DelayAVF: Calculating Architectural Vulnerability Factors for Delay Faults

Peter W. Deutsch*
MIT
Cambridge, USA
pwd@mit.edu

Vincent Quentin Ulitzsch*
MIT/TU Berlin
Berlin, Germany
viniul@mit.edu

Sudhanva Gurumurthi
Advanced Micro Devices, Inc.
Austin, USA
sudhanva.gurumurthi@amd.com

Vilas Sridharan
Advanced Micro Devices, Inc.
Boxborough, USA
vilas.sridharan@amd.com

Joel S. Emer
MIT
Cambridge, USA
emer@csail.mit.edu

Mengjia Yan
MIT
Cambridge, USA
mengjiay@mit.edu

*Both authors contributed equally to this research.

***Abstract*—Reliability is a key design consideration for modern microprocessors. A surge of reports from major cloud vendors describing new silent data corruption (SDC) behaviours at scale suggest a recent change in the nature of faults in the wild. Recent publications have suggested that one root cause of these SDCs may be small delay faults (SDFs) induced by *marginal defects* that increase a circuit's propagation time by a small (sub-cycle) delay. Reasoning about the effects of these faults early in the design of a processor is thus of increasing importance for reliability at-scale.**

Computer architects currently reason about the resilience of microarchitectures against *particle strike induced* faults using Architectural Vulnerability Factor (AVF) which describes the probability that a particle strike impacting a particular microarchitectural structure results in a program-visible failure. In this paper, we develop an AVF-like metric to quantify a processor's vulnerability to SDFs. We conduct a systematic analysis of the potential impacts of SDFs and determine that particle strike AVF is insufficient to reason about SDFs. Considering SDFs requires additional reasoning about the timing characteristics of a circuit, the state element(s) that experience an error due to a fault, and whether the resulting state element errors cause a program-visible failure.

In this paper we present *DelayAVF*, a metric that quantifies microarchitectural vulnerability to small delay faults. We develop a two-step methodology to analyze the DelayAVF of a hardware design. We then analyze the DelayAVF of an open-source RISC-V core, finding new architectural reliability insights that do not present themselves through traditional AVF analysis. Finally, we provide approximations for DelayAVF that allow for the reuse of particle strike AVF data (for instance, from existing fault injection studies).

I. Introduction

Reliability is a key design consideration for modern microprocessors. A significant area of focus to improve reliability has been in the study of particle strike-induced transient faults (sometimes referred to as *soft errors*) caused by neutrons from cosmic rays and alpha particles. These faults have posed a particular threat to microprocessor reliability as they occur randomly in the field and cannot be screened out at test time, making them unpredictable and hard to reproduce. These faults have been observed in the field, occurring on CPUs [34], [46], [51], GPUs [21], [22], and devices such as FPGAs and ASICs [6], [11].

It has recently become evident that particle strikes are no longer the only major contributing factor to errors in the field. Hyperscalers have recently observed an increase in the number of silent data corruptions (SDCs) that cannot be attributed solely to particle strike-induced transient faults [16], [17], [24], [44], [53], leading to industry-wide concerns about system correctness at-scale. Recent industry publications have suggested that one root cause of these newly appearing faults are *marginal defects* [20], [29], [30], [45], [48]. Marginal defects result in additional signal propagation delays under very specific (marginal) conditions, with the locations of these defects and the conditions required for these defects to trigger faulty behaviour being unknown during test time. These marginal defects culminate in *small delay faults* (SDFs) which increase the propagation delay of a signal for a fraction of a clock period [28], [40], [52]. An SDF may in turn result in incorrect values being latched by downstream state elements, potentially resulting in program-visible failures.

Existing test methodologies that aim to root out defective chips during manufacturing [4], [15], [41] are not sufficient to detect all marginal defects, leading to chips with marginal defects being shipped into the field. Chip designers therefore must add resilience to the design to protect against faults with the anticipation that some defects may escape testing. This approach is similar to adding resilience to protect against particle strike-induced transient faults, which inevitably occur due to environmental effects.

Chip designers currently reason about the vulnerability of microarchitectural structures to particle strike-induced transient faults using Architectural Vulnerability Factor (AVF) [33]. A structure's AVF quantifies the probability that a particle strike induced fault in that structure will result in a program-visible failure, providing a metric that allows designers to target resilience where it is most useful. To tractably compute AVF early in the design process, computer architects

perform *ACE analysis*, determining the proportion of bits in a structure whose correctness is required for architecturally correct execution (i.e., are ACE) in each cycle.

An AVF for Small Delay Faults. In this paper we conduct a systematic analysis of the impacts of SDFs across different microarchitectural structures. This analysis reveals that AVF for particle strikes does not provide transferable insights into a structure's vulnerability to SDFs. AVF analysis only reasons about the system-level impact of a single error from a single fault. However, this is insufficient to reason about the impact of an SDF, which may not have a one-to-one mapping between the fault and the resulting error. In contrast to particle strikes, an SDF changes a circuit's timing characteristics. While this change may result in a single state element error, it is also possible for no state element error to occur, e.g., if the delay is too small or the delayed signal's propagation is masked out by combinational logic. In addition, a single SDF can even result in multiple simultaneous state element errors. As a consequence, a structure's vulnerability towards small delay faults cannot be estimated solely using traditional ACE analysis, and a new approach that consider a circuit's timing and state is required.

In this paper we define *DelayAVF*, the probability that a small delay fault in a microarchitectural structure results in a program-visible failure. We define a circuit element (e.g., a wire or gate) to be *DelayACE* in cycle i if a small delay fault in that element during cycle i results in a program-visible failure. The DelayAVF of a structure can then be stated as the average proportion of the structure's circuit elements which are DelayACE during execution.

For DelayAVF to be useful in practice, the determination of whether a circuit element is DelayACE must be computationally tractable. This in itself can be challenging, as examining the consequences of a small delay fault can require computationally expensive timing-aware simulation to track a delay's propagation through a circuit. In this paper, we present a method to determine whether a circuit element is DelayACE in two sub-steps, minimizing the amount of timing-aware simulation required to determine DelayACEness. In our method, timing-aware simulation is only required to determine the set of state elements that experience an error due to an SDF. Determining whether the state element errors result in a program-visible failure can be done in a timing-agnostic manner, similarly to the work by Entrena et al. [18] and Hari et al. [22]. In some cases there may be compounding effects between multiple state element errors, such as when multiple state element errors cancel each other out. In this paper we explicitly enumerate these confounding effects, allowing us to identify their impact. In cases where the impacts of these effects is small, DelayAVF can be approximated by reusing (particle strike) AVF data, which in many cases may already be generated for a design.

As we will demonstrate, computer architects can use DelayAVF to systematically study the impact of SDFs on real hardware designs early on in the design process. DelayAVF analysis provides key architectural insights that can be used to identify structures which are particularly vulnerable to SDFs, helping to guide targeted protections against these faults.

Key Contributions. The key contributions of this paper are:

1) DelayAVF, a methodology to measure the vulnerability of architectural structures to small delay faults (SDFs), capturing the intrinsic circuit and system-level behaviours that result due to these faults.
2) A systematic methodology to compute DelayAVF for real processor designs, utilizing a tractable two-step derivation approach.
3) A case study analysis of the impact of SDFs on various microarchitectural structures in the Ibex RISC-V core [2], revealing key architectural insights.
4) An approximation for DelayAVF which allows for the reuse of existing data from fault injections or pre-existing ACE analysis flows.

Experimental Insights. In studying the Ibex core we find that:

1) Vulnerability to small delay faults can vary significantly across different microarchitectural structures and benchmarks, indicating a strong dependence on architectural and program-level effects.
2) A structure's path timing distribution is not sufficient to reason about the impact of small delay faults. Instead, while the level of vulnerability of a structure is dominated by static circuit timing characteristics for shorter duration SDFs, program and architectural characteristics play a more prominent role for longer duration SDFs.
3) Mitigations that are effective in protecting hardware structures against particle strikes may not be as effective in protecting against SDFs.
4) Approximations to estimate DelayAVF can sometimes provide useful insights, however care should be taken when certain confounding multi-bit error interactions may occur.

Paper Roadmap. Section II provides a background on marginal defects, small delay faults, and existing vulnerability measures and protections. Section III describes why existing particle strike vulnerability measures are insufficient to reason about SDFs, highlighting the need for DelayAVF. Section IV describes how to model the circuit-level impacts of SDFs. Using this model, we formally define DelayAVF and describe how to compute it in Section V. We apply DelayAVF to study the impact of SDFs on an open-source RISC-V core in Section VI, and present a DelayAVF approximation leveraging existing particle strike ACE data in Section VII.

II. Background

A. Terminology

In this paper we use a terminology largely following that of the original AVF work [33]. We begin by highlighting the difference between a *fault*, *defect*, *error*, and *failure*. In this work, a *fault* denotes an undesired change to the hardware's

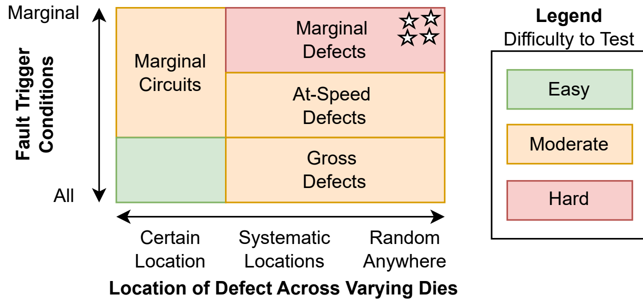

Fig. 1. Characteristics of different circuit defects, adapted from [42]. Colors represent the difficulty to detect the defect during testing. Stars represent this paper's defects of interest, which trigger faults under extremely specific (marginal) operating conditions.

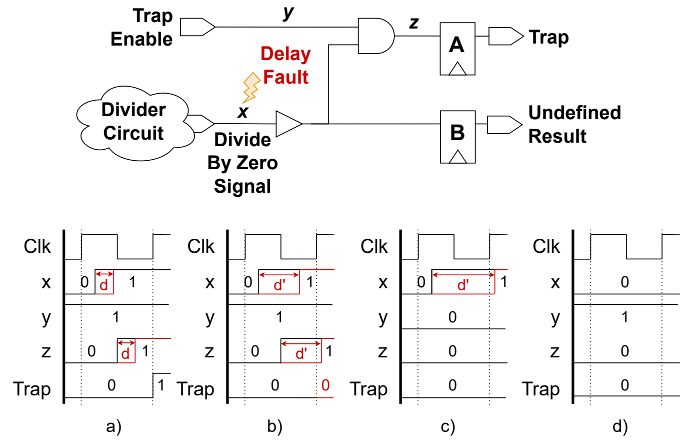

Fig. 2. Example circuit which can experience a state element error due to a small delay fault. a) an added delay d on x results in no error, b) a larger delay d' on x results in an error in A, c) the larger delay d' resulting in no error due to logical masking, d) the signal does not change resulting in no error.

behaviour. For instance, a particle strike can result in a fault wherein there is an unanticipated injection of charge at a given node in a circuit. A fault may also be a result of a *defect*, which is an underlying physical flaw in the hardware. For example, a physical defect on a wire may result in a delay fault, resulting in a signal propagation delay. A fault can result in a *state element error*, causing the value held in a stateful circuit element (e.g., a flip-flop) to be incorrect. For example, if a signal is sufficiently delayed due to a delay fault, an incorrect value may be latched in a downstream flip-flop.

The program-level consequence of a state element error depends on the chip's design and state. A state element error may result in a change in program output, we denote this case as a *program-visible failure*. A state element error may alternatively result in no change to the program's output, resulting in architecturally correct execution, if the state element error is corrected or masked at the circuit/architecture levels.

A program-visible failure can be classified as a *silent data corruption* (SDC) if the program's output is incorrect without any notification by the hardware. Alternatively, a program-visible failure can be a *detected unrecoverable error* (DUE), where the program does not complete successfully but does not produce an incorrect output.

B. Circuit Defects

Defects which result in delay faults under marginal circumstances are a rising concern in industry, and are suspected to be a root cause of recently-documented reliability issues at scale [20], [29], [30], [45], [48]. A delay fault causes a signal on a wire or gate to experience an additional delay (beyond its normal delay) as it propagates through the circuit. In contrast to particle strike induced faults, delay faults can be caused by an underlying defect in the circuit. Circuit defects can be attributed to a wide variety of underlying physical root-causes, including lithographic errors [23], cracks [14], and delamination [13]. At a high level, circuit defects can broadly be classified by two factors: where they occur and under which conditions they cause a fault. A summary of circuit defect classes from [42] is reproduced in Figure 1.

First, defects can be classified based on where they occur across different chips. Defects may occur in the same circuit location across different manufactured chips due to a systematic production error (Figure 1 left) or may exist in random chip locations (Figure 1 right).

Second, defects can be further classified under which operating conditions they induce a fault. Some defects may result in faults under all operating conditions (i.e., a permanent fault, Figure 1 bottom). Other defects may only result in a fault under certain operating conditions, such as specific voltages, frequencies, temperatures, and workload patterns (Figure 1 top). Testing for defects which only result in faults under these *marginal* conditions can be challenging due to the difficulty to replicate the exact (and often unknown) conditions required for a fault to occur.

Randomly located defects which only occur under specific operating conditions are referred to as *marginal defects* (highlighted as the starred region in Figure 1). Identifying all marginal defects at test time requires testing all paths in every circuit at all operating conditions. This may entail years of test time for every chip produced, making such testing economically and practically infeasible. Thus, it is likely that some chips deemed good at test time will in fact contain marginal defects that can cause program-visible failures in the field.

C. Small Delay Faults Resulting from Marginal Defects

Marginal defects can manifest in *small delay faults* (SDFs), resulting in a signal's propagation delay increasing by a small (sub-cycle) duration [28], [52]. To observe the circuit-level impact of an SDF, consider the circuit in Figure 2. This example circuit consists of wires (x, y, and z), an AND gate, and clocked state elements (A and B). Signals propagate through the circuit from the inputs, and these signals have some amount of propagation delay through the circuit. The length of these delays is a function of the physical circuit (e.g.

RC parameters and the strength of the driving gates). Under normal operation, the input signals will propagate through the circuit and arrive at the state elements before the next clock cycle.

A delay fault increases the propagation delay of a signal along a wire, or the output of a gate. Consider some possible outcomes of adding an additional delay on wire x in Figure 2. If the delay is small, the correct signal might still arrive in time (2a). If the delay is larger, the signal may not arrive in time (2b), resulting in an incorrect value being latched in A (a state element error). Despite the delay being large, the signal may be logically masked (2c) resulting in no state element error. Finally, a delay fault cannot affect the correctness of the signal if the signal does not change (2d).

D. Protecting Against Faults

To reduce the impact of a fault, resiliency can be added at both the circuit and architectural levels. At the circuit level, the driving strength of transistors and the layout of the circuit's elements can be changed to reduce the likelihood that a state element error occurs due to a circuit defect. The impact of state element errors can further be mitigated by applying a combination of three architectural strategies: spatial, temporal, and informational redundancy. *Spatial redundancy* uses duplicated components on the chip to perform the same operation (e.g., using Razor [19]). *Temporal redundancy* repeats an operation multiple times, reducing the latency/throughput of the system in exchange for reliability. *Informational redundancy* encodes data in a redundant way, computing on both the raw and redundant forms of the data. Well-established techniques exist to protect storage structures and interconnect, e.g., Hamming codes and Cyclic Redundancy Check codes [26], [49], [55], while arithmetic/logic operations can be protected using residue codes [27], [32], [37].

E. Architectural Vulnerability Factor

A key goal of resilient design is to efficiently protect hardware against faults. The Architectural Vulnerability Factor (AVF) [33] methodology is commonly used to quantify the efficiency of protection for particle strike-induced transient faults. A structure's AVF denotes the probability that a particle strike induced fault in that structure results in a program-visible failure. If the correctness of a specific state element in a given cycle is required for architecturally correct execution (i.e., correct program output), that state element is said to be *ACE*. Using this notion of *ACEness*, the architectural vulnerability factor of a hardware structure H containing a total of B bits running for N cycles is expressed as:

$$AVF(H) = \frac{\sum_{i=1}^{N}[\text{\# of ACE bits in } H \text{ at cycle } i]}{B \cdot N} \quad (1)$$

Computing the AVF for different microarchitectural structures allows computer architects to examine the relative contributions of these structures towards a system's overall failure rate due to particle strikes, and to identify the most cost-effective areas to add resilience. Follow-on work has looked to extend ACE analysis to study broader contexts, including multi-bit transient faults [39], [54], intermittent faults [36], and permanent faults [8]. A full discussion of prior work is presented in Section VIII.

III. Motivation

The emergence of small delay faults as a greater threat to industry-wide reliability calls for methods to evaluate their impact on system-level resilience. In this section we will first outline why methods which reason about the impact of particle strikes do not transfer to study the impact of SDFs. We then systematically reason about the conditions under which an SDF can result in a program-visible failure in Section III-B, leading to a natural definition for $DelayAVF$.

A. Inability of Existing Measures to Study the Impact of SDFs

We will now examine why vulnerability measures that assume a particle strike fault model (such as AVF) are insufficient to reason about vulnerability to small delay faults. Recall that SDFs only change a circuit's timing behavior, while particle strikes result in an immediate bit-flip at the site of the fault. This results in key differences which can be exemplified in the circuit depicted in Figure 2. In this example, if the attached divider circuit detects a divide-by-zero event, it forwards a signal to the core that the computed result is undefined (via Register B). Further, if the trap enable setting is enabled by the core, a divide-by-zero event will signal the core to hang by setting Register A.

For an SDF to cause a state element error, first observe that it must cause the input of downstream state element(s) to transition incorrectly. Assume that in the example circuit the trap enable setting is never enabled by the core and therefore the value stored in Register A (and its corresponding input signal) will never change ('toggle') during execution. In such a case, an SDF on wire x, y, or z can never result in a state element error in A (as A will always latch a 0 due to the AND gate) and thus the state element A should not be considered vulnerable to an SDF. This toggle-dependent error behaviour is not captured under a particle strike model however, as Register A could flip if it is struck, regardless of the value of the trap enable signal. In this scenario Register A could be considered ACE under a particle strike fault model (and contribute to the structure's AVF) while not being vulnerable to SDFs. This behaviour still holds if we additionally consider particle strikes which cause signal bit-flips on wires/gates (such as on wire z).

Second, observe that the delay duration and timing characteristics of a circuit can play a large role in a structure's vulnerability, influencing which state elements could experience a state element error. Let's now consider the case when the trap enable bit is set and again examine the consequences of an SDF on wire x. If the delay is too small, it is possible that no state element errors occur. For a larger delay duration, B may latch a correct value while A latches an incorrect value (due to the additional delay caused by the AND gate), possibly

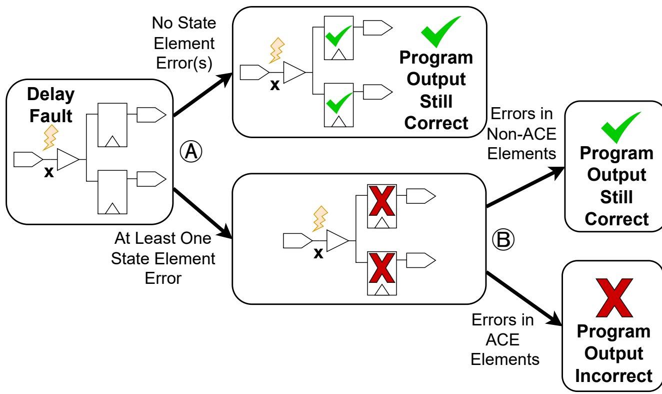

Fig. 3. The conditions required for a small delay fault to result in a program-visible failure.

resulting in an inconsistent architectural state. Particle strike fault models do not describe such a timing-dependent behavior.

Finally, an SDF can result in multiple simultaneous failing state elements, and the group of failing state elements cannot be determined a priori based on their physical positioning since the timing and state of the circuit must also be considered. In the example circuit, it may also be the case that both Register A and Register B hold incorrect values due to an SDF on x, causing the circuit to fail to catch the effects of the divide-by-zero signal entirely. Follow-on work to AVF [54] attempts to reason about simultaneous spatially-adjacent state element errors (e.g., those caused by a single particle strike hitting two or more physically adjacent state elements) in memory arrays. However, while a group of simultaneous spatially-adjacent state element errors can be determined a-priori, the set of failing state elements due to an SDF can change per cycle depending on the circuit's timing and input.

B. Systematically Reasoning About SDFs

The example in Section III-A motivates the need for a dedicated metric that can quantify a structure's vulnerability towards SDFs. To this end, we need to examine the conditions under which an SDF results in a program-visible failure – we summarize the conditions required for this to occur in Figure 3.

First, for a program-visible failure to occur, at least one state element error must occur (Figure 3 Ⓐ). For an SDF to result in an error in a given state element, two sub-conditions must hold. First, the additional delay must induce a path leading to that state element to exceed the clock period in length. Not every SDF which results in a path length exceeding the clock period will result in a state element error. For instance, recall from Figure 2 that logical masking effects can prevent a state element error. Therefore, to result in a state element error, the additional delay must also result in an incorrect value being latched by the state element. Note that a single SDF can affect multiple paths, and therefore potentially result in multiple state element errors. The resultant state element error(s) must then culminate in a failure visible at the program level (Figure 3 Ⓑ).

DelayACE and DelayAVF. To quantitatively reason about vulnerability under the conditions above, we cannot solely reason about whether individual state elements are ACE. Rather, in this paper we will define a new notion of ACEness which allows us to reason about an individual circuit element's (e.g., wire's or gate's) vulnerability to SDFs, which we coin *DelayACE*. Leveraging this notion of DelayACE, we can naturally compute a structure's *DelayAVF* as the average number of circuit elements in a structure H which are DelayACE across all N cycles of a program's execution, as shown in Equation (2).

$$DelayAVF(H) = \frac{\sum_{i=1}^{N}[\text{\# of DelayACE elements in H at cycle } i]}{\text{\# of circuit-elements in } H \cdot N} \quad (2)$$

Using DelayAVF we can identify structures which are particularly vulnerable to SDFs, and deploy targeted mitigations. Analogous to AVF, to estimate the failure rate of a structure, DelayAVF can be multiplied with the rate at which a given structure experiences a small delay fault.

IV. Modeling Small Delay Faults

Prior to formally defining DelayAVF and DelayACE, we now describe the underlying small delay fault model that we will assume throughout the paper. To this end, in the following subsections we will describe how we model 1) a circuit's timing behaviour, 2) when SDFs can manifest due to a marginal defect, and 3) how SDFs can affect the circuit's timing.

A. Circuit Timing Model

In this paper we model a circuit as being composed of clocked state elements, logic elements (e.g., AND, NOR gates), and a set of wires, E, which connect these elements together. This circuit executes a given workload (with fixed inputs) over the course of N cycles. We associate each wire (defined to be between two circuit elements) with a wire-specific signal propagation delay. The signal propagation delay for a wire e is a function of the strength of the circuit element which is driving that wire and the capacitive load that the wire is driving (e.g., the downstream circuit elements). Without loss of generality, we assume that a wire's propagation delay is a fixed data-independent value. Our model is not limited to this assumption and can account for this factor through using state-of-the-art (proprietary) EDA tools such as static timing analysis toolchains. In cases where the varying circuit operating conditions can result in different wire delays (such as when examining different process corners), our model can be repeatedly applied to study fault behaviours across these different delay behaviours.

While our model internally represents delays as being associated with wires, it can easily be utilized to consider faults on gates or state element outputs. To consider SDFs on a circuit element's output (e.g., at the output of a gate or state element), one can introduce a single additional wire x at the output of the element. This wire is connected to the

gate's downstream elements via additional wires which are connected to x. The additional wire will be modeled to have the SDF's delay duration, affecting the propagation delay of the signal to all downstream components. As wire-level delays are a generalization of gate- and state element-level delays, we focus our analysis on wire-level delays for the remainder of the paper.

B. Manifestation of Small Delay Faults

Marginal defects occur in random circuit components and only briefly manifest in SDFs under very specific operating conditions [42]. In studying the effects of marginal defects, we leverage the fact that these defects are non-deterministic in space and time. Since marginal defects do not systematically affect circuits in the same location, we model marginal defects as occurring randomly across the circuit's wires. We make the assumption that a circuit will have a single defect which evaded testing, in line with standard reliability methodologies [3]. There is an extreme sensitivity of marginal defects to specific operating conditions [42]. As such, it is unlikely that the conditions required for a fault hold for longer than the clock period. Therefore, we model SDFs which result from these defects to only affect the circuit for a single cycle.

C. Delay Characteristics of SDFs

Upon an SDF on a wire, we assume a wire's propagation delay experiences an *additional* delay lasting for less than one cycle. We denote the duration of the added delay as d, where d is a small delay adding less than a clock period of additional delay. If a wire's value does not change (i.e., 'toggle') when a fault occurs, the added d has no effect. If a state element latches an incorrect value due to an added delay, we denote this case as a state element error. It may be the case that a single delay results in *multiple* state element errors – for instance, if one wire is on multiple paths to different state elements.

The consideration of delay duration d is an important factor as it affects the set of downstream state elements which can experience errors. If a chip designer does not know an appropriate value of d ahead of time (e.g., if they are designing for a new process node) they can examine the entire space of d (ranging from $0\% - 100\%$ of the clock period) to see how different delays may manifest in program-visible failures. A chip designer may also be able to ascertain an appropriate value of d via the examination of defect-induced delays on existing defective chips. To do so, soft defect localization (SDL) techniques can be used to identify actual delay defects in silicon, using techniques such as optical analysis [9] and laser stimulation [12]. These SDL techniques can be used in conjunction with additional testing data to reproduce the conditions under which the part failed. The resistance associated with the identified defect can then be derived and mapped to a delay magnitude d.

V. DelayAVF: A Delay Fault-Aware AVF

Equipped with a model of the timing characteristics of a circuit and the SDFs that can occur, we now formalize the definition and computation of DelayACE and DelayAVF. In Section V-A, we provide a definition for when a circuit element is DelayACE, and how to use this notion to compute a structure's DelayAVF. In Section V-B we then present a two-step approach to tractably compute these metrics.

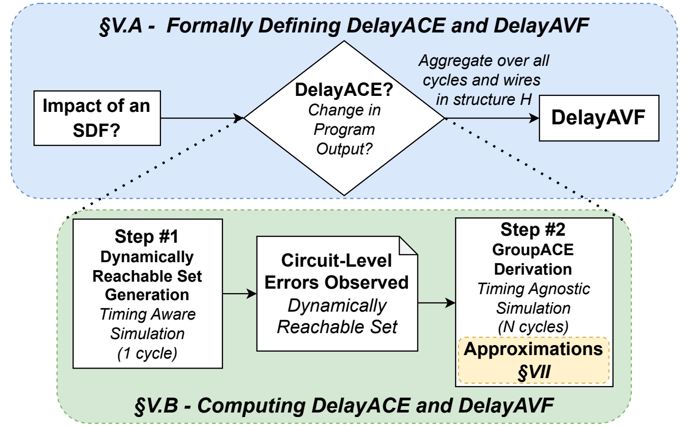

Fig. 4. Our two-step approach to determine if a circuit element is DelayACE, leading to a derivation of DelayAVF.

A. Formally Defining DelayACE and DelayAVF

We begin by defining what it means for a circuit element to be DelayACE. As reasoning about wires allows us to reason about delays at arbitrary circuit elements (as described in Section IV-A), we will focus on determining the DelayACEness of wires for the remainder of the paper.

Definition 1: A wire e is $DelayACE$ in cycle i if a small delay fault adding an additional delay of duration d on that wire in cycle i results in a program-visible failure.

Given the definition of DelayACE, the DelayAVF of a structure naturally follows from Equation (2). Recall that the DelayAVF describes the average number of DelayACE wires in a given microarchitectural structure over all N cycles of a program's execution. For a structure H, we let E be the set of wires associated with the structure. The structure's DelayAVF can then be formulated as:

$$DelayAVF_d(E) = \frac{\sum_{i=1}^{N} \sum_{e \in E} DelayACE_d(e, i)}{N \cdot |E|} \quad (3)$$

where $DelayACE_d(e, i)$ evaluates to 1 if wire e is DelayACE in cycle i with respect to an SDF of duration d, and 0 otherwise.

B. Computing DelayACE and DelayAVF

One strategy to determine whether a wire is DelayACE is to perform a full timing-aware circuit simulation (e.g., via SPICE). Using this simulation, we can study whether an additional delay d on wire e in cycle i results in a change to the program-level output. This strategy is computationally intractable however, as one has to simulate the circuit in a timing-aware fashion for the entire execution.

To make the computation more efficient, the determination of whether a wire is DelayACE can be further broken down into two sub-steps, as shown in Figure 4. First, in a timing-aware step, the set of state element errors which occur due to the SDF (the *dynamically reachable set*) is determined. Second, in a timing-agnostic step, we determine whether the resulting state element errors actually result in a program-visible failure (i.e., whether the dynamically reachable set is *GroupACE*). This two-step approach to computing DelayACE can be encapsulated using the following formula:

$$DelayACE_d(e,i) = GroupACE(DynamicReachable_d(e,i), i+1) \quad (4)$$

In other words, a wire e is DelayACE if the set of state elements which is dynamically reachable due an SDF on e in cycle i is GroupACE in cycle $i+1$. Using the two-step approach is an exact way to determine whether a wire is DelayACE, and is not a heuristic. We now describe the two components in this formula in more detail.

Step #1: Identifying the Dynamically Reachable Set. Our first step is to determine which state elements experience an error due to an SDF. For a state element error to occur, an added delay must induce at least one path in the circuit to have a propagation delay which statically exceeds the clock period.

> **Definition 2:** A state element is statically reachable with respect to an SDF of duration d on wire e in cycle i if it terminates a path exceeding the clock period as a result of the SDF. We call the set of state elements that are statically reachable with respect to an SDF that SDF's *statically reachable set*.

Further, an additional requirement for a state element error to occur is that the state element latches a wrong value due to the delay. We denote the set of state elements which latch an incorrect value as a result of an SDF as *dynamically reachable*.

> **Definition 3:** The dynamically reachable set $\mathcal{S}$ of an SDF of duration d on wire e in cycle i is the set of state elements which are statically reachable and also latch an incorrect value due to the SDF ($\mathcal{S} = DynamicReachable_d(e,i)$).

Note that not all state elements which are statically reachable by a fault are necessarily dynamically reachable – for instance, an incorrect signal may not propagate to a state element due to masking (recall Figure 2).

Step #2: Identifying Program-Visible Failures. Once the dynamically reachable set $\mathcal{S}$ for a given SDF is identified, we then need to determine whether the combined effect of the errors described in $\mathcal{S}$ results in a program-visible failure.

> **Definition 4:** A set of state elements $\mathcal{S}$ is $GroupACE$ in cycle i if a simultaneous error in all state elements $s \in \mathcal{S}$ in that cycle results in a program-visible failure.

Recall that a single delay fault can result in multiple simultaneous state element errors. As such, it is important to note that GroupACE must consider the system-level impact of multiple simultaneous state element errors. For such an analysis to be exact, GroupACE must reason about the combined effect of these state element errors, as the following confounding effects can occur. First, *ACE compounding* occurs when no state element $s \in \mathcal{S}$ is individually ACE, but the state elements together are GroupACE. In contrast, *ACE interference* [54] occurs when a group of state elements $s \in \mathcal{S}$ are individually ACE but the state element errors negate each other, resulting in the group not being GroupACE.

Computational Complexity of the Two-Step Approach. The two-step derivation of $DelayACE$ lends itself to efficient computation, minimizing the need for expensive timing-aware simulations. To compute the dynamically reachable set of a wire e in cycle i, it is sufficient to reason about signal propagation from one state element to another within the single cycle. As such, only cycle i needs to be simulated in an expensive *timing-aware* manner. All other required simulations (including to determine the state at the beginning of cycle i) can be conducted in a timing-agnostic manner.

Importantly, identifying whether state element errors propagate to program-visible failures can be done without considering the circuit's sub-cycle timing behaviour. Computing whether a set of state elements is GroupACE in a given cycle requires the simulation of N total cycles to observe the eventual program output. Together, computing the DelayAVF of a structure H containing $|E|$ wires with respect to a workload running for N cycles requires simulating $O(|E| \cdot N)$ cycles in a timing-aware fashion, and $O(|E| \cdot N^2)$ cycles (i.e., N separate simulations each of length $|E| \cdot N$ cycles) in a timing-agnostic fashion. We note that these simulations are heavily parallelizable in practice.

C. Further Optimizations and Possible Heuristics

While the two-step approach already significantly reduces computational complexity, further optimizations can be implemented in practice to allow for the computation of DelayAVF for large cores. The simplest way to reduce the total number of timing-aware and agnostic simulations is to employ temporal sampling, only considering a statistical subset of cycles in which to inject SDFs. Evaluating DelayAVF on substructures/macros (e.g., examining the adder instead of the entire ALU all at once) can also reduce the total number of simulations, which scales linearly with the number of wires in the examined structure (rather than in the whole core).

Further implementation optimizations can be made to the timing-aware simulation. Recall that the timing-aware simulation aims to determine the set of state elements experiencing errors. As such, the timing-aware simulation only needs to track the propagation of signals from state elements feeding into the statically reachable set of an affected wire, and therefore only needs to simulate a subset of the circuit. Further, state elements which are not statically reachable will trivially latch the correct value, and as such do not need to be considered.

Additionally, if the state elements feeding into the statically reachable set do not toggle, the timing-aware simulation step can be skipped entirely as the dynamically reachable set is trivially empty. Finally, the result of timing-aware simulations for the same circuit, input, and delay condition pairs can be cached. All of the above optimizations retain fidelity. Note that the computational complexity of the timing-aware simulation also scales linearly in the number of wires $|E|$ of the examined structure, again allowing for scalability gains through evaluating smaller sub-structures.

The timing-agnostic step can also be optimized, albeit by trading accuracy for performance. Observe that given an SDF's dynamically reachable set $\mathcal{S}$, determining whether the set is also GroupACE holds similarities to traditional ACE analysis. As such, methodologies which allow for the computation (or approximation [39]) of the ACEness or AVF of individual state elements can be used to approximate GroupACEness. Re-using particle strike ACE data to estimate whether a group of state elements is GroupACE poses a trade-off between computation complexity and accuracy. We evaluate the accuracy of one possible approximation method in Section VII.

VI. Empirical Case Study

We now use $DelayAVF$ to examine the effects of small delay faults on a real hardware implementation. As reasoning about small delay faults requires reasoning about circuit timing information, we implement an RTL-level fault injection framework to compute the DelayACE and DelayAVF metrics.

A. Experimental Setup

In this evaluation we study Ibex [2], a pipelined in-order RISC-V core. Ibex is an open-source core which has seen multiple tape-outs, including in the OpenTitan root of trust [31]. In the following analysis we study five structures from Ibex: the register file, ALU, decoder, load-store queue (LSQ), and prefetcher. The core's register file is an state element array structure to which we have added optional single-error correction ECC (without any double-error detection capabilities). The core's decoder and ALU are logic-heavy structures which themselves contain no state elements, but affect the values stored in state elements outside of their boundary. The core's load-store queue and prefetch buffer are ancillary structures which are smaller than the other structures considered. For each structure we define the structure H as a set of circuit elements which are associated with the examined chip functionality. To assess the vulnerability of a particular structure to SDFs, we solely examine the impact of delays on the wires E in the microarchitectural structure H.

To derive the DelayAVF for each of Ibex's structures we implement a statistical delay fault injection framework, shown in Figure 5. The framework leverages knowledge of the core's gate and state element level timing characteristics. As the impact of an SDF is dependent on the program executing on the core we study five benchmark applications from the Beebs benchmark suite [35]: *md5*, *bubblesort*, *libstrstr*, *matmult*, and *libfibcall*. For a specified Ibex microarchitectural structure H, our experimental framework determines whether each wire $e \in E$ is DelayACE for a subset of faulty cycles. In the following experiments, the injection points were chosen to be equally spaced out throughout the whole program execution, injecting faults on each wire and 4% of all execution cycles. Tables I and II list statistics about the structures and benchmarks examined, respectively.

For each fault-injection cycle examined, our framework simulates the circuit according to the two-step approach outlined in Section V-B. First, a *timing-aware* stage analyzes the circuit-level impact of an SDF. This stage takes in the core's RTL and a technology library describing the delays of each gate in the design. In this evaluation, we use the open-source NanGate 45nm technology library [1]. Using this timing information, this stage simulates the injection of an SDF of duration d in a given cycle, and outputs the resultant state element errors. Second, a *timing-agnostic* stage determines whether a set of state element errors results in a program-visible failure. This stage simulates the impact of state element errors by injecting faults into a Verilator [50] simulation. Any deviation with the fault-free simulation output indicates that the state elements are GroupACE.

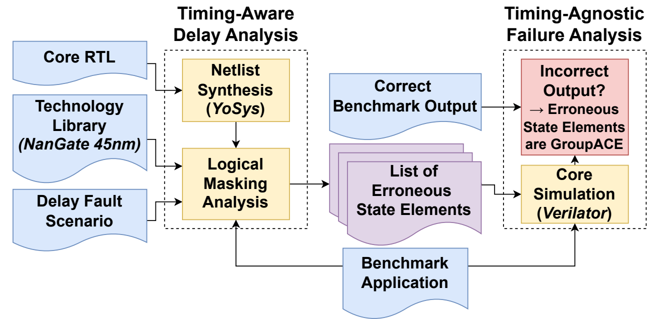

Fig. 5. Simulation flow overview. Toolchains in parentheses are chosen for this case study, but are not strictly required for our modeling approach.

TABLE I
STATISTICS ABOUT THE EXAMINED STRUCTURES

Structure	# Injected Wires (E)
ALU	3668
Decoder	1007
Regfile	17816
Regfile (ECC)	19611
LSU	2027
Prefetch	3249

TABLE II
NUMBER OF CYCLES EXECUTED PER BENCHMARK

Benchmark	# Cycles (N)
md5	1720
libbubblesort	3829
libstrstr	1051
libfibcall	2448
matmult	8903

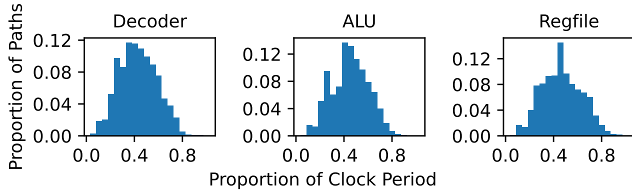

Fig. 6. Path length distributions for different structures in the Ibex core.

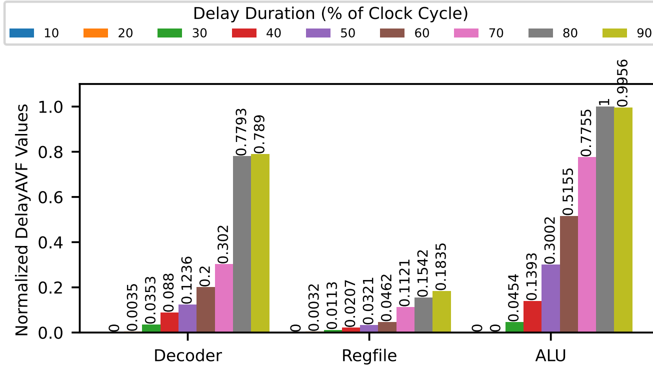

Fig. 7. Normalized geomean DelayAVF values across different Ibex microarchitectural structures (geometric mean of Beebs benchmark suite results).

Modeling Delays. In this evaluation we model delays on wires, where a single wire is defined to connect two circuit elements. The delay on a wire is computed based on the timing characteristics of the source circuit element and the capacitive load of the circuit elements downstream of the source element. We do not consider the added capacitive load of the interconnect itself (in line with pre-layout static timing analysis workflows [7]) and further assume data-independent propagation delays. While timing behaviours may change after the place+route and post-silicon stages, note that logic heavy structures (similar to those studied) are typically dominated by gate delays, with interconnect delays having little effect on overall timing behaviour [38]. These assumptions do not limit the use of DelayAVF in studying more realistic circuit timing models. If desired, DelayAVF could be (re)calculated when more accurate timing information is available to the designer, or with state-of-the-art commercial EDA tools.

Ibex Path Distributions. The path length distributions for Ibex's synthesized microarchitectural structures is shown in Figure 6. The clock period of the Ibex core is set to equal the length of the longest path in the entire design.

B. Evaluating DelayAVF for Different Structures and Delays

We begin by examining the DelayAVF behaviours for Ibex's most consequential microarchitectural structures: the decoder, register file, and ALU. The DelayAVF for these structures is shown in Figure 7. For each structure, we evaluate the DelayAVF of that structure for varying durations of d, ranging from 10% to 90% of the system's clock period, geometrically averaged over the benchmarks considered (and further normalized to facilitate comparison between structures).

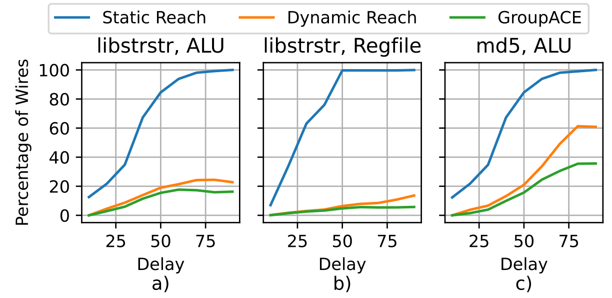

Fig. 8. DelayAVF components for selected structures and benchmarks. *Static Reach* is the % of delayed wires which result in at least one observed statically reachable state element. *Dynamic Reach* is the % of delayed wires which result in at least one observed state element error. *GroupACE* is the % of delayed wires which result in at least one observed program-visible failure.

Observation 1: The average vulnerability of a microarchitectural structure to small delay faults can vary significantly, with the ALU having upwards of $5\times$ the DelayAVF compared to the register file.

As observed in Figure 7, across each delay (with few exceptions), the Ibex's ALU has the highest average DelayAVF, followed by the decoder and the register file. To understand why the DelayAVF behaviours differ significantly across different structures, we revisit the factors which contribute to a circuit element being DelayACE in Figure 8. For a circuit element to be DelayACE, recall that a delay on that circuit element must result in:

1) Path length(s) to downstream state elements statically exceeding the clock period (i.e., being *statically reachable*).
2) State element(s) latching an incorrect value (i.e., being *dynamically reachable*).
3) These state element errors propagating to a program-visible failure (i.e., being $GroupACE$).

Consider the DelayAVF behaviour of the ALU and register file under the `libstrstr` benchmark, shown in Figure 8a) and b) respectively. As seen in both examples, a delay of 50% of the clock period results in at least one statically reachable state element on 85% and 100% of the wires in the ALU and register file respectively. Despite the fact that delays in the register file result in more instances where a state element is statically reachable, far fewer of the SDFs result in at least one state element becoming dynamically reachable (i.e., experiencing a state element error). This low dynamic reachability (and low DelayAVF) can be attributed to the lower *toggle rate* experienced by the register file's state elements.

To see why the register file has a low toggle rate, consider SDFs on word-lines within a similar memory structure in Figure 11. Only two rows can experience a correctness issue due to an SDF in a given cycle: the row which is being activated (signal $0 \rightarrow 1$), and the row that is being deactivated ($1 \rightarrow 0$). Delays on the remaining majority of wordlines ($0 \rightarrow 0$) cannot result in any state elements being dynamically

reachable, resulting in an overall low $DelayAVF$.

Reasoning about DelayAVF's Dependence on Delay. As shown in Figure 7, the DelayAVF highly depends on the SDF duration d examined. To better understand the behaviour of DelayAVF across varying delay durations, we again examine the DelayAVF behaviours shown in Figure 8.

> **Observation 2:** Vulnerability to program-visible failures caused by SDFs is dominated by the circuit's static timing characteristics for small durations of d, with program and architectural-level effects playing a more prominent role for larger delay durations.

For small durations of d, there are few instances where an SDF can result in a path statically exceeding the clock period (and thus no state element errors nor program-visible failures can ever occur due to such a fault). Observe that the static reachability of a state element is solely a function of the circuit's structure, and is independent of any masking at the logical or architectural levels. As the delay duration considered is increased, the number of SDFs that result in at least one statically reachable state element increases. This does not necessarily translate into a corresponding increase in DelayAVF, however, as circuit-level or architectural masking effects can still prevent a program-visible failure. We highlight that the strength of these masking effects are not represented in the circuit path distributions (Figure 6), emphasizing the importance of comprehensive DelayAVF analysis. These observations further suggest that for cores with tighter timing/significantly more critical paths (as seen in highly-optimized cores [5]), and therefore greater proportions of statically reachable state elements for smaller delays, the prominence of program and architectural-level effects is further increased.

Our analysis indicates that multi-bit state element errors are common. Averaged across all benchmarks and all structures, half of the SDFs that induce at least one state element error introduce multi-bit state element errors. The smallest observed fraction of multi-bit state element errors occurs at d = 10%, where an average of 21% of state element errors are multi-bit errors. For all other examined delay durations d, the fraction of multi-bit state element errors fluctuated around 50%, with no clear trend over different values of d.

While the DelayAVF typically increases as the delay duration d is increased, a DelayAVF considering a larger delay d does not necessarily upper bound one with a smaller delay d for the same structure and benchmark. It may be the case that a larger delay sometimes results in smaller dynamically reachable sets of state elements (e.g., if a larger delay induces a state element to latch a correct value due to glitching effects).

Dependence on Benchmarks. In addition to being influenced by circuit-level and microarchitectural characteristics, the DelayAVF of a microarchitectural structure is also influenced by the application running on the core. Figure 9 shows the detailed breakdown of the ALU's DelayAVF characteristics of the Beebs benchmarks considered.

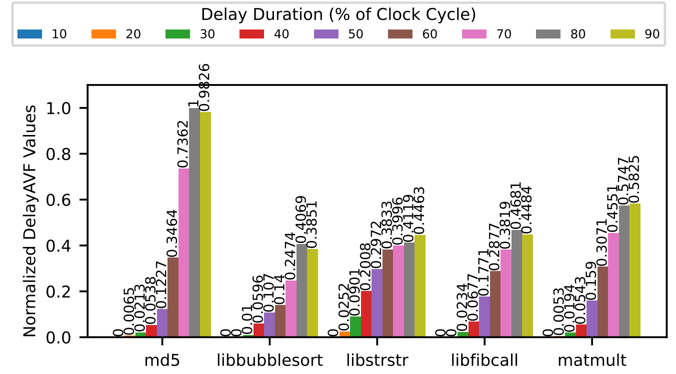

Fig. 9. Normalized DelayAVF values for the Ibex ALU across each Beebs benchmark examined.

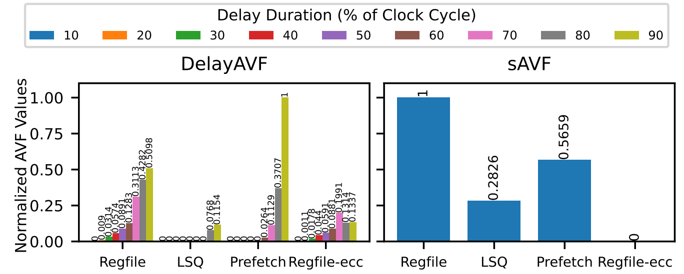

Fig. 10. Normalized geomean $sAVF$ and $DelayAVF$ values for stateful structures in the Ibex core. Values are normalized to the largest $sAVF$ and $DelayAVF$ values observed, respectively.

> **Observation 3:** The DelayAVF of a microarchitectural structure can vary considerably across different benchmarks, further indicating a strong dependence on architectural and program-level effects.

Variations of DelayAVF across different benchmarks can be associated to differences in the rates of state element errors (e.g., due to data-dependent masking), resilience at the architectural level, or the resilience of the benchmark itself to faults at the program-level. The difference between the ALU's DelayAVF behaviours across benchmarks is further highlighted when contrasting the `libstrstr` and `md5` benchmarks, shown in Figure 8a) and c) respectively. `md5` computes hash values (which are highly random in nature), while `libstrstr` performs string comparisons which operate on very regular data. The highly random-looking nature of operations in `md5` entail a higher toggle rate in the ALU, resulting in high dynamic reachability and overall consequent DelayAVF as seen in Figure 8c) and Figure 9.

C. Comparing Delay vs. Particle Strike Vulnerability Factors

We now further illuminate the difference between the insights gained using DelayAVF compared to standard AVF (which we denote as sAVF), as originally highlighted in Section III-A. The DelayAVF and sAVF computed for Ibex's stateful microarchitectural structures (including a comparison of

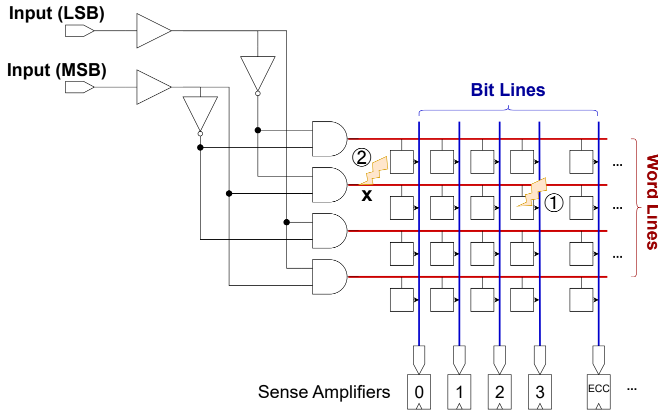

Fig. 11. Example of a memory which is vulnerable to both particle strikes and delay faults. The effects of a particle strikes on a state element (e.g., ①), which can be protected by adding a single-error correct ECC. A small delay fault in x (②) may still result in a state element error, even with ECC.

the register file with and without single-error ECC protection) is shown in Figure 10.

Observation 4: Ranking structural vulnerability via $DelayAVF$ reveals different rank-orderings than particle strike AVF.

In some cases, some structures may simultaneously have a high DelayAVF and sAVF. For instance, the Ibex prefetcher is vulnerable to both particle strikes (the prefetcher has an internal buffer to store lines before they are passed to the pipeline) and SDFs. It can also be the case that the DelayAVF and sAVF do not correlate. For instance, we experimentally observe that adding a single-error correcting ECC to the register file reduces its sAVF to zero, while does not result in an equivalent reduction in DelayAVF.

Observation 5: Protections added to structures which are effective in protecting against particle strikes may not be as effective in protecting against SDFs.

To better illuminate why a memory array may have a different sAVF and DelayAVF, again consider Figure 11. For the purposes of this example, assume that the correctness of data stored in the memory is always required for program-level correctness. This will result in the memory's sAVF being 1, as a particle strike in any of the memory's state elements (e.g., at ①) will result in a program-visible failure. Adding a single-error correction ECC reduces the sAVF of the memory array to 0, however observe that ECC does not correspondingly reduce the DelayAVF to 0. Consider an SDF ② on wire x, which drives the word line associated with address `01`. Assume that in cycle $i-1$ the core reads from address `00` and reads from `01` in cycle i. If a SDF occurs on x in cycle i, the wordline associated with `01` will have a delayed activation, resulting in the sense amplifiers re-latching the previously read values from address `00`. If the data at address `00` and `01` is not identical, this SDF will result in at least one state element error. Despite

TABLE III
MAXIMUM AND AVERAGE OCCURRENCE OF ACE INTERFERENCE AND COMPOUNDING BEHAVIOUR FOR IBEX'S STRUCTURES, AS PERCENTAGE OF ALL DYNAMICALLY REACHABLE SETS OBSERVED (AGGREGATED OVER ALL BENCHMARKS AND ASSUMING AN ADDED DELAY d OF 90% OF THE CLOCK PERIOD). THE EFFECTS ARE CONTRASTED WITH THE MAXIMUM AND AVERAGE RELATIVE CHANGE BETWEEN THE DELAYAVF AND ORDELAYAVF.

Structure	Max ACE Int. (%)	Avg. ACE Int. (%)	Max ACE Comp. (%)	Avg. ACE Comp. (%)	Max Rel. Change (%)	Avg. Rel. Change (%)
ALU	0.98	0.58	0.17	0.09	3.00	1.73
Decoder	13.03	6.73	2.47	1.14	21.80	10.45
Regfile	0.13	0.07	0.17	0.07	0.69	0.30
Regfile (ECC)	0.13	0.07	21.95	11.57	92.45	50.38

these state element error(s) the ECC checks will still pass (as the core has read valid data from another address), and ECC will therefore not detect or correct these errors, leading to a non-zero DelayAVF.

VII. Using Approximations to Estimate DelayAVF

In cases where existing fault-injection or state element (particle strike) ACE data is available, it may be desirable to re-use this data to estimate DelayAVF to reduce required simulation during design time. To this end, we look towards computing the GroupACEness of a set of state elements using each state element's individual (particle strike) ACEness.

Recall from Section V-B that there are two confounding effects when considering the impact of multiple simultaneous state element errors that forced us to reason about these errors in conjunction: ACE compounding and ACE interference. Without the presence of these two effects, we can compute the GroupACEness of a set of state elements through the state elements' individual ACEness. To do so, we use the notion of ORACE:

Definition 5: A set of state elements $\mathcal{S}$ is $ORACE$ if any state element $s \in \mathcal{S}$ is individually ACE [54].

In practice, we observe that ACE compounding and interference do exist, making the use of ORACE an approximation.

Definition 6: The DelayAVF approximation computed by using ORACE instead of GroupACE is denoted as *OrDelayAVF*.

To understand the fidelity of the OrDelayAVF approximation, we evaluate the rates of ACE interference and ACE compounding across the different hardware structures examined. We present these rates and the resulting relative change between DelayAVF and OrDelayAVF in Table III. The importance of an accurate computation of GroupACEness increases as the proportion of SDFs which result in multiple state element errors increases. We therefore evaluated OrDelayAVF with respect to a delay of 90% of the clock period.

Observation 6: Care should be taken to avoid the usage of OrDelayAVF on structures with substantial ACE interference or compounding rates.

For the examined structures, the ORACE approximation works well on average. However, two cases stand out where the ORACE approximation falls short:

First, the decoder experiences a higher degree of ACE interference. This can be partially explained through architectural effects. For example, in some cases multi-bit state element errors result in the core's program counter to rewind by several instructions, while single-bit state element errors cause the program counter to rewind by a single instruction. In some cases, replaying the last instruction may cause a program-visible failure, while replaying several older instructions in sequence may result in no error.

Second, the ECC protected register file experiences a high degree of ACE compounding. This can be directly attributed to the use of single-error-correcting ECC: While a multi-bit state element error is not correctable by the implemented ECC and therefore is GroupACE, the state elements are not individually ACE. ORACE does not account for this compounding effect and therefore under-approximates DelayAVF.

VIII. Related Work

Wilkening et al. [54] examine the impact of *spatially adjacent* multi-bit faults caused by a single particle strike. When considering small delay faults, a key difference arises compared to the spatial multi-bit model presented in [54]. Under the spatial multi-bit fault model, the group of state elements that fail in conjunction can be pre-determined based on their spatial locality. When considering the impact of SDFs the group of state elements that fail in conjunction cannot be determined a priori based on their physical positioning, as the timing and state of the circuit must also be considered.

Pan et al. [36] present intermittent vulnerability factor (IVF): a methodology to estimate architectural vulnerability to intermittent fault models, including delay faults. IVF performs a coarse-grained evaluation of a structure's delay fault vulnerability, assuming that any delay fault during the structure's "write operation" always results in a state element error. IVF thus fails to identify whether the circuit's internal state masks the delay fault. IVF further does not reason about structures without well-defined write operations (e.g., the decoder). IVF further ignores both compounding and interference effects.

Chang et al. [10] reason about the potential effects of timing errors at an instruction-level, and leverage this analysis to reason about when instruction-level errors may propagate to program-visible failures. Chang et al. only consider timing errors in arithmetic units that result in instruction output errors, and cannot systematically reason about the vulnerability of other microarchitectural structures (such as the decoder or register file) to delay faults. Varius [43] estimates the rate of delay-induced state element errors resulting from standard manufacturing variations. Varius does not model logical masking, nor the architectural impact of erroneous state elements.

Shivakumar et al. [47] estimate microprocessor error rates due to particle strikes onto combinational logic gates. Shivakumar et al. do not reason about the circuit's state and timing behavior (required to reason about delay faults), and ignore logical masking effects. Raasch et al. [39] propose an analytical model which approximates the vulnerability of RTL-defined state elements to particle strikes via the AVF of a structure's read and write ports. While [39] is focused on particle strikes and does not provide a method to reason about SDFs, its techniques to approximate AVF could potentially be used to derive ORACE (as discussed in Section V-C).

Entrena et al. [18] and Hari et al. [22] present two-stage models to determine whether particle-induced faults (at the gate- and state element-levels, respectively) result in a program-visible failure. These works first identify state element errors, and perform a faster simulation (at the RTL- or architecture-levels, respectively) to determine whether a program-visible failure occurs. While the approaches taken by these two works to reduce simulation complexity are similar to our two-step approach, both again solely consider the behaviour of particle strikes (which are distinct from SDFs).

Further efforts exist to improve the likelihood of catching SDFs during test time. Czutro et al. [15] evaluate the effectiveness of scan-chain tests by simulating whether a given test pattern can induce at least one state element error due an SDF on a studied wire. This paper also presents an analytical approach to estimate the delay from a resistive defect, offering an additional approach to parameterize d. Ahmed et al. [4] leverage information about a circuit's timing to prioritize ATPG tests which exercise the circuit's most critical paths (increasing the likelihood of detecting an SDF). Riefert et al. [41] employ bounded model checking to generate functional programs that are likely to sensitize a given path and propagate potential state element errors to an output. We highlight that the marginal nature of the defects considered in this paper can still result in test escapes despite employing such testing strategies, requiring us to reason about structural vulnerability using DelayAVF when such a test escape occurs.

In this paper we have focused on the use of DelayAVF to assess the relative vulnerability of different microarchitectural structures to SDFs. However, DelayAVF may also be useful in generating functional tests that are particularly suited to increase the observability of SDFs for a given hardware design. This approach could be applied within hardware-aware functional test generation frameworks such as Harpocrates [25].

IX. Conclusion

We have presented a novel methodology to reason about the impact of small delay faults. Transferring the notion of ACEness to SDFs, we can derive and tractably compute DelayAVF, a metric that quantifies the vulnerability of a hardware structure to SDFs. In a case-study on the open-source Ibex core, we demonstrate the practical efficacy of our methods, providing insights into the factors influencing DelayAVF. Finally, we identify an approximation that can be used to leverage existing ACE data to determine DelayAVF.

ACKNOWLEDGMENT

The authors thank Divya Prasad (AMD), Jeff Rearick (AMD), Jean-Pierre Seifert (TU Berlin), and the anonymous MICRO reviewers for their assistance. The authors acknowledge the financial support of the MIT AI Hardware Program, the Google Research Scholar Program, and the Federal Ministry of Education and Research of Germany in the program "Souverän. Digital. Vernetzt." Joint project 6G-RIC, project identification number: 16KISK030. Further funding was provided by NSF under grant CCF 2217099; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

©2024 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

This paper reflects collaborative work between the authors.

APPENDIX

A. Abstract

Our artifact contains the SDF evaluation framework used to derive the results outlined in Section VI. In particular, the provided framework includes scripts to synthesize the Ibex core and perform a timing analysis, yielding DelayAVF estimates for various submodules in the Ibex design. The source code for the Beebs benchmarks and the timing libraries used are both included in this artifact. All experiments are run in a Dockerfile for convenience.

This artifact can be used in the future to evaluate the DelayAVF of the Ibex core for new benchmarks, and offers a template to calculate DelayAVF on different cores.

B. Artifact check-list (meta-information)

- **Algorithm:** DelayAVF calculation for the Ibex core under the Beebs benchmark suite.
- **Program:** Workflow includes Yosys, RTL to JSON parsers, a custom SDF-injection framework (used to generate dynamically reachable sets), and Verilator.
- **Run-time environment:** Docker.
- **Hardware:** No particular requirements, however a server with 48+ cores is recommended to reduce overall runtime.
- **Output:** DelayAVF values for a specific core, benchmark, and delay range.
- **How much disk space required (approximately)?:** 10 GB.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** 24 hours.
- **Publicly available?:** Yes, available at https://github.com/viniul/micro-artifact.
- **Code licenses (if publicly available)?:** DelayAVF framework under MIT license, various licenses for dependencies.
- **Archived (provide DOI)?:** 10.5281/zenodo.13743439

C. Description

How to access. Our DelayAVF infrastructure is available at https://github.com/viniul/micro-artifact.

D. Installation

Our infrastructure is available in a Docker container. To build and launch the docker environment, clone the git repository (including submodules through `git submodule init; git submodule update`) then run:

```
./build_and_run_docker.sh
```

E. Experiment workflow

To compute the DelayAVF of a given Ibex structure with respect to a certain benchmark, the framework must first be configured appropriately using a *configuration json*. Example configuration jsons are stored in `tests/ibex/testbench/configs/beeps/`. The configuration file contains the following parameters:

- **synth_file** and **sub_synth_file**: The output location of the synthesized top-level and submodule verilog files.
- **submodule_name** and **short_submodule_name**: The RTL structure to be examined.
- **pdk_path**: The location of the timing library (PDK).
- **top_path** and **clk_path**: The RTL paths to the top module and the system clock.
- **hex_payload**: The benchmark/payload to run.
- **delay_range**: The lengths of small delay faults to simulate.
- **percent_sampled_cycles_delay**: The sampling rate of the DelayAVF simulation (i.e., what percentage of cycles to inject delays).
- **percent_sampled_cycles_particle**: The sampling rate of the sAVF simulation (i.e., what percentage of cycles to inject bit-flips).
- **ecc_on**: Whether to enable the single-error correct ECC.
- **output_dir**: The directory to output results.

F. Evaluation and expected results

The workflow will output the DelayAVF results for the structure, benchmark, and delays defined in the configuration file. The entire workflow can be launched by passing a configuration file to the `run_all.sh` script. For instance, to compute the ALU's's DelayAVF for the MD5 benchmark under varying delays (as in Figure 9), execute the following commands in the Docker environment:

```
cd /current_dir/tests/ibex/testbench/

./run_all.sh configs/beeps/md5_alu.dict
```

The above command will generate the DelayAVF information for the MD5 group in Figure 9. For the remaining benchmarks (if desired), follow a similar approach, calling `./run_all.sh` for each Beebs configuration in the figure.

Once all the simulations have completed, call the plotting script to obtain a plot similar to Figure 9:

```
python3 ../../../util_scripts/plot_beeps.py
```

G. Notes

Additional details on the flow are described in the repository's `README` file.

H. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

[1] "FreePDK45 and the Nangate Open Cell Library — mflowgen documentation." [Online]. Available: https://mflowgen.readthedocs.io/en/latest/stdlib-freepdk45.html

[2] "Ibex: An embedded 32 bit RISC-V CPU core." [Online]. Available: https://ibex-core.readthedocs.io/en/latest/

[3] M. Abramovici, M. A. Breuer, A. D. Friedman *et al.*, *Digital systems testing and testable design*. Computer science press New York, 1990, vol. 2.

[4] N. Ahmed, M. Tehranipoor, and V. Jayaram, "Timing-based delay test for screening small delay defects," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 320–325.

[5] R. Aitken, V. Chandra, and D. Pietromonaco, "Implications of variability on resilient design," in *2015 IEEE International Electron Devices Meeting (IEDM)*, 2015, pp. 20.8.1–20.8.3.

[6] H. Asadi, M. B. Tahoori, B. Mullins, D. Kaeli, and K. Granlund, "Soft error susceptibility analysis of sram-based fpgas in high-performance information systems," *IEEE Transactions on Nuclear Science*, vol. 54, no. 6, pp. 2714–2726, 2007.

[7] J. Bhasker and R. Chadha, *Static timing analysis for nanometer designs: A practical approach*. Springer Science & Business Media, 2009.

[8] F. A. Bower, D. Hower, M. Yilmaz, D. J. Sorin, and S. Ozev, "Applying architectural vulnerability analysis to hard faults in the microprocessor," *SIGMETRICS Perform. Eval. Rev.*, vol. 34, no. 1, p. 375–376, jun 2006.

[9] M. Bruce, V. Bruce, D. Eppes, J. Wilcox, E. Cole, P. Tangyunyong, C. Hawkins, and R. Ring, "Soft defect localization (sdl) in integrated circuits using laser scanning microscopy," in *The 16th Annual Meeting of the IEEE Lasers and Electro-Optics Society, 2003. LEOS 2003.*, vol. 2, 2003, pp. 662–663 vol.2.

[10] C.-K. Chang, W. Yin, and M. Erez, "Assessing the impact of timing errors on hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–19.

[11] C.-H. Chen, P. Knag, and Z. Zhang, "Characterization of heavy-ion-induced single-event effects in 65 nm bulk cmos asic test chips," *IEEE Transactions on Nuclear Science*, vol. 61, no. 5, pp. 2694–2701, 2014.

[12] E. J. Cole, P. Tangyunyong, and D. Barton, "Backside localization of open and shorted ic interconnections," in *1998 IEEE International Reliability Physics Symposium Proceedings. 36th Annual (Cat. No.98CH36173)*, 1998, pp. 129–136.

[13] C. Constantinescu, "Impact of intermittent faults on nanocomputing devices," in *DSN 2007 Workshop on Dependable and Secure Nanocomputing*, 2007.

[14] C. Constantinescu, "Intermittent faults and effects on reliability of integrated circuits," in *2008 Annual Reliability and Maintainability Symposium*. IEEE, 2008, pp. 370–374.

[15] A. Czutro, N. Houarche, P. Engelke, I. Polian, M. Comte, M. Renovell, and B. Becker, "A simulator of small-delay faults caused by resistive-open defects," in *2008 13th European Test Symposium*. IEEE, 2008, pp. 113–118.

[16] H. D. Dixit, L. Boyle, G. Vunnam, S. Pendharkar, M. Beadon, and S. Sankar, "Detecting silent data corruptions in the wild," *arXiv preprint arXiv:2203.08989*, 2022.

[17] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent data corruptions at scale," *arXiv preprint arXiv:2102.11245*, 2021.

[18] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela, and C. Lopez-Ongil, "Soft Error Sensitivity Evaluation of Microprocessors by Multilevel Emulation-Based Fault Injection," *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 313–322, Mar. 2012. [Online]. Available: https://ieeexplore.ieee.org/document/5669284/

[19] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, "Razor: circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 24, no. 6, pp. 10–20, 2004.

[20] S. Gurumurthi, V. Sridharan, and S. Gurumurthy, "Emerging Fault Modes: Challenges and Research Opportunities," Jul. 2023. [Online]. Available: https://www.sigarch.org/emerging-fault-modes-challenges-and-research-opportunities/

[21] I. S. Haque and V. S. Pande, "Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 691–696.

[22] S. K. S. Hari, P. Rech, T. Tsai, M. Stephenson, A. Zulfiqar, M. Sullivan, P. Shirvani, P. Racunas, J. Emer, and S. W. Keckler, "Estimating silent data corruption rates using a two-level model," *arXiv preprint arXiv:2005.01445*, 2020.

[23] C. Hawkins, A. Keshavarzi, and J. Segura, "Parametric timing failures and defect-based testing in nanotechnology cmos digital ics," in *Proc. of NASA Symp*. Citeseer, 2003.

[24] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 9–16.

[25] N. Karystinos, O. Chatzopoulos, G.-M. Fragkoulis, G. Papadimitriou, D. Gizopoulos, and S. Gurumurthi, "Harpocrates: Breaking the silence of cpu faults through hardware-in-the-loop program generation," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 516–531.

[26] S. Lin and J. D. Costello, *Error Control Coding*. Peason Prenticle Hall, 1983.

[27] D. Lipetz and E. Schwarz, "Self checking in current floating-point units," in *2011 IEEE 20th Symposium on Computer Arithmetic*, 2011, pp. 73–76.

[28] J. Mahmod, S. Millican, U. Guin, and V. Agrawal, "Special session: Delay fault testing - present and future," in *2019 IEEE 37th VLSI Test Symposium (VTS)*, 2019, pp. 1–10.

[29] A. Meixner, "Screening For Silent Data Errors," Jan. 2023. [Online]. Available: https://semiengineering.com/screening-for-silent-data-errors/

[30] A. Meixner, "Strategies for Detecting Sources of Silent Data Corruption," 2024. [Online]. Available: https://semiengineering.com/strategies-for-detecting-sources-of-silent-data-corruption/

[31] A. Meza, F. Restuccia, J. Oberg, D. Rizzo, and R. Kastner, "Security verification of the opentitan hardware root of trust," *IEEE Security & Privacy*, 2023.

[32] S. Mukherjee, *Architecture design for soft errors*. Morgan Kaufmann, 2011.

[33] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 29–40.

[34] E. Normand, "Single event upset at ground level," *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, pp. 2742–2750, 1996.

[35] J. Pallister, S. Hollis, and J. Bennett, "Beebs: Open benchmarks for energy measurements on embedded platforms," *arXiv preprint arXiv:1308.5174*, 2013.

[36] S. Pan, Y. Hu, and X. Li, "IVF: Characterizing the Vulnerability of Microprocessor Structures to Intermittent Faults," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 5, pp. 777–790, May 2012. [Online]. Available: http://ieeexplore.ieee.org/document/5752262/

[37] S. Payer, C. Lichtenau, M. Klein, K. Schelm, P. Leber, N. Hofmann, and T. Babinsky, "Simd multi format floating-point unit on the ibm z15 (tm)," in *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2020, pp. 125–128.

[38] D. Prasad, S. Sinha, B. Cline, S. Moore, and A. Naeemi, "Accurate processor-level wirelength distribution model for technology pathfinding using a modernized interpretation of rent's rule," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3195970.3195980

[39] S. Raasch, A. Biswas, J. Stephan, P. Racunas, and J. Emer, "A fast and accurate analytical technique to compute the avf of sequential bits in a processor," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 738–749.

[40] J. Rajski, V. Chickermane, J.-F. Côté, S. Eggersglüß, N. Mukherjee, and J. Tyszer, "The future of design for test and silicon lifecycle management," *IEEE Design & Test*, pp. 1–1, 2023.

[41] A. Riefert, L. Ciganda, M. Sauer, P. Bernardi, M. S. Reorda, and B. Becker, "An effective approach to automatic functional processor test generation for small-delay faults," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014, pp. 1–6.

[42] P. G. Ryan, I. Aziz, W. B. Howell, T. K. Janczak, and D. J. Lu, "Process defect trends and strategic test gaps," in *2014 International Test Conference*. Seattle, WA, USA: IEEE, Oct. 2014, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/7035276/

[43] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, "Varius: A model of process variation and resulting timing errors for microarchitects," *IEEE Transactions on Semiconductor Manufacturing*, vol. 21, no. 1, pp. 3–13, 2008.

[44] K. Serebryany, M. Lifantsev, K. Shtoyk, D. Kwan, and P. Hochschild, "Silifuzz: Fuzzing cpus by proxy," *arXiv preprint arXiv:2110.11519*, 2021.

[45] M. Shamsa and D. Lerner, "Defect mechanisms responsible for silent data errors," in *2024 IEEE International Reliability Physics Symposium (IRPS)*. IEEE, 2024, pp. 1–5.

[46] S. Z. Shazli, M. Abdul-Aziz, M. B. Tahoori, and D. R. Kaeli, "A field analysis of system-level effects of soft errors occurring in microprocessors used in information systems," in *2008 IEEE International Test Conference*, 2008, pp. 1–10.

[47] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proceedings International Conference on Dependable Systems and Networks*, 2002, pp. 389–398.

[48] A. Singh, S. Chakravarty, G. Papadimitriou, and D. Gizopoulos, "Silent data errors: Sources, detection, and modeling," in *2023 IEEE 41st VLSI Test Symposium (VTS)*. IEEE, 2023, pp. 1–12.

[49] C. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 397–404, 2005.

[50] W. Snyder, "Verilator and systemperl," in *North American SystemC Users' Group, Design Automation Conference*, 2004.

[51] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng shui of supercomputer memory positional effects in dram and sram faults," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–11.

[52] M. Tehranipoor, K. Peng, and K. Chakrabarty, *Test and diagnosis for small-delay defects*. Springer, 2011.

[53] S. Wang, G. Zhang, J. Wei, Y. Wang, J. Wu, and Q. Luo, "Understanding silent data corruptions in a large production cpu population," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 216–230.

[54] M. Wilkening, V. Sridharan, S. Li, F. Previlon, S. Gurumurthi, and D. R. Kaeli, "Calculating architectural vulnerability factors for spatial multi-bit transient faults," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 293–305.

[55] D. H. Yoon and M. Erez, "Virtualized and flexible ecc for main memory," *SIGARCH Comput. Archit. News*, vol. 38, no. 1, p. 397–408, mar 2010. [Online]. Available: https://doi-org.libproxy.mit.edu/10.1145/1735970.1736064