Harpocrates: Breaking the Silence of CPU Faults through Hardware-in-the-Loop Program Generation

Nikos Karystinos[§] Odysseas Chatzopoulos[§] George-Marios Fragkoulis[§]
George Papadimitriou[§] Dimitris Gizopoulos[§] Sudhanva Gurumurthi[†]
[§]*University of Athens, Department of Informatics and Telecomm., Computer Architecture Lab,* Greece
{n.karystinos | od.chatzopoulos | gm.fragkoulis | georgepap | dgizop}@di.uoa.gr
[†]*RAS Architecture, Advanced Micro Devices, Inc, Austin, Texas, USA*, sudhanva.gurumurthi@amd.com

***Abstract*—Several hyperscalers have recently disclosed the occurrence of Silent Data Corruptions (SDCs) in their systems fleets, sparking concerns about the severity of known and the existence of unidentified root causes of faults in CPUs. These incidents reveal that CPU chips have the potential to generate incorrect results for different tasks due to latent manufacturing defects, variability, marginalities, bugs, and aging. To tackle this problem, we present *Harpocrates*, an automated methodology for the generation of short, constrained-random functional test programs that maximize fault detection in target CPU structures and can be employed at different stages of system lifetime. Harpocrates stands out by adopting a hardware-model-in-the-loop approach, which iteratively refines the generated test programs using a detailed simulation-based microarchitecture engine. The engine models and grades for multiple hardware fault types that can lead to data corruptions during system operation. Harpocrates is versatile and can adapt to various program generators, ISAs, microarchitectures, and fault types. Our results on six important CPU hardware structures show that Harpocrates attains much shorter test generation times than hardware-agnostic publicly available frameworks and outperforms open-source test suites in terms of fault detection capability.**

I. Introduction

As modern CPUs become increasingly complex and powerful, their ability to execute demanding workloads and handle large amounts of data becomes more critical. This complexity raises the probability of unexpected faults occurring during data processing increasing the risk of data corruptions, leading to wrong results, information loss, or system crashes [1]–[6]. Data corruption in CPUs can occur due to hardware defects and marginalities (e.g., temperature conditions), particle strikes, hardware design bugs, and even malicious attacks [7]. The risk of data corruption has further increased as CPU designs have become more complex [3], [8]. Traditional verification and testing methods are often insufficient for detecting all potential defects and root causes of Silent Data Corruptions (SDCs) in modern CPUs [9]. Shrinking technology, skyrocketing counts of CPU and memory chips, and new failure mechanisms all pose extreme reliability challenges. For example, the emergence of marginal defect-induced [10] or degrading faults [11] is a major issue, which cannot be tackled solely relying on conventional manufacturing testing and screening [12]–[14].

In this context, recent hyperscaler reports suggest high levels of *defective parts per million* (DPPM) for CPUs deployed in their fleets. Fig. 1 shows the values disclosed by [1] ("hundreds of CPUs detected for SDCs in hundreds of thousands of machines" ≈1,000 DPPM), by [2] ("the order of a few mercurial cores per several thousand machines" = a bit less than 1,000 DPPM) and recently by [3] ("3.61 CPUs per 10,000" = 361 DPPM). Acceptable values vary across domains: safety-critical domains (e.g., automotive) require less than 10 DPPM [15], while general purpose computing in the cloud or HPC can afford few 100's DPPM, though driving the DPPM as low as possible is important as data center deployments scale out[1]. The daunting DPPM values of Fig. 1 call the computing community to action to *verify* the rates and better understand the *root causes*.

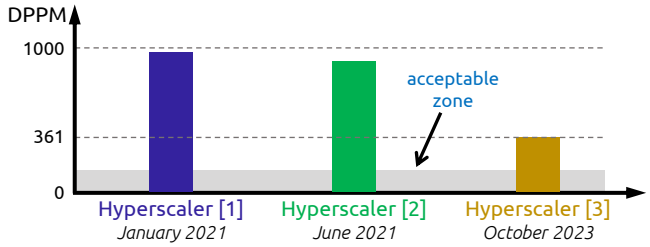

Fig. 1. Reported CPU defective parts per million (DPPM) by hyperscalers.

Several hyperscalers [1]–[3] disclosures *extend* the quest about the origins of SDCs: from usual suspects (memory, storage, network) to the heart of the computation: *the CPU*. In CPUs the overheads of redundancy (area, performance, power) and protection decisions for hardware structures are critical. CPUs are traditionally protected against particle-induced transients (soft errors) using industry best practices such as AVF (Architectural Vulnerability Factor) modeling [16] to determine which structures to protect and to what extent. Yet, with the rise of new defect mechanisms and resulting escaping faults, vulnerability estimates relying solely on transients as the primary root cause might not be entirely accurate, potentially increasing the SDC risk. To this end, it is important for CPU designers and users to employ effective ways for the detection of potential defects in cores (at manufacturing as well as in the field) and expose the root causes of data corruptions.

[1]The need to attain very low DPPM was emphasized at recent panels held at architecture (ISCA), design automation (DAC), testing (ITC) conferences.

CPU fault detection through execution of programs composed of normal instructions has been studied extensively [17]–[27]. This method is known variously as *functional testing*, *instruction-based testing*, *software-based testing*, or *native mode testing*. Unlike *structural testing* methods employed at manufacturing time placing the CPU in a special mode (such as scan), functional testing aims at detecting hardware faults that affect the CPU during normal operating mode. The generation of efficient functional test programs can be deterministic or directed random. Grading of the quality of functional test programs is performed by fault simulation EDA tools on top of detailed gate-level models of the CPU.

We present *Harpocrates*[2], a novel methodology and unique toolchain for the automatic generation and grading of effective functional test programs for modern CPUs. Harpocrates is aligned to related industry-led efforts [3], [28]–[31], which present functional test program generation frameworks for CPUs. Functional testing of CPUs at native instruction execution mode aims to detect any type of fault *before it manifests in the field and affects user programs*. The goal is to maximize the *coverage* of the CPU hardware (go through as much of the hardware as possible) by generating effective but short test programs. By monitoring the CPU's behavior for irregularities (erroneous outputs or crashes) during the execution of these functional test programs, hardware faults can be identified and faulty CPUs isolated. Therefore, an efficient method for generating functional test programs should ideally be *automated* and *hardware-aware*.

For Harpocrates (or any other approach) to be practically plausible, some important challenges should be addressed:

1) Dealing with modern CPU design and deployment: modern CPUs are complex, with multiple interacting layers of hardware and software, and therefore all layers should be considered.
2) Targeting specific hardware structures: most types of hardware faults tend to be localized in specific structures. Aiming to be generically applicable, existing frameworks are hardware-agnostic and do not grade tests on specific CPU microarchitectures. Localizing the fault is important for root cause analysis of defective CPUs and provides valuable field data for driving design improvements in the next generations.
3) Generating fast functional test programs: both for root causing (CPU designer) and quick detection (CPU user), the functional test programs should have short execution times. Datacenter providers aim for near zero fleet downtime. Test programs (on demand or periodically executed) should minimally affect downtime; this aspect should be enhanced in existing frameworks.

Altogether, effective functional program generation for hardware faults detection should be *hardware-aware* using microarchitectural knowledge, consider *the full system stack*, and aim at *short programs* with *high fault detection capability*.

[2]God of silence and secrecy: https://en.wikipedia.org/wiki/Harpocrates.

Our method and tools framework *Harpocrates* is a novel contributor to these challenges, by:

1) Developing a constrained-random program generator and mutator called *MuSeqGen* (Mutator and Sequence Generator). *MuSeqGen's* code generator is built on MicroProbe [32], an established framework for the generation of assembly-level programs for POWER and RISC-V architectures. *MuSeqGen* extends the ISA support to x86-64 and, more importantly, offers versatile utilities for configurable random generation under any set of constraints. *MuSeqGen* integrates a mutation engine that alters or combines generated sequences to generate refined variants of programs. (Challenges #2, #3).
2) Increasing the hardware coverage and the detection capability of generated programs to hardware faults by employing a *hardware-in-the-loop* approach: a detailed CPU model using the gem5 simulator. Gem5, and the tooling built around it, is a key piece of *Harpocrates* in grading and refining the automatically generated programs. (Challenges #1, #2, #3).
3) Proposing a feedback-based automated methodology for the generation of functional programs. The methodology comprises three distinct components: (i) the Generator, (ii) the Mutator, and (iii) the Evaluator, which, together, construct the full *Harpocrates* program generation loop. (Challenges #1, #2).

Harpocrates is optimized for high throughput of functional test program generation and delivers fast programs with high fault detection ability. We demonstrate its effectiveness in detecting different faults in multiple hardware structures of a modern x86 CPU in which it excels in terms of fault detection capability and speed compared to both open-source programs and regular benchmark workloads we evaluate.

II. Background: Essential Concepts and Definitions

A. Silent Data Corruptions (SDCs)

SDCs have become a common concern, impacting critical infrastructures [1], [5], [33]. They are now increasingly linked to CPU chips alongside memory, storage, and networking. These corruptions are termed 'silent' because they escape hardware-level detection mechanisms, causing errors that may propagate through the system unnoticed until they manifest as application-level issues, leading to potential data loss [1], [2]. SDCs may arise from various factors like soft errors, manufacturing defects, and design flaws. They often go unnoticed since the software does not always check for them, especially in cloud and data center environments. Redundancy methods, such as duplicating or triplicating execution, can help mitigate SDCs, but they come with performance and power costs. Both software and hardware-based redundancy methods are expensive in terms of performance, power, and design complexities.

B. Defects, Faults, Errors

A *fault* models a physical mechanism that leads to a mismatch between a specification and the delivered service; in a CPU the service is the correct execution of programs. Every fault has a physical root cause, and in the case of processor faults, high-energy particles like neutrons or alpha particles, as well as silicon manufacturing *defects* or device lifetime degradation, are well-understood root causes. The temporal behavior of a fault - transient, intermittent, or permanent - is closely tied to its physical root cause ("a bit" in the definitions refers either to a storage element or to a gate output) [18], [34], [35]. The following *fault types* (in terms of their temporal behavior) are widely used; we use the broad term *fault type* instead of *fault model* to imply only the temporal behavior (duration of existence) of faults and not their logical behavior.

1) **Transient faults** exist for a finite period of time (disappear when the bit is overwritten) and are nonrecurring.
2) **Intermittent faults** oscillate between faulty and fault-free operation.
3) **Permanent faults** do not correct over time (constantly stuck-at faults) [36].

In the context of hardware faults, *errors* are observable outcomes stemming from faults in hardware resources like storage elements or gates. When a defective resource is used during computation, it can lead to Silent Data Corruption (SDC), system crashes, error notifications, or corrections through hardware resilience mechanisms like Error Correcting Codes (ECC). SDC is the most dangerous outcome, as it can potentially compromise the correctness of software running on the hardware without any observable indication of the error [10]. In the realm of Reliability, Availability, and Serviceability (RAS) design for CPUs, emphasis has been placed on mitigating particle-induced faults, often referred to as "soft errors" [37]. However, recent reports from hyperscalers highlight new fault causes, such as marginal defects, capable of inducing faults under specific conditions like temperature, voltage, and workload patterns [10], [11], [38], [39]. These defects, along with device degradation in scaled technologies, pose severe concerns for transient, intermittent, and permanent faults over a processor's lifespan. To ensure high reliability, it is crucial to comprehend various physical root causes, their associated fault types, and the resulting errors based on the location of these faults within a processor.

C. Fault Detection Capability and Hardware Coverage

Fault detection capability of a program in a hardware structure for a specific fault type or model, is the fraction of the faults in the structure that the program *detects* (i.e., the faulty run of a diagnostic program deviates from the fault-free run). Detection capability is measured in our simulation infrastructure through statistical fault injection. If N faults are injected and the program detects n of them, its detection capability is n/N. For our fault injection runs in the experimental evaluation, we use the latest version of the gem5-based injector (GeFIN) [40] employed in many reliability studies [41]–[47]; a major extension of the tool is the injection of faults in gate level models of CPU functional units.

Hardware coverage (for the purposes of our methodology) is defined as any objective (reward) function tied to a specific CPU hardware structure of any granularity (bit, register, cache line, functional unit, decoder, etc.) that is expected to correlate well with the fault detection capability of functional programs targeting the structure. Coverage functions can incorporate different measures of utilization (both spatial and temporal) in the relevant hardware unit but need to be measured quickly to facilitate iterative refinement of program generation[3].

Thus, coverage is a generic first-order (fast to measure) hardware proxy employed by our method to grade and enhance the *fault detection capability* of programs, which is the metric employed in the eventual evaluation of the quality of the programs that *Harpocrates* generates. Coverage is essentially a proxy of the ability of a program to both activate (sensitize) faults and make them observable (propagate).

D. Fault Types Interplay and Coverage Measurement

Fig. 2 shows the relationship among fault types used to model silicon defect mechanisms. Ideally, a perfect functional test program would detect *all* fault types across all hardware areas, including faults that arise during the CPU's lifetime but were not present during manufacturing. The diagram's "transients" area encompasses all fault combinations over time (i.e., all combinations of a [bit] x [cycles of fault existence]), representing the entire fault universe. Permanent faults are a subset of transients (i.e., the cycles the fault exists is equal to the total program cycles), persisting throughout the program. Intermittent faults exist for a specific number of cycles and can behave as permanent or transient within that interval. A program that detects all transient faults is also very likely[4] to detect the other two types of faults in a hardware bit.

To this end, to guide the *Harpocrates* methodology for array-based components, we focus on a hardware coverage metric related to transient faults. Architecturally Correct Execution (ACE) analysis [16], [48], [49] identifies the fraction of bits crucial for correct execution thus estimating the program's vulnerability to transient faults [50]. The ACE lifetime analysis computes, in each cycle (incorporating the notion of time which is needed for transient faults), the bits that are necessary for Architecturally Correct Execution (ACE) and labels them as ACE. We use the ACE lifetime analysis result (the "vulnerability"; ranging from 0 to 1 or 0% to 100%) as our *hardware*

[3]For some structures, hardware coverage is as simple as a usage counter. For others, extra information is needed to reflect the diversity of their utilization.

[4]But not certain because detection also depends on specific bit patterns.

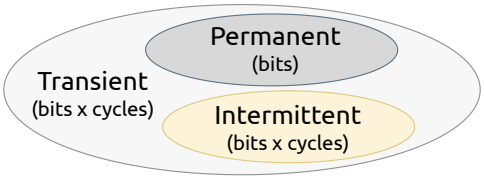

Fig. 2. Hardware fault types (in terms of duration of existence).

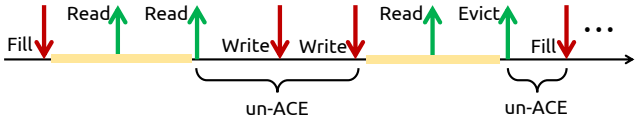

Fig. 3. ACE lifetime analysis of a cache bit. Yellow intervals are ACE. Arrows refer to accesses. For other components, only Reads and Writes are needed.

coverage measurement for the bits of a CPU array structure when a program runs. Fig. 3 shows ACE lifetime stages for a hardware bit (e.g., in the L1D cache). ACE identifies intervals crucial for program correctness (write-to-read or read-to-read), labeling them as ACE periods, with the remaining labeled un-ACE. In summary, for transient faults in bit array components, we measure the hardware coverage of a program by calculating the ratio of ACE cycles to total cycles for each storage bit.

However, the ACE lifetime analysis metric is not suitable for other fault types, such as permanent faults in arithmetic units or gates. For those cases, we define the Input Bit Ratio (IBR) as a more fitting coverage metric. IBR measures the total input bits to a unit across program execution divided by the theoretical maximum count of input bits, (assuming the unit's inputs are utilized at every cycle of program execution)[5].

E. Measuring Fault Detection Capability

We employ Statistical Fault Injection (SFI) to estimate a program's fault detection capability. SFI injects faults at the microarchitecture level and observes outcomes at the software level [45], [51]–[54]. This approach assesses a program's ability to detect hardware faults across the layers. In particular, if the injected fault propagates to the output of the program without any observable indication, the effect is a Silent Data Corruption (SDC). Another case is when the injected fault propagates to the software and results in an Application or System Crash. Conversely, if the fault is masked at either the hardware or software level, it remains undetected. CPU protection schemes like parity and ECC are considered in fault injection modeling. For example, a single bit flip in a fully unprotected cache will be either Masked or lead to an SDC or Crash, while in a SECDED (single error correction double error detection) cache it will be Masked (Corrected). For different fault types (e.g., transient or permanent faults), we grade the final fault detection capability of the refined functional programs of *Harpocrates* using this approach.

III. Existing Frameworks and Experimental Setup

A. Open-Source CPU Testing Frameworks

Aiming to drive community-wide efforts in improving defects detection in CPUs, some open-source frameworks, contributed by the industry, have been publicly released. Many of these test frameworks have been released to the community through the broad industry-wide effort in the context of the Open Compute Project (OCP) and are summarized in [55]. They present generic approaches that can be further enhanced and provide guidelines for more community contributions. Our aspiration for Harpocrates is to be a major contributor to these efforts. In our experimental evaluation section, we present results from the open-source tests these suites publicly offer and compare them with Harpocrates. In this paper, we choose SiliFuzz [28], [29] and OpenDCDiag [30], [56], as these frameworks also include tests, which we use in our evaluations. Wang *et al.* in [3] mention that the company's in-house test programs are not publicly available, but they also employ SiliFuzz and OpenDCDiag to test the fleet.

1) SiliFuzz [29]: SiliFuzz employs fuzzing on (hardware-like) software proxies like CPU simulators and disassemblers. It uses software-coverage metrics to create a diverse input set, known as the corpus. High code coverage in proxies, is conjectured to uncover interesting CPU behavior and detect defects. Silifuzz is hardware and ISA agnostic, generating programs by randomly mutating byte sequences. Valid and deterministic result programs are retained as snapshots to run on the CPU. While still a work in progress, SiliFuzz is yielding results in the field. To achieve statistically significant results in our fault injection-based evaluation, instructions from multiple snapshots (maximum of 100 bytes of binary code each) are aggregated into a single 10K instructions test (the typical size of our methodology's test programs).

2) OpenDCDiag [56]: OpenDCDiag is an open-source project designed for identifying defects and bugs in CPUs, focusing on datacenter CPUs, though not limited to them. It consists of a set of manually specified tests built around a CPU testing framework. The framework permits the addition of custom, user-specified tests, offering convenient primitives. The open-source tests provided include compression, cryptographic operations, matrix multiplication and singular value decomposition. These algorithms are sensitive to data corruption; corruption in their inputs or intermediate results is highly likely to result in corruption in the output data.

In our evaluation of OpenDCDiag we consider each test separately, and evaluate a single execution of the full test. Input sizes are configured so that no test exceeds 100 million cycles of execution.

B. Harpocrates Experimental Setup

1) gem5 simulator: Harpocrates takes a novel *hardware-in-the-loop* approach to complement the SiliFuzz and OpenDCDiag frameworks, which are hardware-agnostic. As discussed in sections II-D and II-E, we employ two important evaluation methodologies to refine and measure the hardware coverage and the detection capability of Harpocrates-generated programs. Both the *coverage* and the *detection capability* can highlight the ability to detect faults and identify the cause of a potential data corruption. For all of our evaluations, we employ the widely used gem5 simulator [57]–[59], which is a well-established microarchitectural simulator. We extend the gem5 simulator to allow the evaluation of the ACE lifetime analysis and the IBR metric (presented in II-D), as well as the injection of faults of different types in the microarchitectural structures.

[5]IBR can be <1 due to several factors. For example, a 64-bit arithmetic unit may not be utilized in every cycle. Moreover, its inputs may not always be 64-bits wide. IBR only considers the effective input bits passed to the unit during the cycles it is actually used. IBR is a fast toggle-count-like measurement, to facilitate many iterations while remaining representative of hardware activity.

Finally, we specify an out-of-order core configuration setting microarchitectural parameters and sizes based on publicly available data for commercial x86 CPUs.

2) Fault Models and Hardware Structure Selection: Our gem5-based fault injection infrastructure can inject transient, intermittent or permanent faults to several hardware structures of the CPU microarchitectural model (e.g., caches, register files, queues, buffers, branch predictors, arithmetic units, etc.).

To demonstrate the effectiveness of Harpocrates, we present results for six key hardware structures: *(a) physical (integer) register file, (b) L1 data cache, (c) integer adder, (d) integer multiplier, (e) SSE floating-point adder and (f) SSE floating-point multiplier.* We intentionally selected the first four hardware structures for a fair comparison of Harpocrates with SiliFuzz and OpenDCDiag (which are hardware-agnostic) and to a general-purpose benchmarks suite (MiBench [60])[6] because all of them bare the hope of a reasonably high utilization of these four main structures. The two SSE FP arithmetic units were chosen to complete the picture since recent reports from hyperscalers clearly identify floating-point units along with their integer counterparts (in scalar and vector hardware) as very likely sources of SDCs in large fleets [3]. Harpocrates can produce effective targeted functional programs for any other hardware structure.

C. Setting the baseline

To set a fair baseline, we measure the effectiveness of the open source frameworks (SiliFuzz, OpenDCDiag) in terms of our driving metrics for Harpocrates: hardware coverage (section II-D) and fault detection capability measured using SFI (section II-E). We also present the same metrics for twelve general purpose benchmarks from the MiBench suite [60].

We showcase SFI results both for transient and permanent faults for different structures. SFI in bit array components, such as the physical register file and L1 data cache, utilizes a transient fault model characterized by a uniform random distribution of bit and cycle selection. In each execution, a single random bit undergoes a flip at a randomly chosen cycle of program execution. On the other hand, SFI in functional unit components, such as the integer adder and multiplier, employs a permanent fault model *at the gate level* because (single event) transients in logic minimally affect the operation of the CPU as they are masked logically and temporally [65]. All functional unit components are modeled at gate level, and a set of random gates is uniformly sampled for injection. In each iteration, one gate from this set is selected, and a stuck-at-0 or stuck-at-1 fault is simulated to the end of execution.

Fig. 4 shows the hardware coverage and the fault detection capability evaluation of MiBench, SiliFuzz, and OpenDCDiag for the Integer Register File (IRF) and L1 data (L1D) cache. The IRF detection capability is very low (less than 5%) across most of the programs considered, with few outliers in MiBench and SiliFuzz that marginally pass the 5% mark. Looking at the L1D cache evaluation results, the detection capability across all three frameworks is significantly higher than the IRF reaching more than 80% for a single OpenDCDiag program. In both components, it is clear that the coverage results are always higher than the detection numbers - a known property of ACE that provides an upper bound of the actual detection [16]. The gap of measured detection from this upper limit is large in most programs tested, due to software masking. Thus, Fig. 4 clearly shows that there is significant room for improvement and underline the objective of *Harpocrates*.

Fig. 5 shows the hardware coverage and the fault detection capability evaluation for the Integer Adder and the Integer Multiplier. When it comes to the Integer Adder, all three sets of workloads exhibit an average detection capability around the 80% mark, with MiBench and OpenDCDiag having single outlier programs that reach 99% and 98% respectively. When it comes to the integer multiplier there is significantly more variability among the three frameworks. MiBench has an average detection capability of 53%, SiliFuzz 70% and OpenDCDiag 37%. The highest detection capability is achieved by a SiliFuzz generated workload and reaches 87% while MiBench and OpenDCDiag programs exhibit maximum detection capabilities of 67% and 58% respectively.

Fig. 6 shows the hardware coverage and the fault detection capability for the SSE FP Adder and Multiplier. Unlike the previous four hardware structures, the two SSE FP units are not utilized by many of the considered workloads. Thus, only 4 MiBench and half of the OpenDCDiag benchmarks have non-zero detection. OpenDCDiag performs best with a maximum detection capability of 98.5% for the SSE FP adder and 58.2% for the SSE FP multiplier. This is expected as much of its workloads are FP-heavy: MxM (Matrix Multiply), SVD (Singular Value Decomposition), etc.

[6]MiBench is widely used in similar studies [43], [45], [51]–[53], [61]–[64].

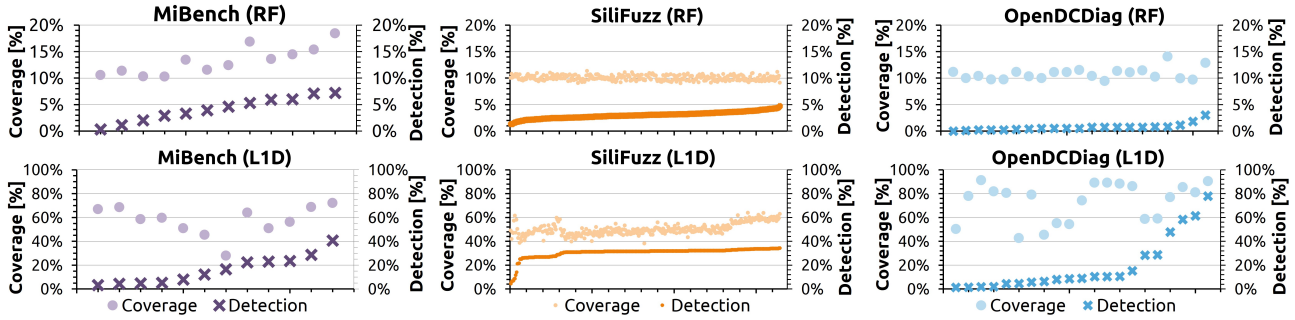

Fig. 4. Coverage (light dots, left y-axis) and Detection (dark crosses, right y-axis) for the IRF and L1D

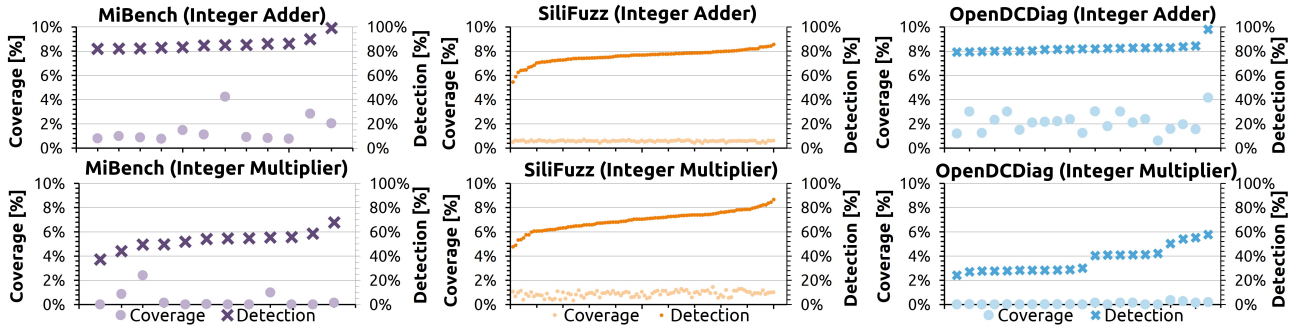

Fig. 5. Coverage (light dots, left y-axis) and Detection (dark crosses, right y-axis) for the integer adder and multiplier.

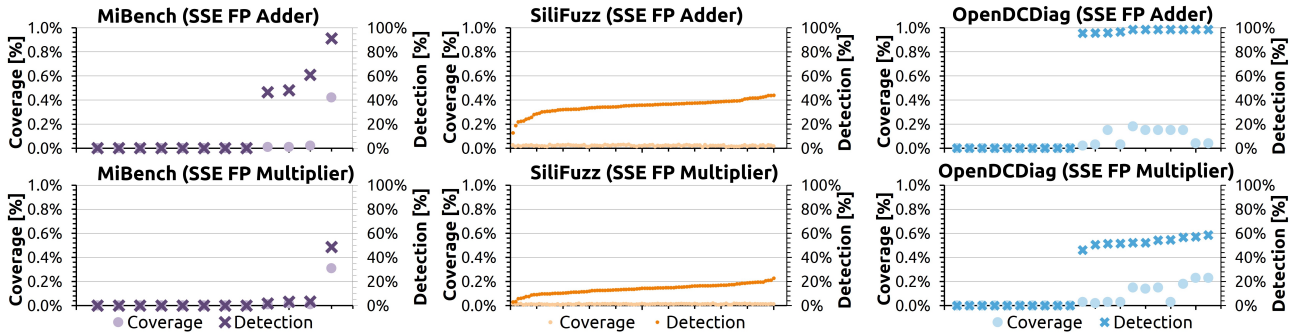

Fig. 6. Coverage (light dots, left y-axis) and Detection (dark crosses, right y-axis) for the SSE FP adder and SSE FP multiplier.

Considering the coverage results for FUs, while IBR does not serve as an upper bound for the detection capability in the manner ACE does for bit arrays, it does quantify the degree of ”exercising” of these components. Thus, the presence of numerous instances illustrating relatively high IBR alongside low detection capability suggests the occurrence of masking at the software level, as these programs seemingly propagate a small portion of the functional unit results to the output. From these results, it is clear, that there is a pressing need for improvement in the development of tests capable of efficiently identifying permanent gate-level faults in functional units.

While the detection capabilities achieved by the best workloads from MiBench, OpenDCDiag and Silifuzz for the Integer and SSE FP adders might appear satisfactory we show that *Harpocrates consistently reaches a detection* $> 99\%$ *in two orders-of-magnitude less CPU cycles*. Harpocrates employs a detailed CPU model in gem5 as a *fast grading engine* to iteratively improve the hardware coverage which positively correlates with fault detection capability in our approach.

IV. The *Harpocrates* System Architecture

The *Harpocrates* system is designed to produce and iteratively increase the quality of functional programs with respect to their hardware coverage and error detection capability. This section provides a high-level overview of the system, while the next section elaborates on our concrete instantiation of Harpocrates.

A. High-Level Description

Harpocrates consists of the main components shown as orange boxes in Fig. 7 which also depicts their interactions. We will elaborate on the specific tools we have chosen in the following section. Different tools that provide a similar functionality can be substituted in each spot.

The three main components are:

- ***Generator***: This component generates valid functional test programs. It also bootstraps the iterative refinement process by producing an initial test program population.
- ***Mutator***: Operating alongside the Generator, the Mutator alters the current set of programs or combines them to craft new variants (see more details in Section V-B1).
- ***Evaluator***: Acting as the driving force of the system, the Evaluator assesses all generated programs against a predefined metric (an objective function that we are optimizing for). Programs that perform best under this metric (fittest) are retained for subsequent mutation iterations, thereby directing the evolution of test programs towards those that optimize the objective function[7].

Harpocrates' flow resembles that of a genetic algorithm. We manage a collection of programs (i.e., the *population*). These programs undergo combination (akin to *genetic crossover*) and modification (*mutation*). The subsequent generation's population is then determined by an evaluation metric that serves as a *fitness function*. Existing systems like *Silifuzz* [29] are built around similar principles. In *Silifuzz*, the generation and mutation is mapped to the fuzzing component that alters and combines byte sequences that serve as test programs and the evaluator is mapped to the computation of the **software** coverage of the proxy (CPU emulator, disassembler, etc.).

[7]Different metrics can be used depending on the target structure, the fault model of interest, or other required parameters of the generated programs.

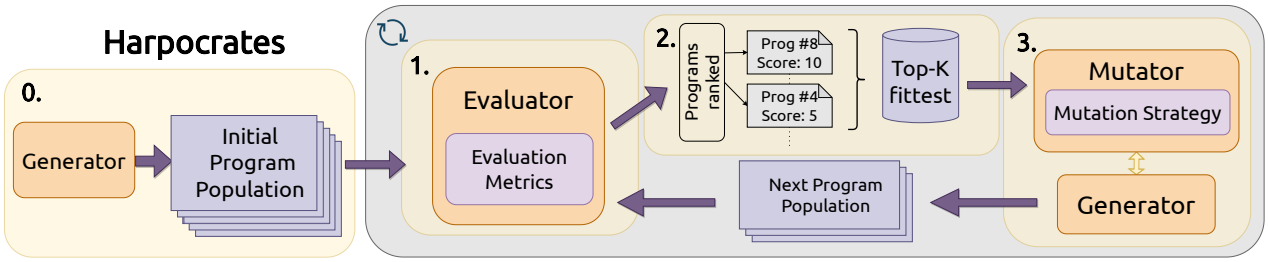

Fig. 7. The *Harpocrates* architecture and complete flow of operation during program refinement.

Therefore, SiliFuzz is *hardware-agnostic*; Harpocrates adopts a novel *hardware-in-the-loop* approach through the grading that the microarchitectural simulation engine provides.

Fig. 8 presents a simple example scenario, showcasing the different approaches of the two frameworks. We assume that the frameworks are tasked with creating test programs for an example OoO core which has two ALUs both of which can handle all integer operations. Moreover, the target structure for these tests is ALU #0. As previously mentioned, Silifuzz represents the program as a byte sequence mutating raw bytes with no internal notion of x86 encoding. This results in more than 2 out of 3 produced sequences being eventually unusable due to illegal instructions, etc. Silifuzz initially adds them to its test corpus, but later discards them as non-runnable. This percentage of discarded sequences coincides with our experimental data for Silifuzz as presented in section VI. Harpocrates on the other hand is ISA-aware and thus generates instruction sequences accurately encoding the operands and their types (notice how Harpocrates encodes that the ADD instruction's operands are both registers). Harpocrates relies on accurate hardware feedback from the execution of the program on the gem5 simulator to decide which of the mutated programs should advance to the next generation. In the example, the fitness function is set to the number of operations executed in ALU #0 (i.e. the target structure). When the SUB instruction is mutated to a DIV, that DIV is issued to ALU #1, stalling the unit for a large number of cycles. The rest of the additions are issued to ALU #0, which now executes 3 operations in total. Based on its increased fitness this program advances to the next generation while the other is dropped.

B. Flexibility and Uses of Harpocrates

The ISA, microarchitecture, optimization, and fault model parameters in Harpocrates can be flexibly adapted as follows:

- any ISA supported by the microarchitectural simulation engine[8] that performs the *Harpocrates* evaluation can be used (gem5 supports x86, Arm and RISC-V ISAs);
- any microarchitectural structure of a complex out-of-order CPU can be analyzed (currently, our fault injection infrastructure in gem5 models and injects faults in more than 70 different hardware structures);
- any "quality" metric can be used to guide the iterative refinement of the functional test program (hardware coverage metric tailored to particular structure);
- any fault model can be used for the evaluation of the final fault detection capability of programs.

Any combination of the above can be picked to match the requirements of a particular use case of *Harpocrates*. A short (non-exhaustive) list of use cases follows:

- Fast periodic scan of a CPUs fleet (the Ripple, in production testing mode, discussed in [33]): in this case, *Harpocrates* can be constrained to generate programs of certain short duration (cycles or instructions) maximizing their fault detection capability under this constraint.
- Comprehensive scan of CPUs fleet (the Fleetscanner, out of production testing mode discussed in [33]): in this case, *Harpocrates* is set to reach a certain (very high) fault detection capability without an execution time constraint.
- Electrical and environment conditions screening before silicon is shipped: silicon designers look for marginal defects [7], [10] that produce faults under certain conditions; when conditions require focus on a particular fault type or structure, *Harpocrates* can be configured for the purpose.

[8]An emulator can also be employed but the point in *Harpocrates* is to bring the hardware awareness which is missing from ISA emulators.

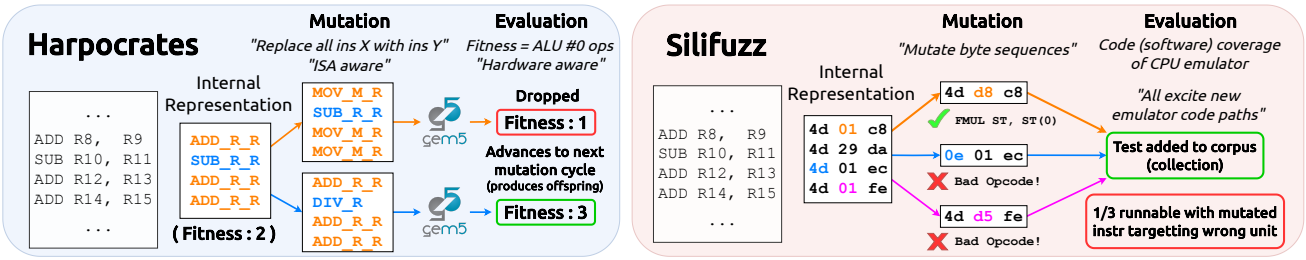

Fig. 8. *Harpocrates vs. Silifuzz* in a single step. Assumptions: Same starting sequence; OoO CPU with 2 ALUs; ALU #0 is the target structure.

V. Our Instantiation of Harpocrates: the Tools

We describe in detail the specific tools employed in our instantiation of *Harpocrates* through which we derived the experimental results present in the next section (and the baseline results presented in III-C). Our evaluation engine is the gem5 simulator enriched with hardware coverage analysis and fault detection measurement (through fault injection) capabilities.

Program generation and mutation is orchestrated via the *MuSeqGen* (Mutator and Sequence Generator) framework we have developed. The complete framework is depicted in Fig. 9.

A. MuSeqGen

A central piece of *MuSeqGen* is the code generator. Harpocrates utilizes and significantly extends the publicly available version of the MicroProbe [66] code generation framework to support a large subset of the (previously completely unsupported in MicroProbe) x86-64 ISA and all the additional complexities of generating valid programs in the x86-64 ISA. MicroProbe's publicly available implementation only covers the POWER and RISC-V ISAs. In addition, we have modified MicroProbe's tooling in order to support constrained *random* generation of programs. We also develop higher-level tooling to streamline the generation process and expose all configurable parameters. The next paragraphs introduce MicroProbe to familiarize the reader with the framework's terminology.

MicroProbe [32], [66] is an ISA-independent, modular, and extensible code generation framework. MicroProbe was initially developed to assist in accurate energy-related characterization of CPUs. The key feature of MicroProbe, is the separation of architecture (i.e., ISA) and microarchitecture-specific details from the code generation process.

Specifically (Fig. 9), the framework consists of two main modules: the *Architecture Module* and the *Code Generation Module*. The **Architecture Module** is composed of a set of configuration text files that specify all the ISA details (registers, instructions, formats, etc.) and, optionally, certain microarchitectural details (e.g., core components, cache configurations, latencies, etc.). All this information, and most importantly the ISA details, is queried during generation to produce the microbenchmark that is valid for the target ISA. The **Code Generation Module** orchestrates the code generation process. The microbenchmark being generated is encoded in an internal representation, which undergoes refining transformations via a set of compiler-like (user-specified) *passes*. These transformations handle: initialization, branch resolution, memory operand resolution, register allocation, etc. The sequence of passes that was specified to produce the final microbenchmark is collectively referred to as a *policy*.

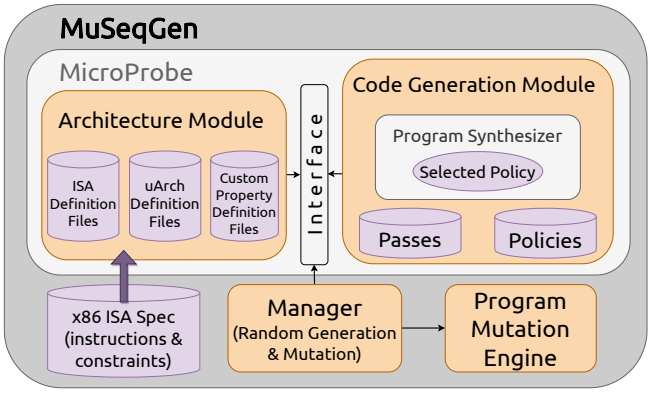

Fig. 9. The MuSeqGen framework structure. It corresponds to module #3 of our instantiation of the *Harpocrates* architecture (Fig. 7).

The collection of *passes* and the *policies* created from those passes are generic and reusable, thanks to the separation provided by the detachment of all architectural dependencies from code generation. For example, existing register allocation passes in the code generation module are compatible with any target ISA specified in the architecture module. The generation process is driven by the synthesizer object, to which we attach our sequence of passes (i.e., our policy). Ultimately, a microbenchmark must be lowered to a valid assembly. This requires a substantial amount of architectural information, which is retrieved through interfaced communication between the **Code Generation Module** and the **Architecture Module**.

MicroProbe's approach is an ideal fit for the generation of programs in Harpocrates due to the following features:

1) Extensibility: The framework is inherently highly extensible and modular, enabling our developed tooling to easily interface with the other *Harpocrates* modules.

2) ISA-awareness: The generated programs are always valid, provided all ISA constraints are encoded.

3) Assembly Level Codegen: This is critical because working at a higher level (e.g., with a high-level programming language) would deprive us of complete control over code generation. The compiler's back end would be responsible for instruction selection, favoring certain instructions over others and translating higher-level programming language constructs into specific patterns. A compiler aims for performance; however, in our case, such behavior during code generation would restrict the explored program space and limit our flexibility.

B. x86 support

Let us now delve into some implementation details concerning the support of the x86 ISA in *MuSeqGen*. For a new ISA to be supported, we need to provide all relevant configuration files in the Architecture Module (see Fig. 9). These configuration files collectively define the ISA. The extension of the Architecture Module for the x86 ISA involved the following key steps:

- Identifying an x86-64 ISA reference file in a structured, machine-readable format (e.g., an XML file).
- Developing a parser to interpret the file and convert it into a set of configuration files with specific formats.
- Incorporating the configuration files into the Architecture Module so that they can be utilized during the code generation step to produce valid x86-64 assembly.

This process results in our generator supporting $\approx 2,000$ x86 instruction variants, closely matching the extent of x86 support in the gem5 simulator. The x86-64 CISC ISA is significantly more complex than the two RISC ISAs already in

Microprobe and several subtle details needed to be accounted for to reliably produce valid, deterministic, and non-crashing x86-64 assembly.

Firstly, x86 supports several addressing modes. For Harpocrates' demonstration purposes in the experimental evaluation on six hardware structures, we implement support for RIP-relative as well as base + offset addressing. However, the flexibility of our framework allows for the implementation of any x86 addressing mode without any restriction.

Moreover, in x86, we have to consider the implicit operands of instructions when encoding them. For example, in x86-64, some variants of the MUL instruction implicitly store part of their result in RAX. If this case is disregarded, the code generator will be unable to prevent a situation like the following: Assume the RAX register is used as an address base register, but the generator inserts a MUL instruction. In this case, a segmentation fault is very likely during execution, as the address is most likely corrupted after the MUL operation.

Stack-related instructions like *PUSH* and *POP* and their variants can also produce crashing sequences, e.g., popping the empty stack. A related issue was the resulting stack alignment after the execution of the generated sequence. For x86-64 (or amd64) architecture, the stack alignment ABI requires the stack to be aligned on a 16-byte boundary when a function is called. We carefully insert a re-aligning snippet of instructions after the end of our generated sequence.

To detect corruptions, it is important to provide programs with deterministic output (i.e., executions of the program always provide the same output). To achieve this, the code generator excludes certain x86-64 instructions that are non-deterministic (as the authors of SiliFuzz also pointed out).

The aforementioned complexities are only a representative subset of the modifications and constraints we had to develop into the generator to reliably produce a non-crashing deterministic x86-64 assembly.

1) Mutation Engine: MuSeqGen introduces a program mutation engine (Fig. 9), closely integrated with the code generator. Having implemented and tested various sequence altering and recombination strategies (e.g., k-point crossover and variations on instruction replacement), we have experimentally settled on a specific variation of instruction replacement that replaces all the occurrences of a randomly selected instruction of the sequence, with another random instruction (the same mnemonics with different operand types are handled as distinct instructions). This strategy provides fairness in replacement due to the uniform selection of the replacement instruction. This policy's benefits are twofold: (1) We can optimize any objective function without tuning the mutation strategy separately, (2) We avoid the pitfall of specifying "too explicit" mutation strategies that would trivialize programs and narrow the explored ISA space or get stuck in local optima. The counterargument might be that a "targeted" mutation strategy converges faster, however, as we present in the evaluation section VI, our instruction replacement policy converges swiftly, without the aforementioned pitfalls.

2) Generation and Mutation Manager: The final piece of *MuSeqGen* is the Manager (bottom middle in Fig. 9), a cooperating collection of scripts that orchestrate the most common flows in the framework. The simplest one is configurable constrained random generation, whereas the more interesting flows involve the mutation engine as well. For example, we could instruct the Manager to orchestrate the following: 1) Generate 10 random programs, 2) Randomly mutate the instruction sequence of each generated program 5 times, 3) Generate programs from the 25 total mutated sequences, etc.; which is how the Harpocrates' loop is set in motion.

C. Complete Flow

The complete Harpocrates flow is shown in Fig. 7.

Step 0: The process starts with the Generator, *MuSeqGen*. The program generation engine produces (in parallel) the initial population of random test programs, based on any user-specified constraints. Given our generator's intrinsic understanding of the x86-64 ISA and its restrictions, it ensures every generated program results in valid assembly. This process is described in subsection V-D.

Step 1: Next is the evaluation step, conducted via the gem5 microarchitectural simulator [57]–[59], which serves as our evaluation/grading platform (III-B). Utilizing gem5 enables us to harness a wide collection of program execution metrics (akin to hardware counters but much more expansive). The rich microarchitectural-level data available through simulation permits the creation of sophisticated metrics and analyses. One such analysis is the ACE-like analysis we presented in subsection II-D. In this phase, the functional test programs are simulated in parallel in gem5 and our predefined evaluation metric is computed, for each one, as a *fitness score* (which corresponds to the hardware coverage metric). The result of the evaluation step is the ranking of the current generation of test programs based on their fitness scores.

Step 2: In the next step, we perform selection. Our strategy in this step is straightforward. We advance the *top-K* programs that maximize our metrics to the following step.

Step 3: In this step, the Mutator alters the *top-K* programs according to the strategy described in V-B1, to produce the next generation of programs. The mutation engine is integrated within *MuSeqGen*, leveraging the available ISA information, which allows us to express various mutation strategies. This integration ensures that mutations comply with ISA constraints.

The new set of test programs goes through evaluation again, starting the next cycle of the iterative assessment and mutation flow. The process is repeated until our metrics converge.

D. The x86 Program Generation of MuSeqGen

MuSeqGen starts the generation process by encoding the program structure via variable-sized basic blocks connected in a control flow graph (CFG). The next step involves selecting instructions for these blocks, which can be done either by importing sequences from external tools like MuSeqGen's mutation engine, manual specification (for smaller programs), or allowing the generator to autonomously choose instructions

based on uniform or user-defined distributions. After populating the basic blocks with instructions, the focus shifts to assigning operands. Register allocation is configurable, allowing strategies (i.e., passes) such as constant register dependency distance, random allocation subject to ISA constraints, round-robin, etc. Any allocation algorithm can be integrated into the framework. Address resolution for memory operands necessitates the definition of memory regions within which the operands are validly addressed. These regions and the access patterns defined on them are completely configurable. Consider an access pattern that iterates over a region with a fixed stride; in a 32 KB region for example, with a stride of 32 bytes, we could resolve operands to 1K different addresses. Finally, immediate operands are resolved by uniform sampling across their whole range, while branch target resolution requires careful design to maintain consistent control flow despite variations in operands, memory, or initial state.

For our objectives and the hardware structures we analyze, we have determined that the generation of instructions within a single basic block is sufficient and provides the desirable level of structural simplicity. Our programs thus consist of a linear sequence of instructions. To ensure this linear execution path, all branches resolve to the subsequent instruction, equating taken and not-taken paths. Register allocation maximizes the dependency distance, to provide a balance between high ILP and data flow propagation. Memory operands are always resolved in a round-robin fashion and within a cache-sized designated memory space with a fixed stride. These parameters were chosen for our target components after carrying out a detailed sensitivity analysis which we omit due to space limitations. Had we been targeting different structures (e.g., caching hierarchy logic, page-fault logic) our parameters and more importantly memory passes and instruction sequence size would have been configured accordingly.

After resolving operands, the final step involves generating valid assembly instructions for the target ISA. However, raw assembly alone cannot produce a standalone binary; initialization of registers and memory is crucial for producing a fixed end-state output, vital for error detection. MuSeqGen handles this through configurable *wrappers*, which encompass raw assembly, allowing for initialization and output computation code. The wrapper, typically a minimal C program, encodes the core test program as an inline assembly within the main function. It manages register and memory initialization and ensures appropriate *warmup* code so that all core test instructions are run under a similar hardware state (i.e., consistency). An added benefit with a C code wrapper is that producing and executable binary for our test is simple; we just pass through a C compiler. The test's output includes the final state of architectural registers and a signature over accessed memory regions. To ensure accurate evaluation metrics, gem5 directives are set to isolate core program instructions from initialization or output computation.

VI. Experimental Evaluation

Below, we discuss the Harpocrates-generated programs fault detection effectiveness, as well as other properties. All experiments are performed on a dual-socket AMD EPYC 7402 24-Core CPU (96 hyperthreads) with 128GB of DDR4 RAM.

A. Harpocrates Performance Evaluation

Harpocrates' generation and evaluation speed can be compared to SiliFuzz, which is the only alternative automated methodology available for this purpose. We calculate the effective instruction generation rate for both frameworks.

1) SiliFuzz: The first step involves fuzzing the proxy to accumulate a set of "interesting" inputs, which adequately cover the proxy's code. We performed fuzzing on the Unicorn x86 emulator that SiliFuzz uses, on all cores, for 40 minutes. This step yielded 635,587 test inputs. Subsequently, only the test inputs (i.e., programs) that are non-crashing and deterministic are picked out. This sorting step lasted 5 minutes. Only about one-third of the programs were kept, amounting to 173,731 runnable, deterministic, short programs. The average SiliFuzz-generated program contains 18.6 instructions, for a total of 3,230,528 runnable instructions produced in 45 minutes, or ≈1,200 instructions per second.

2) Harpocrates: As Table I shows Harpocrates completes a mutation / generation / evaluation cycle in 13.35 seconds, for 96 programs of 5K instructions each. This amounts to 480,000 generated and evaluated instructions, or ≈36,000 runnable instructions per second. **Harpocrates (runnable and evaluated) instruction generation rate is 30x that of Silifuzz.**

Another point to be drawn out, is the impracticality of a flow that refines programs by directly measuring detection capability through SFI. In such a flow, a single iteration takes several hours, instead of the seconds shown in Table I.

B. Harpocrates Parameters and Convergence

Harpocrates is configured and operates as follows.

1) Integer Register File: Program size is set to 10K instructions in the generator. Mutation is performed via instruction replacement as presented in V-B1, thus the size remains constant through iterations. The hardware coverage (reward function) being optimized with every iteration is computed by the ACE lifetime analysis detailed in II-D. In each iteration, we evaluate 96 programs (i.e., the maximum number of hardware threads in our setup; Harpocrates exploits the full parallelism of any CPU configuration), only keeping the top 16. The next population is produced by mutating each of the top 16 programs 6 times, producing 96 offspring programs. The convergence of this process is shown in the top-left graph of Fig. 10. Harpocrates iterated 10,000 times in total; the graph only shows the coverage values measured for the top-16 programs every 1,000 iterations. This explains the upward "gaps" seen every 16 dots in the graph; we advance by

TABLE I
Harpocrates Single Loop Step Duration Breakdown.

Step	Mutation	Generation	Compilation	Evaluation	**Total**
Time	0.51s	9.18s	1.12s	2.54s	13.35s

1,000 iterations. Harpocrates has reached convergence much earlier than the 10,000 iterations, at about 5,000 iterations. We showcase the full graph to illustrate that the maximum coverage is retained for subsequent iterations. Harpocrates attains **more than 3x** the coverage of the best-performing tests (w.r.t coverage, Fig. 4 top graphs) in the other frameworks. The final measure of test quality is, however, the detection capability, which we compare in the following subsection.

2) L1 Data Cache: The setup is similar to the Register File with two differences. First, the program size is now set to 30K instructions. Second, all memory references are resolved sequentially with a stride of 8 bytes in a 32KB region. Both of these constraints enable the generated programs to adequately exercise the data cache. The 32KB memory region is intentionally chosen; our data cache has that exact capacity. Coverage is mapped once again to the ACE analysis but only considering the bits of the data cache this time. Harpocrates converges after approximately 2,000 iterations, attaining a maximum coverage of **95%**, as shown in the top-right graph in Fig. 10. Unlike the coverage graph for the IRF the coverage progression has no upward jumps due to the smaller sampling interval. Specifically, we show the coverage of the top 16 programs every 100 iterations. Finally, notice the high starting point (≈77% in the first generation) due to the cache-aware constraints we imposed on program generation.

3) Integer Adder: Program size is set to 5K instructions. Mutation is still the instruction replacement presented in V-B1, however we now map our coverage function to IBR, explained in the last paragraph of II-D. Each iteration evaluates 32 programs, only keeping the top 8. Each of the top 8 programs is mutated 4 times to produce the subsequent generation. In this scenario, Harpocrates manages to converge in just over 1,000 iterations, saturating to ≈10% coverage (IBR), as shown in the middle-left graph in Fig. 10. Again, the graph is sampled, showing the coverage of the top programs every 100 iterations. It is clear that the smaller population and program size chosen are perfectly adequate for strong convergence, speeding up the full execution of Harpocrates to under 2 hrs.

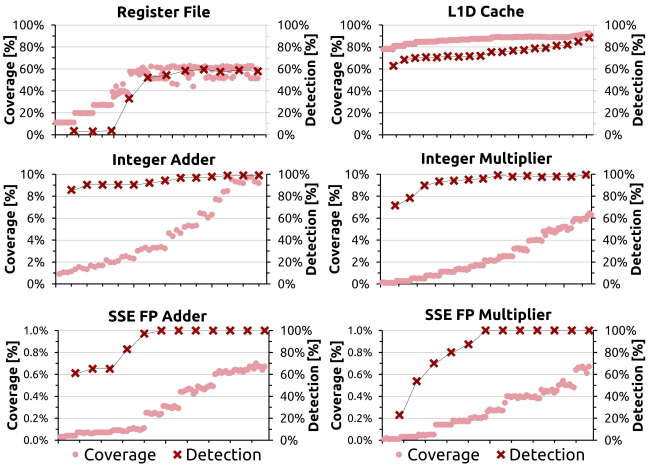

Fig. 10. Coverage (light dots, left y-axis) and Detection (dark crosses, right y-axis) for all components, measured across Harpocrates optimization.

4) Integer Multiplier: We employ an identical setup to the integer adder. Again, Harpocrates converges in 1,000 iterations, saturating to a 6% coverage (IBR). In both functional units examined, Harpocrates achieves **2x higher** coverage than the best programs from other frameworks (as seen when comparing the middle graphs of Fig. 10 to Fig. 5).

5) SSE FP Adder: The setup is similar to the integer functional units. For the SSE FP adder, Harpocrates' converges in 5,000 iterations, saturating to a 7% coverage (IBR metric). However, our SFI experiments revealed that detection capability peaks much earlier (step 500) stabilizing just below 100% and thus we present data up to step 1,100 (sampling every 100 steps) to better illustrate the progress curve of Harpocrates.

6) SSE FP Multiplier: For the SSE FP multiplier, we configure Harpocrates similarly to the previous units and observe convergence in 5,000 iterations, saturating to a 5% coverage (IBR metric). Again, detection capability peaks much earlier (step 600) and thus we truncate the data to step 1,100 to show the progress of Harpocrates generation. The early peaks in detection for both SSE FP components can be attributed to the fact that the XMM registers are accessed with relatively fewer instructions than general registers, and thus there is a smaller probability that a fault that has propagated in those registers will be masked by a subsequent operation. Compared to the three other frameworks for both the SSE FP units Harpocrates achieves more than **10x increased** coverage.

A critical observation, which confirms the crux assumption in our methodology, can be drawn from Fig. 10: For all the components it is evident that **increasing the coverage** of our test program population (through the Harpocrates' methodology) **translates to increases in detection capability**. Thus, our coverage metrics, carefully selected for the combination of fault type and hardware structure, are positively correlated with the measured detection capability of our programs. This is the key insight of Harpocrates; relying on *accurate, hardware-aware, quantitative feedback* to evolve highly-specialized programs targeted at specific microarchitectural structures. This automated, targeted, hardware-aware evaluation is the key novelty of our framework, contrary to other approaches which might rely on a priori expert knowledge on microarchitecture or hardware-"blind" evaluation metrics like Silifuzz.

Another interesting point is the discrepancy observed in detection capability. Specifically, injected transient faults in the two bit-array components are much harder to detect (for all frameworks) compared to permanent faults in the functional units. This is mainly due to fault characteristics: transients are easily overwritten, while permanents persist across the whole execution. Another reason is that the IRF and L1 data cache are microarchitected and present complex dynamic behaviour (adequately covering them in a generated test is harder), whereas the functional units are primarily exercised by specific instructions. The "difficulty" of the two scenarios is also apparent from the convergence of Harpocrates: For all functional units ≈1000 loops of refinement produce excellent programs. For the data cache, we need more than 2000 iterations and for the IRF coverage (and detection) reach

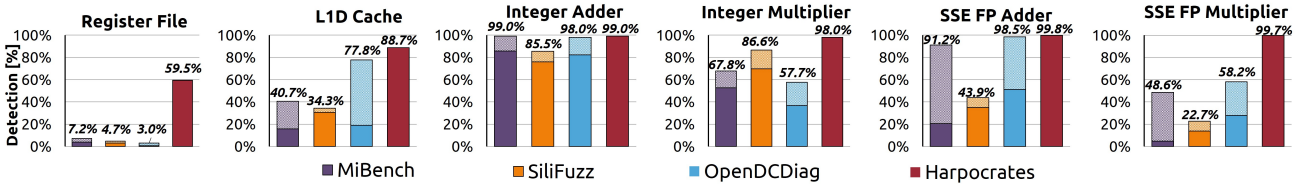

Fig. 11. Maximum (top of bars) and Average (middle of bars when shown) detection values for each method and for each of the six hardware structures.

their peak at about 5000 iterations. It is thus experimentally "harder" for Harpocrates to converge on good solutions for bit arrays (studied for transient faults) compared to functional units (studied for permanent faults).

C. Harpocrates Fault Detection and Full Comparisons

This section compares and discusses the fault detection capability for all frameworks for the six hardware components of this study (Fig. 11). The results from the open source tests (Silifuzz and OpenDCDiag) and the MiBench workloads are shown to put things in perspective and showcase how Harpocrates can generate tests that adapt to different components and fault modes, providing excellent detection capability.

1) Integer Register File: the Harpocrates-generated program manages to detect nearly **10x** more faults compared to the other frameworks, as shown in the leftmost graph in Fig. 11. The register file structure is very challenging for transient fault detection due to the very high rates of register allocation and freeing during program execution. Our uniform random instruction replacement policy guided by feedback from the ACE coverage metric allows us to generate instruction patterns that maximize program bits exposed to transient faults in the IRF. It is evident that these patterns do not occur in "typical" workloads (MiBench, OpenDCDiag) nor in Silifuzz-generated programs..

2) L1 Data Cache: Some OpenDCDiag framework tests reach high fault detection capability for the L1D cache (almost 80%) as we can see in the second from left graph in Fig. 11. Harpocrates' final tests for the data cache manage an even higher detection capability for transients, **approaching 90%**.

3) Integer Adder: In this structure, every framework's best programs nearly fully detect all gate-level stuck-at faults injected (with SiliFuzz slightly behind the rest). However, the average test of the other frameworks provides poor coverage of the permanent faults (third from the left graph Fig. 11).

4) Integer Multiplier: In this structure, SiliFuzz performs much better than OpenDCDiag and MiBench, which are lacking a test with significant activity in the integer multiplier. Harpocrates' final tests reach **nearly 100%** detection.

5) SSE FP Adder: On average across MiBench, SiliFuzz and OpenDCDiag the detection capability of SSE FP adder faults is low, as seen in the second from the right graph of Fig. 11, mainly due to lack of SSE activity in most workloads as mentioned in Sec. III-C. There are multiple outlier programs (mainly in OpenDCDiag) that achieve high detection capability in this component (up to 98.5%). Harpocrates achieves **99.8% detection** in orders-of-magnitude less CPU cycles.

6) SSE FP Multiplier: Similarly to the adder, SSE FP multiplier detection capability is even lower on average for non-Harpocrates workloads. In contrast to the adder, maximum detection capability is achieved by a single OpenDCDiag program and is only 58.2%, very low for permanent faults. The Harpocrates generated program reaches **99.7% detection**. Overall, Harpocrates is the only framework to manage **nearly full fault detection in all functional units we compared**.

The excellent detection capability demonstrated by Harpocrates' tests can be elucidated by referring to the coverage metrics outlined in Section II-D. For bit array components, ACE serves as a coverage metric that establishes an upper limit for detection capability. As illustrated in Fig. 10, the gaps between the coverage and detection capability of our generated programs are minimal, indicating negligible software masking, due to the careful parameterization of our generator.

A similar observation can be made for the functional units. The IBR metric signifies the "exercising" of functional units, and although it does not establish an upper limit for detection, a strong correlation between the two is evident. This suggests that our programs propagate most functional unit results to their output. In contrast, a significant portion of workloads in MiBench, SiliFuzz, OpenDCDiag exhibit a low correlation between these metrics, indicating significant software masking.

An additional advantage of our approach lies in the detection speed facilitated by our tests. While a single MiBench program matches our 99% detection in the integer adder, it does so **in more than 11 million cycles**. In contrast, Harpocrates' programs can attain the same, or even superior, detection capability in a significantly shorter timeframe. Specifically, 99% detection of the injected permanent faults is accomplished **in only 50,000 cycles**, approximately **220 times faster**. This rapid detection speed is also consistently observed for test programs directed at the multiplier. In comparison to the best competition program (SiliFuzz-generated), which only manages to detect **86.6%** of all injected faults with a similar runtime to our generated test, **our hardware-in-the-loop approach achieves 99.5%**.

D. The unexpected bug detection in gem5 via Harpocrates

An unexpected finding during our experimental evaluation, shed extra light into Harpocrates' capabilities. A set of Harpocrates-generated programs *revealed* an internal assertion error in gem5 simulator (v22.0.0.2). Tracking down the root cause of this assertion error, we found an instruction emulation bug with the RCR x86 instruction, fixed in later versions [67] of gem5. The simulation crashes in the corner-case where the rotate amount is equal to the size of the rotated register.

VII. Related Work

Various approaches and studies related to the detection, tolerance, and impact of Silent Data Corruptions (SDCs) in computing systems have been presented. Some employ redundancy by executing identical processes on multiple replicas and comparing their outcomes to detect and potentially correct errors [68]–[71]. Hardware-based redundancy, like the dual-core lockstep technique [72] is too costly for widespread use, despite its effectiveness for critical applications. Other works use ML [73]–[79] to forecast SDC occurrences. Some predict result ranges and identify errors when the actual result falls outside the predicted range [73]–[75]. Previous studies focused on silent errors induced primarily by transient radiation rays [80]–[82] and noted SDCs caused by faulty processors in cloud service environments [1], [33], [83]. The text refers to Meta's case study on SDCs in production [1].

SDCs also originate from other components: disks, memory [81], [82], [84], TPUs [85], and corner-case or electrical bugs that may occur under certain conditions in a hardware unit and escape into volume production [12], [13], [86]–[88]. Environmental conditions such as temperature and humidity affect electronic devices [89], [90] with core temperature identified as a trigger for SDCs. Further, prior studies provide in-depth evaluations of SDCs under diverse conditions, using physical experiments or simulators. Fault injection is widely used to evaluate system reliability. Methods range from using neutron beams based on irradiation models [80], [82], [85] to simulators, experimental devices for fault injection [4]–[6], [45], [52], [91] or combining both beam experiments and fault injection [41], [51], [64]. The aim is to improve injector designs based on observations, to better assess SDC solutions.

Previous research has explored the generation of stress tests via genetic algorithms in different contexts and objectives: maximization of power consumption and dI/dt voltage noise [92]–[97], or stress viruses for soft errors [62], [98], [99]. Most of these approaches aim for *worst-case instruction sequences* only, which naturally results in a few distinct instructions executed in a tight loop. In particular [99] operates at the register transfer level and aims to maximize metrics that can potentially increase the failure rate due to soft errors. Other types of faults are not analyzed and, as the authors admit, the paper does not provide any validation for the correlation between the targeted metrics and the failure rate (through fault injection). In the context of the CPU faults problem, such approaches would be primarily useful as a prologue: to establish extreme stress conditions that can trigger an intermittent fault resulting from a defect, particularly a marginal defect [7], [11]. Marginal defects are nowadays the most challenging to detect through manufacturing and in-package testing, thus driving the development of continuous fault screening methodologies through the system lifetime, such as Harpocrates. Therefore, Harpocrates is orthogonal to these studies; it mutates much longer, diverse instruction sequences to stimulate hardware components, increasing coverage and thus the defect detection capability.

VIII. Conclusion

Harpocrates is a novel methodology for the automatic generation of functional test programs for the detection of CPU faults throughout the lifetime of computing systems. It is a major contributor to existing frameworks, complementing them with the novel hardware-model-in-loop approach to iteratively refine hardware coverage leading to increased detection capability. It can be configured to produce effective programs for different microarchitectural structures, ISAs, fault types, and for different optimization targets (e.g., short duration or high fault detection). We demonstrated Harpocrates' effectiveness on multiple important hardware structures of a modern x86 microarchitecture for different fault types.

Acknowledgements

This work was supported by research gifts from AMD and Meta, as well as the European Union's Horizon Europe research and innovation programme under grant agreements No 101093062 (Vitamin-V), No 101070238 (NEUROPULS), and No 101097224 (REBECCA). Views and opinions expressed are however, those of the authors only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them. The authors would like to thank Ramon Bertran from IBM Research and the anonymous ISCA 2024 reviewers for their insightful comments and feedback.

References

[1] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent Data Corruptions at Scale," 2021. [Online]. Available: https://arxiv.org/abs/2102.11245

[2] 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*, ser. HotOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 9–16. [Online]. Available: https://doi.org/10.1145/3458336.3465297

[3] 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*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 216–230. [Online]. Available: https://doi.org/10.1145/3600006.3613149

[4] G. Papadimitriou and D. Gizopoulos, "Silent data corruptions: Microarchitectural perspectives," *IEEE Transactions on Computers*, vol. 72, no. 11, pp. 3072–3085, 2023.

[5] G. Papadimitriou, D. Gizopoulos, H. D. Dixit, and S. Sankar, "Silent data corruptions: The stealthy saboteurs of digital integrity," in *2023 IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2023, pp. 1–7.

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

[7] 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*, 2014, pp. 1–8.

[8] U. Baruch, "Assuring reliable processor performance at scale," Semiconductor Engineering, March 8, 2022. [Online]. Available: https://semiengineering.com/assuring-reliable-processor-performance-at-scale/

[9] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," 2021. [Online]. Available: https://arxiv.org/abs/2102.02308

[10] S. Gurumurthi, V. Sridharan, and S. Gurumurthy, "Emerging fault modes: Challenges and research opportunities," ACM SIGARCH, July 17, 2023. [Online]. Available: https://www.sigarch.org/emerging-fault-modes-challenges-and-research-opportunities/#

[11] T. C. I. S. . I. 43.040.10, "Iso/tr 9839:2023, road vehicles, application of predictive maintenance to hardware with iso 26262-5," ISO, August, 2023. [Online]. Available: https://www.iso.org/standard/83605.html

[12] G. Papadimitriou, D. Gizopoulos, A. Chatzidimitriou, T. Kolan, A. Koyfman, R. Morad, and V. Sokhin, "Unveiling difficult bugs in address translation caching arrays for effective post-silicon validation," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016, pp. 544–551.

[13] Y. Sazeides, A. Gerber, R. Gabor, A. Bramnik, G. Papadimitriou, D. Gizopoulos, C. Nicopoulos, G. Dimitrakopoulos, and K. Patsidis, "Idld: Instantaneous detection of leakage and duplication of identifiers used for register renaming," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 799–814.

[14] T. Kolan, H. Mendelson, V. Sokhin, S. Doron, H. Theiler, S. Aviv, H. Hadad, N. Freidman, E. Tsanko, J. Ludden, and B. Cockcroft, "Post silicon validation of the mmu," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 212–217.

[15] A. Meixner, "Auto chipmakers dig down to 10ppb," Semiconductor Engineering, March 8, 2022. [Online]. Available: https://semiengineering.com/auto-chipmakers-dig-down-to-10ppb/

[16] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 29–.

[17] J. Shen and J. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," in *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, 1998, pp. 990–999.

[18] D. Gizopoulos, A. Paschalis, and Y. Zorian, *Embedded Processor-Based Self-Test*. Springer, 2004. [Online]. Available: https://doi.org/10.1007/978-1-4020-2801-4

[19] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing," *IEEE Design & Test of Computers*, vol. 27, no. 3, pp. 4–19, 2010.

[20] S. V. Kodakara, M. V. Sagar, and J. Yuen, "Extracting effective functional tests from commercial programs," in *2015 IEEE 33rd VLSI Test Symposium (VTS)*, 2015, pp. 1–6.

[21] S. Gurumurthy, S. Vasudevan, and J. A. Abraham, "Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor," in *2006 IEEE International Test Conference, ITC 2006, Santa Clara, CA, USA, October 22-27, 2006*, S. Davidson and A. Gattiker, Eds. IEEE Computer Society, 2006, pp. 1–9. [Online]. Available: https://doi.org/10.1109/TEST.2006.297676

[22] S. Gurumurthy, M. Pratapgarhwala, C. Gilgan, and J. Rearick, "Comparing the effectiveness of cache-resident tests against cycleaccurate deterministic functional patterns," in *2014 International Test Conference, ITC 2014, Seattle, WA, USA, October 20-23, 2014*. IEEE Computer Society, 2014, pp. 1–8. [Online]. Available: https://doi.org/10.1109/TEST.2014.7035348

[23] N. Foutris, M. Psarakis, D. Gizopoulos, A. Apostolakis, X. Vera, and A. Gonzalez, "Mt-sbst: Self-test optimization in multithreaded multicore architectures," in *2010 IEEE International Test Conference*, 2010, pp. 1–10.

[24] F. Angione, P. Bernardi, A. Calabrese, L. Cardone, A. Niccoletti, D. Piumatti, S. Quer, D. Appello, V. Tancorre, and R. Ugioli, "An innovative strategy to quickly grade functional test programs," in *2022 IEEE International Test Conference (ITC)*, 2022, pp. 355–364.

[25] C.-P. Wen, L.-C. Wang, K.-T. Cheng, K. Yang, W.-T. Liu, and J.-J. Chen, "On a software-based self-test methodology and its application," in *23rd IEEE VLSI Test Symposium (VTS'05)*, 2005, pp. 107–113.

[26] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto, and M. S. Reorda, "Test program generation for communication peripherals in processor-based soc devices," *IEEE Design & Test of Computers*, vol. 26, no. 2, pp. 52–63, 2009.

[27] M. Hatzimihail, M. Psarakis, D. Gizopoulos, and A. Paschalis, "A methodology for detecting performance faults in microprocessors via performance monitoring hardware," in *2007 IEEE International Test Conference*, 2007, pp. 1–10.

[28] "Google, SiliFuzz - Fuzzing CPUs by proxy," https://github.com/google/silifuzz.

[29] K. Serebryany, M. Lifantsev, K. Shtoyk, D. Kwan, and P. Hochschild, "Silifuzz: Fuzzing cpus by proxy," 2021. [Online]. Available: https://arxiv.org/abs/2110.11519

[30] "Intel Corporation, OpenDCDiag," https://github.com/opendcdiag/opendcdiag.

[31] "AMD, OpenFieldHealthCheck," https://github.com/amd/Open-Field-Health-Check.

[32] R. Bertran, A. Buyuktosunoglu, M. S. Gupta, M. Gonzalez, and P. Bose, "Systematic energy characterization of cmp/smt processor systems via automated micro-benchmarks," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 199–211.

[33] H. D. Dixit, L. Boyle, G. Vunnam, S. Pendharkar, M. Beadon, and S. Sankar, "Detecting silent data corruptions in the wild," 2022.

[34] *Digital System Testing and Testable Design*. Wiley - IEEE Press, 1990. [Online]. Available: https://ieeexplore.ieee.org/servlet/opac?bknumber=5266057

[35] D. Gil-Tomás, J. Gracia-Morán, J.-C. Baraza-Calvo, L.-J. Saiz-Adalid, and P.-J. Gil-Vicente, "Analyzing the impact of intermittent faults on microprocessors applying fault injection," *IEEE Design & Test of Computers*, vol. 29, no. 6, pp. 66–73, 2012.

[36] V. Nelson, "Fault-tolerant computing: fundamental concepts," *Computer*, vol. 23, no. 7, pp. 19–25, 1990.

[37] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, 2004, pp. 264–275.

[38] L. Peters, "New insights into ic process defectivity," Semiconductor Engineering, November 7, 2023. [Online]. Available: https://semiengineering.com/new-insights-into-ic-process-defectivity/

[39] A. Meixner, "Screening for silent data errors," Semiconductor Engineering, January 10, 2023. [Online]. Available: https://semiengineering.com/screening-for-silent-data-errors/

[40] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos, "Differential fault injection on microarchitectural simulators," in *2015 IEEE International Symposium on Workload Characterization*, 2015, pp. 172–182.

[41] P. R. Bodmann, G. Papadimitriou, R. L. Rech Junior, D. Gizopoulos, and P. Rech, "Soft error effects on arm microprocessors: Early estimations versus chip measurements," *Computer*, vol. 56, no. 7, pp. 4–6, 2023.

[42] A. Chatzidimitriou, G. Papadimitriou, C. Gavanas, G. Katsoridas, and D. Gizopoulos, "Multi-bit upsets vulnerability analysis of modern microprocessors," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, 2019, pp. 119–130.

[43] A. Chatzidimitriou, G. Papadimitriou, D. Gizopoulos, S. Ganapathy, and J. Kalamatianos, "Assessing the Effects of Low Voltage in Branch Prediction Units," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 127–136. [Online]. Available: https://doi.org/10.1109/ISPASS.2019.00020

[44] I. Tsiokanos, G. Papadimitriou, D. Gizopoulos, and G. Karakonstantis, "Boosting Microprocessor Efficiency: Circuit- and Workload-Aware Assessment of Timing Errors," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021, pp. 125–137. [Online]. Available: https://doi.org/10.1109/IISWC53511.2021.00022

[45] G. Papadimitriou and D. Gizopoulos, "Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA '21)*, 2021, pp. 902–915. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00075

[46] ——, "Characterizing Soft Error Vulnerability of CPUs Across Compiler Optimizations and Microarchitectures," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021, pp. 113–124. [Online]. Available: https://doi.org/10.1109/IISWC53511.2021.00021

[47] ——, "Avgi: Microarchitecture-driven, fast and accurate vulnerability assessment," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 935–948.

[48] 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.

[49] S. Mukherjee, *Architecture Design for Soft Errors*. Morgan Kaufmann, 2011. [Online]. Available: https://doi.org/10.1016/b978-0-12-369529-1.x5001-0

[50] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, 2009, pp. 502–506.

[51] P. R. Bodmann, G. Papadimitriou, R. L. R. Junior, D. Gizopoulos, and P. Rech, "Soft Error Effects on Arm Microprocessors: Early Estimations versus Chip Measurements," *IEEE Transactions on Computers*, vol. 71, no. 10, pp. 2358–2369, 2022. [Online]. Available: https://doi.org/10.1109/TC.2021.3128501

[52] G. Papadimitriou and D. Gizopoulos, "Anatomy of On-Chip Memory Hardware Fault Effects Across the Layers," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–12, 2022. [Online]. Available: https://doi.org/10.1109/TETC.2022.3205808

[53] P. Bodmann, G. Papadimitriou, D. Gizopoulos, and P. Rech, "The Impact of SoC Integration and OS Deployment on the Reliability of Arm Processors," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021, pp. 223–225. [Online]. Available: https://doi.org/10.1109/ISPASS51385.2021.00040

[54] G. Papadimitriou, A. Chatzidimitriou, and D. Gizopoulos, "Adaptive Voltage/Frequency Scaling and Core Allocation for Balanced Energy and Performance on Multicore CPUs," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 133–146. [Online]. Available: https://doi.org/10.1109/HPCA.2019.00033

[55] "OCP, Hardware Management Talks: Server Component Resilience," https://www.opencompute.org/events/past-events/2023-ocp-hardware-management-tech-talks.

[56] D. P. Lerner, B. Inkley, S. H. Sahasrabudhe, E. Hansen, L. D. R. Munoz, and A. v. de Ven, "Optimization of tests for managing silicon defects in data centers," in *2022 IEEE International Test Conference (ITC)*, 2022, pp. 578–582.

[57] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[58] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," 2020. [Online]. Available: https://arxiv.org/abs/2007.03152

[59] "gem5 GitHub Repository," https://github.com/gem5/gem5, accessed: 2023-04-25.

[60] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14. [Online]. Available: https://doi.org/10.1109/WWC.2001.990739

[61] N. J. George, C. R. Elks, B. W. Johnson, and J. Lach, "Transient fault models and avf estimation revisited," in *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, 2010, pp. 477–486.

[62] A. A. Nair, L. K. John, and L. Eeckhout, "Avf stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 125–136.

[63] D. S. Khudia and S. Mahlke, "Harnessing soft computations for low-budget fault tolerance," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 319–330.

[64] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos, and P. Rech, "Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2019, pp. 26–38. [Online]. Available: https://doi.org/10.1109/DSN.2019.00018

[65] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–10.

[66] "IBM, Microprobe," https://github.com/IBM/microprobe.

[67] "gem5 RCR emulation bugfix," https://github.com/gem5/gem5/commit/1dd30723c4370526a092c536889b1deeab2658eb.

[68] S. Mukherjee, J. Emer, and S. Reinhardt, "The soft error problem: an architectural perspective," in *11th International Symposium on High-Performance Computer Architecture*, 2005, pp. 243–247.

[69] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Washington, DC, USA: IEEE Computer Society Press, 2012.

[70] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about eve: Execute-Verify replication for Multi-Core servers," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 237–250. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kapritsos

[71] G. Zuo, "Tolerate silent data errors with coded computation," 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:253100909

[72] A. B. de Oliveira, G. S. Rodrigues, F. L. Kastensmidt, N. Added, E. L. A. Macchione, V. A. P. Aguiar, N. H. Medina, and M. A. G. Silveira, "Lockstep dual-core arm a9: Implementation and resilience analysis under heavy ion-induced soft errors," *IEEE Transactions on Nuclear Science*, vol. 65, no. 8, pp. 1783–1790, 2018.

[73] L. Bautista-Gomez and F. Cappello, "Exploiting spatial smoothness in hpc applications to detect silent data corruption," in *Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conf on Embedded Software and Systems*, ser. HPCC-CSS-ICESS '15. USA: IEEE Computer Society, 2015, p. 128–133. [Online]. Available: https://doi.org/10.1109/HPCC-CSS-ICESS.2015.9

[74] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello, "Lightweight silent data corruption detection based on runtime data analysis for hpc applications," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 275–278. [Online]. Available: https://doi.org/10.1145/2749246.2749253

[75] S. Di, E. Berrocal, and F. Cappello, "An efficient silent data corruption detection method with error-feedback control and even sampling for hpc applications," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015, pp. 271–280.

[76] C. Liu, J. Gu, Z. Yan, F. Zhuang, and Y. Wang, "Sdc-causing error detection based on lightweight vulnerability prediction," in *Asian Conference on Machine Learning*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:204857839

[77] N. Yang and Y. Wang, "Identify silent data corruption vulnerable instructions using svm," *IEEE Access*, vol. 7, pp. 40 210–40 219, 2019.

[78] ——, "Predicting the silent data corruption vulnerability of instructions in programs," in *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, 2019, pp. 862–869.

[79] S. Li, S. Di, K. Zhao, X. Liang, Z. Chen, and F. Cappello, "Resilient error-bounded lossy compressor for data transfer," in *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[80] D. Agiakatsikas, G. Papadimitriou, V. Karakostas, D. Gizopoulos, M. Psarakis, C. Bélanger-Champagne, and E. Blackmore, "Impact of voltage scaling on soft errors susceptibility of multicore server cpus," in *MICRO-56: 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-56. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3613424.3614304

[81] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender, "Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 329–335, 2005.

[82] L. M. Luza, D. Söderström, H. Puchner, R. G. Alía, M. Letiche, A. Bosio, and L. Dilillo, "Effects of thermal neutron irradiation on a self-refresh dram," in *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2020, pp. 1–6.

[83] D. F. Bacon, "Detection and prevention of silent data corruption in an exabyte-scale database system," in *The 18th IEEE Workshop on Silicon Errors in Logic – System Effects*, 2022.

[84] Q. Guan, N. DeBardeleben, S. Blanchard, and S. Fu, "Empirical studies of the soft error susceptibility ofsorting algorithms to statistical fault injection," in *Proceedings of the 5th Workshop on Fault Tolerance for HPC at EXtreme Scale*, ser. FTXS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 35–40. [Online]. Available: https://doi.org/10.1145/2751504.2751512

[85] R. L. Rech and P. Rech, "Reliability of google's tensor processing units for embedded applications," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 376–381.

[86] G. Papadimitriou, A. Chatzidimitriou, D. Gizopoulos, and R. Morad, "An agile post-silicon validation methodology for the address translation mechanisms of modern microprocessors," *IEEE Transactions on Device and Materials Reliability*, vol. 17, no. 1, pp. 3–11, 2017.

[87] I. Wagner and V. Bertacco, "Reversi: Post-silicon validation system for modern microprocessors," in *2008 IEEE International Conference on Computer Design*, 2008, pp. 307–314.

[88] N. Foutris, D. Gizopoulos, M. Psarakis, X. Vera, and A. Gonzalez, "Accelerating microprocessor silicon validation by exposing isa diversity," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 386–397.

[89] D. Cimmino and S. Ferrero, "High-voltage temperature humidity bias test (hv-thb): Overview of current test methodologies and reliability performances," *Electronics*, vol. 9, no. 11, 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/11/1884

[90] C. Zorn and N. Kaminski, "Temperature humidity bias (thb) testing on igbt modules at high bias levels," in *CIPS 2014; 8th International Conference on Integrated Power Electronics Systems*, 2014, pp. 1–7.

[91] D. Sartzetakis, G. Papadimitriou, and D. Gizopoulos, "gpufi-4: A microarchitecture-level framework for assessing the cross-layer resilience of nvidia gpus," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022, pp. 35–45.

[92] Z. Hadjilambrou, S. Das, P. N. Whatmough, D. Bull, and Y. Sazeides, "Gest: An automatic framework for generating cpu stress-tests," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 1–10.

[93] Z. Hadjilambrou, S. Das, M. A. Antoniades, and Y. Sazeides, "Leveraging cpu electromagnetic emanations for voltage noise characterization," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 573–585.

[94] K. Ganesan, J. Jo, W. L. Bircher, D. Kaseridis, Z. Yu, and L. K. John, "System-level max power (sympo) - a systematic approach for escalating system-level power consumption using synthetic benchmarks," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 19–28.

[95] K. Ganesan and L. K. John, "Maximum multicore power (mampo) — an automatic multithreaded synthetic power virus generation framework for multicore systems," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[96] Y. Kim and L. K. John, "Automated di/dt stressmark generation for microprocessor power delivery networks," in *IEEE/ACM International Symposium on Low Power Electronics and Design*, 2011, pp. 253–258.

[97] Y. Kim, L. K. John, S. Pant, S. Manne, M. Schulte, W. L. Bircher, and M. S. S. Govindan, "Audit: Stress testing the automatic way," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 212–223.

[98] L. Mukhanov, D. S. Nikolopoulos, and G. Karakonstantis, "Dstress: Automatic synthesis of dram reliability stress viruses using genetic algorithms," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 298–312.

[99] K. Swaminathan, R. Bertran, H. Jacobson, P. Kudva, and P. Bose, "Generation of stressmarks for early stage soft-error modeling," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks – Supplemental Volume (DSN-S)*, 2019, pp. 42–48.