DEFCON: Generating and Detecting Failure-prone Instruction Sequences via Stochastic Search

Ioannis Tsiokanos[*], Lev Mukhanov[*], Giorgis Georgakoudis[†], Dimitrios S. Nikolopoulos[‡], and Georgios Karakonstantis[*]

[*]*Institute of Electronics, Communications and Information Technology, Queen's University Belfast, UK*
Email: {*itsiokanos01, l.mukhanov, g.karakonstantis*}*@qub.ac.uk*
[†]*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, USA*
Email: *giorgis@llnl.gov*
[‡]*Department of Computer Science, Virginia Tech, USA*
Email: *dsn@vt.edu*

Abstract—**The increased variability and adopted low supply voltages render nanometer devices prone to timing failures, which threaten the functionality of digital circuits. Recent schemes focused on developing instruction-aware failure prediction models and adapting voltage/frequency to avoid errors while saving energy. However, such schemes may be inaccurate when applied to pipelined cores since they consider only the currently executed instruction and the preceding one, thereby neglecting the impact of all the concurrently executing instructions on failure occurrence. In this paper, we first demonstrate that the order and type of instructions in sequences with a length equal to the pipeline depth affect significantly the failure rate. To overcome the practically impossible evaluation of the impact of all possible sequences on failures, we present DEFCON, a fully automated framework that stochastically searches for the most failure-prone instruction sequences (ISQs). DEFCON generates such sequences by integrating a properly formulated genetic algorithm with accurate post-layout dynamic timing analysis, considering the data-dependent path sensitization and instruction execution history. The generated micro-architecture aware ISQs are then used by DEFCON to estimate the failure vulnerability of any application. To evaluate the efficacy of the proposed framework, we implement a pipelined floating-point unit and perform dynamic timing analysis based on input data that we extract from a variety of applications consisting of up-to 43.5M ISQs. Our results show that DEFCON reveals quickly ISQs that maximize the output quality loss and correctly detects 99.7% of the actual faulty ISQs in different applications under various levels of variation-induced delay increase. Finally, DEFCON enable us to identify failure-prone ISQs early at the design cycle, and save 26.8% of energy on average when combined with a clock stretching mechanism.**

I. Introduction

Shrinking transistor sizes and improving energy efficiency are forcing chip manufacturers to scale down supply voltages [1]. At the same time, nanometer nodes are susceptible to static and dynamic variations [1], [2] during manufacturing or operation, e.g., imperfections in the power grid or high power surges triggered by running workloads. Due to these reasons, circuits are increasingly prone to timing failures [2]. Such failures threaten system functionality and output correctness, and may be triggered randomly or even inflicted on purpose through malicious code that compromises security [3].

State-of-the-Art. Therefore, there is a growing interest in techniques that study the sources of timing failures [1], [4] to early identify them [5] and estimate their impact on applications [6]. These works revealed that subject to delay variations, the type and operands of executing instructions determine the occurrence of a timing failure. Finding the factors that affect such failures becomes more complicated within pipelined cores, where instruction execution history also seems to play a role [7].

Due to the inherent complexity of timing error modeling, recent techniques exploited machine learning methods to predict timing failures [8]–[10], and guide the development of clock adjustment techniques [11] or estimate the program error rate [12]. However, these methods overlook the impact of deeper instruction execution history (i.e. type and order of instructions within a pipeline at any instant) on timing failures. Thus, they may fail to accurately predict timing failures in deep pipelined designs where multiple instructions are executed simultaneously.

Lately, studies [13], [14] on pipelined cores proposed evolutionary search algorithms to develop stress codes that maximize voltage noise and predict worst-case voltage fluctuations in advance. However, such stochastic search methods are limited only to the context of voltage noise and cannot identify bit-level timing errors and quality degradation incurred by a potential error. Also, they are agnostic of the underlying micro-architecture and lack post-layout circuit information, requiring custom-made infrastructure for measuring complex phenomena (e.g. voltage droop). In fact, such search algorithms have never been considered jointly with accurate post-layout gatelevel simulation (PGS) for both generating and detecting instruction sequences (ISQs) that maximize failure-induced quality loss, while considering instruction execution history, instruction type and operand dependent path excitation.

Contributions. In this paper, we present DEFCON[1], a fully automated framework that generates micro-architecture aware,

[1]**DE**tecting **F**ailure-prone instru**C**ti**O**n seque**N**ces (DEFCON)

failure-prone ISQs and detects vulnerable code regions in applications. The main contributions are:

- We analyze the impact of instruction execution history within a pipelined, multi-cycle, IEEE-754 compliant [15] floating-point unit (FPU). Our analysis presents conclusive evidence that deep instruction execution history impacts timing failure rates, thus enhancing prior studies [8]–[10].
- We develop a framework based on stochastic search and PGS to generate failure-prone ISQs, considering the execution history in the pipeline and type and operands of instructions. The generated ISQs maximize the output quality loss incurred due to timing failures under an assumed delay increase. Such sequences are then being used to identify vulnerable code regions within various application binaries.
- We examine the DEFCON efficacy by comparing the failure-induced quality loss obtained for the generated ISQs and the quality loss observed for the ISQs that we extract from a representative set of applications.
- We quantify the degree to which various applications are prone to timing failures using a new application vulnerability metric, namely code vulnerability factor (CVF).
- We evaluate the DEFCON effectiveness in detecting faulty ISQs by estimating the True Positive Rate (TPR), which measures the fraction of actual faulty ISQs detected by DEFCON in real applications.

Overall, DEFCON effectively generates stress code that induces much higher output quality loss than conventional application codes and accurately detects faulty ISQs. We also demonstrate that DEFCON can save energy when combined with a dynamic clock stretching mechanism.

This paper is organized as follows. Section II presents the overview and motivation of our work, while Section III discusses the proposed framework. In Section IV, we present the experimental results; and conclusions are drawn in Section V.

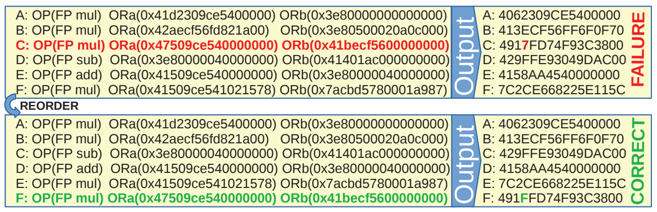
II. Overview and Motivation

Any pipelined core with S stages consists of a set of N combinatorial paths $P = \{p_1, p_2..p_N\}$, which are characterized by their delays $D(p_i)$ for $i = 1, 2..N$. In such a core, each of these paths can be found within exactly one pipeline stage $s = 1, 2..S$ and only few of them will be excited every time depending on the executed instruction. Note that in general an instruction is composed of an opcode (OP) and two input operands (ORa and ORb).

In any synchronous pipeline design, a setup timing failure [16] occurs if at any clock cycle the executed instruction activates a path P_i that has a delay (i.e. $D(p_i)$) more than the set clock period. In fact, the possibility of a path to fail under any delay variation depends on many parameters, such as the type of the in-flight instruction and its operands, as well as the instruction execution history of the pipeline. We elaborate with more details in the following paragraphs.

A. Timing Failures - Type of Instruction and Input Operands

The number and distribution of faults in a design strongly depend on the type of the executed instruction. Instructions



```
A: OP(FP mul)  ORa(0x41d2309ce5400000) ORb(0x3e80000000000000)
B: OP(FP mul)  ORa(0x42aecf56fd821a00)  ORb(0x3e80500020a0c000)
C: OP(FP mul) ORa(0x47509ce540000000) ORb(0x41becf5600000000)
D: OP(FP sub)  ORa(0x3e80000040000000) ORb(0x41401ac000000000)
E: OP(FP add)  ORa(0x41509ce540000000) ORb(0x3e80000040000000)
F: OP(FP mul)  ORa(0x41509ce541021578) ORb(0x7acbd5780001a987)
```

```
A: 4062309CE5400000
B: 413ECF56FF6F0F70
C: 4917FD74F93C3800
D: 429FFE93049DAC00
E: 4158AA4540000000
F: 7C2CE668225E115C
```

```
A: OP(FP mul)  ORa(0x41d2309ce5400000) ORb(0x3e80000000000000)
B: OP(FP mul)  ORa(0x42aecf56fd821a00)  ORb(0x3e80500020a0c000)
C: OP(FP sub)  ORa(0x3e80000040000000) ORb(0x41401ac000000000)
D: OP(FP add)  ORa(0x41509ce540000000) ORb(0x3e80000040000000)
E: OP(FP mul)  ORa(0x41509ce541021578) ORb(0x7acbd5780001a987)
F: OP(FP mul) ORa(0x47509ce540000000) ORb(0x41becf5600000000)
```

```
A: 4062309CE5400000
B: 413ECF56FF6F0F70
C: 429FFE93049DAC00
D: 4158AA4540000000
E: 7C2CE668225E115C
F: 491FFD74F93C3800
```

Fig. 1: Impact of instruction order on timing failures.

which activate critical long paths tend to fail more frequently [7], [10]. For example, one of the previous studies [8] shows that the slow floating-point addition instructions can fail more often than their integer counterparts, which excite less critical paths. Furthermore, depending on input operands, the same instruction may activate different paths of different latency requirements leading to different error rates [4], [8].

B. Timing Failures - Instruction History in Pipelines

Apart from the input operands and instruction type, parallel execution of instructions may also affect the possibility of timing failures. This is because concurrently executing instructions share control signals and execution stages, affecting the state of the forwarding logic, and thereby place great demand on circuit timing deadlines. In a previous study [7], authors show that ISQs have a significant impact on timing error rates, but they have not indicated how many instructions within a sequence affect this dynamic timing behavior nor suggest any method to identify these error-prone sequences. Fig. 1 shows an example that we encountered during our experiments. The top ISQ has exactly the same instruction opcodes and input operands to the bottom ISQ. However, instruction C has a timing failure that corrupts its output (highlighted in red) when subjected to a 15% delay increase (see Section IV.C). Changing the order of instructions, without violating execution dependencies, as shown in the bottom ISQ, the timing failure does not occur. Such an observation indicates that the order of instructions within an ISQ and thus the previously executed instructions may affect the failure probability.

The recent works [8]–[10], which studied the correlation between instruction history and failures, have indicated that, besides the currently executed instruction, only the instruction in the previous cycle affects timing failures. However, such an observation holds only in the case of simple, non-pipelined, functional units. Intuitively, all those instructions that precede an instruction in the pipeline may have an effect on the timing error behavior of this instruction. To investigate this, we extract a trace of $1M$ floating-point ISQs from the *bt* application (see Section IV.A) and run PGS of an FPU, the details of which will be discussed later. We simulate this trace in windows of increasing number of concurrently executed instructions, assuming a 15% delay increase. Specifically, we start simulation with a window size of 1 instruction and increase it up-to the pipeline depth, which is 6 stages in this FPU under test. Each experiment records the timing error rate (ER), defined as $\frac{Faulty\ ISQs}{Total\ ISQs}$, and the average relative error (avgRE), defined as:

$$avgRE = \frac{\sum_{i=1}^{K} \left| \frac{O_{gold}(i) - O_{sim}(i)}{O_{gold}(i)} \right|}{K} \quad (1)$$

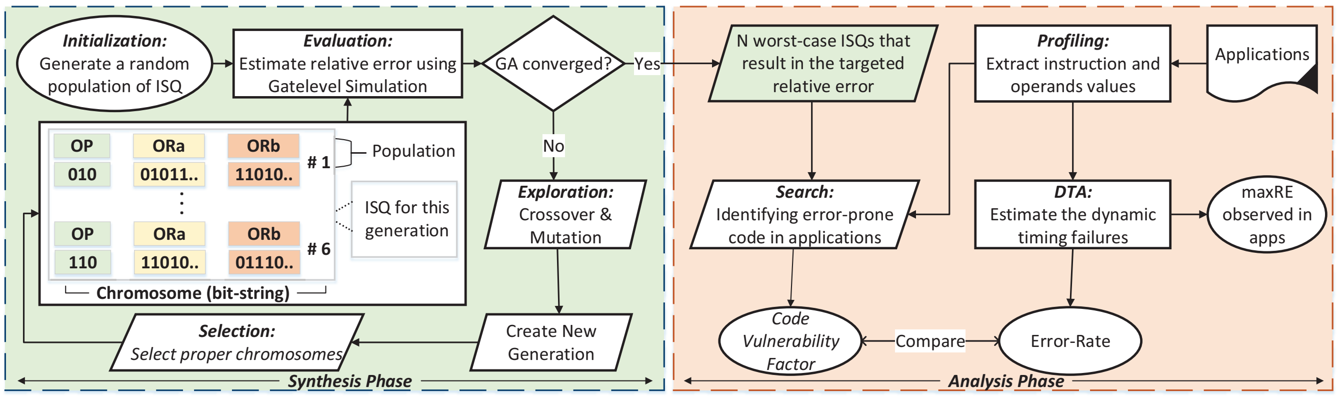

Initialization:
Generate a random population of ISQ
Evaluation:
Estimate relative error using Gatelevel Simulation
GA converged?
Yes
No
OP
ORa
ORb
010
01011..
11010..
# 1
Population
ISQ for this generation
110
11010..
01110..
# 6
Chromosome (bit-string)
Exploration:
Crossover & Mutation
Selection:
Select proper chromosomes
Create New Generation
Synthesis Phase
N worst-case ISQs that result in the targeted relative error
Profiling:
Extract instruction and operands values
Applications
Search:
Identifying error-prone code in applications
DTA:
Estimate the dynamic timing failures
maxRE observed in apps
Code Vulnerability Factor
Compare
Error-Rate
Analysis Phase

Fig. 2: Synthesis and analysis phases of DEFCON.

where $O_{gold}(i)$ and $Osim(i)$ denote the error-free output and simulated output, respectively, obtained by PGS for a specific ISQ (i). For this experiment, i varies from 1 to K = 1M (number of extracted ISQs). Fig. 3 indicates that sequences consisting of 6 instructions, i.e. a window of 6, have exactly the same ER and avgRE when running the full trace (full history) through simulation. By contrast, a window size of 1 instruction leads to $\sim 46.6\%$ smaller ER and ~ 5 orders of magnitude lower avgRE when compared to the window of 6 instructions. Based on these findings, we conclude that all the instructions preceding the currently executed instruction in the pipeline may trigger a failure in this instruction.

Challenge. Using PGS to exhaustively evaluate all possible parameters that affect dynamic timing failures (type of instructions, input operands, instruction execution history) is extremely time-consuming, if not practically infeasible. Particularly, in the target FPU, there are 6 instruction types (see Section IV.A) and thus 6 different OPs, with two double-precision 64-bit instruction operands and a pipeline depth of 6 stages. The number of all possible combinations of these execution parameters is: $\left(6 \times 2^{64} \times 2^{64}\right)^6 \approx 7.24 \times 10^{235}$. To address this challenge, we propose DEFCON, discussed next.

III. Design and Implementation of DEFCON

DEFCON is a framework for fast and accurate generation and detection of ISQs that are prone to timing failures. Initially, DEFCON formulates a genetic algorithm (GA) [17] to generate stressful ISQs for the execution unit under test, including input data and instruction execution history. Then it characterizes the fault vulnerability of applications by identifying error-prone ISQs in the application's binary. The overall DEFCON workflow is presented in Fig. 2. The *synthesis phase* is being executed only once to generate stressful ISQs, while the *analysis phase* runs for each application.

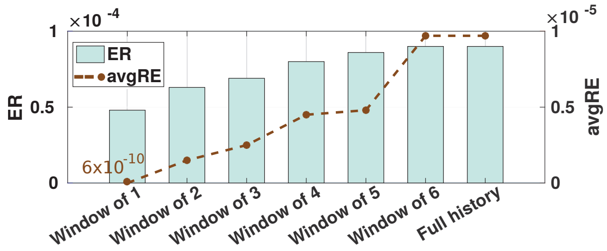


Fig. 3: Impact of instruction execution history on error rate (ER) and average relative error (avgRE).

A. Synthesis Phase

This phase synthesizes the ISQs that occur the maximum or a targeted failure-induced output error. As explained, the extremely large number of all possible combinations of parameters that affect failure rate renders the evaluation of every combination practically impossible. Instead, DEFCON formulates properly a GA, which implements a stochastic search to generate ISQs that maximize the output quality loss due to timing failures. GAs are stochastic search algorithms inspired by evolutionary biology to search for a solution to optimization problems. Similarly to the natural selection process, GA algorithm defines the following parameters.

Chromosomes. Chromosomes are used to encode information about parameters of a search. Particularly, for DEFCON, each chromosome defines a specific ISQ of pipeline depth (i.e. 6) instructions with specified OPs and input operands.

Selection. The algorithm selects chromosomes only when they fit the target evaluation function. Specifically, DEFCON searches for chromosomes that maximize the relative error induced by timing failures (see Eq. 2). GA measures this error for a specific chromosome by invoking PGS, supported by ModelSim (see Fig. 2, Evaluation stage). DEFCON records all sequences with a non-zero relative error, up-to the maximum found through this selection process.

Mutation. GA varies chromosomes through mutation to expand the search space. In particular, the mutation operation stochastically changes parts of a chromosome, in our case, the instruction type, order or input operands.

Crossover. GA also generates new chromosomes through the crossover operation. The crossover operation stochastically combines different parts of parent chromosomes to generate new chromosomes (offsprings).

Generations. The number of offspring generations (or algorithm iterations) generated by GA is in principle unlimited. However, DEFCON interrupts GA when the offspring obtaining the highest relative error target function does not change for several generations. It should be noted that chromosomes for the initial generation are generated randomly.

B. Validation of Synthesis Phase

To validate the generated worst-case ISQs, we perform DTA using Modelsim for PGS. DTA identifies the actual timing margins of the target core at runtime by including path activation information (instruction type, operand values, pipeline sequence) that are unavailable during static timing analysis. To feed ModelSim with real input data, we use a profiling tool to extract all ISQs from various applications. Applying DTA to the extracted traces allows us to measure all the dynamic timing failures. By those failures it is possible to calculate the output quality loss. We evaluate the efficacy of the synthesis phase by comparing the quality loss obtained for the worst-case ISQs synthesized by DEFCON and quality loss obtained for the faulty ISQs in applications.

C. Application Analysis Phase

At this phase, we apply DEFCON to detect error-prone code regions in real applications without long-running simulation campaigns. To this end, we compare ISQs generated at the synthesis phase of DEFCON with ISQs extracted from the considered applications. In particular, DEFCON implements a hash table that checks quickly an application binary for the presence of error-prone ISQs and splits all the application sequences into 2 groups: {failure-prone, failure-free}.

It is important to note that during the analysis phase, DEFCON matches only instruction opcodes but not operands. We use such an approach to minimize the risk of missing an ISQ which, depending on the operand values, may lead or not to timing failures. In other words, if we were matching the combination of instruction opcodes and operands, then DEFCON might mischaracterize an actual faulty ISQ as error-free when a specific set of operands is used. Thus, to enable operand-agnostic application fault vulnerability analysis, DEFCON investigates ISQs consisting of only opcodes during the analysis phase. In our framework, we analyze the sequences of 6 opcode types which correspond to the depth of the pipeline. We refer to the set of error-prone instruction opcodes used by DEFCON in this phase as EIS_{OP}. Due to the fact that the same error-prone OP sequence (without considering the operands) may lead to different degrees of quality loss, any sequence will be found exactly once in EIS_{OP}.

IV. Experimental Results

In this section, we first present our experimental setup. Second, we show convergence of the GA search. Then, we evaluate the efficacy of DEFCON for characterizing the application vulnerability and detecting erroneous ISQs. Lastly, we discuss several use cases of DEFCON.

A. Experimental Setup and Application Profiling

We apply our approach to a 6-stage pipelined, multi-cycle, IEEE-754 compatible FPU that supports both single and double precision operations. This unit is a part of the mor1kx MAROCCHINO pipeline, which is a processor implementing the OpenRISC 1000 instruction set architecture [18]. The FPU supports the following floating-point instructions: multiplication, division, addition and subtraction, integer-to-float and float-to-integer conversions. This design is implemented using the typical corner of the CCS NanGate 45 nm library (@1.1V). For hardware synthesis and place-and-route, we use the Design Compiler from Synopsys and Innovus from Cadence.

To measure the efficacy of DEFCON, we collect floating-point instruction traces by instrumenting various programs from the Rodinia [19] and NAS [20] benchmark suites. Specifically, those programs are *k-means* with 1000 input data points and 34 features, *hotspot* with 1024x1024 grid and *cg*, *is*, *mg* and *bt* with S input sizes. For collecting floating point traces from real size inputs in reasonable time (from 8 to 114 secs), we use DynamoRIO [21] to instrument program binaries executing on an ARM A7 board. The ARM FPU has an 1-to-1 correspondence of instructions to the target OpenRISC FPU.

B. Evaluation of the GA Search Engine

Experimentally, we found that GA converges to a solution faster when the number of chromosomes in each generation, the crossover probability and the mutation probability (Section III.A) are set to 40, 0.9 and 0.6, respectively. Also, to accelerate the GA search process, we ran it in parallel on 16 cores of an Intel Xeon system clocked at 2.0 GHz. Note that the avgRE of the generated ISQs varies with the number of generations and converges to 1.0 and does not change after 120 generations. The search process running on one core takes 20 hours on average to converge.

C. Worst-case Instruction Sequences Generation

In this subsection, we evaluate the effectiveness of DEFCON in generating stressful ISQs under different clock reduction (CR) levels, which represent potential degrees of variation-induced worst-case delay increase. In this work, we use two different CR levels, CR1 and CR2 that correspond to 15% and 13% worst-case delay increase, respectively. CR1 and CR2 are consistent with the levels of variation-induced delay increase that have been reported in literature [2], [4].

To measure the maximum quality loss incurred by the timing failures under those two CR levels, we compare maxRE of the most stressful ISQs generated by DEFCON with maxRE observed in the considered applications. Based on Eq. 1, maxRE is defined as:

$$maxRE = \max_{i=1...K} \left| \frac{O_{gold}(i) - O_{sim}(i)}{O_{gold}(i)} \right| \quad (2)$$

Fig. 4 compares the maxRE obtained by DEFCON with the one measured for each program under the different CR regimes. Under CR2, i.e. increasing delay by 13%, 5 out

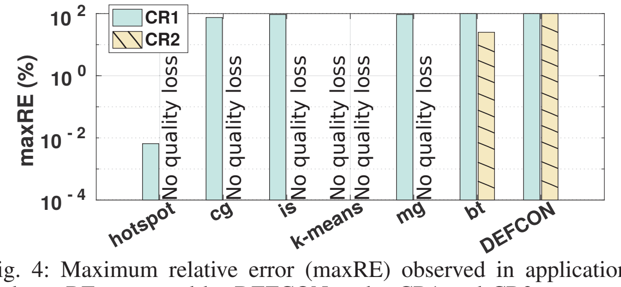


Fig. 4: Maximum relative error (maxRE) observed in applications and maxRE generated by DEFCON under CR1 and CR2.

of 6 benchmarks (*hotspot*, *cg*, *is*, *k-means*, *mg*) have no timing failures, hence there is no quality loss. In the case of *bt* benchmark the measured maxRE is 25.1%. DEFCON supersedes the measured maxRE of applications by generating ISQs with maxRE up-to 100%, which is 95.8% on average higher than the one observed for the applications under CR2.

For CR1, which corresponds to a 15% delay increase, all programs apart from *k-means* manifest timing failures, and the measured maxRE ranges from $7 \cdot 10^{-3}\%$ to 100%. This difference in maxRE (and the number of timing failures) between CR1 and CR2 is consistent with the *timing wall* phenomenon [4], [5]. We see that in both cases, CR1 and CR2, the measured maxRE is always equal or less than the maxRE obtained for the worst-case ISQs synthesized by DEFCON. These results imply that DEFCON is effective for generating error-prone ISQs that stress the system, in terms of output quality loss, more than conventional workloads.

D. Analysis of Application Vulnerability

Apart from generating stressful ISQs, DEFCON characterizes the vulnerability of applications, by inspecting their binary code. To achieve this, as explained in Section III.C, we implement a hash-based search engine that quickly detects instances of the EIS_{OP} set (error-prone ISQs discovered by DEFCON) for each program. We refer to the error-prone opcode sequence set obtained by this application specific search as EIS_{app}. To quantify the vulnerability of each application, we introduce a new metric named as Code Vulnerability Factor (CVF). CVF is defined as follows: $CVF = \frac{|EIS_{app}|}{Total\ ISQs}$.

CVF measures for each benchmark under CR1 and CR2 are provided in Fig. 5. To verify the proposed metric, we estimate ER for each application using DTA. Intuitively, we suggest that applications with a high CVF are more vulnerable to failures and thus have a higher ER. ER measures are depicted in the right y axis of Fig. 5. We see that ER at CR1 for each application (apart from k-means where no timing failures are manifested) is significantly higher than the one obtained at CR2. Similarly, the proposed vulnerability metric grows with the CR level: CVF at CR1 is 18.3× higher than CVF at CR2 on average. Furthermore, in the same figure, we also observe that the *bt* benchmark at CR1 incurs the highest ER (10^{-4}), while in the case of *k-means* ER is 0. Identically, *bt* and *k-means* benchmarks incur the highest CVF (0.44) and lowest CVF (0), respectively. The similar correlation between CVF and ER metrics are observed also for *hotspot*, *cg*, *mg* and *is* benchmarks. Based on these results, we conclude that CVF efficiently estimates erroneous application behavior and can be applied for characterization of the application vulnerability.

E. DEFCON Accuracy

In this section, we analyze the accuracy of DEFCON in detecting faulty ISQs within applications. DEFCON outcomes of inspecting a program binary are divided in 4 categories: True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN). TP represents the number of ISQs that DEFCON correctly detects as faulty and TN counts the number of ISQs that DEFCON correctly detects as non-faulty.

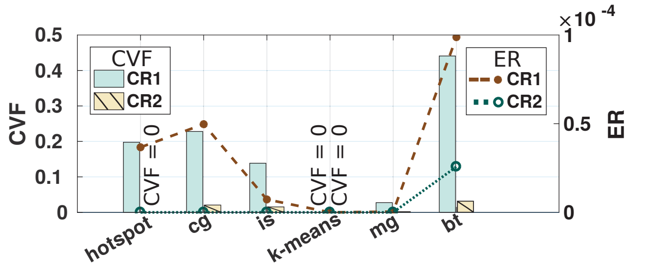


Fig. 5: Code vulnerability factor (CVF) extracted by DEFCON and error rate (ER) obtained by DTA in applications under CR1 and CR2.

FP and FN count the number of incorrect detections. For high detection accuracy, it is essential to avoid FN outcomes and maximize TP outcomes. The True Positive Rate (TPR) measures detection accuracy and is defined formally as: $TPR = \frac{TP}{TP+FN}$. Also, we evaluate DEFCON's operation using the metric of the True Negative Rate (TNR), which shows the ratio of correctly identified non-faulty ISQs: $TNR = \frac{TN}{TN+FP}$. Thus, the accuracy of DEFCON is a trade-off between these two metrics that may negatively affect each other. Note that target goal of this work is to achieve high TPR by identifying correctly all the actual faulty ISQs and avoiding FNs.

Table I presents DEFCON outcomes and accuracy measured across all the programs under CR1. The average TPR achieves 99.7%, which implies that the vast majority of faulty ISQs were correctly detected, while the average TNR is 82.2%, indicating that the number of correctly identified non-faulty ISQs is also high. We see that for *hotspot*, *cg*, *is* and *mg* programs, DEFCON detects accurately all faulty ISQs (TPR=100%) with considerably high TNR ranging from 77.2% to 97.2%. Additionally, DEFCON identifies correctly that there are no timing failures for *k-means* program, i.e., $|EIS_{app}|$ = faulty ISQs = 0. For *bt* program, TPR is 98.9%, however, TNR is relatively low (55.9%), which indicates that there were many FP outcomes. We explain this by the fact that *bt* program contains a lot of error-prone ISQs which data operands, however, do not trigger errors. Nonetheless, these ISQs may manifest failures for different input data set.

F. Potential Use Cases

As shown, DEFCON effectively generates and detects error-prone code. Such a code can be used early at the design cycle for evaluating the susceptibility of any design to timing errors. Additional use cases of DEFCON are discussed next.

1) Generation of stress code: The synthesis phase of DEFCON synthesizes error-prone ISQs that incur a significant failure-induced quality loss under different levels of delay variation. Sources of this variation can be process variation, voltage droop, operating temperature or aging. Hence, DEFCON generates micro-architecture aware stress code that maximizes the quality degradation under potential delay increase that may be induced by different types of variations. Existing works generate stress viruses that maximize delay increase considering only voltage droops [13], [14] and are agnostic of underlying hardware, requiring complicated infrastructure.

2) Detection of Error-prone ISQs: DEFCON can be also applied to detect error-prone ISQs in real applications without

TABLE I: DEFCON outcomes and accuracy measures using different metrics across all benchmarks under 15% worst-case delay increase.

Apps	**ISQs**	**EIS$_{app}$**	**Faulty ISQs**	**FP**	**TP**	**FN**	**TN**	**TPR (%)**	**TNR (%)**
hotspot	408806	80759	15	80744	15	0	328047	100	80.2
cg	704046	160543	35	160508	35	0	543503	100	77.2
is	4259984	589903	31	589872	31	0	3670081	100	86.2
k-means	7351820	0	0	0	0	0	7351820	100*[2]	100
mg	7396151	203075	3	203072	3	0	7193076	100	97.2
bt	43533307	19193326	4300	19189074	4252	48	24339933	98.9	55.9

long-running simulation campaigns. In particular, performing DTA based on ISQs extracted from *bt* program takes almost 2 days. The proposed framework enable us to quickly (up-to 55 seconds in the case of *bt*) check if a particular application is tolerant to failures (CVF=0) when FPU operates at reduced voltages. For example, in the case of *k-means*, DEFCON indicates that no timing failures will occur under 15% and 13% delay increase. Accordingly, we can reduce the voltage from 1.1V down to 0.95V without obtaining failures when running this workload and thus improve power/energy efficiency by 29.3%. Note that for our implementation 15% delay increase is imposed by the available slow corner library (@0.95V).

In addition, when considered jointly with a design-centric technique, such as dynamic cycle adjustment [22], DEFCON facilitates operations at lowered voltages for every application. The main idea behind this approach is that the design under test constantly operates at a lowered voltage (e.g. 0.95V), while the clock period is dynamically stretched when an error-prone ISQs is detected. Specifically, every instruction is characterized by DEFCON as critical (error-prone) or off-critical depending on the group of ISQs (see Section III.C). Then, we implement a lookup table that contains the type of each executed instruction to check if a particular instruction in the pipeline is critical. Once such an instruction is detected, the clock period increases (by 15% for 0.95V), providing sufficient time to mitigate timing failures. As shown in Table II, such a technique saves from 21.2% up-to 29.3% of energy. Note that we estimate these savings considering the execution time overhead incurred by the occasional clock stretching.

Finally, our framework facilitates accurate error injection during the evaluation of application resiliency [6], as errors can be injected based on error-prone ISQs detected by DEFCON rather than using random injection strategies. DEFCON-driven error injection enables the user to evaluate the impact of timing errors on applications in a realistic manner.

TABLE II: Energy savings across all benchmarks when DEFCON considered jointly with a cycle adjustment technique [22].

Apps	**hotspot**	**cg**	**is**	**k-means**	**mg**	**bt**
Energy gains (%)	26.1	27.2	28.4	29.3	28.9	21.2

V. Conclusions

In this paper, we present DEFCON, a framework for generating and detecting error-prone ISQs in pipelined cores, considering for the first time the full pipeline execution history, instruction type and data dependent path excitation. DEFCON formulates a genetic algorithm driven by accurate PGS to stochastically search for micro-architecture aware ISQs that trigger timing failures under delay variations. Our framework use these sequences to detect error-prone code regions in application binaries and to characterize the application vulnerability. Our results show that DEFCON effectively generates stressful ISQs that maximize the failure-induced output quality loss in an FPU. We also demonstrate that DEFCON correctly detects faulty ISQs in a set of various benchmarks with 99.7% accuracy on average. Finally, we present several use cases of DEFCON, which enables us to achieve energy savings.

[2]TPR cannot be calculated (division by zero), but since $|EIS_{app}|$ = faulty ISQs = FN = 0, we set TPR = 100%.

Acknowledgments

This work is partially supported by the European Commission (H2020) under grant 688540 and 732631. This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 (LLNL-CONF-770077).

References

[1] Y. Zhang *et al.*, "irazor: Current-based error detection and correction scheme for pvt variation in 40-nm arm cortex-r4 processor," *JSSC*, 2018.
[2] P. Gupta *et al*, "Underdesigned and opportunistic computing in presence of hardware variability," *TCAD*, 2013.
[3] D. Karaklaji *et al.*, "Hardware designer's guide to fault attacks," *IEEE TVLSI*, vol. 21, no. 12, pp. 2295–2306, Dec 2013.
[4] I. Tsiokanos *et al.*, "Significance-driven data truncation for preventing timing failures," *IEEE TDMR*, vol. 19, no. 1, pp. 25–36, March 2019.
[5] I. Tsiokanos *et al.*, "Low-power variation-aware cores based on dynamic data-dependent bitwidth truncation," in *DATE*, 2019, pp. 698–703.
[6] M. Kaliorakis *et al.*, "Merlin: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," in *ISCA*, 2017, pp. 241–254.
[7] G. Hoang *et al.*, "Exploring circuit timing-aware language and compilation," in *ASPLOS*. ACM, 2011, pp. 345–356.
[8] X. Jiao *et al.*, "Clim: A cross-level workload-aware timing error prediction model for functional units," *Trans. on Comp.*, pp. 771–783, 2018.
[9] J. J. Zhang *et al.*, "Fate: Fast and accurate timing error prediction framework for low power dnn accelerator design," in *ICCAD*, Nov 2018.
[10] G. Tziantzioulis *et al.*, "b-hive: A bit-level history-based error model with value correlation for voltage-scaled integer and floating point units," in *DAC*, June 2015, pp. 1–6.
[11] Y. Fan *et al.*, "Compiler-guided instruction-level clock scheduling for timing speculative processors," in *DAC*, June 2018, pp. 1–6.
[12] O. Assare *et al.*, "Accurate estimation of program error rate for timing-speculative processors," in *DAC*. ACM, 2019, p. 180.
[13] Y. Kim *et al.*, "AUDIT: stress testing the automatic way," in *MICRO*, 2012, pp. 212–223.
[14] Z. Hadjilambrou *et al.*, "Sensing cpu voltage noise through electromagnetic emanations," *IEEE CAL*, vol. 17, no. 1, pp. 68–71, Jan 2018.
[15] IEEE 754-2008 Standard for Floating-Point Arithmetic.
[16] J. Bhasker *et al.*, *Static Timing Analysis for Nanometer Designs: A Practical Approach*, New York, USA: Springer, 2009.
[17] M. Mitchel, *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT press, 1998.
[18] OpenRISC, "OpenRISC 1000 architecture manual".
[19] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, Washington, DC, USA, 2009, pp. 44–54.
[20] D. H. Bailey *et al.*, "The nas parallel benchmarks&mdash;summary and preliminary results," in *Supercomputing*, NY, USA, 1991, pp. 158–165.
[21] D. L. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Cambridge, MA, USA, 2004.
[22] J. Constantin *et al.*, "Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment," in *DATE*, 2015, pp. 381–386.