Effects of thermal neutron irradiation on a self-refresh dram. In *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. IEEE, 2020.

[59] Stefan Mach, Fabian Schuiki, Florian Zaruba, and Luca Benini. Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(4):774–787, 2020.

[60] D. MacMillen, R. Camposano, D. Hill, and T.W. Williams. An industrial view of electronic design automation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1428–1448, 2000.

[61] Erik Jan Marinissen, Gilbert Vandling, Sandeep Kumar Goel, Friedrich Hapke, Jason Rivers, Nikolaus Mittermaier, and Swapnil Bahl. Eda solutions to new-defect detection in advanced process technologies. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 123–128, 2012.

[62] S.E. Michalak, K.W. Harris, N.W. Hengartner, B.E. Takala, and S.A. Wender. Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, 2005.

[63] Laurence W. Nagel and D.O. Pederson. Spice (simulation program with integrated circuit emphasis). Technical Report UCB/ERL M382, EECS Department, University of California, Berkeley, Apr 1973.

[64] S Novak, C Parker, D Becher, M Liu, Marty Agostinelli, M Chahal, P Packan, P Nayak, Stephen Ramey, and S Natarajan. Transistor aging and reliability in 14nm tri-gate technology. In *2015 IEEE International Reliability Physics Symposium*, pages 2F–2. IEEE, 2015.

[65] Fabian Oboril and Mehdi B Tahoori. Extratime: Modeling and analysis of wearout due to transistor aging at microarchitecture-level. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

[66] Shigeo Ogawa and Noboru Shiono. Generalized diffusion-reaction model for the low-field charge-buildup instability at the si-sio$_2$ interface. *Phys. Rev. B*, 51:4218–4230, Feb 1995.

[67] C. L. Portmann and T. H. Y. Meng. Metastability in cmos library elements in reduced supply and technology scaled applications. *IEEE J Solid-State Circuits*, 30:39–46, 1995. https://doi.org/10.1109/4.350196.

[68] H. Quinn and P. Graham. Terrestrial-based radiation upsets: a cautionary tale. In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 193–202, 2005.

[69] Kostya Serebryany, Maxim Lifantsev, Konstantin Shtoyk, Doug Kwan, and Peter Hochschild. Silifuzz: Fuzzing cpus by proxy. *arXiv preprint arXiv:2110.11519*, 2021.

[70] Wilson Snyder. https://www.veripool.org/verilator/, 2021.

[71] Rakesh Vattikonda, Wenping Wang, and Yu Cao. Modeling and minimization of pmos nbti effect for robust nanometer design. In *Proceedings of the 43rd annual Design Automation Conference*, pages 1047–1052, 2006.

[72] Jyothi Bhaskarr Velamala. Compact modeling and simulation for digital circuit aging. PhD dissertation, 2012. https://repository.asu.edu/attachments/97628/content//tmp/package-sAOTlT/Velamala_asu_0010E_12271.pdf.

[73] Steve Wadsworth and Dennis Brophy. Advanced asic sign-off features of ieee 1076.4-2000 and standards updates to verilog and sdf. In *System on Chip Design Languages: Extended papers: best of FDL'01 and HDLCon'01*, pages 35–42. Springer, 2002.

[74] Guosai Wang, Lifei Zhang, and Wei Xu. What can we learn from four years of data center hardware failures? In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 25–36. IEEE, 2017.

[75] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. Understanding silent data corruptions in a large production cpu population. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 216–230, 2023.

[76] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[77] Kenneth M. Zick, Chien-Chih Yu, John Paul Walters, and Matthew French. Silent data corruption and embedded processing with nasa's spacecube. *IEEE Embedded Systems Letters*, 4(2):33–36, 2012.