ExHero: Execution History-aware Error-rate Estimation in Pipelined Designs

ExHero: Execution History-aware Error-rate Estimation in Pipelined Designs

Ioannis Tsiokanos and Georgios Karakonstantis

Institute of Electronics, Communications and Information Technology, Queen's University Belfast, UK

Email: {*itsiokanos01, g.karakonstantis*}*@qub.ac.uk*

Abstract—The increased variability renders nanometer devices prone to timing errors. Recent works focused on the development of error prediction models for either evaluating the effects of timing errors on applications or guiding the voltage/frequency settings. Such models may have considered the data-dependent excitation of paths, but they have neglected the impact of all the concurrently executed instructions on error occurrence; which may lead to inaccurate error estimation in pipelined designs. To investigate such a limitation, we develop ExHero, a fully automated framework that performs dynamic timing analysis considering the execution history of a number of in-flight instructions. Using ExHero, we first demonstrate that the order and type of instructions within sequences that have length equal to the pipeline depth significantly affect the error-rate. When applied to a pipelined floating-point unit, ExHero reveals that existing approaches estimate on average 46.5% and 32.1% lower error-rate and absolute error, respectively than the actual ones.

Index Terms—Timing errors, instruction execution history, pipeline, microarchitecture, error resilience, multi-cycle FPU.

1 Introduction

Advanced technology scaling and increased static and dynamic variability caused by process, temperature, voltage, and aging effects make circuits prone to timing errors. Such errors, the probability of which is further worsening upon supply voltage scaling and dynamically changing environmental conditions, threaten the system functionality [1].

State-of-the-Art. Such a reality led researchers to model timing errors [1], [2] and evaluate their effects on applications [3], [4]. Recently, there is a growing interest in techniques that improve the accuracy of conventional data-agnostic error models and frameworks which assume fixed error probabilities [4]. Existing timing error models tried to capture the data-dependent dynamic timing behavior of arithmetic operations [3], [5], [6] by considering microarchitectural (i.e., instruction type) and workload (i.e., operands) features, under various potential delay variations. Although such frameworks paved the way for improving the timing error estimation, they were applied to simple independent functional units and not in pipelined microarchitectures that support the concurrent execution of multiple instructions. In particular, such work considers only the currently executed instruction and the immediately preceding one, neglecting the influence of all the in-flight instructions. Such a limitation led them to overlook the impact of ***instruction execution history*** (i.e., the type and order of instructions within a pipeline at any instance) on timing errors, which may lead to inaccurate error prediction.

Contributions. In this paper, we present ExHero[1], a microarchitecture-aware framework that takes steps towards analyzing the impact of instruction execution history on the error-rate of pipelined units, while jointly considering all the other critical factors affecting the error occurrence. The main contributions of our work are the following:

- We develop a framework based on commercial Electronic Design Automation (EDA) tools that allow us to capture the effects of instruction execution history on timing error occurrence. To understand how timing errors vary across instruction sequences, we conduct a detailed simulation study using four benchmarks with different number of instructions within a sequence, and quantify the relative contribution of instruction sequences to the overall error-rate.
- We analyze the impact of instruction execution history within a pipelined, out-of-order, multi-cycle, IEEE-754 compliant [7] floating-point unit (FPU). Our analysis presents conclusive evidence that deep instruction execution history impacts timing error-rates, thus complementing and advancing prior studies [3], [5], [8].
- We estimate the error-rate (ER) under sequences consisting of different number of instructions. Based on ER, we also evaluate the error-induced output quality loss quantified in terms of the absolute error.

The rest of the paper is organized as follows: Section 2 provides the background and motivation of our work, while Section 3 discusses the ExHero implementation. In Section 4, we present the experimental results. Section 5 describes our work with relation to other research; and conclusions are drawn in Section 6.

2 Background and Motivation

Any pipelined core with K 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 $k = 1, 2..K$ and only few of them will be excited at

1. **Ex**ecution **H**istory-aware **Er**ror-rate Estimati**o**n (ExHero)

every instance depending on the executed instruction [6]. Typically, an instruction is composed of an operation code (OP) which denotes the type of the instruction, a destination (ORd) and two input operands (ORa and ORb) represented as binary vector. Note that each pipeline stage processes a specific part of one instruction at a time, allowing the parallel execution of multiple instructions. By the terms *parallel* or *concurrent* execution of instructions, we mean that up-to K instructions share the same hardware circuitry (i.e., pipeline) in a time-sharing fashion.

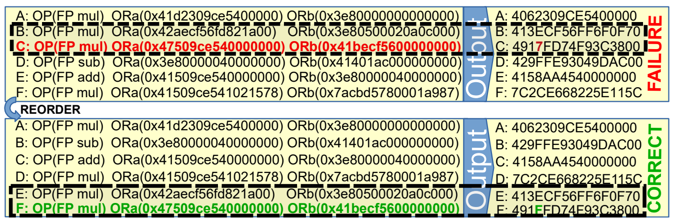
In any synchronous pipeline design, a setup timing error occurs if at any clock cycle the executed instruction activates a path P_i that has a delay (i.e. $D(p_i)$) more than the set clock period. In fact, the possibility of a path to fail under any delay variation depends on many parameters [2], [3], [8]:
(1) the type of the in-flight instruction, (2) the input operands of the executed instruction, as well as (3) the instruction execution history of the pipeline. We elaborate with more details in the following paragraphs.

2.1 Background: Type of Instruction and Operands

The number and distribution of faults in a design strongly depend on the type of the executed instruction. Instructions which activate long critical paths tend to fail more frequently [5], [8]. For example, one of the previous studies [3] shows that the slow floating-point addition instructions can fail more often than their integer counterparts, which excite less timing critical paths. Furthermore, depending on input operands, the same instruction may activate different paths of different latency requirements leading to different error-rates [2], [3].

2.2 Motivation: Instruction History in Pipelines

Apart from the input operands and instruction type, parallel execution of instructions may also affect the possibility of timing errors. 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 fact, the delay of each sensitized path depends on the state of its nodes (input/output ports and gate pins), which is set by the previously executed inputs. Notably, in a pipelined microarchitecture where different instructions may be sharing common circuitry within each stage, the node state of each path at each stage depends also on the previously executed instructions. That is, the joint effects of the in-flight instruction and a number of previously executed instructions determine the delay of the excited path in every pipeline stage and thus the error-rate. This has been observed in a previous study [8], where authors show that instruction sequences have a significant impact on timing error-rates. This work showcased the spatial timing error locality where static instructions exhibit consistent error behavior over a period of program execution. However, this study is agnostic of the underlying microarchitecture and thus it does not reveal how many instructions within a sequence affect the dynamic timing behavior nor suggests any method to identify the location (i.e., bit) of a potential timing error. To provide accurate, instruction-aware, bit-level error models,



```
A: OP(FP mul)  ORa(0x41d2309ce5400000) ORb(0x3e80000000000000)    A: 4062309CE5400000
B: OP(FP mul)  ORa(0x42aecf56fd821a00)  ORb(0x3e80500020a0c000)    B: 413ECF56FF6F0F70
C: OP(FP mul) ORa(0x47509ce540000000) ORb(0x41becf5600000000)     C: 4917FD74F93C3800
D: OP(FP sub)  ORa(0x3e80000040000000) ORb(0x41401ac000000000)    D: 429FFE93049DAC00
E: OP(FP add)  ORa(0x41509ce540000000) ORb(0x3e80000040000000)    E: 4158AA4540000000
F: OP(FP mul)  ORa(0x41509ce541021578) ORb(0x7acbd5780001a987)    F: 7C2CE668225E115C

A: OP(FP mul)  ORa(0x41d2309ce5400000) ORb(0x3e80000000000000)    A: 4062309CE5400000
B: OP(FP sub)  ORa(0x3e80000040000000) ORb(0x41401ac000000000)    B: 429FFE93049DAC00
C: OP(FP add)  ORa(0x41509ce540000000) ORb(0x3e80000040000000)    C: 4158AA4540000000
D: OP(FP mul)  ORa(0x41509ce541021578) ORb(0x7acbd5780001a987)    D: 7C2CE668225E115C
E: OP(FP mul)  ORa(0x42aecf56fd821a00)  ORb(0x3e80500020a0c000)    E: 413ECF56FF6F0F70
F: OP(FP mul) ORa(0x47509ce540000000) ORb(0x41becf5600000000)     F: 491FFD74F93C3800
```

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

subsequent works [3], [5] studied the correlation between instruction history and errors, and indicated that timing errors in the currently executed instruction can be triggered either explicitly by this instruction (and its operands) or implicitly by the previous instruction. Therefore, they point out that a window of two instructions (i.e., the current instruction and the previous one) is necessary and sufficient to obtain fully accurate error-rate estimation. However, such an observation holds only in the case of simple, non-pipelined, functional units. Intuitively, a window of more than two instructions may have an effect on the timing error behavior of pipelined units where multiple instructions are executed (are in-flight) in the pipeline. To verify our hypothesis, we illustrate on Figure 1 an example that we encountered during our experiments. The top instruction sequence has exactly the same instruction opcodes and input operands to the bottom instruction sequence. When we run post-layout gate-level simulation (see Section 3.2) of a pipelined FPU the details of which will be discussed later, instruction C has a timing error that corrupts its output (highlighted in red). Such an error occurs when the target FPU is subjected to a 15% delay increase representing possible variations induced by various sources [1] (see Section 4) through simulation. Based on prior works [3], [5], if we fix both the faulty instruction C and the immediately preceding instruction B, while changing the order of the other instructions in the sequence, we expect exactly the same timing behavior of instruction C. Nonetheless, if we change the order of instructions without violating execution dependencies, as shown in the bottom instruction sequence, the timing error does not occur. Such a finding indicates that a window consisting of more than two instructions affects the timing error probability in pipelined units. For instance, in the top instruction sequence depicted in Figure 1, the timing error in instruction C is determined by the joint effects of an execution history window of three instructions: the previous two instructions (i.e., instructions A and B) and current instruction (i.e., instruction C). This is attributed to the fact that these instructions excite paths in shared circuits in some of the stages, as we explained above.

3 Design and Implementation of ExHero

To investigate how many instructions that precede an instruction in the pipeline play a role in the timing error behavior of this instruction, we propose ExHero. The overall ExHero workflow, as shown in in Figure 2, consists of the *design phase* and *analysis phase*. The *design phase* is executed only once, while the *analysis phase* runs for each application.

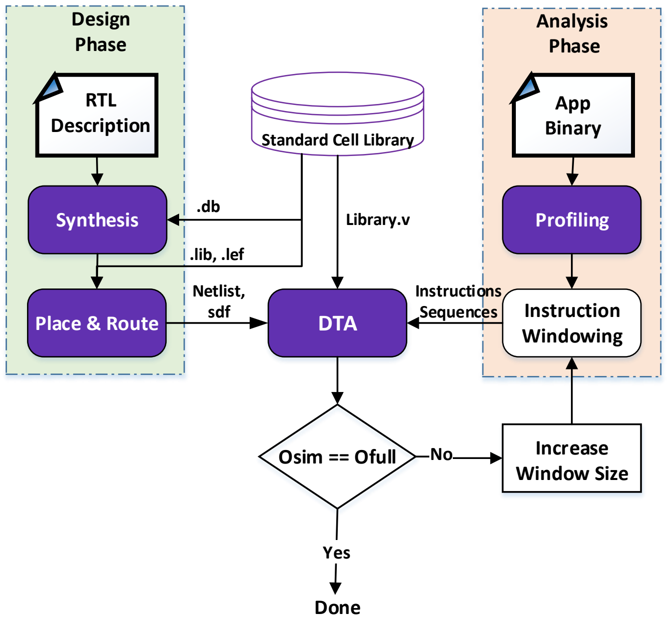


Fig. 2: Overall workflow of ExHero.

3.1 Design Phase

The first step of this phase is the Synthesis which is followed by the Place and Route steps. Note that these steps are performed utilizing optimization steps which aim at achieving maximum performance. *Design phase* outputs the following files:
i) A library verilog file which specifies the logic and the rise and fall times of the standard cells.
ii) A gate-level netlist which is stored in a verilog format (.v). This file consists of a list of the electronic components in the circuit and a list of the nodes they are connected to.
iii) A standard delay format (SDF) file which describes the cell and interconnect delay.

3.2 Analysis Phase

At this phase, we first extended a sampling-based profiling tool to extract program traces running different applications on a real hardware. In particular, we extract one million instruction sequences from each of the considered applications. Then, we split the extracted instructions into small subsets of instruction sequences by applying instruction windowing. The number of the instructions within a sequence depends on the window size (WS): $WS = 1, 2, .., K$, where K denotes the maximum pipeline depth. Each WS ties the circuit's behavior on the current instruction to the history of all the preceding instructions in the pipeline. Note that the window size indicates the number of the instructions that are executed concurrently. For example, a window size equal to two (we refer to this as $WS = 2$) shows that two instructions are in-flight in the pipeline.

Dynamic Timing Analysis (DTA). Then, using the extracted traces and the outputs of the *design phase*, we perform DTA to estimate the manifestation of dynamic timing errors under any potential delay increase. DTA identifies the actual timing margins of the target design at runtime by including path activation information (instruction type, operand values, pipeline sequence) that are unavailable during static timing analysis [2]. To achieve this, we use detailed post-layout gate-level simulation in windows of increasing numbers of concurrently executing instructions, while assuming a potential variation-induced delay increase (see Section 4). We refer to the magnitude of this delay increase as ΔT. We start the simulation with a window size of one instruction (i.e., one instruction each time in the pipeline). If the simulation output is not identical with the output when simulating the full trace (full history), we increase the window size. We refer to the simulation output of the full trace as $Ofull$. We repeat this step until the simulation output (O_{sim}) matches O_{full} and thus the actual timing behavior.

4 Evaluation Results

In this section, we first present our experimental setup. Then, we evaluate ExHero, assuming a $\Delta T = 15\%$ worst-case delay increase. This delay increase is consistent with the levels of variation-induced delay increase that have been reported in literature [1], [2]. To represent potential degrees of worst-case delay increase, we reduce the clock period accordingly. For example, 15% worst-case delay increase is represented by reducing the clock period by 15%. Finally, we quantify the inaccuracy involved in existing execution history-aware error prediction frameworks.

4.1 Experimental Setup and Application Profiling

For the case study of using ExHero, we focus on arithmetic, floating-point operations since those are more prone to timing errors, as also reported by existing studies [2], [3], [5]. Moreover, FPUs typically determine the clock frequency and are excellent representatives of complex pipelined designs. Nonetheless, ExHero can be applied to any other design since the Exhero workflow (see Figure 2) is implemented in a fully automated way. As a result, this automated procedure increases the generality of our framework since it only requires the register-transfer level (RTL) description of the target core.

We apply our evaluation framework to a pipelined, out-of-order, 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 (ISA) [9]. The FPU supports the following floating-point instructions: multiplication, addition and subtraction, integer to floating-point and floating-point to integer conversions. Figure 3 illustrates the pipeline of the targeted FPU. As a multi cycle unit, this FPU provides different pipelined depths, and thus different latency, across the considered operations. In particular, the first one (i.e., OCB) and the last two pipeline stages (i.e, R1, R2) of Figure 3 are common across all the floating-operations, while the pipeline depth of each operation can be observed in Table 1. Figure 4 highlights the micro-architecture of the floating-point addition/subtraction operations. At Stage 1, an Order Control Buffer (OCB) and a Pre-Normalize block are implemented, which permits data dependencies detection and adjustment of the exponent and mantissa, respectively. Stage 2

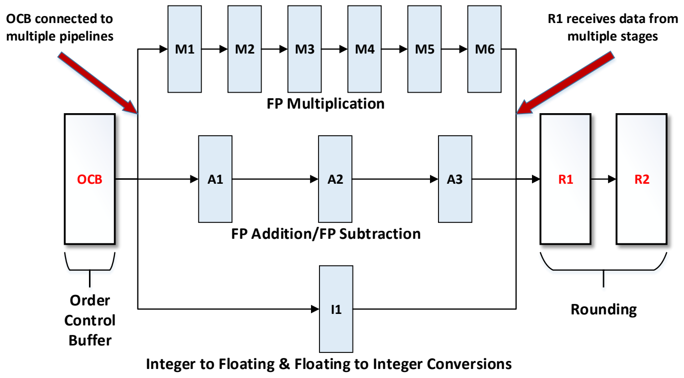

OCB connected to multiple pipelines
R1 receives data from multiple stages
M1
M2
M3
M4
M5
M6
FP Multiplication
OCB
A1
A2
A3
R1
R2
FP Addition/FP Subtraction
I1
Order Control Buffer
Rounding
Integer to Floating & Floating to Integer Conversions

Fig. 3: Pipeline of the target FPU. This pipeline supports multiple outstanding floating-point (FP) operations. OCB, R1 and R2 stages are common for every operation.

TABLE 1: Pipeline depth across the considered floating-point operations. Both single and double precision operations of the same instruction type have the same pipeline depth.

Operation (single and double precision)	Pipeline depth
Integer to floating-point conversion	4
Floating-point to integer conversion	4
Floating-point addition	6
Floating-point subtraction	6
Floating-point multiplication	9

is responsible for the pre-addition/subtraction alignment, while Stage 3 performs the necessary multiplexing and shifting of the operands. Mantissa addition and exponent update are performed at Stage 4; post-normalization and rounding occur in the last two stages. We refer to the last two rounding stages as R1 and R2, respectively.

This FPU design is implemented by applying the *design phase* depicted in Figure 2 on the CCS NanGate 45 nm standard cell library (@1.1V). In this phase, the Synthesis and Place and Route steps are conducted using Synopsys Design Compiler and Cadence Innovus, respectively. The maximum clock frequency achieved is 480MHz. As for the post-layout gate-level simulation, we use ModelSim from Mentor Graphics.

We collect floating-point instruction traces by instrumenting various programs from the NAS [10] benchmark suite. Specifically, those programs are *is, mg, ft* and *ep* with S input sizes. For collecting floating point traces from real size inputs in reasonable time (from 13 to 120 secs), we extend an open-source profiling tool [11] to instrument program binaries executing sequentially on a single thread of an ARM A7 board clocked at 2.1 GHz. The ARM FPU has an 1-to-1 correspondence of instructions to the target OpenRISC FPU.

4.2 Impact of Instruction Execution History on Timing Error-rate

To quantify the impact of instruction execution history on error manifestation, we perform DTA using different window sizes. We start with $WS = 1$ and increase it until the simulation output matches the simulation output of the full

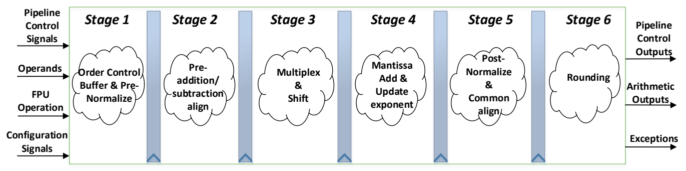


Fig. 4: Microarchitecture of the floating-point addition/subtraction operations.

history. Note that the full history simulation corresponds to the actual dynamic timing behavior of the design under test. Each step records the timing error-rate (ER) defined as:

$$ER = \frac{Faulty\ Instruction\ Sequences}{Total\ Instruction\ Sequences} \tag{1}$$

and the average absolute error (avgAE), defined as:

$$avgAE = \frac{\sum_{i=1}^{I} |O_{gold}(i) - O_{sim}(i)|}{I} \tag{2}$$

where $O_{gold}(i)$ and $Osim(i)$ denote the error-free output and simulated output, respectively, obtained by simulating a specific instruction sequence (i). For this experiment, i varies from 1 to I = 1 M (number of extracted instruction sequences for each program).

Figure 5 shows the ER and $avgAE$ of each benchmark under 15% delay increase. We observe that different benchmarks incur different ER and $avgAE$. This happens because different input instructions activate different paths contributing to the calculation of different output bits. We also observe that the number of the previously executed instructions that affects the error-rate of the instruction executed in the current cycle varies significantly across the considered benchmarks. In the case of the *is* program, sequences consisting of 6 instructions (i.e., $WS = 6$) have exactly the same ER and $avgAE$ when running the full trace (i.e., full history) through simulation. By contrast, $WS = 1$ leads to 51.1% smaller ER and 26.4% lower $avgAE$ when compared to $WS = 6$. Similar results are reported in the case of the *ft* program where $WS = 6$ results in the same behavior with the full history simulation (i.e., the actual manifested errors). When compared to the full history, $WS = 1$ occurs 85.7% smaller ER and 46.2% less $avgAE$. In the case of the *mg* program, only the currently executing instruction and the preceding one (i.e., $WS = 2$) affect the ER and the $avgAE$. In the case of the *ep* program, when we take into account windows of 1 and 2 instructions, no timing errors are manifested and thus there is no quality loss ($avgAE = 0$). Conversely, $WS = 9$, which is equivalent to the full history for this program, results in 32 faulty instruction sequences with ER and $avgAE$ equal to 3.2×10^{-5} and 1.3×10^{7}, respectively.

4.2.1 Comparison with Prior History-aware Frameworks

As explained in Section 2.1, existing work [3], [5] showcased the need to incorporate instruction execution history for accurate error prediction. Specifically, this work indicates that both current and previous instruction (i.e., WS=2) determine the delay of the excited path and thus the error manifestation. However, $WS = 2$ leads to on average 46.5% and 32.1% lower error-rate and absolute error, respectively

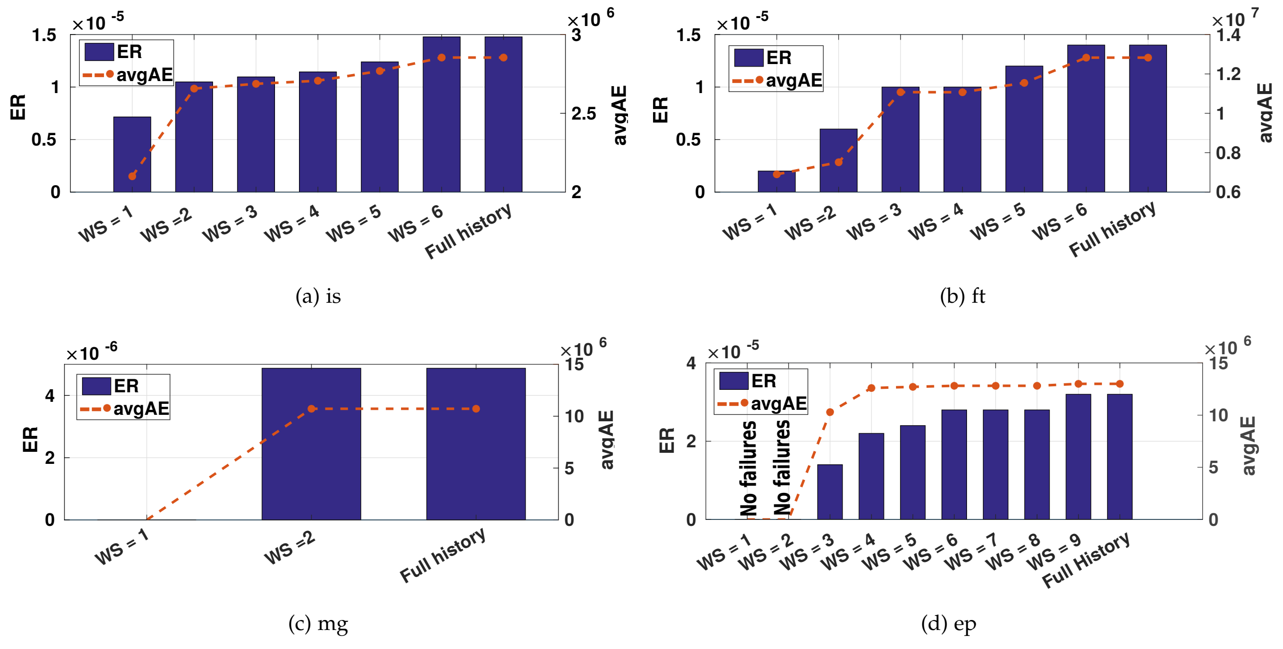


Fig. 5: Error-rate (ER) and average absolute error ($avgAE$) under 15% delay increase and various levels of window size (WS) across all benchmarks.

when compared with the actual ones estimated by taking into account an extended number of executed instructions in the pipeline.

Overall, these results imply that the order and type of instructions that have been executed in the previous cycles have a significant impact on the error-rate of the currently executed instruction. The number of the instructions within sequences that affects this dynamic timing behavior may vary depending on the input trace and pipeline depth. In our experiments where the maximum pipeline depth is 9 (see Table 1), a window of 9 instructions (8 preceding instructions and the current one) is necessary and sufficient to determine the error-rate behavior of the FPU under test.

5 Related Work

The raising importance of timing errors led researchers to the development of error detection and correction techniques [12]. Such design-centric techniques integrate additional circuitry (e.g., tunable replica circuit, special flip-flops) to monitor the timing critical paths and adopt recovery mechanisms in case of detected errors. However, such schemes may lead to large recovery overheads, especially if the activation probability of the error-prone paths is high. To overcome this, energy efficient frameworks have been proposed that are guided by accurate high-level timing models for predicting timing errors [3] and evaluating their effects on applications [4]. In this section, we review the state-of-the-art in error modeling and evaluation.

5.1 Data-agnostic Error Models

Several error evaluation frameworks are based on timing error models with fixed error probability under a given voltage/frequency setting [4]. Such models assume the same bit error probability across every instruction irrespective of the executed data. Nonetheless, such straightforward approaches are highly inaccurate since they overlook the dependence of errors on the input workload [3], [6].

5.2 Instruction-aware Error Models

To improve the accuracy of data-agnostic models, statistical instruction-aware models have been developed [6]. These models employ detailed DTA to extract instruction aware statistics considering the instruction type and data-dependent path activation. The extracted dynamic statistics are then used to determine the probabilities of an instruction to face a timing error at a specific bit location. Despite its statistical model, instruction-ware models also suffer from inaccuracies, though in a lesser degree than data-agnostic models, because error estimation relies on an aggregate error rate rather than the exact manifestation of an error. Further, they ignore the impact of previously executed instructions on error occurrence.

5.3 History-aware Error Models

To accurately predict bit-level timing errors, recent studies incorporate instruction execution history into their error models [3], [5]. These studies indicate that besides the in-flight instruction, the previous instruction affects the manifestation of timing errors. Although effective and more accurate than previous models, such models may also lead to misinterpretation of error behavior since they overlook an important timing property in pipelined designs. In fact, prior history-aware models neglect the influence of all the concurrently executed instructions that are in-flight in the pipeline and their role in error manifestation.

This work enhances the state-of-the-art by jointly considering the type and input operands of the executed instruction as well as the type and input operands of all the preceding instructions that are in-flight in the pipeline.

6 Conclusions

In this paper, we presented ExHero, a fully automated framework demonstrating that sequences within instructions up-to maximum pipeline depth significantly affect the error-rate of pipelined designs. By developing a microarchitecture-aware framework that considers both data dependencies and execution history of different depths, we simulate several traces extracted by various benchmarks. Our results indicate that existing work underestimates the occurrence of a timing error in pipelined units. This may lead to incorrect evaluation of errors on applications and inaccurate reliability assessments, guiding non-optimal design decisions. Even though we demonstrated the ExHero efficacy by applying it to a multi-cycle, pipelined FPU, the presented steps can be applied to any other pipelined unit.

Acknowledgments

This work is partially supported by the European Community Horizon 2020 programme under grant no. 688540 (UniServer) and grant no. 732631 (OPRECOMP).

References

[1] P. Gupta *et al*, "Underdesigned and opportunistic computing in presence of hardware variability," *TCAD*, 2013.
[2] I. Tsiokanos *et al.*, "Significance-driven data truncation for preventing timing failures," *IEEE TDMR*, vol. 19, no. 1, pp. 25–36, March 2019.
[3] X. Jiao *et al.*, "Clim: A cross-level workload-aware timing error prediction model for functional units," *Trans. on Comp.*, pp. 771–783, 2018.
[4] K. Parasyris *et al.*, "Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates," in *44th Annual IEEE/IFIP DSN*, 2014, pp. 622–629.
[5] 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.
[6] J. Constantin *et al.*, "Statistical fault injection for impact-evaluation of timing errors on application performance," in *DAC*, 2016.
[7] IEEE 754-2008 Standard for Floating-Point Arithmetic.
[8] G. Hoang *et al.*, "Exploring circuit timing-aware language and compilation," in *ASPLOS*. ACM, 2011, pp. 345–356.
[9] OpenRISC, "OpenRISC 1000 architecture manual".
[10] D. H. Bailey *et al.*, "The nas parallel benchmarks&mdash;summary and preliminary results," in *Supercomputing*, NY, USA, 1991, pp. 158–165.
[11] L. Mukhanov *et al.*, "Alea: Fine-grain energy profiling with basic block sampling," in *PACT*, 2015.
[12] K. Bowman *et al.*, "Circuit techniques for dynamic variation tolerance," in *DAC*, 2009, p. 4–7.

Ioannis Tsiokanos currently pursues his Ph.D. on design of energy efficient and variation tolerant pipelined micro-architectures at the School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, UK. He received the B.Sc. and M.Sc. degree from the Department of Electrical and Computer Engineering of University of Thessaly, Greece, in 2016. He is the recipient of the prestigious DATE Best Paper Award. His current research interests include low-power designs, fault tolerance circuits and hardware/software co-design with an emphasis on robustness.

Georgios Karakonstantis is an Associate Professor at the School of Electronics, Electrical Engineering and Computer Science of Queen's University Belfast, United Kingdom and senior member of the IEEE. He has published more than 85 papers in peer reviewed journals and conferences, and he is inventor of a US patent and author of two book chapters. He is the recipient of two HiPEAC paper awards, two best paper awards at DATE and of a prize at the Altera Innovate Design Contest. He received the MSc and Ph.D. degree in Electrical and Computer Engineering from Purdue University, West-Lafayette, USA. His research focuses on energy-efficient and error-resilient computing and storage architectures for embedded and high-performance applications.