Rapid Fault Injection Simulation by Hash-based Differential Fault Effect Equivalence Checks

Johannes Geier*, Leonidas Kontopoulos*, Daniel Mueller-Gritschneder†, Ulf Schlichtmann*
*Technical University of Munich, Germany, {johannes.geier, leonidas.kontopoulos, ulf.schlichtmann}@tum.de
†TU Wien, Austria, daniel.mueller-gritschneder@tuwien.ac.at

***Abstract*—Assessing a computational system's resilience to hardware faults is essential for safety and security-related systems. Fault Injection (FI) simulation is a valuable tool that can increase confidence in computational systems and guide hardware and software design decisions in the early stages of development. However, simulating hardware at low levels of abstraction, such as Register Transfer Level (RTL), is costly, and minimizing the effort required for large-scale FI campaigns is a significant objective. This work introduces Hash-based Differential Fault Effect Equivalence Checks to automatically terminate experiments early based on predicting their outcome. We achieve this by matching observed fault effects to ones already encountered in previous experiments. We generate these hashes from differentials computed by repurposing existing fast boot checkpoints from a state-of-the-art acceleration method. By integrating these approaches in an automated manner, we can accelerate a large-scale FI simulation of a CPU at RTL. We reduce the average simulation time by a factor of up to 25 compared to a factor of around 2 to 5 for state-of-the-art techniques. While maintaining 100 % accuracy, we can recover the faulty state through the stored differentials.**

***Index Terms*—Fault injection, Simulation, Checkpoints, Register-transfer-level, Fault effect equivalence**

I. Introduction

Faults or defects can occur in different variations in computational systems. In integrated circuits, we can distinguish between hard (permanent) and soft (temporary) faults. Permanent faults refer to (mainly) irreversible defects in the hardware, e.g., a shorted wire or defect transistor. Soft errors [1], on the other hand, often occur due to transient faults that do not destroy hardware and are, therefore, reversible. They result from temporary upsets, seemingly at random due to radiation-induced Single Event Upsets (SETs) [1], [2], or deliberately by an adversary via electromagnetic pulses [3], [4] or lasers [5], [6]. In the context of road vehicle safety, ISO 26262 [7] norms software-level (part 6) and hardware-level (part 11) Fault Injections (FIs) as methods to increase confidence in a system's resilience to hard and soft errors. In contrast to post-silicon FI, where experiments are conducted by exposing the device to radiation, FI simulation allows evaluation at the early stages of the development, i.e., pre-silicon. Works such as [8]–[10] have shown that simulations at lower levels of abstraction, e.g., micro-architecture or Register Transfer Level (RTL), outperform higher levels of abstraction in terms of accuracy. However, simulating hardware at low levels of abstraction, such as RTL, is costly: Firstly, the fault space spanning all possible fault configurations (space and time) grows larger the lower the level of abstraction. Secondly, the simulation performance at lower levels of abstraction is much lower, e.g., in the range of thousands of instructions per second vs. millions of instructions per second for Instruction Set Simulator (ISS) [11]. Speed-up techniques that address this issue can be put in two categories: *Pre-FI* and *post-FI*. Among others *pre-FI* methods include: Statistical Fault Injection (SFI) [12], [13] target to reduce the required number of simulations to sample the fault space. Fault Equivalence Pruning (FEP) [14]–[16] collapse multiple injections to a single equivalent one but was so far only applicable to errors in memories or register files, but not micro-architectural states. Checkpoint Restore Boots (CRBs) [17]–[19] save the effort to boot the system for each FI simulation repeatedly. *Post-FI* techniques try to minimize the effort to observe a faulted system's behavior: Mixed-level Simulations (MLSs), such as [11] and [20], switch from a lower to a higher level of abstraction to speed-up the simulation when possible. However, such mixed-level simulations require considerable knowledge of the underlying Device Under Test (DUT) and, more importantly, a large effort to set up a simulator that allows switching between abstraction levels. This is why most of the works target Central Processing Units (CPUs) where an ISS enables switching between Instruction Set Architecture (ISA) and RTL. In [17], Dynamic Fault Collapsing (DFC) was proposed as another technique that checks whether a single bit during a previous FI simulation was corrupted at a certain point in time, assuming any FI targeting this bit at this time will lead to the same error effect and, hence, can be omitted.

This research was funded by the German Federal Ministry of Education and Research within the project MANNHEIM-FlexKI, funding no. 01IS22086L.

In this work, we propose a new method for further accelerating FI simulation campaigns without compromising accuracy. We build on the idea of DFC of [17]. As a major improvement, the proposed approach not only omits simulation runs that corrupt a bit, as seen in previous simulations, but instead checks in narrow checkpoint intervals whether the possible multi-bit fault effect pattern was already observed in any previous FI run. If the same error pattern is found, the simulation can be terminated early as the outcome is known from the previous FI run. A straightforward implementation of this causes large overheads to store and compare the immense number of checkpointed fault effect patterns as many FI campaigns run millions of FIs, diminishing the speed-up gained by the proposed method. To counter this effect, we additionally propose an efficient storage technique that only stores the difference to the golden reference run and a hashing technique for highly efficient comparisons. In summary, this work has the following contributions:

(1) **Dynamic Fault Effect Equivalence Checks**: A new method that terminates FI runs early when a previous run already observed the fault effect.
(2) **Differential Checkpointing (diffs)** for efficient storage of fault effect patterns. These checkpoints can also be exploited for post-simulation analysis of fault propagation.
(3) **A Hash-based method to compare fault effect checkpoints**: We use a database to host hashes of (2) to allow active experiments comparison of their own diffs against all previously seen ones enabling early termination via low-cost Fault Effect Equivalence Check (FEEC).

To demonstrate the approach, we extend an existing open-source, cycle-accurate RTL FI tool to support (1) and (2) without modification or knowledge of the DUT, i.e., treat the RTL as a black box. We implement (3) in a server-client framework. Clients simulate a RISC-V-based System on Chip (SoC) with the CPU being the FI target. For benchmarking, we also implemented the state-of-the-art techniques CRB, MLS, and masking checks. We reduce the average simulation time for FI campaigns by a factor of up to 25 against pure RTL compared to a factor of around 2 to 5 for state-of-the-art speed-up techniques CRB and MLS.

II. Related Work

Reducing the number of required simulations: In [12], Leuveugle et al. adapted established statistical methods to FI simulations of a hardware accelerator. One major contribution is their finding that for randomly sampled campaigns, not the fault space (population F) defines the number of required experiments (chosen sample N), but the chosen error margin and confidence level with which the campaign is set. [13] extends this by increasing the sample size until a target error margin is reached. Unfortunately, SFI is only relevant for FI campaigns whose goal is to be statistical in nature but can lack the observability of rare behaviors or strong localities of fault patterns. FEP [14]–[16] is another notable technique that aims to reduce the number of simulations needed to be executed. The underlying method is based on finding equivalent fault configurations before conducting the experiments. For example, if a variable is written (DEF) and read (USE) at a later time, all same-value FIs between these two points are equivalent. Only one experiment from the equivalence set would be simulated, and the others would be pruned from the simulation set, assuming their outcome would be equivalent. Unfortunately, equivalence classes are harder to find and not as frequent on lower levels of abstraction, such as synchronous RTL. For example, micro-architectural registers of a pipelined CPU may be read and updated in the same cycle, minimizing potential DEF/USE intervals. In [17], Berrojo et al. describe DFC, a variant of FEP. During the simulation of an experiment a, they compute a diff against the golden reference. If the diff indicates exactly one-bit deviation in bit b at time t, then the outcome of future experiments injecting a bit-flip in b at time t equals the already conducted experiment. DFC allows FEP by building up a database of fault effect records but does perform FEEC of an active experiment against this database.

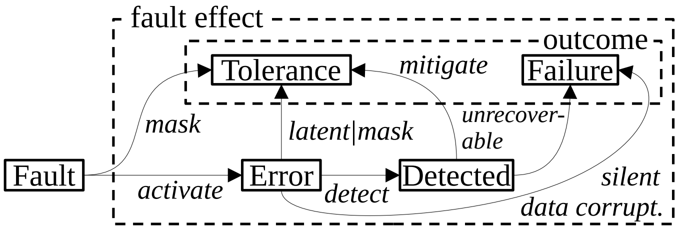

Fig. 1: Fault and fault effect transitions and classes

Accelerating remaining experiments: Both SFI and FEP are powerful techniques. However, there will be a number of experiments left that have to be conducted. A simulation tool can try to minimize the *pre-FI* simulation cost (*warmup*) by fast-forwarding the simulation state as close as possible to the FI point. In [11], a MLS is introduced that allows switching between ISS and RTL models of a CPU, where the ISS conducts the *warmup* phase. For deterministic FI campaigns, a low-cost *warmup* can be achieved via CRB, notably applied to RTL FI in works such as [17]–[19]. The idea is to record the simulation state at certain checkpoints of a fault-free reference simulation, which are then used in the subsequent FI simulations. A checkpoint that is as close as possible to the FI point is selected for each fault configuration. For a uniformly sampled SFI, CRB can cut the simulation cost in half [19]. While CRB effectively removes the *warmup* segments from a FI campaign, after the fault is injected, its impact needs to be observed. [11] and [20] implement two *post-FI* acceleration techniques in their MLS: Firstly, masking checks and secondly switch-back to ISS from RTL. Whereas [20] compares the faulty micro-architecture against a golden reference state, [11] simulates a modified RTL of a CPU, which contains shadows of its internal registers. This allows tracking fault propagation. If the primary and shadow registers match by value, the fault has vanished, and the experiment can terminate early. If the fault is only present in the ISA registers or has left the DUT, e.g., corrupted memory, the simulation can switch up to the ISS. A major drawback of this approach is its complexity and limit to abstractable DUTs, e.g., CPUs. The MLS has to be extended w.r.t. knowledge of abstractable states, i.e., which RTL registers map to which ISA registers. Furthermore, to allow observing the fault propagation, the simulator must evaluate both primary and shadow states, doubling the simulation cost during *cooldown*. Furthermore, the simulation must stay at RTL to avoid losing information due to abstraction if the micro-architecture is corrupt.

III. Hash-based Differential Fault Effect Equivalence Checks

In this work, we slightly adjust the FI terminology introduced by [21] to categorize faults and their effects. Figure 1 depicts this in a simplified diagram. We distinguish between an injected fault and its effects, which means a fault might *mask* or *activate* to an error, after which it may either lead to the outcomes

TABLE I: Symbols used in this work

T	set of time steps of reference simulation
B	set of injectable states (e.g., bits)
I	set of injectable time steps $I \subset T$
F	fault space, for single-bit faults: $F = B \times I$
$(\alpha,\beta)^n$	experiment $n \in N$, with $(\alpha,\beta)^n \in F$
N	sample of unique experiment configs $N \subset F \wedge (\alpha,\beta)^i \neq (\alpha,\beta)^j \, \forall_{i,j} \in N$
o^n	outcome of experiment n
μ_t^n	state of an experiment n at time $t \in T$
T^c	set of time steps where a checkpoint exists
$\mu_{t^c}^R$	state of the reference simulation at time $t^c \in T^c$
$\oplus$	bit-wise exclusive OR
⬠	checkpoints $\mu_{t^c}^R$ *with* $t^c \in T^c$

failure or *tolerance*. If detection mechanisms exist in software or hardware, an error might be mitigated (e.g., corrected) or deemed unrecoverable (e.g., reset needed). Undetected, a failure might also occur if the FI results in *erroneous* behavior (e.g., a Silent Data Corruption (SDC)). Lastly, an error could not affect the system behavior or output by being masked or latent (*benign*). We will use the symbols presented in Tab. I for mathematical equations, algorithms, and results.

A. Checkpoint Differentials (CDIF)

Given that the simulation states for a fault-free simulation exists (μ^R), we can compute the differential (diff) for an experiment a at time t with:

$$\delta_t^a = \mu_t^a \oplus \mu_t^R \Rightarrow \mu_t^a = \mu_t^R \oplus \delta_t^a \tag{1}$$

To compute the diff, we need a reference state μ_t^R during simulation. Works such as [11] simulated a fault-free pipeline alongside the faulty one. However, since we already have fault-free states from the golden reference run that generated the fast-forward CRB checkpoints, we can reuse them to compute the diff at the points in time where a checkpoint exists: Checkpoint Differential (CDIF). Fig. 2 (right) depicts this for a set of checkpoints at times $t^c \in T^c$. Experiments a, b, l, and m choose checkpoints as close as possible to the FI. Next, they advance the simulation time until it reaches another checkpoint t^c where they can compute a diff against the checkpoint state $\mu_{t^c}^R$ per (1).

1) Masking Checks: The first trivial check that can be performed is Checkpoint Masking (CMSK), e.g., the fault in experiment m in Fig. 2 has vanished which is indicated by the diff being 0 at t_2^c. The experiment can assume the reference simulation's outcome o^R via (2):

$$\delta_t = 0 \Leftrightarrow \mu_t \oplus \mu_t^R = 0 \Leftrightarrow \mu_t = \mu_t^R \Rightarrow o = o^R \tag{2}$$

2) Database of Diffs (DoD) Δ: Since $\oplus$ is symmetric, we can also reverse the faulty state μ_t^a from the golden state μ_t^R and δ_t^a (right-hand side (1)). This gives us a powerful tool to analyze fault effect propagation after the campaign. For most experiments, the Hamming Weight lets us store individual diffs sparsely, e.g., Coordinate list format (COO). We can describe this DoD Δ as a set of tuples from experiment identifiers (n), timestamps (t), and diff values (δ):

$$\Delta = \{(n, t^c, \delta_{t^c}^n)_1, (n, t^c, \delta_{t^c}^n)_2, ...\} \text{ with } n_i \in N \wedge t_i^c \in T^c$$

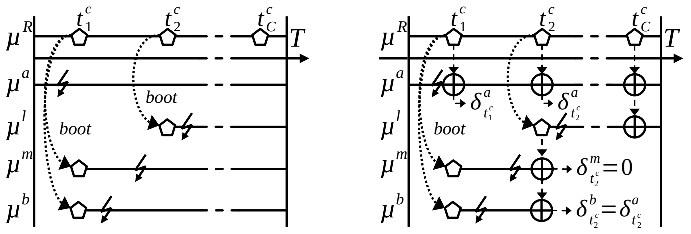

Fig. 2: Checkpoint Restore Boot (CRB) (left) and CDIF (right)

B. Fault Effect Equivalence Checks (FEEC)

In contrast to FEP and Fault Outcome Prediction (FOP), FEEC is not limited by analysis of the executed workload or knowledge of the DUT. If the states of two experiments are equal at any point in time, their outcomes will be equivalent:

$$\mu_x^a = \mu_y^b \Rightarrow o^a = o^b \quad x, y \in T \tag{3}$$

Based on (3), we could build up a database that hosts snapshots of simulation states of all conducted experiments at all simulated times. Although this would allow FEEC for future experiments, the storage and search logic costs would be immense. A possible solution would be to compress the information while retaining the capability to perform FEEC.

1) Reusing Differentials: We can also perform FEEC for an experiment b by finding a matching diff in the DoD Δ.The idea is that since (1) lets us reverse from a diff back to the full state, we can perform (3) by comparing an active experiment's diff against entries in the database. However, since we are computing the diffs at CRB checkpoints to allow reusing the checkpoints, using (1) to reverse from a diff back to a full state needed in (3) only works for equal points in time:

$$\begin{gathered}\delta_x^a = \delta_y^b \Leftrightarrow \mu_x^R \oplus \mu_x^a = \mu_y^R \oplus \mu_y^b, \text{ with } x \neq y \wedge \mu_x^R \neq \mu_y^R \\ \not\Leftrightarrow \mu_x^a = \mu_y^b \not\Rightarrow o^a = o^b\end{gathered} \tag{4}$$

As an example, the accumulate register of a Multiply-Accumulate (MAC) unit is activated (read) at time z by a MAC instruction. Two diffs, δ_x^a and δ_y^b, with $x < z < y$, might be equivalent, i.e., indicate the same faulty state in the register, although experiment a will see the fault being activated at time z and b not because it was injected after z. By selecting an equivalent point in time $t \in T$, the left-hand and right-hand side μ_t^R of the equivalence check cancel out and leave the states μ_t^a and μ_t^b from which FEEC can be derived:

$$\begin{gathered}\delta_t^a = \delta_t^b \Leftrightarrow \mu_t^R \oplus \mu_t^a = \mu_t^b \oplus \mu_t^R, \text{ with } \mu_t^R \oplus \mu_t^R = 0 \\ \Leftrightarrow \mu_t^a = \mu_t^b \Rightarrow o^a = o^b\end{gathered} \tag{5}$$

Limitation: Searching for matches in the diff database Δ is costly due to element-wise comparisons. Compressing the fault effect into a single value is more efficient, e.g., a hash value.

2) Hashing Fault Effect: Let h be the value of a collision-free [22] hash function H on. Next, we can compute a signature $h_x^a = H(\mu_x^a)$ of a fault effect μ_x^a. If we can not find the signature h_x^a in a database containing signatures of already conducted experiments, we add h_x^a to the database and keep simulating until the experiment finishes. In contrast, if we can find the signature in the campaign database, added by a previous experiment, e.g., h_y^b, we can infer FEEC from (3) via

$$h_x^a = h_y^b \Rightarrow \mu_x^a = \mu_y^b \Rightarrow o^a = o^b, \text{ with } x, y \in T. \tag{6}$$

Limitations: Hash functions are one-way, i.e., non-reversible [23] which means we lose all information about an experiment's state by hashing it. If we want to analyze the experiment, e.g., fault propagation, we must re-execute it. Furthermore, hash functions are computationally expensive, so constantly computing them can result in considerable overhead.

3) Hash-based Differential FEEC: From Equations (4) and (5), we know that differential FEEC is only safe at equivalent points in time such that simply hashing diffs $H(\delta_t)$ and storing them without the timing information could lead to incorrect matches in the database lookup. Let $\hat{h}$ be the value of a collision-free hash on a diff δ_t^a combined with a seed s:

$$\hat{h}_s^a = \hat{H}(s, \delta_t^a)$$

By choosing s to be the diffs timestamp, we can enforce that the hashes of equal-value diffs from different points in time do not match, even if the diffs would match:

$$\hat{h}_x^a \neq \hat{h}_y^b, \text{ with } \delta_x^a = \delta_y^b \ \wedge \ x \neq y$$

Assuming the hashes of two matching diffs $\delta_t^a = \delta_t^b$ were built with $\hat{H}$ and the same time seed t, their values would match and allow safe FEEC:

$$\hat{h}_t^a = \hat{h}_t^b \Rightarrow \mu_t^a = \mu_t^b \Rightarrow o^a = o^b, \text{ with } \delta_t^a = \delta_t^b$$

4) Database of (Diff) Hashes Γ*:* We can describe the Database of (Diff) Hashes (DoH) Γ as a set of tuples from hash values $\hat{h}$ and experiment identifiers (n):

$$\Gamma = \{(n, \hat{h}_{t^c}^n)_1, (n, \hat{h}_{t^c}^n)_2, ...\}, \text{ with } n_i \in N \wedge t_i^c \in T^c$$

IV. Implementation

We implemented our method, Hash-based Differential Fault Effect Equivalence (CDIF+FEEC), in an RTL FI tool and a server-client simulation framework.

A. Simulation Flow

Our CDIF+FEEC simulation flow is described in Alg. 1. It expects unique fault experiments N to be run on the simulator S. In the preparation phase (L. 2-4), the algorithm resets its outputs O, Δ, and Γ, issues a fault-free reference simulation to generate checkpoints, and sorts the faults by their injection times (earliest first). The elaboration phase starts in L. 5, where a run is started for each fault in the campaign. lines 6-9 describe the CRB as depicted in Fig. 2. Next, the simulator performs the *warmup* by advancing to FI time β (lines 10-11) and applies the fault (L. 12, shown here as an addition of the fault α). In lines 13-29, the experiment is in *cooldown*: As long as no FEEC succeeded or the simulator reports End-Of-Simulation (EOS), the next closest checkpoint is selected and advanced to (L. 14-16). Next, the diff is computed. In case we encounter EOS, we select the corresponding final simulator state as the 2nd operand (L. 20); else, we use the next checkpoint state once the simulation advanced to it (L.18).

Algorithm 1: CDIF+FEEC simulation pseudo-code

```
1  RunCampaign ($N, \mathcal{S}$)
      In: Fault configurations $N$, Simulator $\mathcal{S}$
      Result: Outcomes $O$, diffs $\Delta$ and hashes $\Gamma$
2     $O \leftarrow \Delta \leftarrow \Gamma \leftarrow \{\}$                ;empty sets
3     $\mu^R, T^c \leftarrow \mathrm{ref}(\mathcal{S})$          ;generate checkpoints
4     $N \leftarrow \mathrm{sort}(N, less)$          ;sort by earliest FI
5     foreach $fault\ (\alpha,\beta)^n \in N$ do
6        EOS ← false  ;init end-of-simulation flag
7        $t^c_{\mathrm{boot}} \leftarrow \max(\{t^c \mid t^c \in T^c \wedge t^c < \beta^n\})$
8        $t \leftarrow t^c_{\mathrm{boot}}$              ;set simulation state to
9        $\mu \leftarrow \mu^R_{t^c_{\mathrm{boot}}}$     ;closest to FI checkpoint state
10       while $t < \beta^n \wedge \neg EOS$ do
11          EOS ← advance($\mathcal{S}$)           ;advance to FI
12       $\mu \leftarrow \mu + \alpha^n$                   ;inject fault
13       while $\neg EOS$ do
14          $t^c_{\mathrm{next}} \leftarrow \min(\{t^c \mid t^c \in T^c \wedge t^c > t\})$  ;select
             next closest checkpoint
15          while $t < t^c_{next} \wedge \neg EOS$ do
16             EOS ← advance($\mathcal{S}$)
17          if $\neg EOS$ then
18             $\delta \leftarrow \mu \oplus \mu^R_{t^c_{\mathrm{next}}}$   ;diff vs sim checkpoint
19          else
20             $\delta \leftarrow \mu \oplus \mu^R_{t^c_{\mathrm{EOS}}}$   ;diff vs EOS checkpoint
21          $\hat{h} \leftarrow \hat{H}(t, \delta)$            ;compute diff hash
22          if $\exists (r, \hat{h}^r)_i \in \Gamma,\ \hat{h} = \hat{h}^r_i$ then
23             $o^n \leftarrow o^r$          ;reference found match
24             $EOS$ ← true           ;stop simulation
25          else
26             $\Gamma \leftarrow \Gamma \cup \{(n, \hat{h})\}$        ;add hash to DoD
27             $\Delta \leftarrow \Delta \cup \{(n, t, \delta)\}$     ;add diff to DoH
28             if $EOS$ then
29                $o^n \leftarrow \mathrm{eval}(\mu, \delta)$  ;determine outcome
30       $O \leftarrow O \cup \{o^n\}$       ;done. add outcome to DoO
```

After computing the hash (L. 21), the DoH Γ is searched for a match in L. 22. If found (L. 23-24), we can set the outcome of our experiment to the match's and break the *cooldown* loop. If we do not find a matching hash, we add our hash to the DoH and the diff to the DoD (L. 26-27). Unless we encountered EOS (L. 28), the *cooldown* continues with the next closest checkpoint. Otherwise, the outcome is evaluated from the final state and diff (L. 29). For all experiments, we add their respective outcome to the Database of Outcomes (DoO) O. This can either be a reference to another experiment (FEEC) or an actual evaluation of the final simulation state per Fig. 1.

B. RTL Fault Injection Simulator

The simulation basis is the open-source software (OSS) *Verilator*, an *Verilog* RTL to *C*++ compiler [24]. Transient FI capability is provided by another OSS vRTLmod [25] that transforms *Verilator's* output, referred to as verilated RTL (vRTL), to allow injection in sequential states, i.e., *flip-flop*-level FI. We implement CRB with Verilator's built-

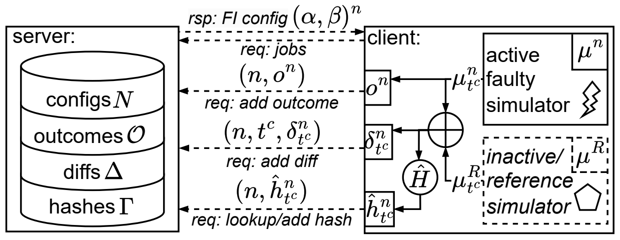

Fig. 3: Simplified FI simulation server/client architecture

in *save/restore* feature. This enables serialization (checkpoint store) and de-serialization (restore) of a current simulation state to a file. To allow generating diffs as described in Sec. III-A, we extend vRTLmod tool's code generation to automatically generate functions that compute a diff in COO format from two instances of the now fault injectable vRTL module. This extension works on the RTL in black-box mode since vRTLmod analyzes its input vRTL for C++ variables representing sequential states. Although we instantiate multiple vRTL modules for the simulator, only one is faulted and stimulated with inputs. A second dummy instance is used to restore checkpoints (μ^R) for the diffs in the *cooldown* phase. An FI campaign is simulated by a server-client system sketched in Fig. 3 inspired by [26], where the authors proposed an open-source ISS-based fault attack framework. Similarily, we use HDF5 [27] to manage our databases.Alg. 1 is mostly implemented client-side, except the FEEC check. Clients initiate it by a *lookup/add hash* request to the server, which updates the DoH Γ and responds accordingly. Tab. II lists the simulator features we implemented. The referenced works are not meant to be interpreted as a one-to-one equivalence. We implement MLS by activating the second simulator, depicted inactive in Fig. 3, and coupling it with the faulty one acting as a live tracking core and allowing us to switch back to ISS once a diff indicates a pure ISA fault.

V. Evaluation

We evaluate the performance of our CDIF+FEEC method against other techniques in a common experimental setup.

A. Experimental Setup

We host the system depicted in Fig. 3 with 63 worker threads on a server with two Intel® Xeon® Silver 4314 processors at 2.4 GHz. As DUT, we choose the open-source 32-bit RISC-V CPU cv32e40p [28], [29] in a minimalistic SoC. The RTL contains 9,380 sequential logic bits (B), including architectural and micro-architectural registers. The DUT executes a subset of Embench™ (Tab. III). Usually, the benchmarks (BMs) run multiple rounds to estimate a CPU's performance, which would result in additional masking and most BMs having a similar runtime. Therefore, we reduce each BM's round count to one. The number of inject-cycles ($|I_{BM}|$) is smaller than the respective BM's execute-cycles ($|T_{BM}|$) because we do not inject in boot and return code. For single-bit FI, the fault space per BM can be expressed as a Cartesian product of inject-cycles and bits. Checkpoints are placed every 10,000 clock cycles (ccs), e.g., `aha-mont` only has one and `wikisort` has 169.

TABLE II: Considered simulator features and related works

simulator	description	related work
RTL	no features	*baseline*
CRB	Checkpoint Restore Boot	[17]–[19]
MLS	CRB, Mixed-level Simulation	[11], [20]
CMSK	CRB, Checkpoint Masking	*ours*
CDIF+FEEC	CRB, Checkpoint Differential, and Fault Effect Equivalence Check	*ours*

TABLE III: Benchmark (BM) execution times $|T_{BM}|$ and single-bit FI fault space F_{BM} from injection bits B and cycles I_{BM}

BM	$\|T_{BM}\|$ [cc]	$\|I_{BM}\|$ [cc]	$\|F_{\mathrm{BM}} = B \times I_{BM}\|$
`aha-mont`	18,616	12,646	$118 \cdot 10^6$
`huffbench`	292,157	285,006	$2.67 \cdot 10^9$
`picojpeg`	757,363	750,979	$7.04 \cdot 10^9$
`wikisort`	1,692,368	1,675,850	$15.7 \cdot 10^9$

B. Experimental Results

All simulator configurations conduct the same set of uniformly distributed and unique single-bit, single-cycle fault experiments per BM. The goal of our experiment is not to evaluate the safety/security of the DUT but to support our claims regarding speed-up without accuracy loss. For this, we assume the pure RTL configurations (RTL and CRB) as the *ground truth*. Tab. IV reports the fault outcomes per BM and simulator feature of equal campaigns N with 111,000 unique experiments. We consider the *erroneous* outcome classes SDC and Detectable Unrecoverable Error (DUE). We classify an experiment as DUE when the CPU enters an exception or hangs (after 2× the expected cc were executed) and as an SDC when the BM output is erroneous. The *Benign* are all experiments that did not affect the BM's output. However, the state of the DUT at EOS is either clean (masked), deviates in execution time, or has corrupted bits in memory or registers (latent). Only MLS deviates with DUEs and SDCs from RTL. The reasons are the ISS's inaccurate timing and a stricter bus protocol (I/O exceptions). For example, hang experiments at RTL became an SDC at ISS due to the ISS executing more instructions than the RTL in the same amount of time. The pure RTL simulators, including CDIF+FEEC, show a 100% match in outcomes. However, masked experiments differ due to low checkpoint count (e.g., `aha-mont64`), no masking checks at ISS level, and CDIF+FEEC performing checks at EOS, which results in no unclassified experiments remaining.

1) Simulation Performance: From Tab. IV, we can see that for the non-FEEC simulators (CMSK and MLS), about 50% of experiments have to finish simulating. This rate can be reduced by deploying FEEC, which automatically filters out many latent fault effects. Next, we evaluate the performance w.r.t. to size of campaigns. We do this by issuing the same uniformly sampled configurations N. As Tab. V reports, the average simulation times per experiment ($\overline{t_s}$) of non-FEEC features are not affected by the sample size, e.g., a single `wikisort` experiment at pure RTL takes 63.9 s to complete for $|N| = 100,000$. Except for `aha-mont64`, we observed a speed-up of roughly 2× when

TABLE IV: Classification in *erroneous* (SDC, DUE) and *benign* outcomes for equal experiments but different simulators

$\|N\|$:111,000	BM	*erroneous*		*benign*			
simulator		SDC	DUE	mask	time	latent	unclass
RTL	`aha-mont`	5461	2783	0	0	0	102756
CRB	`aha-mont`	5461	2783	0	0	0	102756
CMSK	`aha-mont`	5461	2783	27738	0	0	75018
MLS	`aha-mont`	5461	2783	47591	0	0	55165
CDIF+FEEC	`aha-mont`	5461	2783	56485	188	46083	0
RTL	`huffbench`	2224	4807	0	0	0	103969
CRB	`huffbench`	2224	4807	0	0	0	103969
CMSK	`huffbench`	2224	4807	48620	0	0	55349
MLS	`huffbench`	2229	4795	47445	0	0	56531
CDIF+FEEC	`huffbench`	2224	4807	53984	310	49675	0
RTL	`picojpeg`	676	2559	0	0	0	107765
CRB	`picojpeg`	676	2559	0	0	0	107765
CMSK	`picojpeg`	676	2559	56661	0	0	51104
MLS	`picojpeg`	681	2553	48560	0	0	59206
CDIF+FEEC	`picojpeg`	676	2559	59357	1719	46689	0
RTL	`wikisort`	330	2864	0	0	0	107806
CRB	`wikisort`	330	2864	0	0	0	107806
CMSK	`wikisort`	330	2864	58141	0	0	49665
MLS	`wikisort`	328	2996	49159	0	0	58517
CDIF+FEEC	`wikisort`	330	2864	59865	2072	45869	0

TABLE V: Performance in average experiment simulation time ($\overline{t_s}$) in seconds and speed-up factor vs. baseline RTL (×).

	BM	$\|N\|$: 10,000		100,000		1,000,000	
simulator		$\overline{t_s}$[s]	×	$\overline{t_s}$[s]	×	$\overline{t_s}$[s]	×
RTL	`aha-mont`	0.74	-	0.73	-		
CRB	`aha-mont`	0.55	1.3	0.55	1.3		
CMSK	`aha-mont`	0.47	1.6	0.49	1.5		
MLS	`aha-mont`	0.60	1.2	0.60	1.2	0.60	~1.2
CDIF+FEEC	`aha-mont`	0.47	1.6	0.39	1.9	0.40	~1.8
RTL	`huffbench`	11.7	-	11.6	-		
CRB	`huffbench`	6.1	1.9	6.1	1.9		
CMSK	`huffbench`	3.7	3.2	3.8	3.0		
MLS	`huffbench`	3.0	3.9	3.0	3.9		
CDIF+FEEC	`huffbench`	3.1	3.8	1.5	7.6	1.0	~11
RTL	`picojpeg`	29.9	-	27.4	-		
CRB	`picojpeg`	15.0	2.0	15.0	1.8		
CMSK	`picojpeg`	7.7	3.9	8.1	3.4		
MLS	`picojpeg`	6.6	4.5	6.8	4.0		
CDIF+FEEC	`picojpeg`	5.9	5.1	2.4	11	1.3	~21
RTL	`wikisort`	65.6	-	63.9	-		
CRB	`wikisort`	33.2	2.0	33.4	1.9		
CMSK	`wikisort`	16.8	3.9	17.3	3.7		
MLS	`wikisort`	14.2	4.6	14.2	4.5		
CDIF+FEEC	`wikisort`	12.8	5.1	5.0	12.8	2.53	~25

enabling CRB, which is as expected and consistent with related works, such as [18], [19]. This can be explained by the uniform distribution of the sample N where, on average, experiments start at half BM runtime $0.5|T_{BM}|$, which means, on average, only the remaining half has to run. Masking checks (CMSK) can speed-up by another factor of about 2× to 4× vs. RTL. This can be explained by the fact that the speed-up seems proportional to the masking rate (50% → 2×, see Tab. IV). MLS allows a total speed-up of around 4.6×. Enabling FEEC has the most significant effect on the simulation performance, where CDIF+FEEC outperforms all other considered simulators. For example, the largest sample on the longest BM (`wikisort`, $|N| = 1,000,000$) has a reduction of $\overline{t_s}$ to 2.53 s, which means an estimated speed-up of 25× vs. RTL. Fig. 4 plots the measurements for $\overline{t_s}$ from Tab. V over the BM's executed cc per experiment. CDIF+FEEC's performance is split for different sample sizes $|N|$. In contrast, Fig. 5 plots $\overline{t_s}$ per BM over a normalized sample size. As expected, for all non-FEEC features, $\overline{t_s}$ is constant, but there is a downward trend for increasing sample sizes for CDIF+FEEC: The probability of later experiments matching increases with the number of fault effects in the DoH, which is higher for larger samples.

VI. Conclusion

We present a fast and accurate speed-up technique to accelerate RTL FI simulations. Our method of Hash-based Differential Fault Effect Equivalence (CDIF+FEEC) can be used without intricate knowledge of the DUT (black box) or additional analysis. To demonstrate the approach, we implemented CDIF+FEEC in vRTLmod [25], [30], an open-source RTL FI tool and a server-client simulation framework simulating a RISC-V-based SoC. We measured a speed-up of up to 25 times against pure RTL simulation, significantly outperforming various existing methods, which we benchmarked at a factor of 2 to 5. While maintaining 100% accuracy, we can recover faulty states through the stored checkpoint differentials, which can be used for detailed analysis, such as fault propagation.

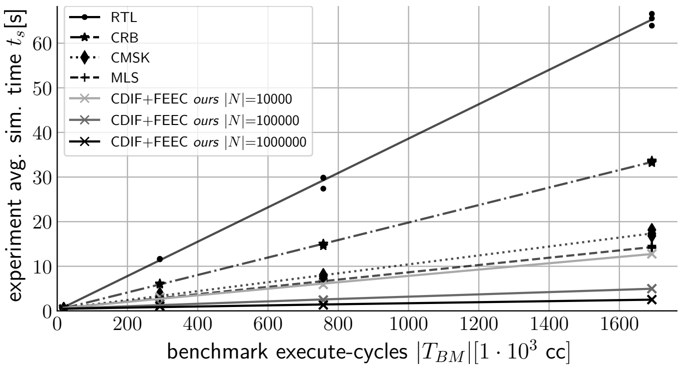

Fig. 4: Average simulation time per experiment $\overline{t_s}$ over benchmark execute-cycles $|T_{BM}|$ in ccs for all considered simulators

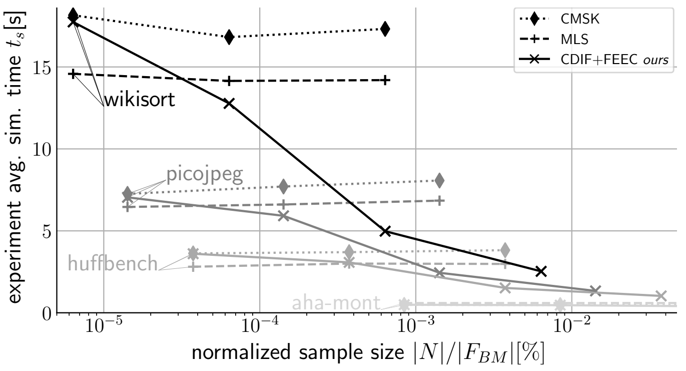

Fig. 5: Average simulation time per experiment $\overline{t_s}$ over chosen sample size $|N|$ normalized for benchmark fault space size $|F_{BM}|$

References

[1] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005.

[2] E. H. Ibe *et al.*, "Radiation-induced soft errors," in *VLSI Design and Test for Systems Dependability*, S. Asai, Ed. Tokyo: Springer Japan, 2019, pp. 57–127.

[3] A. Dehbaoui, J.-M. Dutertre, B. Robisson, P. Orsatelli, P. Maurine, and A. Tria, "Injection of transient faults using electromagnetic pulses Practical results on a cryptographic system," 2012, journal of Cryptology ePrint Archive: Report 2012/123.

[4] N. Moro *et al.*, "Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller," in *Proceedings - 10th Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2013*, 2013, pp. 77–88.

[5] J. G. van Woudenberg, M. F. Witteman, and F. Menarini, "Practical optical fault injection on secure microcontrollers," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2011, pp. 91–99.

[6] B. Selmke, J. Heyszl, and G. Sigl, "Attack on a dfa protected aes by simultaneous laser fault injections," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2016, pp. 36–46.

[7] International Organization for Standardization, "ISO 26262:2018 (all parts): Road vehicles - functional safety," 2018.

[8] H. Cho, S. Mirkhani, C. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: Association for Computing Machinery, 2013.

[9] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas, and Y. Makris, "Instruction-level impact analysis of low-level faults in a modern microprocessor controller," *IEEE Transactions on Computers*, vol. 60, no. 9, pp. 1260–1273, 2011.

[10] M. Maniatakos, C. Tirumurti, A. Jas, and Y. Makris, "AVF analysis acceleration via hierarchical fault pruning," in *2011 Sixteenth IEEE European Test Symposium*, 2011, pp. 87–92.

[11] D. Mueller-Gritschneder, U. Sharif, and U. Schlichtmann, "Performance and accuracy in soft-error resilience evaluation using the multi-level processor simulator ETISS-ML," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE Press, 2018, p. 1–8.

[12] 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.

[13] I. Tuzov, D. De Andrés, and J. C. Ruiz, "Accurate robustness assessment of hdl models through iterative statistical fault injection," *Proceedings - 2018 14th European Dependable Computing Conference, EDCC 2018*, pp. 1–8, 2018.

[14] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: Association for Computing Machinery, 2012, p. 123–134.

[15] J. Li and Q. Tan, "Smartinjector: Exploiting intelligent fault injection for sdc rate analysis," in *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, 2013, pp. 236–242.

[16] H. Schirmeier, C. Borchert, and O. Spinczyk, "Rapid fault-space exploration by evolutionary pruning," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8666 LNCS, pp. 17–32, 2014.

[17] L. Berrojo *et al.*, "New techniques for speeding-up fault-injection campaigns," in *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, 2002, pp. 847–852.

[18] J. Na and D. Lee, "Acceleration of simulated fault injection using a checkpoint forwarding technique," *ETRI Journal*, vol. 39, no. 4, pp. 605–613, 2017.

[19] B. Tabacaru, M. Chaari, W. Ecker, T. Kruse, and C. Novello, "Efficient checkpointing-based safety-verification flow using compiled-code simulation," in *2016 Euromicro Conference on Digital System Design (DSD)*, 2016, pp. 364–371.

[20] H. Cho, E. Cheng, T. Shepherd, C.-Y. Cher, and S. Mitra, "System-level effects of soft errors in uncore components," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 9, pp. 1497–1510, 2017.

[21] J. Arlat, A. Costes, Y. Crouzet, J. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 913–923, 1993.

[22] A. Russell, "Necessary and sufficient conditions for collision-free hashing," *Journal of Cryptology*, vol. 8, p. 87–99, 1995.

[23] Y. Zheng, T. Matsumoto, and H. Imai, "Structural properties of one-way hash functions," in *Advances in Cryptology-CRYPTO' 90*, A. J. Menezes and S. A. Vanstone, Eds. Springer Berlin Heidelberg, 1991, pp. 285–302.

[24] W. Snyder, P. Wasson, D. Galbi, and et al. Verilator. [Online]. Available: https://github.com/verilator/verilator

[25] J. Geier and D. Mueller-Gritschneder, "vrtlmod: An llvm based open-source tool to enable fault injection in verilator rtl simulations," in *Proceedings of the 20th ACM International Conference on Computing Frontiers*, ser. CF '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 387–388. [Online]. Available: https://doi.org/10.1145/3587135.3591435

[26] F. Hauschild, K. Garb, L. Auer, B. Selmke, and J. Obermaier, "ARCHIE: A QEMU-based framework for architecture-independent evaluation of faults," in *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, 2019, pp. 20–30.

[27] S. Koranne, "Hierarchical data format 5: HDF5," in *Handbook of Open Source Tools*. Boston, MA: Springer US, 2011, pp. 191–200.

[28] M. Gautschi *et al.*, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[29] OpenHW Group. (2024) CV32E40P - GitHub repository. [Online]. Available: https://github.com/openhwgroup/cv32e40p

[30] vRTLmod. (2024). [Online]. Available: https://github.com/tum-ei-eda/vrtlmod