UNIVERSITY OF CALIFORNIA, SAN DIEGO

Improved Timing Error Resilience of Microelectronic Computing Systems using Cross-layer Optimizations

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Xun Jiao

Committee in charge:

Professor Rajesh K. Gupta, Chair
Professor Gert Cauwenberghs
Professor Ryan Kastner
Professor Sorin Lerner
Professor Julian McAuley

2018

The dissertation of Xun Jiao is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2018

DEDICATION

I dedicate this dissertation to my loving family. Without their everlasting love and support this journey would not have been finished.

EPIGRAPH

The word “impossible” is not in my dictionary.
Napoleon Bonaparte.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

During the past five years as a Ph.D. student, I am so blessed to meet so many great people — advisors, colleagues, family, friends, and God, without whom this dissertation cannot be finished.

First and foremost, I was honored and fortunate to spend five years with my Ph.D. advisor Professor Rajesh K. Gupta, who brought me to beautiful San Diego and gave me the opportunity to conduct the research in an extremely free environment. I dedicate my deepest gratitude for his guidance, encouragement, thoughts, and supports that will remain in my heart forever. I would also like to thank my senior colleague, Dr. Abbas Rahimi, who has walked and guided me through the initial stages of research. I would also like to give special thanks to my brother and colleague, Dr. Yu Jiang, who has been guiding me both in school work and personal life since the day I went to middle school.

Besides my advisor and colleagues, I want to express my everlasting gratitude to my parents, Zhaoqun Jiao and Xiaoli Hou. They have been giving me unconditional love and believing in me in every choice that I have made in my life, which made this dissertation possible. I would not be the person I am today without their love and faith in me. To my wife, Yaxin Fu, who has sacrificed the stable life in China and courageously moved all the way to the U.S. to marry me, I thank you for your love, care, and support. Thanks for making me a better man. I owe everything to you.

I would also like to thank my committee members, Professor Gert Cauwenberghs, Professor Ryan Kastner, Professor Sorin Lerner, and Professor Julian McAuley, for their advice and comments in my dissertation. I would also like to give special thanks to my industry mentors, Professor Jose Pineda De Gyvez and Dr. Hamed Fatemi, who have given me tremendous advice and support over the past years. I would also like to thank Professor Lui Sha from UIUC, for his advice and support in my career. I am also grateful to work with Vincent Camus and Professor Christian Enz from EPFL.

I am blessed to be surrounded by so many good friends and colleagues during my Ph.D. time who made San Diego a truly unforgettable memory in my life. I thank MESL group friends Muhammad Adnan, Bharathan Balaji, Manish Gupta, Atieh Lotfi, Vahideh Akhlaghi, Zhou Fang, Jeng-Hau Lin, and many more. I thank my UCSD friends

Chunbin Lin, Jianguo Wang, Hao Zhuang, Jingwei Lu, Ilgweon Kang, Jiajia Li, Weiting Chan, Mulong Luo, Jiajun Lu, Xinchi Gu, and many more. I give my special thanks to my brothers and sisters in church, Chee Yap, Jingxue Lu, Xiaowen Zhang, Jian Zhang, Ye Tian, Zheng Fu, and many more.

Last but not least, I want to thank God for his unconditional love, which I know will continue to be with me throughout my whole life.

The material in this dissertation is based on the following publications.

Chapter 2 contains reprints of Xun Jiao, Abbas Rahimi, Balakrishnan Narayanaswamy, Hamed Fatemi, Jose Pineda de Gyvez, and Rajesh Gupta, "Supervised Learning Based Model for Predicting Variability-Induced Timing Errors", *Proc. IEEE International NEW Circuits And Systems (NEWCAS) conference*, 2015. Xun Jiao, Yu Jiang, Abbas Rahimi, and Rajesh Gupta, "SLoT: A Supervised Learning Model to Predict Dynamic Timing Errors of Functional Units", *Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE)*, 2017. Xun Jiao, Abbas Rahimi, Yu Jiang, Jianguo Wang, Hamed Fatemi, Jose Pineda de Gyvez, and Rajesh Gupta, "CLIM: A Cross-level Workload-aware Timing Error Prediction Model for Functional Units", *IEEE Transactions on Computers (TC)*, 2017. The dissertation author is the primary author of the papers.

Chapter 3 contains reprints of Xun Jiao, Vincent Camus, Mattia Cacciotti, Yu Jiang, Christian Enz, and Rajesh Gupta, "Combining Structural and Timing Error in Overclocked Inexact Speculative Adders", *Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE)*, 2017. The dissertation author is the primary author of the paper.

Chapter 4 contains reprints of Xun Jiao, Yu Jiang, Abbas Rahimi, and Rajesh Gupta, "WILD: A Workload-Based Learning Model to Predict Dynamic Delay of Functional Units", *Proc. IEEE International Conference on Computer Design (ICCD)*, 2016. The dissertation author is the primary author of the paper.

Chapter 5 contains reprints of Xun Jiao, Mulong Luo, Jeng-Hau Lin, and Rajesh Gupta, "An Assessment of Vulnerability of Hardware Neural Networks to Dynamic Voltage and Temperature Variations", *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017. The dissertation author is the primary author of the paper.

Chapter 6 contains reprints of Xun Jiao, Vahideh Akhlaghi, Yu Jiang, and Rajesh

Gupta, “Energy-Efficient Neural Networks using Approximate Computation Reuse”, *Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE)*, 2018. The dissertation author is the primary author of the paper.

My coauthors (Yu Jiang, Abbas Rahimi, Vahideh Akhlaghi, Mulong Luo, Jeng-Hau Lin, Jianguo Wang, Vincent Camus, Mattia Cacciotti, Christian Enz, Balakrishnan Narayanaswamy, Hamed Fatemi, Jose Pineda de Gyvez, and Rajesh Gupta) have all kindly approved the inclusion of the aforementioned publications in my dissertation.

VITA

1991	Born, Jinxian, Jiangxi, China
2013	B.Sc., Telecommunication Engineering and Management, joint program between Beijing University of Posts and Telecommunications and Queen Mary University of London
2016	M.Sc., Computer Science (Computer Engineering), University of California, San Diego
2018	Ph.D., Computer Science (Computer Engineering), University of California, San Diego

- Xun Jiao, Abbas Rahimi, Yu Jiang, Jianguo Wang, Hamed Fatemi, Jose Pineda de Gyvez, and Rajesh Gupta, “CLIM: A Cross-level Workload-aware Timing Error Prediction Model for Functional Units”, IEEE Transactions on Computers (TC), 2017.

- Xun Jiao, Vahideh Akhlaghi, Yu Jiang, and Rajesh Gupta, “Energy-Efficient Neural Networks using Approximate Computation Reuse”, in Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE), 2018.

- Xun Jiao, Mulong Luo, Jeng-Hau Lin, and Rajesh Gupta, “An Assessment of Vulnerability of Hardware Neural Networks to Dynamic Voltage and Temperature Variations”, in Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2017.

- Xun Jiao, Vincent Camus, Mattia Cacciotti, Yu Jiang, Christian Enz, and Rajesh Gupta, “Combining Structural and Timing Error in Overclocked Inexact Speculative Adders”, in Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE), 2017.

- Xun Jiao, Yu Jiang, Abbas Rahimi, and Rajesh Gupta, “SLoT: A Supervised Learning Model to Predict Dynamic Timing Errors of Functional Units”, in Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE), 2017.

- Xun Jiao, Yu Jiang, Abbas Rahimi, and Rajesh Gupta, “WILD: A Workload-Based Learning Model to Predict Dynamic Delay of Functional Units”, in Proc. IEEE International Conference on Computer Design (ICCD), 2016.

- Xun Jiao, Abbas Rahimi, Balakrishnan Narayanaswamy, Hamed Fatemi, Jose Pineda de Gyvez, and Rajesh Gupta, “Supervised Learning Based Model for Predicting Variability-Induced Timing Errors”, in Proc. IEEE International NEW Circuits And Systems (NEWCAS) conference, 2015.

ABSTRACT OF THE DISSERTATION

Improved Timing Error Resilience of Microelectronic Computing Systems using Cross-layer Optimizations

by

Xun Jiao

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California, San Diego, 2018

Professor Rajesh K. Gupta, Chair

Microelectronic scaling has entered into the nanoscale era with tremendous capacity and performance advantages that continue to drive new devices and systems from high-performance computing to ultra-low power Internet endpoints. This scaling, however, faces challenges due to serious effects of microelectronic variability that results in significant variation in individual device parameters. The most common manifestation of this variability is increased susceptibility to timing errors. Combating these errors usually results in increased guardbands in the circuit and architectural design, thus reducing the gains from process technology advances.

This dissertation focuses on methods to improve the timing error resilience of microelectronic computing systems by reducing the guardbands which also results in improved operational efficiency if microelectronic circuits. Timing errors can show at various abstraction levels — the circuit layer, the architecture layer, and the software layer. Accordingly, we have proposed error tolerance methods that correspond to the layer where such errors manifest. Considering the interdependence of overall system performance or application quality on the design choices made at various abstraction levels, an integrated view of the overall effects of error tolerance strategies is necessary to evaluate the effects of these approaches in the system or application layer. Cross-layer optimizations are thus important in addressing the effects of timing errors. At the circuit layer, we examine the root cause of "timing error" via analysis of dynamic path sensitization of the circuit. We use machine learning methods to build a prediction model for the timing errors based on the useful features extracted from computation history, circuit workload, and circuit switching. Results show high prediction accuracy and fast computing performance that make the model useful in early circuit reliability evaluation. Second, at the architecture layer, by characterizing delay of various instructions, we dynamically adjust the clock frequency, that reduces timing errors and improve the operational efficiency. Finally, at the software layer, by utilizing the inherent "error-tolerance" of emerging applications such as neural networks, we reduce design margins under the premise that the application quality is acceptable. Specifically, we have investigated the vulnerability of emerging neural networks to timing errors and deliver an approximate computing hardware for neural networks that achieves significant energy savings with negligible accuracy loss. Stemming from this dissertation, our future research concerns building emerging high-performance, low-power, reliable, and intelligent non-conventional computing systems.

Chapter 1

Introduction

The scaling of microelectronics into nanometer era with tremendous capacity and performance advantages continues to drive new devices and systems ranging from low-power Internet-of-Things (IoT) nodes to high-performance computing platforms. However, with continuous scaling, microelectronic systems are increasingly susceptible to timing errors that are caused by increased microelectronic variability, making them a notable threat to circuit reliability. To protect circuits from timing errors, designers typically use conservative timing margins acting as guardbands, computed from a multi-corner worst-case analysis at design time.

In this section, we first introduce the sources of microelectronic variability and its consequences on timing errors. Then, we discuss the state-of-art approaches to improve the timing error resilience of microelectronic computing systems and their associated challenges. Finally, we present our approach to improving timing error resilience by applying cross-layer optimizations.

1.1 Microelectronic Variability Causes Timing Errors

The microelectronic variability arises from three broad sources: 1) Spatial variations: process variations induced by random dopant fluctuations and sub-wavelength lithography can cause static variations in channel length (L) and threshold voltage (V_{th}). Process variations can be either die-to-die (D2D) or within-die (WID) variations [8]. D2D variations affect all the computing devices equally on one die and WID variations affect each device differently on one die [8]. 2) Temporal variations: Aging and wear-out effects such as negative bias temperature instability (NBTI), electromigration, time-dependent dielectric breakdown, gate oxide integrity, and thermal cycling can change the electrical characteristics of devices, causing temporal degradation in hardware reliability [57]. 3) Dynamic variations: This is often caused by environmental factors such as supply voltage droop and temperature fluctuation. Voltage droop is known as *dI/dt* problem that voltage varies instantaneously with the current fluctuation on the power delivery network (PDN). On the other hand, temperature fluctuation alters the circuit parameters such as carrier mobility, threshold voltage, etc. Technology scaling is known to further exacerbates the variations [3].

Table 1.1: Inverter delay for different PVT corners [45].

Process		**Voltage (V)**	**Temperature (°C)**	**Delay (ps)**
NMOS	PMOS			
Fast	Fast	1.0	-40	22.17
Fast	Fast	1.0	125	22.54
Fast	Fast	0.9	-40	27.21
Fast	Fast	0.9	125	26.16
Slow	Slow	1.0	-40	31.44
Slow	Slow	1.0	125	30.63
Slow	Slow	0.9	-40	42.78
Slow	Slow	0.9	125	38.89

The most immediate manifestations of manufacturing variability are in power and path delay variations. For instance, up to 13X variance in sleep power among 10 ARM Cortex M3 cores in the temperature range of 22-60°C is seen in [77]. In this dissertation, we mainly focus on the path delay variations.

The path delay variations can alter the path delay of a microelectronic element. For example, Table 1.1 presents the inverter delay at different PVT corners. We can observe up to 1.8X delay variation between the worst-case and best-case delay. We can also observe that all three variation sources can affect the inverter delay. Going up to the

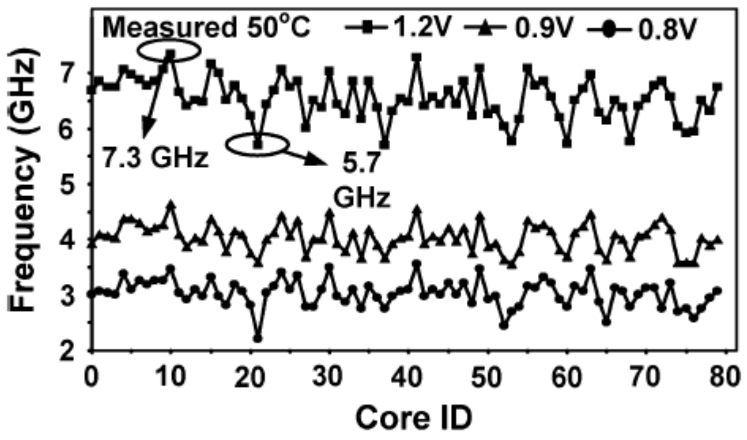


Figure 1.1: WID core-to-core maximum clock frequency variation for 80 cores on a single chip [30].

system level, Fig. 1.1 shows the Within-Die (WID) core-to-core maximum clock frequency (Fmax) variations among 80 cores in a 65nm processor. The measurement was done at a fixed temperature at 50°C with varying voltage at 1.2V, 0.9V and 0.8V respectively. Within the same operating condition at 1.2V, maximum frequency at 7.3GHz and minimum frequency at 5.7GHz could be observed, leading to a 28% Fmax variation. Once the voltage downscales to 0.8V, the maximum frequency is below 4GHz for every core, indicating the effects of voltage scaling on the path delay.

Delay uncertainty might prevent circuits from meeting its timing specification. This violation eventually causes invalid state being stored in sequential elements resulting in timing errors in circuits. Such timing errors will propagate up from corrupted state information to higher computing layers, leading to erroneous instructions or invalid computation results in software programs, that culminates in a degraded application quality or even system corruption [64]. A cross-layer approach not only examines the effects at various layers but also how their effects propagate to higher layers. Before presenting our approach, we review the state-of-art approaches that seek to improve timing error resilience as well as their challenges in the next section.

1.2 Challenges in Improving Timing Error Resilience

To protect microelectronic systems from timing errors, designers typically use conservative timing margins acting as guardbands, computed from a multi-corner worst-case analysis. However, this results in loss of operational efficiency measured as performance or energy efficiency of the circuits. Furthermore, increasing variability caused by process, voltage, temperature, and aging (PVTA) in advanced processes exacerbates this problem and increases the already-conservative guardbands. Attempting to improve the timing error resilience while reducing such conservative guardbands, various methods have been proposed. In this section, we will be discussing three typical approaches, **error prevention**, **error correction**, and **error acceptance** as well as their associated challenges: **inaccurate error modeling**, **high correction cost**, and **unacceptable application quality**. The observation of such challenges motivates our solutions, which are the main content of this dissertation.

1.2.1 Inaccurate Error Modeling

This challenge is associated with **error prevention** approach, whose main idea is to develop an accurate model that can predict the occurrences of timing errors, based on which the system can proactively adjust the guardbands to prevent the timing errors. The technique requires two mechanisms: 1) prediction of timing errors, which is usually accomplished by a prediction model; 2) prevention of timing errors, which is usually accomplished by dynamic adaptation of guardbands. For prediction, several works predict timing errors at instruction-level [64] [79] [67]. They claim that some instructions, which they call critical instructions, are more prone to timing errors. They rely on large-scale gate-level simulations and determine critical instructions based on their past behavior — if some instructions had timing errors in the past then they are likely to have timing errors in the future. Other works use machine learning to predict timing errors [63] [74]. A linear discriminant classifier predicts the timing error rate of functional units by obtaining variation information and then adjust timing guardbands accordingly [63]. However, it overlooks the effect of input operands by predicting errors purely based on PVTA parameters. B-Hive [74], predicts bit-level timing errors for voltage-scaled functional units but also claims that incorporating input operands does not improve the prediction accuracy significantly. For prevention, the system typically adjusts the guardbands based on the timing error prediction model. For example, [79] [67] extends the pipeline by one cycle in case of a positive timing error prediction.

Challenge: The main challenge of this approach is that the current timing error prediction model, or timing error behavior description, is inaccurate and pessimistic. The main reason for this problem is that there has been a missing consideration of the impact of input operands on timing error behaviors. For example, in [64] [79] [67], timing errors are predicted based on a worst-case scenario of path sensitization, but in reality, the instruction might not face timing errors because its input operands are not sensitizing the critical path. This results in pessimistic modeling. Timing error prediction models such as [63] [74] overlook the effects of input operands by predicting errors purely based on variation and circuit parameters. In order to solve this problem, we use the machine learning methods to establish a model that can predict timing error given arbitrary input operands, thus establishing a model that can accurately depict timing error behavior (**Chapter 2**). This model can be used by circuits designers to evaluate the guardbands configurations during design- or runtime.

1.2.2 High Correction Cost

This challenge is associated with **error correction** approach, whose main idea is to actively reduce and even remove the guardbands, and then use a correction mechanism to correct the timing errors in the system in case of their occurrences. This technique requires two mechanisms: 1) detection of timing errors, which is usually accomplished by a shadow latch [32] [9] [72]; 2) correction of timing errors which is usually accomplished by masking timing errors [9] [79] [23]. For detection, a shadow flip-flop was used in [32] to detect timing errors induced by speculated voltage scaling. Such shadow flip-flop approaches were also used in error-detection sequential circuit (EDS) [9] Error-detection sequential circuit (EDS) [9] introduces in-situ monitors in the critical path where it double samples the signal with a shadow latch. Once a mismatch occurs between flip-flop value and latch sampled value, an error signal will be activated to notify the incorrect operation. A less intrusive way would be using a tunable replica circuits (TRC) [72] adopted in each pipeline stage to monitor the timing error. For correction, Razor [32], for example, once error detection signal is activated, the pipeline is stalled for one cycle and replace the errant value with the correct value stored in Razor latch. A slack redistribution technique could help the timing speculation further to leverage the guardband reduction on some specific target [51].

Challenge: The main challenge of this intrusive error detection and correction approach is that it can lead to very high hardware overhead and performance penalty, thereby (partially) offsetting the benefits of reducing the guardbands. First, the hardware overhead can be large, e.g., 18% [61] and 21% [34] overheads in area, and 8% [17] power overhead. Second, the performance penalty can be large and will increase monotonically as the timing error rates increase. For example, when a timing error is detected during an instruction execution, the Intel resilient core [10] prevents the errors from propagating by flushing the pipeline and replaying the errant instruction multiple times (multiple-issue instruction replay). The recovery penalty is $3 * N$ cycles where N is the number of pipeline stages. In order to solve this challenge and reduce the cost of error correction, we propose a method to proactively prevent timing error at the instruction level, thus avoiding the cost of error correction (**Chapter 4**). We also design the approximate circuit and use the structure to efficiently correct the timing errors (**Chapter 3**).

1.2.3 Unacceptable Application Quality

This challenge is associated with **error acceptance** approach. The main idea is that by utilizing intrinsic error tolerance at the application level, the system is allowed to have occasional error occurrences as long as the output quality is still acceptable by users. This makes it possible to reduce the system guardbands without violating the users' requirements, leading to an emerging computing paradigm referred to as approximate computing. Some typical error-tolerant applications include multimedia applications, machine learning applications and emerging fields such as recognition, mining, and synthesis (RMS) applications [16, 21, 33, 36, 59].

The approximation can be realized by designing approximate hardware or relaxing the operating constraints. An accuracy-configurable integer adder offers two operating modes: exact and approximate [50]. During the exact operating mode, error detection and correction must be applied, while in the approximate mode the errors can be ignored and left out uncorrected. Similarly, floating-point units can dynamically switch between exact and approximate modes [66]. Its approximate mode ignores the timing errors on the less significant N bits of the fraction part where N is a reprogrammable memory-mapped register. Another technique is proposed for timing error acceptance to improve the quality-energy tradeoff for DCT/IDCT components [40]. Rely [16], is a language for expressing approximate computation that allows developers to define a *reliability specification*, which identifies the minimum required probability with which a program must produce an exact result. Chisel [59], further enhances the capabilities of Rely by providing combined *reliability* and/or *accuracy* specification.

Challenge: The main challenge of this approach is to guarantee the application quality after applying approximation techniques because the quality usually depends on the input data. For a fixed hardware-level approximation, it is possible that the application quality is acceptable under one input dataset but fails under another. To solve this problem, we developed a "controllable" and "reconfigurable" approximation technique that can change the approximation level to satisfy the quality requirements (**Chapter 6**). Furthermore, the application developers and researchers are usually unclear about error behavior of the underlying hardware and cannot accurately expose the approximation errors to applications. This leads to either conservative or pessimistic approximation level. Thus, we proposed a cross-layer assessment approach to expose the hardware errors to applications (**Chapter 5**). And finally, application developers usually need a hardware error model to help them estimate the application quality. For example, in Rely [16] and Chisel [59], the accuracy specification determines a maximum acceptable difference between the approximate and exact result values, while the reliability specification specifies the probability that a computation will produce an acceptably accurate result. Meeting the latter specification is a challenge for the automatic model generation since the model must provide reliable information about the possibility of an error occurrence under different workload conditions, i.e., accurate *error prediction*. To solve this problem, we propose a supervised learning-based model to predict timing errors for

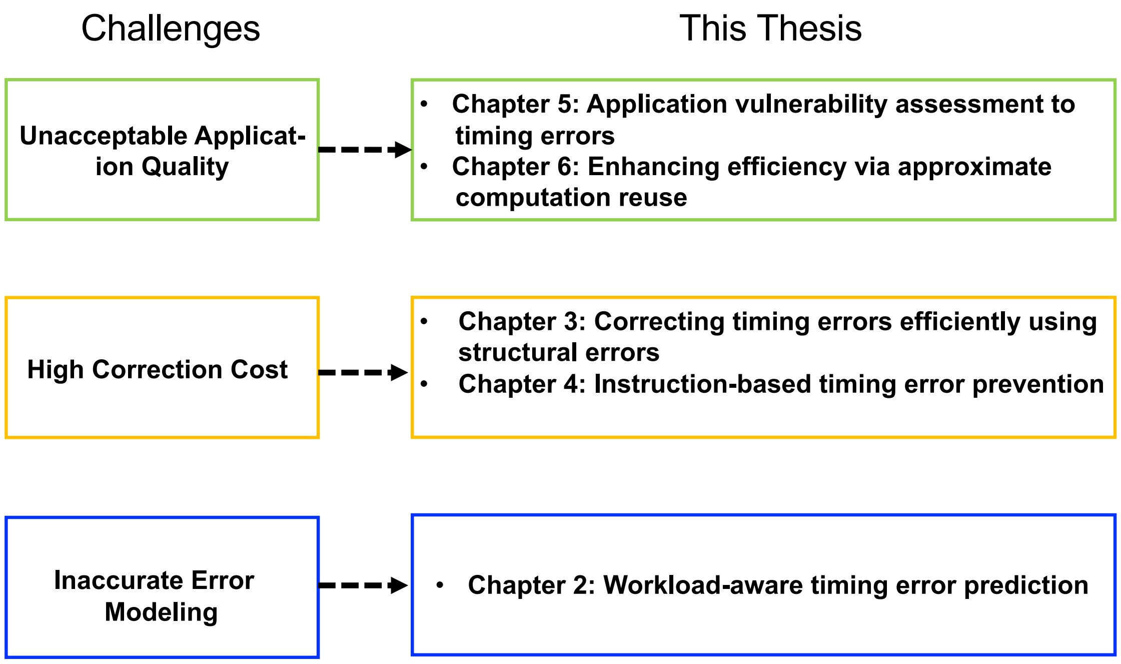

Challenges
This Thesis
Unacceptable Applicat-ion Quality
Chapter 5: Application vulnerability assessment to timing errors
Chapter 6: Enhancing efficiency via approximate computation reuse
High Correction Cost
Chapter 3: Correcting timing errors efficiently using structural errors
Chapter 4: Instruction-based timing error prevention
Inaccurate Error Modeling
Chapter 2: Workload-aware timing error prediction

Figure 1.2: Dissertation organization.

any given input workload (**Chapter 2**).

1.3 Our Approach and Dissertation Organization

To address the challenges in improving timing error resilience, this dissertation presents cross-layer optimization techniques and methodologies. The manifestation of timing errors at various abstraction levels — the circuit layer, the architecture layer, and the software layer has motivated our approaches that span various levels. For example, at the circuit layer, we examine the root cause of "timing error" via analysis of dynamic path sensitization to build an error prediction model (Chapter 2). At the architecture layer, we characterize delay of various instructions to dynamically adjust the clock frequency (Chapter 4). Unlike hardware where correct execution is predicted at each instruction or even gate level, software likely presents a greater resilience to numerical errors depending upon their significance to the end application. Operating under that hypothesis, we explore use of approximate hardware executions in Chapter 6.

Although these approaches focus on individual layers, it is important to consider the interdependence of system performance or application quality on design choices at each of these layers because the effects of timing errors can propagate to higher layers. Cross-layer optimizations are thus important in addressing the effects of timing errors. For example, at the circuit layer, we expose the predicted timing errors to software layer to estimate the quality of error-tolerant applications (Chapter 2). At the architecture layer, we evaluate the system performance improvement by performing instruction-level dynamic frequency scaling (Chapter 4). At the software layer, we dynamically reconfigure and control the approximation level of hardware to ensure acceptable application quality (Chapter 6). Fig. 1.2 illustrates the scope and organization of this thesis, in which each major challenge is addressed respectively.

To enable accurate error modeling, **Chapter 2** presents the approach to integrate the impact of input operands in the error modeling using machine learning methods. We have proposed using machine learning methods to establish the relationship between input workload and dynamic path sensitization that can be used in predicting timing errors. We have used gate-level simulation to locate the key features in the input workload that determine the dynamic path sensitization and have used it as input features in applying machine learning methods. We also utilized such model to estimate the circuit reliability under different datasets. Compared to the state-of-art simulation tools, the model can achieve significant execution acceleration.

To enable reduced correction cost, **Chapter 3** describes a methodology to efficiently correct the timing errors using the structural errors of inexact adders. By combining the structural errors and timing errors, we show that inexact adders are more resilient to overclocking than conventional adders and the combination of speculation and overclocking is a complementary combination to maximize circuit robustness. **Chapter 4** describes an approach to proactively prevent timing errors instead of correcting them. We use supervised learning model to predict dynamic delay of instructions. Using the model-directed dynamic frequency scaling (DFS), we are able to improve performance while preventing timing errors proactively.

To enable acceptable application quality, **Chapter 5** describes a cross-layer approach to assess the vulnerability of hardware neural networks to dynamic voltage and temperature variations. We extract the timing errors from hardware using gate-level simulations and examine their effects in the software using error injections. **Chapter 6** goes beyond timing errors but also look into the tradeoff between errors due to *approximation* and energy efficiency. To utilize the inherent error-tolerance of neural networks, we proposed a controllable and reconfigurable approximate computation reuse approach to improve the energy efficiency of hardware neural networks, with insignificant accuracy degradation.

Chapter 7 concludes the dissertation and gives future directions and ongoing work.

Chapter 2

Workload-aware Timing Error Prediction

In Chapter 1, we introduce the sources of timing errors and the typical approaches to combat timing errors and associated challenges. These approaches can be broadly classified into three categories: **error prevention**, **error correction** and **error acceptance**. Beginning with this chapter, we propose our approaches to combat these challenges. This chapter takes on the challenge of **inaccurate error modeling** associated with **error prevention** approaches. We describe a method to integrate the impact of input operands in the error modeling using machine learning methods and present evaluation results.

2.1 Introduction

To protect circuits from timing errors, designers typically use a conservative timing margin, that is, clocking circuits at speeds slower than what could be supported by the underlying circuits. The additional margin on path delay provides a guardband against varying delays caused by variability in manufactured chips. However, it leads to operational inefficiency. This loss of efficiency is already approaching limits that it compromises any performance gains enabled by migration to new process nodes. Thus, it is critical to reduce this margin. Existing **error prevention** approaches reduce such conservative margins by predicting the timing errors in advance and adjusting the clock period adaptively. Several prior works have devised methods to predict the timing errors via instruction-level models [25, 67, 79]. At their core, these methods rely upon identification of *critical instructions*, that is, instructions which are likely to fail in a reduced margin operation, by estimating maximum path delays and using this information to guide runtime adaptation of the clock speed. As an alternative, Rahimi *et al.* proposed a timing error rate prediction model for functional units based on hardware variation information [63].

A common limitation of these approaches is assumption of a worst-case scenario for path sensitization that overlooks the effect of input operands, leading to pessimistic modeling. In fact, during execution, the sensitized paths can vary with different input workload [54]. An instruction or functional unit may exhibit a different delay under different input operands, resulting in different timing error rates (TERs) [79]. Unfortunately, due to the extremely large input space, incorporating input operands into timing error modeling becomes very difficult, if not impossible. Our attempt to capture the path sensitization behavior under arbitrary input workload and represent it in the error modeling faces the following challenges:

Challenge 1: Dynamic path sensitization is affected by various parameters, such as operand values, instruction types, and computation history. These become more complex as we move up the level of abstraction in an attempt to identify useful "features" from the input parameter space for effective timing error models.

Challenge 2: There might be numerous failed circuit paths in the design, and the timing errors might be caused by any one of them. It is unclear how these features will determine which paths to sensitize and, therefore, how they induce timing violations. More importantly, a detailed path analysis may not be possible since we often do not have detailed circuit diagrams available. Further, under cryptographic assumptions Probably Approximately Correct (PAC) learning of Boolean circuits is a difficult problem, even under uniform distribution over the inputs [53].

Proposed Approach: To overcome these challenges and provide an accurate error prediction model, we propose **`CLIM`**, a cross-level supervised learning-based model to predict timing errors for a given input workload, clock period and functional unit (**FU**) type. The key idea of **`CLIM`** is to establish a prediction model that can best explore the relationship from input features to sensitized circuit paths by learning the existing patterns and their corresponding output classes. For a given input data and clock period, **CLIM** predicts output data to be one of two predefined classes–{*timing correct, timing erroneous*} at two levels: bit-level and value-level.

First, we measure the timing errors at each cycle to generate output class labels using gate-level simulation (**GLS**) of post-layout designs in TSMC 45nm technology. We also perform a trial-and-error process to extract useful features from input data. Second, we apply supervised learning methods to construct and train **CLIM** for four functional units: (INT_ADD, FP_ADD, INT_MUL, FP_MUL) at two levels with extracted input features and output class labels. Third, we evaluate the prediction accuracy of **CLIM** by comparing its predicted results with simulation-based ground truth.

Contribution: We make following contributions:

1. We present a detailed bit-level and value-level timing error behavior characterization using standard ASIC flow and gate-level simulation. We show that different input operands lead to different error behavior. We then extract useful "features" from input operands to train the model. We apply *random forest tree* on the training data to develop **CLIM**, an input workload-aware learning model to predict bit-level and value-level timing errors.

2. We evaluate the performance of **CLIM** at two granularities under various datasets and circuit parameters such as circuit structures and clock periods. **CLIM** demonstrates average prediction accuracy of 95% and 97% at the value-level and at the bit-level respectively, exceeding baseline models.

3. We quantify the degree of error tolerance of arithmetic operations in error-tolerant applications by deriving their bit-level reliability specifications. By comparing such bit-level reliability specifications with **CLIM**-predicted bit-level reliability, we predict output quality of such applications into two classes: {*acceptable*, *non-acceptable*}. This prediction is on average 97% consistent with simulation-based classification. We also utilize **CLIM** to analyze the value-level reliability of functional units, which exhibits deviation within 2.8% on average of detailed gate-level simulation ground truth. We demonstrate the efficiency of **CLIM** by comparing it

to the execution speed of a gate-level simulation.

2.2 Problem Formulation

We represent the timing errors of a circuit as a function of circuit parameters and the input workload. More specifically, we abstract a circuit as a mapping from an input space $\mathcal{I}$ consisting of p circuit parameters (e.g., the circuit structure, and clock speed) and m input bits, to create an input I. Suppose the function implemented by an ideal circuit, without timing errors is ϕ_i and the function of the real physical circuit is ϕ_r, which includes the effect of timing errors. The output value in error is $\psi(I) = \phi_i(I) \oplus \phi_r(I)$, where $\oplus$ is the XOR operator. Our goal is to learn (an approximation of) ψ given a range of inputs and circuit parameters.

However, in general we do not know the structure of the ψ function – it is not even clear *a priori* if the structure of ψ is similar to the structure of the circuit function ϕ. We thus propose evaluating a sequence of *non-parametric* classification methods to classify the inputs and thereby map them into different outputs as shown in Section 2.3.2.

We define $x[t]$ as the input operands vector, $y[t]$ as the gate-level simulation output and $y\_gold[t]$ as the pure-RTL simulation output value, all at cycle t. Note that $y[t]$ may contain timing errors while $y\_gold[t]$ is always clean. We denote $y_i[t]$ and $y\_gold_i[t]$ as i^{th} bit position of the gate-level simulation and RTL simulation output respectively, where $i = 1, 2, ...N$ and N is the number of output bits. We define the two classes for output value: C_e representing *timing erroneous* and C_c representing *timing correct*, and we define the class of $y[t]$ and $y_i[t]$ as $C[t]$ and $C_i[t]$ respectively. At cycle t, if $y[t]$ = $y\_gold[t]$, then $C[t]$ is marked as class C_c. If mismatched, then $C[t]$ is marked as class C_e. The same principle applies to bit-level to determine the bit-level timing class. Our goal is to predict the output class $C[t]$ ($C_i[t]$) at cycle t as a function of input workload, clock period and functional unit type, denoted as follows:

$$C[t] = f(t_{clk}, FU_{type}, x[t], x[t-1], x[t-2], ..., x[1]) \tag{2.1}$$

$$C_i[t] = f(t_{clk}, FU_{type}, x[t], x[t-1], x[t-2], ..., x[1]) \tag{2.2}$$

where t_{clk} is the clock period, FU_{type} is FU type, $x[t]$, $x[t-1]$, ... $x[1]$ are the input workloads at cycle $t, t-1, ...1$. The reason for putting the entire input stimuli history is that we do not know whether previous input workload would set a circuit state and thereby have an effect on the timing error behavior of current cycle t. In instruction-level models [79] [25] [67], the effects of input workload are not considered. Therefore, we later investigate the features from input data which affect the output timing error behaviors, as shown in Section 2.3.2. This becomes a binary classification problem: for a given input data and circuit parameters at cycle $t, t-1, ...1$, **CLIM** predicts the output $C[t]$ ($C_i[t]$) to be one of two classes: C_c or C_e.

2.3 Cross-level Timing Error Prediction Model (CLIM)

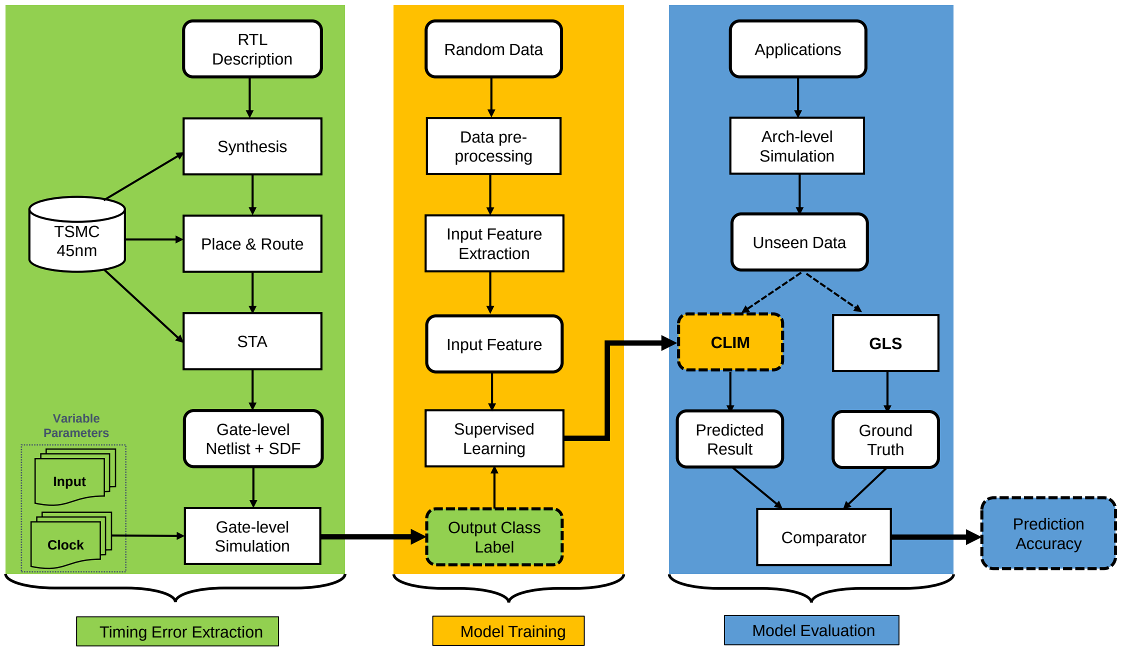


Figure 2.1: **CLIM** model: a) Timing Error Extraction to examine the timing errors and to generate output timing class labels; b) Model Training to apply random forest classification (RFC) to construct **CLIM**; c) Model Evaluation to evaluate **CLIM** prediction accuracy.

Model Overview: It is composed of three phases as shown in Fig. 2.1: *Timing Error Extraction, Model Training* and *Model Evaluation*. a) The *Timing Error Extraction* phase implements the standard ASIC flow and uses gate-level simulation to generate timing class: C_c if matched, otherwise C_e. b) In the *Model Training* phase, we preprocess the training data and extract useful features from them, which will then be incorporated into modeling. We then apply RFC method to construct the model with the input features and output timing class labels generated from last phase. c) In the *Model Evaluation* phase, we use **`CLIM`** to predict the timing class of the functional unit output value and then compare the predicted class with gate-level simulation ground truth to compute prediction accuracy. More details about the three phases are illustrated as follows.

2.3.1 Timing Error Extraction

We use both 32-bit integer and single-precision floating point units (FPUs) as our experimental platforms: INT_ADD, INT_MUL, FP_ADD, FP_MUL, implemented in VHDL. FPUs are fully compatible with the IEEE-754 standard and can provide more complex structures compared to their integer counterparts. We change the data types and circuit structures to better evaluate the robustness of our model. We extract the value-level and bit-level timing errors through *Timing Error Extraction* module as illustrated in Fig. 2.1, which is divided into several steps.

We use FloPoCo [28] to generate the synthesizable VHDL codes of combinational circuits. We put wrappers at input and output ports to have better timing notations. We then use *Synopsys Design Compiler* to synthesize the VHDL codes and use *Synopsys IC Compiler* to generate post place-and-route netlist in TSMC 45nm technology. Next, we use *Synopsys PrimeTime* to perform static timing analysis, generating a Standard Delay Format (SDF) file. Then, we vary clock periods to simulate the netlist with *Mentor Graphics Modelsim* to do SDF back-annotation gate-level simulation to generate output data $y[t]$. The input stimuli of simulation x[t], comes from two sources: Python-written random data generator and the application input data profiled using *Multi2Sim* [75], a cycle-accurate CPU-GPU heterogeneous architectural simulator. At cycle t, the input stimuli vector $x[t]$ is applied to gate-level simulation to generate output $y[t]$ ($y_i[t]$) and compare with pure-RTL simulation result $y\_gold[t]$ ($y\_gold_i[t]$) to derive timing errors as shown in Section 2.2.

2.3.2 Model Training

Data Preprocessing

To collect the training input data, we generate the random input data as stimuli for simulations. For a 32-bit bit vector, we randomly set each bit independently to produce the training data. But note that for test input data, which might come from application profiling, its format could be in decimal format. We need to preprocess such input data to convert it into the correct format, for example, 0.5 should be converted to 00111111000000000000000000000000 if the functional unit is of IEEE-754 single-precision format. The reason for doing this is that the functional unit accepts 32-bit input vectors and each bit value could affect the dynamic path sensitization, hence the final timing class. The decimal value cannot precisely represent the impact of each bit location. Therefore, in our model training, we use each bit value to compose input features rather than the decimal value alone.

As a matter of methodology, we remove the repetitive pair of $\{x[t-1], x[t]\}$ in the dataset because the same pair of current and preceding input leads to same timing class (as shown next). We also exclude an ambiguous case where the preceding input $x[t-1]$ is the same with current input $x[t]$, because even if a timing violation occurs at cycle t, the output could still appear to be correct. We note that these two situations are unlikely especially with randomly chosen 32-bit operands.

Feature Extraction

From the processed training input data, we need to find out the useful input features that determine the output timing class. Empirically, the current cycle input workload $x[t]$ directly affects the dynamic path sensitization at cycle t, hence the final output timing class. However, it is not clear whether the preceding input has impact on the current cycle path sensitization and timing behavior. To explore the effect of history input workload, we use a trial-and-error process, which iteratively varies the preceding input while fixing the current input workload. We set the experiment as follows:

- Case 1: We fix the current input $x[t]$ and randomly vary the preceding cycle input $x[t-1]$, where we set cycle $t = 10, 30, 50, 70, ...$ We use this to evaluate the effects of immediately preceding input.

- Case 2: We fix both the current input $x[t]$ and the immediately preceding input $x[t-1]$, while randomly varying the preceding input of immediately preceding input $x[t-2]$, where we set t as above. We use this to evaluate the effects of the deeper history.

We use 100K cycles for simulation and use different clock periods. At value-level, in Case 1, we found the timing class $C[t]$ varies irregularly. More specifically, by comparing every two examined neighboring outputs, e.g., $c[30]$ and $c[50]$, we found 44% of neighboring pairs exhibit different timing classes. In Case 2, we found all output timing classes $C[t]$ exhibit exactly the same behaviors, i.e., all C_c or C_e. At the bit-level, we examine the hamming distance between every two examined neighboring timing class outputs, where each output is a 32-bit vector of $C_i[t]$, where $i = 0, 1, ...31$. The hamming distance between two vectors is defined as the number of mismatched bit positions, e.g., 10001 and 10000 has a hamming distance of 1. In case 1, we can see most pairs have a positive hamming distance, indicating that the resulting output timing classes are different. In case 2, the neighboring hamming distance is always 0, which means the bit-level timing class output is exactly the same for every bit position.

This key observation shows that only the preceding and current cycle input vectors $x[t-1]$, $x[t]$ are accountable for timing errors in the current cycle t. For a combinational logic placed between sequential elements, it is natural that the preceding input workload sets a state for the circuit, and then the current input toggles nets based on the current state. Thus, the path sensitization depends on both the current circuit state and current circuit input. However, since most previous works do not consider input operands as features for timing error modeling [63] [79] and some work points out that including a deeper history would increase the accuracy [74], we investigate the effects of input operands and history. This key observation locates the source factors that determine the dynamic path sensitization and motivates an workload history-aware modeling approach.

On the other hand, we explore circuit parameters that can reflect or partially reflect the timing violation behaviors. One parameter that can be used is timing class output. At the value-level, the circuit output timing errors occur if and only if at least one output bit location faces a timing violation. The timing violation of a particular output bit occurs only when there is at least one sensitized circuit path ending at that bit facing violation. A sensitized path would have all of its nodes toggled [11]. Hence, the end point, i.e., the output bit, should also be toggled. Thus, we also take the final output value into our modeling as part of the input feature. By composing aforementioned features, our final input features are $\{x[t-1], x[t], y\_gold[t-1], y\_gold[t]\}$. At bit-level, the same principle applies and leads to the final input features are $\{x[t-1], x[t], y\_gold_i[t-1], y\_gold_i[t]\}$.

Training Process

Since the model has two levels, we also need to train the model at two-levels respectively. At the value-level, we set $\{x[t-1], x[t], y\_gold[t-1], y\_gold[t]\}$ as the input feature and C_t as output class labels; At the bit-level, we set $\{x[t-1], x[t], y\_gold_i[t-1], y\_gold_i[t]\}$ as the input feature and $C_i[t]$ as output class labels. Therefore, for a given circuit with *K-bit* output, a set of *K+1* binary classifiers is developed. *Model Training* stage in Fig. 2.1 illustrates the process of constructing the model. First, we apply 500K random data points as training input data. We extract the input feature through *Feature Extraction* module and output labels through *Timing Error Extraction* stage. We then apply and evaluate several supervised learning methods on these training data to train **CLIM**.

While certain positive learnability results exist for specific classes of circuits [58], they do not cover the circuits we consider here. In contrast to these aforementioned methods (which essentially learn a model of the circuit under consideration), we focus on learning when a circuit does not work as desired, i.e., the circuit contains timing errors. Capturing the timing errors will require learning a binary classifier. Thus, we evaluate four supervised learning methods for their increased sophistication and practical use: k-nearest neighbor (**k-NN**), support vector machine (**SVM**), logistic regression (**LR**) and random forest tree (**RFC**) classifiers [6]. These learning methods are very popular in classifying various kinds of tasks and we want to see whether they fit for the timing error classification tasks. By comparing them we can also conclude why we choose a particular method. The machine learning module is provided by Scikit learning module [62] in Python, and we use the default configurations for the classifiers.

We evaluate **k-NN** because it provides useful theoretical properties [26] and has limited parameters to train. Given an input vector x, k-NN classifier predicts a timing error if the majority of the k nearest neighbors of x in the dataset $\mathcal{D}$ has timing errors. However, in our case, K-NN finds its nearest neighbors based on hamming distance, which actually overlooks the situations wherein different bit positions would have disparity of significance on path sensitization. Thus, we would expect the k-NN model perform badly. In addition to this, k-NN classifiers typically have sub-par generalization performance (i.e., performance on new data) when available labeled data is limited, which could potentially lead to appropriate feature normalization and scaling issues.

To address these problems, we evaluate **LR** and **SVM** because they can learn weights w on each bit position, which considers the disparity of significance of different bit positions.

In **LR**, we learn weights to compare the logic functions that perform well on the training data $\mathcal{D}$. In particular, for an input x we predict 1, or the timing error, if the ratio of $\frac{F(x)}{1-F(x)} >= 1$ where $F(x)$ is given by

$$F(x) = \frac{1}{1+e^{-w \cdot x}} \tag{2.3}$$

In **SVM**, given labels y_i for the N training data points x_i, SVM learn w based on the following large margin optimization problem:

$$\begin{aligned} \min_{w,\eta,b} \quad & \frac{1}{2}||w||^2 + C\sum_i \eta_i & (2.4) \\ \text{s.t.} \quad & y_i(w \cdot x_i + b) \geq 1 - \eta_i & (2.5) \end{aligned}$$

where w is weights and b is offset, which jointly determine a separating hyperplane. Essentially, weights are learned that maximize the margin (η_i) by which examples are classified correctly. Typically, input examples are mapped to a higher dimensional kernel space (we use the popular Gaussian Radial Basis Function (RBF) kernel in our experiments).

LR and **SVM** can learn the disparity of significance of different bit positions.

However, one potential limitation is that, they put a fixed weight on each bit position. It is unclear whether each bit position contributes linearly to the final timing error, and the contribution of each bit position might be changed along with the change of other bit values. Think about an ”AND” gate – if one input is zero, then the final result will always remain the same regardless of the value of the other input.

To address this problem, we propose to use **RFC**. **RFC** is an ensemble-learning method that constructs multiple decision trees at training time and uses their averaging to improve accuracy and control overfitting. Decision trees are a non-parametric supervised learning method that aims to establish a tree-like model by learning decision rules from training data. As a white box model, the decision rules are based on Boolean logic; thus it is easy to understand and interpret. However, decision trees can easily create overly complex trees and become very deep by learning many irregular patterns with a large variance. This will lead to the notorious overfitting problem, which cannot generalize the data well. RFC alleviates this problem by constructing multiple decision trees. In our scenario, **RFC** can predict the timing errors based on the decision rules it learned from the data patterns. This method emphasizes the disparity of different bit positions and also considers the interaction between the input bits. Although it may lose the opportunity to learn some ”irregular” patterns, overall it reduces overfitting and boosts performance.

$$S = \begin{bmatrix} f_{1A} & f_{1B} & f_{1C} & C[1] \\ f_{2A} & f_{2B} & f_{2C} & C[2] \\ \vdots & \vdots & \vdots & \vdots \\ f_{dA} & f_{dB} & f_{dC} & C[d] \end{bmatrix} \tag{2.6}$$

$$S_1 = \begin{bmatrix} f_{5A} & f_{5B} & f_{5C} & C[5] \\ f_{10A} & f_{10B} & f_{10C} & C[10] \\ \vdots & \vdots & \vdots & \vdots \\ f_{100A} & f_{100B} & f_{100C} & C[100] \end{bmatrix} \tag{2.7}$$

$$S_2 = \begin{bmatrix} f_{15A} & f_{15B} & f_{15C} & C[15] \\ f_{20A} & f_{20B} & f_{20C} & C[20] \\ \vdots & \vdots & \vdots & \vdots \\ f_{200A} & f_{200B} & f_{200C} & C[200] \end{bmatrix} \tag{2.8}$$

$$S_M = \begin{bmatrix} f_{3A} & f_{3B} & f_{3C} & C[3] \\ f_{40A} & f_{40B} & f_{40C} & C[40] \\ \vdots & \vdots & \vdots & \vdots \\ f_{400A} & f_{400B} & f_{400C} & C[400] \end{bmatrix} \tag{2.9}$$

We use equation 2.6 to equation 2.9 to illustrate the process of creating a random forest classifier. Equation 2.6 is the original training dataset, where we have d input samples, each of which is composed of 3 features, that lead to a particular class C. We split the entire training data into M independent sub-sample datasets, S_1, S_2,...,S_M. Then, we use M decision tree classifiers to fit all sub-sample datasets. Hence, M decision trees are developed. Finally, each decision tree predicts the class and we use the majority vote of all M votes as the final prediction result. In the model construction, we need to tune several important parameters such as number of trees in the forest, the depth of trees, and the number of features to test at each node. Increasing these parameters could possibly improve the prediction accuracy but incurs more computational overhead. Thus, we use the default settings recommended by Scikit learning module [62].

Table.2.1 presents the prediction accuracy, training and testing time of four methods using 100K random training data and 10K random test data under a computer configuration of 2-core Intel(R) Xeon(R) CPU E5504@2.00GHz and 50GB memory. More specifically according to the Table, LR is fastest because of its relatively easy computation process, which assigns weight to each bit position. However, it achieves the lowest accuracy because the contribution of each bit position is not identical to the final output. Although SVM achieves good accuracy, compared with the other three classifiers its long running time impedes its use. Comparing to the other three baseline classifiers, we can emphasize why RFC is the choice because it can interpret the difference at each bit position (compared with KNN) as well as interactions among bits (compare with SVM and LR). Finally, we choose RFC due to its high accuracy, fast computing time and superior interpretability. Note that the training process is a one-shot activity, so the testing time is more important for model usage.

Table 2.1: Prediction accuracy, training time, and testing time of four learning methods.

method	Accuracy	Training Time	Testing Time
LR	85%	42.8s	0.21s
KNN	87%	4224s	849s
SVM	92%	18600s	1968s
RFC	93%	94.74s	0.26s

2.3.3 Model Evaluation

We evaluate the model performance by comparing with gate-level simulation under various functional units, clock periods, and datasets.

Evaluation Metrics

Prediction Accuracy: Prediction accuracy is an intuitive measurement of how accurate the predictions are. We define mean bit-level prediction accuracy (MBPA) and mean value-level prediction accuracy (MVPA) as follows:

$$MBPA[clk] = \frac{\sum\limits_{bit\ i} \left(\frac{\sum\limits_{cycle\ t} |C^{(pred)}_{clk,i,t} == C^{(real)}_{clk,i,t}|}{\#cycles} \right)}{\#bit\_positions} \tag{2.10}$$

$$MVPA[clk] = \frac{\sum\limits_{cycle\ t} |C^{(pred)}_{clk,t} == C^{(real)}_{clk,t}|}{\#cycles} \tag{2.11}$$

where $C^{(pred)}_{clk,i,t}$ and $C^{(real)}_{clk,i,t}$ are the predicted and real timing classes (1 for timing-erroneous and 0 for timing-correct) for bit position i at a given clock period clk and cycle t. $C^{(pred)}_{clk,t}$and $C^{(real)}_{clk,t}$ are the predicted and real timing classes (1 for timing-erroneous and

0 for timing-correct) for the entire value at a given clock period clk and cycle t. Its best value is 0 and worst value is 1.

Comparison Methods

We compare **`CLIM`** against following baseline methods, which can help us evaluate the true performance of our model:

- **rand [69]:** This model is adopted from [69]. We call it **rand** model because it predicts the timing class with random guessing without considering the effects of input operands.
- **fixed [25, 67, 79]:** This model is adopted from [25] [67] [79]. We call it **fixed** model because it always predicts a fixed timing class based on the pre-characterized information, i.e., it predicts C_c (C_e) when the clock period does (not) meet the measured maximum instruction-level timing delay. At the bit-level, it always predicts the particular timing class that has more instances in training data. For example, if in the training data more data are timing correct than erroneous, then this model always predicts *timing correct*. Note that this model can lead to high prediction accuracy if the dataset is heavily biased, e.g., 99% of the output data is C_c. Then its prediction accuracy is 99% by always predicting *timing correct*.

2.4 Experimental Results

In this section, we first describe our experimental setup. Second, we characterize the hardware timing behavior. Third, we evaluate **`CLIM`** performance at both the bit-level and the value-level. Lastly, we examine **`CLIM`** efficiency.

2.4.1 Experimental Setup

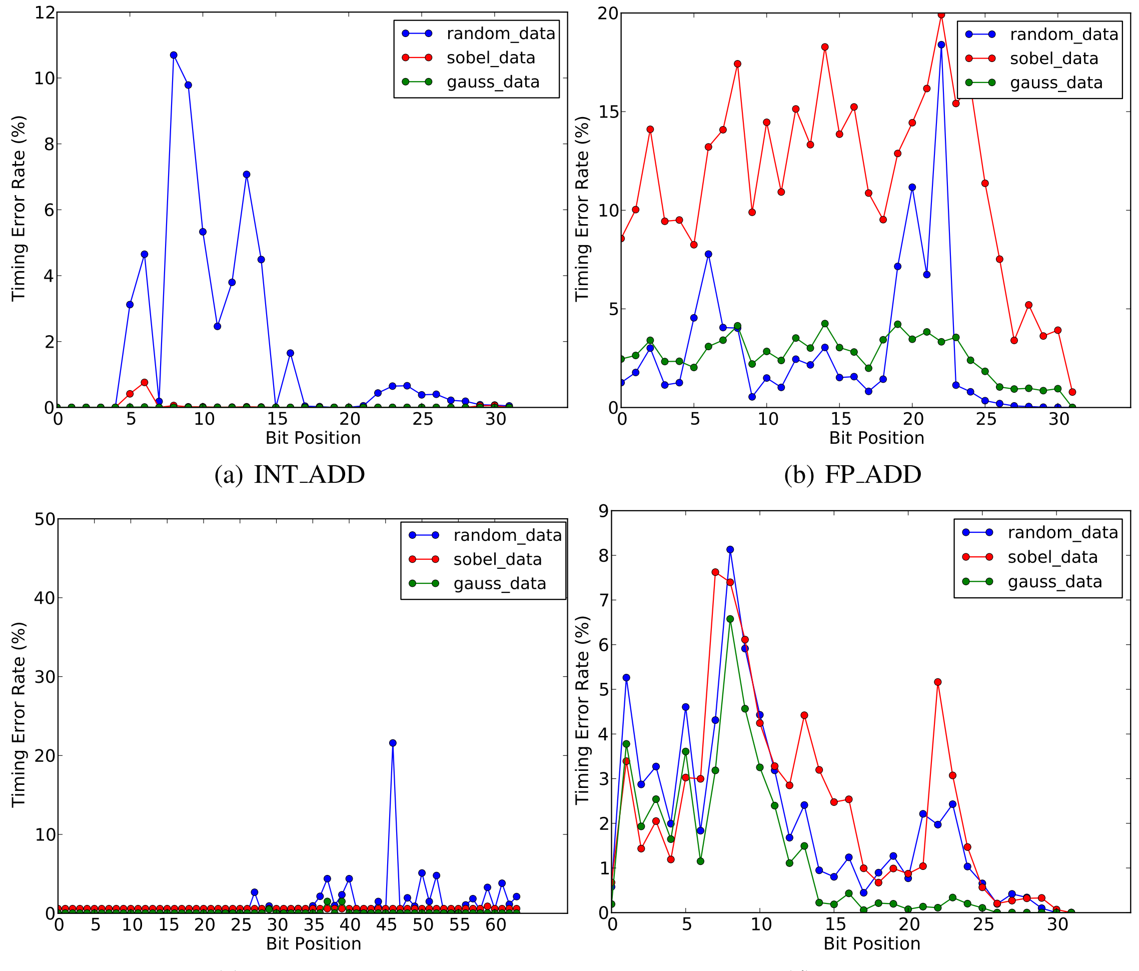
We characterize timing error rate (TER) as the ratio of $\#erroneous\_cycles$ and $\#total\_cycles$. To explore a range of timing errors, we set the experimental clock period for each functional unit (FU) so that their value-level timing error rate (TER) reaches 10%, 20%, and 30% under random data approximately. Through the rest of this chapter, we refer to the *clock period reduction* (CPR) as a 3-tuple, $\{$*CPR1, CPR2, CPR3*$\}$m that leads to three corresponding timing error rates. Note that such CPR pair values are different for each functional unit.

Using a trial-and-error gate-level simulation to compute CPR is time-consuming since we need to iterate clock periods until the target timing error rates is met. This process could take exponentially large number of gate-level simulations as the number of functional units grows. Therefore, we derive such clock periods through the characterization of dynamic delays of all simulation cycles. We know a timing error occurs if the clock period is less than the dynamic delay for a given cycle period. Therefore we only need to sort all the dynamic delays and find the top 10%, 20%, and 30% dynamic delay as the $\{$*CPR1, CPR2, CPR3*$\}$. To do this, we first extract all the dynamic delays; we parse the value change dump (VCD) file that is generated by gate-level simulation at a relatively slow clock period to make sure there are no timing violations. The VCD file records the toggled endpoints of each circuit path at each cycle. Second, for each clock cycle, we use the last toggle event time of the input pin of all sequential elements (flip-flop, registers, etc.) to subtract the last positive clock edge arrival time to get the maximum delay at that cycle. For example, at cycle N the positive clock edge occurs at time t, and the very last toggled event at the data input pin of all sequential elements occurs at time t', then the dynamic delay at this cycle is $t' - t$. Third, we sort all the dynamic delays in a descending order, and locate the delay at the top 10%, 20%, and 30% position.

We use three datasets to evaluate and utilize the model: random data, Sobel filter and Gaussian filter. The two image processing applications are adopted from AMD APP SDK [1]. The openCL code of these applications are simulated by Multi2Sim to profile input data. The images are adopted from Caltech-UCSD Birds 200 vision dataset [78].

2.4.2 Hardware Characterization

As mentioned earlier, timing errors are caused by the violations of circuit timing specification where the sensitized path delay is larger than the clock period. Thus, the key to modeling timing errors is to model the path sensitization behavior. We demonstrate the effect of input operands on timing errors using gate-level simulations and



(a) INT_ADD (b) FP_ADD

(c) INT_MUL (d) FP_MUL

Figure 2.2: Bit-level timing error rate (%) under different input datasets.

characterize the timing error rates of different functional units. We depict bit-level timing errors at *CPR3* under different input datasets as illustrated in Fig. 2.2, where we observe several important facts.

First, under the same input dataset, different bit positions exhibit different timing error rates. This is because different output bits lie on different paths with different delays. Second, under a different input dataset, the same bit positions exhibit different timing error rates. For example, in Fig. 2.2(c), some bit positions under the *sobel* and *gauss* datasets exhibit a nearly zero timing error rate while those same bits under random dataset exhibit up to a 20% timing error rate. This is because different input data exercise different paths towards an output bit, thus causing different delays. Third, some bit positions might exhibit similar timing error rates under different datasets. For example, in Fig. 2.2(d), some bit positions exhibit a similar timing error rate under three datasets. From this observation, we infer that the path sensitization behavior in FP_MUL is relatively similar under these three input datasets, thus resulting in similar timing error rates. These observations of the effects of input workload on timing error behavior has motivated us to develop an workload-aware model.

2.4.3 Bit-level `CLIM`

Table 2.2: Bit-level CLIM on INT_ADD for timing error prediction.

datasets	CPR1			CPR2			CPR3		
	`CLIM`	**fixed**	**rand**	**`CLIM`**	**fixed**	**rand**	**`CLIM`**	**fixed**	**rand**
random	96.7%	92.8%	50.0%	96.1%	92.3%	49.9%	95.6%	88.7%	50.0%
sobel	99.8%	99.9%	49.9%	99.9%	99.9%	49.9%	99.9%	99.8%	50.0%
gauss	99.9%	99.9%	49.9%	99.9%	99.9%	49.9%	99.9%	99.9%	49.9%

Table 2.3: Bit-level CLIM on FP_ADD for timing error prediction.

datasets	CPR1			CPR2			CPR3		
	`CLIM`	**fixed**	**rand**	**`CLIM`**	**fixed**	**rand**	**`CLIM`**	**fixed**	**rand**
random	97.6%	91.1%	49.8%	95.5%	88.6%	50.1%	94.8%	87.9%	50.0%
sobel	96.3%	93.4%	49.9%	94.4%	89.4%	50.0%	93.5%	88.6%	49.9%
gauss	98.7%	97.5%	50.0%	98.1%	96.7%	49.9%	98.1%	96.2%	50.0%

Table 2.4: Bit-level CLIM on INT_MUL for application quality prediction.

datasets	CPR1			CPR2			CPR3		
	`CLIM`	**fixed**	**rand**	**`CLIM`**	**fixed**	**rand**	**`CLIM`**	**fixed**	**rand**
sobel	100%	100%	3.1%	100%	100%	3.1%	100%	100%	3.1%
gauss	100%	100%	4.6%	100%	100%	4.6%	98.4%	95.3%	4.6%

We first evaluate the bit-level model of **`CLIM`** on four functional units and compare with baseline models.

Table 2.5: Bit-level CLIM on FP_MUL for application quality prediction.

datasets	CPR1			CPR2			CPR3		
	`CLIM`	**fixed**	**rand**	**`CLIM`**	**fixed**	**rand**	**`CLIM`**	**fixed**	**rand**
sobel	100%	68.7%	87.5%	100%	68.7%	87.5%	96.8%	68.7%	87.5%
gauss	96.8%	75.0%	78.1%	93.7%	75.0%	78.1%	93.7%	75.0%	78.1%

Table 2.2 and Table 2.3 present the MBPA of **`CLIM`** for INT_ADD and FP_ADD, where we can observe several facts. For INT_ADD, **`CLIM`** exhibits prediction accuracy between 95.6%-99.9% across three datasets and CPRs. Meanwhile, **fixed** can deliver prediction accuracy between 88.7%-99.9% and **rand** almost always achieves 50% accuracy. More specifically, **fixed** only achieves 99.9% accuracy when the input dataset is *sobel* or *gauss*. These two datasets are heavily biased with almost zero TERs, according to Fig. 2.2(a). For *rand* dataset, which is more representative, **`CLIM`** achieves 96.2% accuracy while **fixed** achieves 91.3% accuracy on average. For FP_ADD, **`CLIM`** exhibits prediction accuracy between 93.5%-98.7% across three datasets and CPRs. Meanwhile, **fixed** can deliver prediction accuracy between 87.9%-97.5% and **rand** almost always achieves 50% accuracy. On average, **`CLIM`** achieves 96.3% accuracy and **fixed** achieves 92.1% accuracy. Under mild TERs, **fixed** classifier can almost always achieve decent accuracy, perhaps leading one to doubt whether it is necessary and worthwhile to develop **`CLIM`**. In fact, **fixed** classifier has no ability to identify any positive output because it always predicts outputs to be C_c. This will severely hurt system reliability as it assumes no error when an error could occur. Thus, we further compare these models on more functional units by using them to predict the output quality of approximate computing applications as presented in Table 2.4 and Table 2.5. Before getting to the result, we first introduce bit-level reliability specification of approximate computing applications.

Bit-level Specification The error-tolerant applications used in the approximate computing field exhibit enhanced error resilience at the application-level when multiple valid output values are permitted. Instead of a single output value, the output value is associated within an application-specific quality metric, such as peak signal-to-noise ratio (PSNR). Therefore, if execution is not numerically precise, the application can still appear to execute correctly from the users' perspective. We focus on error-tolerant appli-

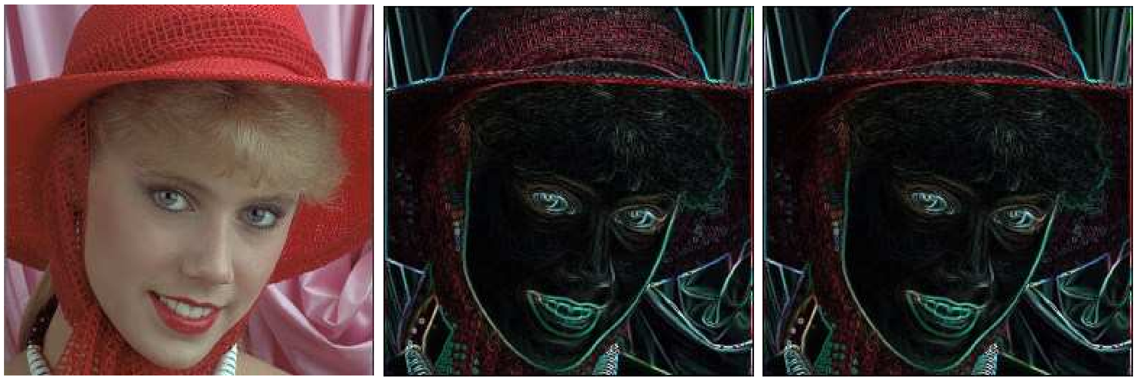

(a) Original Input (b) Exact Output (c) Approximate Output

Figure 2.3: (a) Original input image. (b) Error-free exact Sobel filter output with PSNR = inf. (c) Error-injected approximate Sobel filter with PSNR = 30dB.

cations mainly from the image processing domain, including Sobel filter and Gaussian filter. In image processing applications, a PSNR larger than 30dB is generally considered as acceptable to users [66]. As illustrated in Fig. 2.3, it is hard to tell the difference between exact output and approximate output.

We quantify the degree of error tolerance of arithmetic operations in these applications by defining the notion of bit-level reliability specification. Similar with Rely [16] described in Section 2.1, it defines the minimum required probability with which the arithmetic operation output bit must be correct so that the application can deliver an acceptable output. For example, if we say reliability specification of 20th bit of FP_MUL operation is 70%, it means if the reliability of this bit is lower than 70%, the application output quality is not acceptable.

We compute the reliability specification for each bit of interested arithmetic operations through an iterative fault injection process as shown in Fig. 2.4. First, we flip the one output bit of our interested operation (e.g., INT_MUL) with an initial probability that is small enough so that the application output quality is acceptable. This fault injection is done using our-modified version of Multi2Sim [75] simulator. Second, we check the output quality (PSNR) of the resulted application using Matlab. Third, if the output quality is acceptable, we increase the bit flip probability and repeat step 1 and 2 until the output quality is not acceptable, then we use the last acceptable probability as the threshold probability. After these steps, we calculate the reliability specification as $1 - threhold\_probability$. We repeat such fault injection processes for every bit position across multiple arithmetic operations and error-tolerant applications.

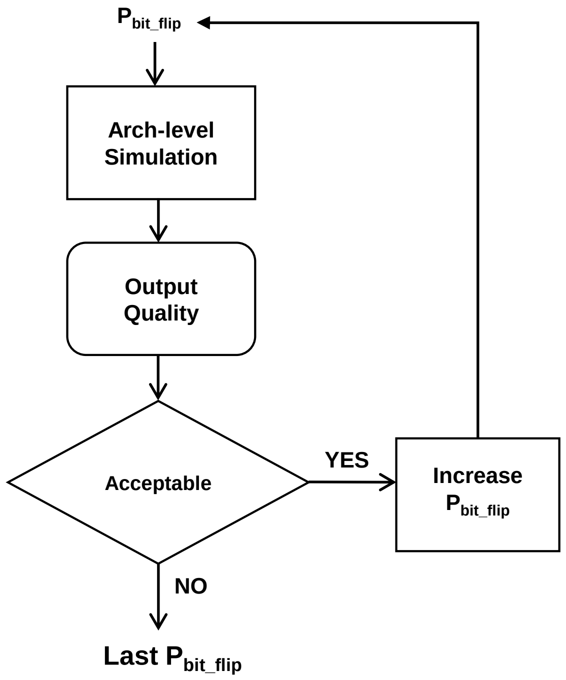


Figure 2.4: Derive bit-level reliability specification for error-tolerant applications through fault injection. P_{bit_flip} is a bit flip probability.

Quality Estimation We then use **`CLIM`** to predict the error-tolerant application quality into two classes: $\{$*acceptable*, *non-acceptable*$\}$ with the following process. First, we obtain the bit-level reliability specification of each bit position. Second, we use **`CLIM`** to predict the bit-level TER of each bit position, and then use $1 - TER$ to derive bit-level reliability. We then compare the predicted reliability with reliability specification. If the predicted reliability is greater than the specification, then **`CLIM`** will predict the application quality is acceptable; otherwise it is unacceptable. For example, if the predicted reliability for 20th bit of FP_MUL is greater than 70%, then **`CLIM`** will predict the application quality is acceptable. Third, we use gate-level simulation to compute the ground-truth reliability for each bit position. Then we use such reliability to determine whether the application quality is acceptable by comparing it with reliability specification, as with the second step. Finally, gate-level simulation will produce a ground truth result on whether an application quality is acceptable or not. Fourth, we then compare the prediction result of **`CLIM`** with gate-level simulation ground truth and compute the prediction accuracy across all the bit positions. We repeat the same process for **fixed**

and **rand** classifier.

Table 2.4 and Table 2.5 compare the accuracy of the three models. For INT_MUL, both **CLIM** and **fixed** achieve high prediction accuracy because according to Fig. 2.2(c), *sobel* and *gauss* have almost zero TERs. Thus, the real reliability is close to 100% which matches the **fixed** classification. The **rand** achieves low accuracy because its predicted reliability is close to 50% while the real reliability is close to 100%. For most bit positions, the bit-level specification is between 50% and 100%, thus **rand** has a different prediction than ground truth, resulting in low accuracy. For FP_MUL, **CLIM** achieves accuracy between 93.7%-100% while **fixed** and **rand** achieves 68.7%-75.0% and 78.1%-87.5% respectively. The low accuracy of **fixed** is due to the fact that most bit positions of FP_MUL have non-zero TERs. For example, for bit position 21 under *sobel* dataset whose bit-level specification is 99%, its ground-truth reliability is 96.7% as computed by gate-level simulation, making the quality non-acceptable. Since **CLIM**-predicted reliability is 95.5%, **fixed**-predicted reliability is 100% and **rand**-predicted reliability is 50%, both **CLIM** and **rand** correctly predict *non-acceptable* while **fixed** predicts *acceptable*, leading to a misprediction. **CLIM** demonstrates robustness across functional units and datasets regardless of whether it is biased, while **fixed** achieves low accuracy due to its inability to identify erroneous instances.

2.4.4 Value-level CLIM

Table 2.6: value-level CLIM on INT_ADD for timing error prediction.

datasets	CPR1			CPR2			CPR3		
	CLIM	**fixed**	**rand**	**CLIM**	**fixed**	**rand**	**CLIM**	**fixed**	**rand**
random	96%	9.9%	49.7%	93.5%	19.0%	50.0%	91.2%	29.6%	49.8%
sobel	99.3%	0.7%	49.9%	99.0%	0.8%	49.8%	98.4%	1%	50.0%
gauss	99.9%	0.1%	50.0%	99.9%	0.1%	50.0%	99.0%	0.1%	49.9%

Table 2.6 and Table 2.7 present the MVPA of **CLIM** for INT_ADD and FP_ADD. For INT_ADD, **CLIM** exhibits average prediction accuracy at 97.4% across three datasets and CPRs. Meanwhile, **fixed** delivers average prediction accuracy at 6.8% and **rand** al-

Table 2.7: Value-level CLIM on FP_ADD for timing error prediction.

datasets	CPR1			CPR2			CPR3		
	`CLIM`	**fixed**	**rand**	**`CLIM`**	**fixed**	**rand**	**`CLIM`**	**fixed**	**rand**
random	95.6%	9.7%	50.2%	93.2%	20.1%	50.1%	92.3%	67.0%	49.8%
sobel	95.3%	33.8%	49.9%	88.8%	39.9%	50.1%	92.2%	48.8%	49.9%
gauss	97.1%	9.6%	49.9%	94.2%	11.9%	50.0%	93.3%	15.6%	49.8%

Table 2.8: Value-level CLIM on INT_MUL for reliability prediction using AE.

datasets	CPR1			CPR2			CPR3		
	`CLIM`	**fixed**	**rand**	**`CLIM`**	**fixed**	**rand**	**`CLIM`**	**fixed**	**rand**
random	0.9%	89.9%	49.9%	0.6%	79.4%	29.4%	0.5%	70.2%	20.3%
sobel	2.4%	91.6%	41.6%	1.8%	84.4%	34.4%	4.6%	69.2%	19.2%
gauss	0.6%	89.8%	39.8%	3.1%	81.6%	31.5%	4.7%	65.1%	15.0%

Table 2.9: Value-level CLIM on FP_MUL for reliability prediction using AE.

datasets	CPR1			CPR2			CPR3		
	`CLIM`	**fixed**	**rand**	**`CLIM`**	**fixed**	**rand**	**`CLIM`**	**fixed**	**rand**
random	3.3%	89.9%	39.9%	3.9%	79.9%	29.9%	3.3%	70.5%	20.5%
sobel	0.4%	89.7%	39.7%	4.0%	80.0%	30.0%	0.5%	69.9%	19.9%
gauss	4.2%	89.9%	39.9%	5.8%	79.5%	29.5%	6.2%	70.9%	20.9%

most always achieves 50% accuracy. For FP_ADD, **CLIM** exhibits average prediction accuracy at 93.6% across three datasets and CPRs. Meanwhile, **fixed** delivers average prediction accuracy at 28.5% and **rand** almost always achieves 50% accuracy. The low accuracy of **fixed** classifier is due to the fact that at instruction-level, it always predicts C_e for all cycles because examined clock periods are all smaller than instruction-level timing delay. It only considers the worst-case scenario to set its instruction-level timing delay. Since **fixed** always predicts C_e when the examined clock period is smaller than instruction-level delay, its predicted value-level reliability is always close to 0. This will severely deviate from the ground truth reliability when the TER is mild. Thus, we further compare these models on more functional units by utilizing them to predict the reliability as presented in Table 2.8 and Table 2.9.

Before getting to the result, we introduce the evaluation metric on assessing the accuracy of reliability predictions: absolute error, as follows

$$AE = |reli_{pred} - reli_{gls}| \tag{2.12}$$

where $reli_{pred}$ is the predicted reliability while $reli_{gls}$ is the ground truth reliability derived by gate-level simulation. This metric defines the difference between the predicted value and the ”true” value, so a smaller value means a better performance.

Table 2.8 and Table 2.9 compare the accuracy of three models. For INT_MUL, **CLIM** achieves AE between 0.5%-4.7% while **fixed** and **rand** achieve 65.1%-91.6% and 15.0%-49.9% respectively. For FP_MUL, **CLIM** achieves AE between 0.4%-6.2% while **fixed** and **rand** achieve 69.9%-89.9% and 19.9%-39.9% respectively. The low accuracy of **fixed** is due to the fact that at the three CPRs, the TERs are approximately 10%, 20%, and 30% respectively and **fixed** always predicts 0 reliability, leading to a huge difference. This indicates that only considering the worst-case instruction-level delay to predict timing errors could lead to a huge deviation from the real scenario, which might be even worse than a random guess. Meanwhile, **CLIM** demonstrates its robustness with average AE at 2.8%.

2.4.5 `CLIM` Efficiency

We compare the **`CLIM`** speed with gate-level simulation. On average across all datasets and functional units, **`CLIM`** computes 173X faster than gate-level simulation. The more complex the circuit structure, the slower speed for simulation. But this might not apply to **`CLIM`**, because it processes input data according to its own rule, which might not scale up with the complexity of the circuit structure. For previous instruction-level models [79], the authors claim that the gate-level simulation is very time-consuming and becomes a bottleneck for research purpose. Thus, **`CLIM`** provides a faster alternative way to examine reliability without performing time-consuming conventional gate-level simulation.

2.4.6 Discussion

Potential Usage: The machine learning approaches can also be used to predict the timing errors for a different implementation of circuits, such as approximate adders [47]. On the other hand, the model could be utilized online to guide dynamic frequency scaling (DFS) with an efficient physical implementation. Recently, a voltage-droop induced delay prediction model has been implemented using SVM to guide online DFS [81], whose hardware overhead is 1.5% for today's processor design. We expect the overhead of **`CLIM`** is less than such a model, since by comparison Table 2.1 shows that SVM computing time is more than 7000X of RFC model.

Potential Limitation: The main limitation of such a learning-driven method is that it only works for arithmetic functional units. It is unclear whether it can work for other micro-architecture parts such as memory. This is because the advantage of machine learning is that it can learn the path sensitization based on input data pattern, which is the main factor that determine timing errors. But for memory, there is not a clear clue as to the source factors of its timing errors.

2.5 Chapter Summary

This chapter presents **`CLIM`**, a supervised learning-based model to predict timing errors of functional units at two levels: the bit-level and the value-level. It considers the impact of input operands on dynamic path sensitization (and hence timing errors). We perform gate-level simulation on a post-layout netlist to extract timing errors and useful "features" from input data and circuit activity. We then apply a random forest classification method to construct the model with extracted input features and output labels. We consider input workload, computation history, and circuit toggling as input features to construct **`CLIM`**. For a given input data and circuit parameter, **`CLIM`** predicts the output to be one of two classes: $\{$*timing correct, timing erroneous*$\}$. On average across several functional units and clock periods, its bit-level and value-level prediction accuracy are 97% and 95% respectively, far more accurate than existing models without considering the effects of input workload. We utilize **`CLIM`** in estimating error-tolerant application output quality, achieving an average of 97% accuracy. **`CLIM`**-based reliability estimation is within 2.8% deviation on average of detailed gate-level simulation. Our ongoing work seeks to improve the efficiency of model building by using efficient and more advanced learning methods.

Chapter 2 contains reprints of Xun Jiao, Abbas Rahimi, Balakrishnan Narayanaswamy, Hamed Fatemi, Jose Pineda de Gyvez, and Rajesh Gupta, "Supervised Learning Based Model for Predicting Variability-Induced Timing Errors", *Proc. IEEE International NEW Circuits And Systems (NEWCAS) conference*, 2015; Xun Jiao, Yu Jiang, Abbas Rahimi, and Rajesh Gupta, "SLoT: A Supervised Learning Model to Predict Dynamic Timing Errors of Functional Units", *Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE)*, 2017; Xun Jiao, Abbas Rahimi, Yu Jiang, Jianguo Wang, Hamed Fatemi, Jose Pineda de Gyvez, and Rajesh Gupta, "CLIM: A Cross-level Workload-aware Timing Error Prediction Model for Functional Units", *IEEE Transactions on Computers (TC)*, 2017. The dissertation author is the primary author of the papers.

Chapter 3

Correcting Timing Errors Efficiently using Structural Errors

In the last chapter, we propose a workload-aware timing error prediction model to combat the **inaccurate error modeling** challenge associated with **error prevention** approaches. In this chapter, to combat the **high correction cost** challenge associated with the **error correction** approaches, we propose a method to efficiently correct timing errors using the structural errors of inexact (approximate) circuits.

3.1 Introduction

Typical error correction approaches actively reduce and even remove the guardbands, and then use a correction mechanism to correct the timing errors in the system in case of their occurrences [10, 22, 32, 34]. As discussed in Chapter 1, such techniques could incur significant silicon overhead for online monitoring [17, 34, 61] and performance penalty [10].

In this chapter, we show how timing errors can be corrected efficiently using structural modifications to circuits such as in the design of Inexact Speculative Adders (ISA). The structural modification is used by a redesign of the adder architecture that shortens the critical path and performs carry speculation. Such structural modification can cause errors, referred to as structural errors. However, these errors can be controlled with the selection of design parameters such as the selection of speculation, error-correction and error-reduction mechanisms. On the other hand, we use overclocking to reduce conservative guardbands at the risk of introducing new timing errors. We build a methodology to combine both structural and timing errors and analyze their interplay with each other to limit the joint errors. We apply machine learning methods to predict timing errors of overclocked Inexact Speculative Adders.

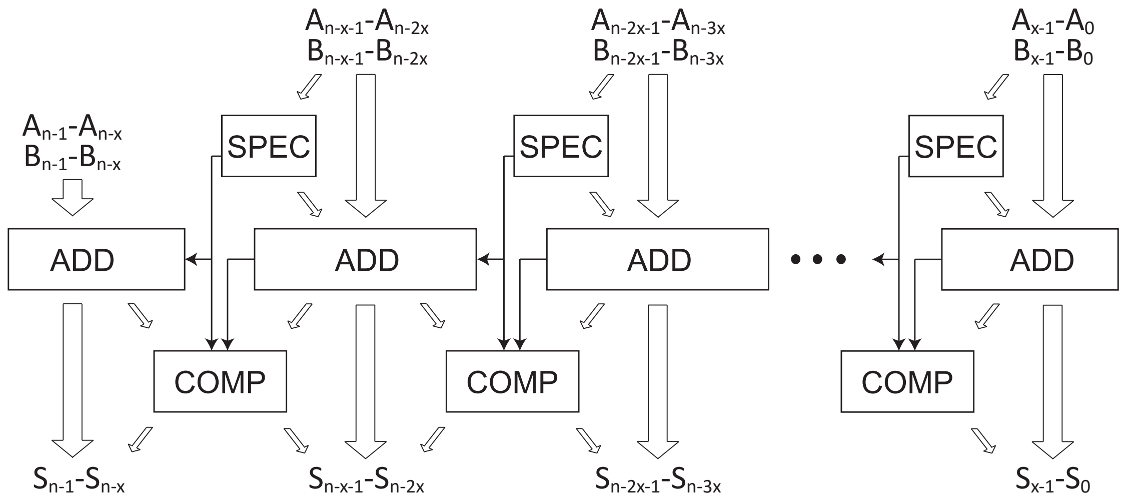
Our contributions are as follows:

- We build a bit-level timing-error prediction model for overclocked ISA evaluating arithmetic effects of errors.
- We develop a methodology to combine both structural and timing errors and to show their joint contribution on the average relative error.
- We characterize the trade-off between structural and timing errors for different overclocked ISA designs.
- We analyze the distribution of both types of errors and how they interplay with each other in an overclocked ISA.

3.2 Inexact Speculative Adders

Since adders are the most common arithmetic blocks used in DSPs, many attempts have been made to design approximate versions of them [14, 70, 71, 83]. To this purpose, *carry speculation* is an interesting technique, exploiting the fact that in additions, carry propagate sequences are typically short [83]. Hence, it is possible to estimate, more or less accurately, an intermediate carry using a limited number of previous stages. This allows to splits the carry chain into two or more shorter paths, relaxing the constraints over the entire design and pushing energy, delay and area beyond the limits imposed by traditional design.

Many speculative adders have been proposed in literature based on the ETAII concept [83]. Among them, the Inexact Speculative Adder [13] has generalized and optimized the architecture for speculative compensated addition to minimize speculation overhead and by implementing a dual-direction compensation mechanism. Moreover,



A_{n-1}-A_{n-x} B_{n-1}-B_{n-x}
A_{n-x-1}-A_{n-2x} B_{n-x-1}-B_{n-2x}
A_{n-2x-1}-A_{n-3x} B_{n-2x-1}-B_{n-3x}
A_{x-1}-A_0 B_{x-1}-B_0
SPEC
SPEC
SPEC
ADD
ADD
ADD
ADD
COMP
COMP
COMP
S_{n-1}-S_{n-x}
S_{n-x-1}-S_{n-2x}
S_{n-2x-1}-S_{n-3x}
S_{x-1}-S_0

Figure 3.1: General block diagram of an Inexact Speculative Adder. Each speculative segment consists of a carry speculator (SPEC), a regular adder (ADD) and an error compensation block (COMP).

it has already been successfully verified and integrated in multiplier circuits [15]. The following subsections presents the ISA in details.

3.2.1 Overall Architecture

The block diagram of an ISA is depicted in Fig. 3.1. It splits the carry chain in multiple paths executed concurrently, each of them consisting of a carry speculator block (SPEC), a sub-adder block (ADD) and an error compensation block (COMP).

The SPEC generates a partial carry signal from a limited number of operand bits using a carry look-ahead approach. When a propagate chain covers the full block, the exact carry cannot be speculated from the partial product and the output carry is guessed. The ADD calculates local sums from the speculated carry. The COMP detects the speculation faults by comparing the carry generated from the SPEC with the carry-out coming from the previous ADD. It then compensates faulty sums either by attempting to correct a few bits of the local sum or by reducing relative error over a few bits of the preceding sum. This allows to avoid massive errors generated from an internal overflow caused by an inconsistent carry. Intuitively, the first speculative path, i.e. the one operating on the least significant bits (LSBs) of the adder, does not have SPEC or COMP blocks since it uses directly the adder carry-in.

3.2.2 Error Compensation Technique

The achieved addition arithmetic is illustrated in Fig. 3.2. The COMP's error-correction technique consists in incrementing or decrementing only a small group of LSBs of the local sum to compensate for the erroneous speculated carry. In most cases, it can fully resolve carry errors, but if those stages are all in propagate modes, correction is impossible as it would lead to an internal overflow. In this situation, the uncorrected bits ensure a low relative error of the result, since they have a higher significance than the error bit. The COMP also uses error balancing to flip a small group of MSBs of the preceding sum to further reduce the relative error.

Thus, using the COMP block reduces simultaneously error rate and relative error. Moreover, as the correction hardware is executed concurrently to the local addition, this technique has a minimal impact on the critical path.

3.2.3 Design Strategy

Inexact Speculative Adders can easily be designed with a delay-accuracy approach: the adequate delay tradeoff is obtained by sizing SPEC and ADD blocks, principal slack elements of the ISA, while the sizing of the COMP techniques can then be used to tune the mean accuracy and limit the worst-case error. Thanks to a custom sizing of each speculative path and each speculative path block, the ISA architecture allows very precise tuning of multiple error characteristics while optimizing circuit performance and efficiency.

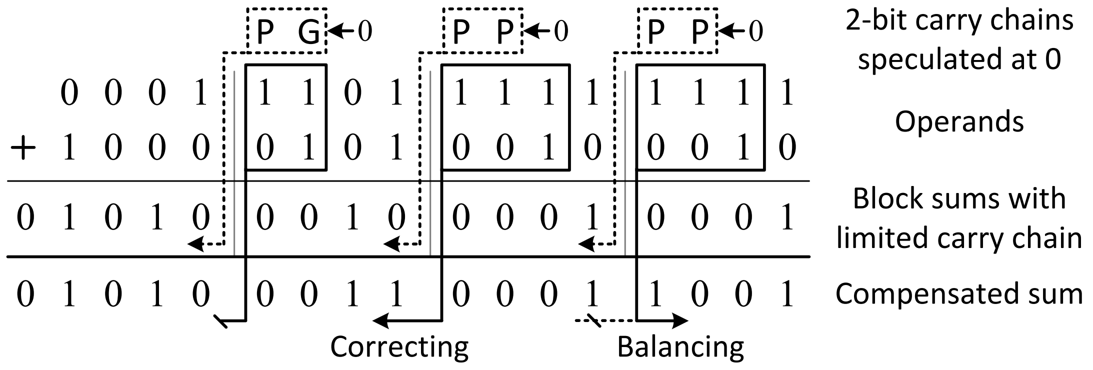


Figure 3.2: Example of ISA addition arithmetic with 2-bit speculation, 1-bit correction and 1-bit error reduction.

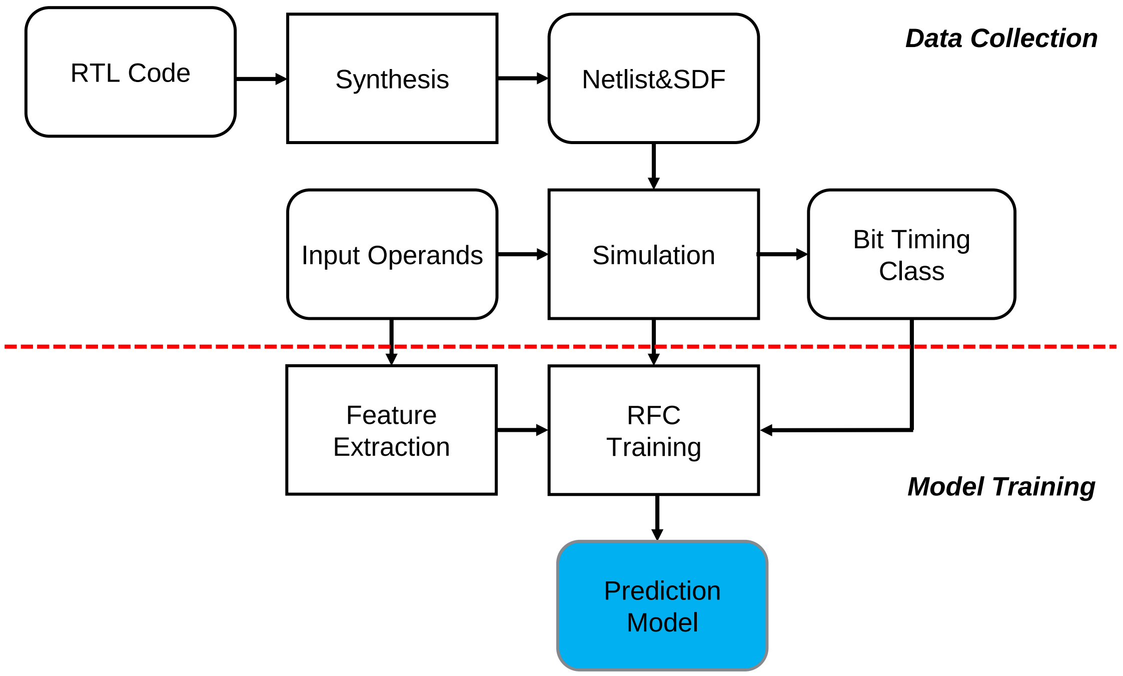


Figure 3.3: Bit-level timing error prediction model construction flow.

3.3 Guardband Reduction with Bit-level Timing Error Prediction

The bit-level timing-error prediction model for guardband reduction uses binary classification method to predict timing errors for a given clock reduction and input load. It captures the dynamic circuit path sensitization behaviors by learning the mapping relationship between input workload and bit-level timing errors. For each bit position, a binary classifier is trained to predict if it is timing-erroneous. The overall model construction flow, containing two parts, *Data Collection* and *Model Training*, is illustrated in Fig. 3.3.

Data Collection:First, we synthesize the RTL code into a netlist and extract the corresponding standard delay format (SDF) file. Second, using random data as input operands, we perform SDF-annotated gate-level simulations to generate output data at unsafe clock periods. At each cycle, a new input vector is fed into the simulation. We define $x[t]$ as the input vector, $y_{RTL}[t]$ as pure-RTL output and $y[t]$ as gate-level output, at cycle t. For an output bit position n at cycle t, we define the timing class of $y_n[t]$

as $c_n[t]$, which is one of two timing classes: $\{$*timing-correct*, *timing-erroneous*$\}$ based on whether it has a timing error. If $y_n[t]$ matches $y_{RTLn}[t]$, then $c_n[t]$ is timing-correct, otherwise timing-erroneous.

Model Training: First, we extract the useful feature vectors from input data. At cycle t, the output $y[t]$ is jointly determined by current input $x[t]$ and preceding cycle input $x[t-1]$. Besides, we also consider output bit value $\{y_{RTLn}[t-1], y_{RTLn}[t]\}$ as input feature because the timing error (bit flip) can only occur when these two bits are different [20]. If these two bits hold same value, the latched value is correct to users even if the clock period does not meet the sensitized path delay. Thus, we consider $\{x[t], x[t-1], y_{RTLn}[t-1], y_{RTLn}[t]\}$ as our input feature and the $c_n[t]$ as output label.

For each bit position, we train a binary classifier using supervised learning methods. We use Random Forest tree Classification (RFC) as our learning method to balance the prediction accuracy and training cost. RFC is an ensemble method composed of a number of decision trees (DT), which learn a set of decision rules based on the pattern of input and their possible outcomes. DT considers the joints effects of different bit positions but could incur overfitting problem. RFC alleviates overfitting issue by developing more than one decision tree and use their average result as final prediction. It may lose the opportunity to learn some “irregular” patterns, overall it reduce the overfitting and boost performance.

3.3.1 Prediction Model Evaluation

To evalutate the model prediction accuracy for a selected overcloking rate, we define the *average bit-level prediction error rate* ($ABPER$) as follows:

$$ABPER[clk] = \frac{\sum_{bit\ n} \left(\frac{\sum_{cycle\ t} |TC^{(pred)}_{clk,n,t} - TC^{(real)}_{clk,n,t}|}{||\#cycles||} \right)}{||\#bit\_positions||} \tag{3.1}$$

where $TC^{(pred)}_{clk,n,t}$ and $TC^{(real)}_{clk,n,t}$ are the predicted and real timing classes (0 for timing-erroneous and 1 for timing-correct) at a given clock period clk, bit position n and cycle t. This metric is a good indicator of bit-level model prediction accuracy.

3.4 Combining Structural and Timing Errors

All previous works have discussed individual use of either approximate circuit design, such as speculative compensated architectures, or guardband-reduction (overclocking) timing-error prediction. But those two approaches targeting different abstraction levels could intuitively be the perfect and

It is possible to combine structural errors due to speculation and timing errors due to guardband reduction to maximize circuit performances and robustness. Indeed, timing errors occur on the critical paths, which would be split into multiple shorter paths in a speculative circuit. The timing errors would thus be distributed among all outputs—instead of only degrading MSBs in conventional circuits—but at the cost of structural errors due to speculation. Parameters of the speculative structure and levels of guardband reduction can be adjusted together in order to find an optimum between timing and structural errors.

This study focuses on the case of binary addition based on the use of ISA adders synthesized for 3.3 GHz in a 65 nm technology. The methodology adopted is the following:

- Several ISA adders have been selected and implemented with design parameters optimizing error and circuit costs.
- Timing-error prediction has been adapted to predict timing errors on these circuits for different overclocking levels.

3.4.1 Error Combination

A new error model needs to be developed to distinguish and combine correctly the error contributions from both abstraction layers. First, at behavioral level, *structural errors* are caused by the design of the ISA architecture. Those deterministic errors vary with the selection of design parameters such as the selection of speculation, error-correction and error-reduction mechanisms. Structural errors are obtained by comparing the outputs from the designed circuit from exact addition results. Then, at gate level, *timing errors* occur when overclocking the ISA circuit, thus are obtained by comparing the over-clocked circuit to the same inexact but properly-clocked circuit. Those errors vary with different clock periods and are less predictable as they also depend from the previous circuit state or inputs.

To simplify the three type of output values used to compute those errors, we define the following types of output values:

- y_{silver}, the *silver output* obtained from the over-clocked ISA circuit, containing both structural and timing errors.
- y_{gold}, the *golden output* the expected value from the implemented circuit, containing the structural errors only.
- $y_{diamond}$, the *diamond output* ideal output value from an exact addition or conventional adder circuit.

Thus, we compute the *arithmetic error* (E) from each abstraction level as:

$$E_{struct} = y_g - y_d \quad E_{timing} = y_s - y_g \,, \tag{3.2}$$

whereas the *relative error* (RE), both contributions being calculated with respect to the exact result, is defined as:

$$RE_{struct} = \frac{y_{gold} - y_{diamond}}{y_{diamond}} \quad RE_{timing} = \frac{y_{silver} - y_{gold}}{y_{diamond}} \,. \tag{3.3}$$

Despite this study only considers unsigned computations, it is important for arithmetic and relative errors to be kept signed. Indeed, if both error contributions are in the same directions, they would add to each other to increase the overall error, such as in Fig. 3.4:

output values			error contributions	
$y_{diamond}$	1000	8	RE_{struct}	$\frac{6-8}{8} = -\frac{2}{8}$
y_{gold}	0110	6	RE_{timing}	$\frac{4-6}{8} = -\frac{2}{8}$
y_{silver}	0010	4	RE_{joint}	$-\frac{2}{8} - \frac{2}{8} = -\frac{4}{8}$

Figure 3.4: Example of additive errors (exact output $y_{diamond}$, exemplary erroneous outputs y_{gold} and y_{silver} from ISA and over-clocked ISA, respectively)

But if two errors happening simultaneously are in opposite directions, they could compensate each other and reduce the overall error, such as in Fig. 3.5:

output values			error contributions	
$y_{diamond}$	1000	8	RE_{struct}	$\frac{6-8}{8} = -\frac{2}{8}$
y_{gold}	0110	6	RE_{timing}	$\frac{7-6}{8} = +\frac{1}{8}$
y_{silver}	0111	7	RE_{joint}	$-\frac{2}{8} + \frac{1}{8} = -\frac{1}{8}$

Figure 3.5: Example of compensating errors

Fig. 3.6 depicts the flow used to combine ISA errors with timing errors in the case of arithmetic errors for example.

3.4.2 Model Evaluation

Although the $ABPER$ is a good metric for bit-level prediction accuracy, it does not represent the misprediction effect on the output arithmetic value of the adder. Thus, we define another metric using output arithmetic error values instead of bit timing classes, the *average value-level predictive error* ($AVPE$):

$AVPE[ISA, clk] =$

```
1  inputs: set of ISA architectures, input set, clock periods
2  outputs: mean arithmetic errors
3  foreach $ISA \in$ ISA architectures do
4      foreach $x \in$ input vectors do
5          compute $y_{diamond}[x]$
6          compute $y_{gold}[x, ISA]$
7          compute $E_{struct}[x, ISA] = y_{gold}[x, ISA] - y_{diamond}[x]$
8          foreach $clk \in$ clock periods do
9              compute $y_{silver}[x, ISA, clk]$
10             compute $E_{timing}[x, ISA, clk]$
                 $= y_{silver}[x, ISA, clk] - y_{gold}[x, ISA]$
11             compute $E_{joint}[x, ISA, clk]$
                 $= E_{timing}[x, ISA, clk] + E_{struct}[x, ISA]$
12     compute means of $|\, E_{joint}[x, ISA, clk] \,|$ over inputs
```

Figure 3.6: Pseudo-code computing the mean arithmetic error of over-clocked ISAs with structural and timing errors.

$$\frac{\displaystyle\sum_{cycle\ t} \frac{|\, y^{(pred)}_{silver}\,[ISA, clk, t] - y^{(real)}_{silver}\,[ISA, clk, t]\,|}{y^{(real)}_{silver}\,[ISA, clk, t]}}{||\#cycles||} \tag{3.4}$$

where $y^{(pred)}_{silver}$ and $y^{(real)}_{silver}$ are the predicted and real arithmetic output values of a given ISA, clock period clk and at cycle t.

Note that the model does not directly generate arithmetic values, it only generates timing-class vectors, which are arrays of bit-flip positions, and deduces the corresponding y_s compared to the expected output y_g.

3.5 Experimental Results

3.5.1 General Considerations

Twelve different ISA designs have been selected from [12], they are the best implementations fitting the 0.3 ns timing constraints. All ISA have regular structures with uniformly sized blocks (i.e. parallel paths of 2x16, 4x8, 8x4 bits only) and are denoted by quadruples of bit-widths: (block size, SPEC size, correction, reduction). They have been confronted to an exact adder, also constrained at 0.3 ns.

Approximate circuits are commonly characterized and validated through the simulation of random sets of inputs. As a matter of fact, the presented results are statistical estimations depending on the random sample distribution (occurrence of specific patterns initiates errors in specific adders). Adders are characterized using a sample of ten million unsigned random inputs. The main metric considered is the Root Mean Square (RMS) of the relative error RE as it is independent of the adder bit-width and proportional to the SNR, which it interesting for many applications, particularly in multimedia processing.

Circuits have been synthesized with Synopsys Design Compiler in an industrial 65 nm technology from high-level descriptions in order to benefit from the compiler's optimization libraries and most favorable architecture choices. Delay-annotated gate-level simulations have been run with Mentor Modelsim in order to extract timing errors for three delays: 0.285 ns, 0.27 ns and 0.255 ns, corresponding to 5, 10 and 15 % of

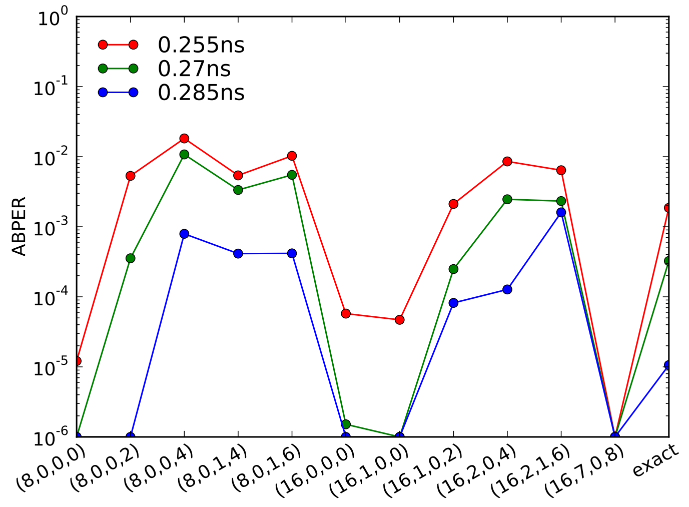


Figure 3.7: Average bit-level prediction error rate (ABPER) under overclocking.

overclocking from the safe-clock period of 0.3 ns. Machine learning methods used to construct the model come from the Scikit-learn Python package [62].

3.5.2 Timing-error Prediction Evaluation

Fig. 3.7 presents the $ABPER$ for each ISA at three overclocking points: 5, 10 and 15 %. From this figure, we firstly observe that almost all $ABPER$ values are around or less than 1 %, demonstrating a high prediction accuracy of the model. Second, $ABPER$ at higher overclocking is always larger than that at less overclocking. For example, the third ISA (8,0,0,4), has $ABPER$ around 0.1% at 0.285ns (5% overclocking), and has $ABPER$ around 1% for 10% and 15% overclocking. This is because more paths violating timing specification resulting in more timing errors, which makes model harder to track all path sensitization behaviors. Some $ABPER$ can reach 0 if there is no timing error, such as ISA (8,0,0,0) at 0.285 ns and 0.27 ns. We use 10^{-6} as $ABPER$ in this case.

Fig. 3.8 presents the $AVPE$ for each ISA at three overclocking points. From this figure, we observe that although bit-level prediction accuracy are always good but the mispredicted bits could sometimes cause a large arithmetic error. For example, the eighth ISA (16,1,0,2) at 0.255 ns and 0.27 ns causes a $AVPE$ around 5. This is because many mispredicted bits are among most significant bits that can cause a large deviation up to 2^{32} from original value. While for most ISA, the third ISA (8,0,0,4) for example, has $AVPE$ less than 0.1$\%$ for all three overclocking points, showing the misprediction effect on arithmetic value is negligible. Similar with Fig. 3.7, we neglect $AVPE$ value when it is lower than 10^{-6}. Overall, most $AVPE$ are lower than 10^{-2}, indicating that misprediction on arithmetic error is tolerable for most ISA designs.

3.5.3 Results of Error Combination

Fig. 3.9 shows the structural and timing relative error RMS as well as their resulting joint contribution for ISA designs under the three overclocking points.

At the lowest overclocking rate of 5 % (Fig. 3.9a), we immediately observe that the exact adder circuit (rightmost of the figure) is subject to large timing errors which make it the worst adder of the group in terms of overall joint error RMS. We find that for most ISA adders, the joint error is dominated by the structural-error contribution coming from the speculative architecture. Low and medium-accuracy ISA circuits (on the left part of the figure) seem very robust to timing errors, having negligible timing errors compared to structural errors. Among the high-accuracy ISA designs, only ISA (16,2,0,4) has succumbed to a massive amount of timing errors. Though, if this specific ISA has a low sensitivity threshold to timing errors, it is still better than the exact adder in terms of joint error.

At 10 % overclocking (Fig. 3.9b), timing-error contributions strongly increase, but stay lower than structural-error contributions for low-accuracy ISA adders. Two additional high-accuracy ISA circuits have fallen to timing errors: ISA (16,0,0,0) and (16,1,0,2) ISA circuits. Yet, they are still operating slightly better than the exact adder, whose average error, entirely due to timing, has been multiplied by 3 compared to 5 % overclocking.

At the highest overclocking of 15 % (Fig. 3.9c), all the selected high-accuracy ISA designs have fallen to timing errors. Yet, some of these designs still exhibits decent overall accuracy such as ISA (16,2,1,6). This latter relegates to the second place ISA (16,7,0,8), which has a more accurate architecture but is found less resilient to aggressive overclocking. Understanding this variability in timing-error robustness as well as the difference of threshold between structural and timing errors could be highly beneficial to low-power and time-constrained circuit design. This would require a deeper

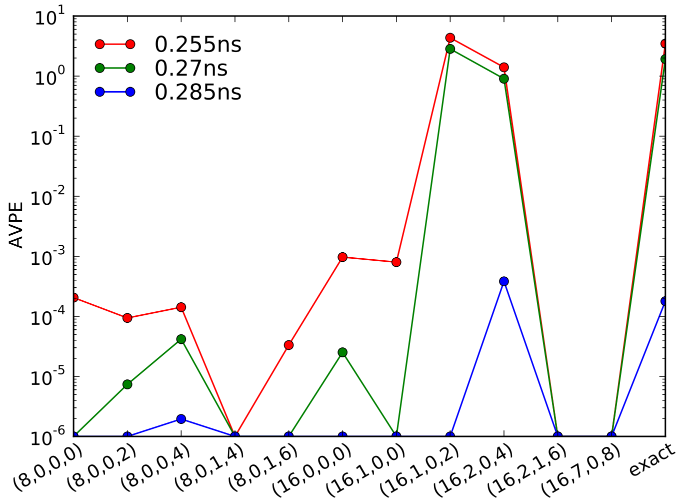


Figure 3.8: Average value-level predictive error (AVPE) under overclocking.

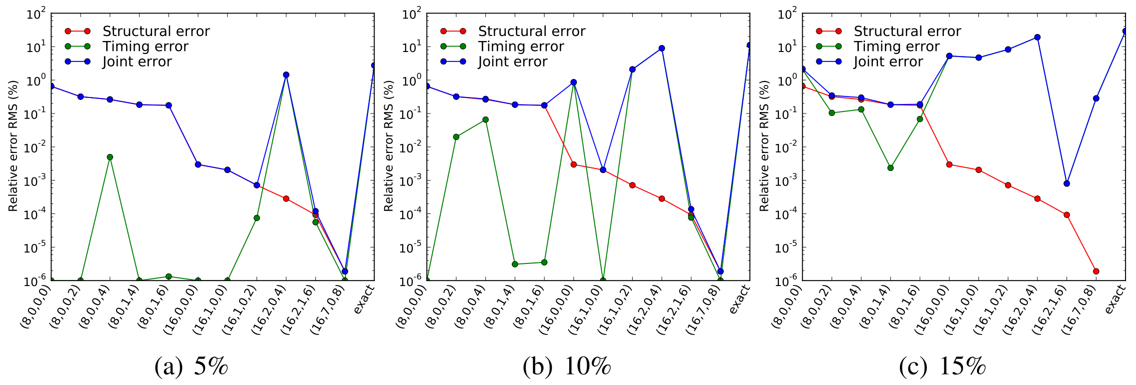


Figure 3.9: Relative error RMS of ISAs under 5 %, 10 % and 15 % overclocking.

analysis combining more speculative designs to better cover the design space offered by inexact speculative circuits.

For low-accuracy ISA overclocked with 15 % overclocking (left part of Fig. 3.9c), it is particularly interesting to note the high balance between timing and structural errors. This compromise between the two error contributions gives generally a better overall accuracy than adders designed with high structural accuracy.

3.5.4 Structural and Timing Error Balance

In order to better understand how the two types of errors interplay with each other, Fig. 3.10 displays the internal distribution of structural and timing errors within

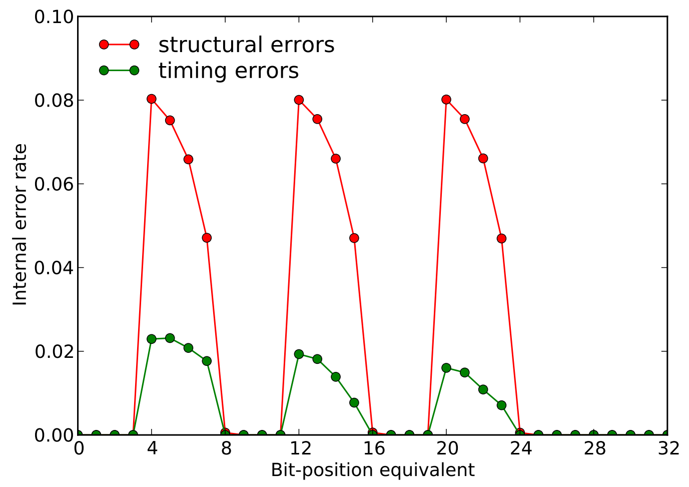


Figure 3.10: Bit-level-equivalent error distribution in ISA (8,0,0,4) under 15 % overclocking.

the example of 15 %-overclocked ISA (8,0,0,4) since this configuration shows the best balance between errors (c.f. Fig. 3.9c). Arithmetic structural errors have been translated into their equivalent bit-level positions. Note that the timing errors distribution is not as regular as the structural errors distribution. While it is easy to distinguish when several arithmetic speculative errors occur simultaneously on different speculative paths and translate independent errors into bit positions, timing errors might span over various outputs.

Structural errors are immediately recognizable on three speculative paths (the first speculative path, operating from the LSB, uses directly the adder carry-in so doesn't have errors). As this ISA only has 4-bit error reduction (no error correction), it only introduces errors on the preceding sub-adder sums, that is why structural-error peaks are slightly shifted on the left of the figure.

In a conventional adder, overclocking would dangerously degrade MSBs. In this ISA, despite causing structural errors, the 4-path speculative structure leads to a split of critical path, distributing the timing errors over those paths instead of the MSBs. Those errors mainly occur on the 4-bit error reduction block, last logic element in the critical path. This trade-off between structural and timing errors demonstrates the good resilience of ISA architectures against conventional circuits.

3.6 Chapter Summary

This Chapter presents a method to efficiently correct timing errors using the structural errors in Inexact Speculative Adders. We show that speculative adders are more resilient to overclocking than conventional adders because the structural errors and timing errors compensate each other. Indeed, the effects of combination of both errors are controllable since the speculative structure induces a split in the critical path that balances timing errors and distribute them along all outputs instead of only degrading MSBs in conventional circuits.

Chapter 3 contains reprints of Xun Jiao, Vincent Camus, Mattia Cacciotti, Yu Jiang, Christian Enz, and Rajesh Gupta, "Combining Structural and Timing Error in Overclocked Inexact Speculative Adders", *Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE)*, 2017. The dissertation author is the primary author of the paper.

Chapter 4

Instruction-based Timing Error Prevention

In the last chapter, we used structural errors of inexact circuits to efficiently correct timing errors so as to reduce the correction cost, a challenge we identified earlier with error correction approaches. In this chapter, focusing on the same challenge, we elevate our abstraction level from circuit level to instruction level and model the delay of instructions. Based on the delay model, we propose an instruction-based timing error prevention approach to avoid the correction cost.

4.1 Introduction

As mentioned in Chapter 1, **error prevention** is a less-intrusive method than **error correction** in the sense that it avoids the performance penalty when correcting errors. For example, Roy *et al.* propose a method to predict critical instructions based on instruction types [67]. Upon the identification of critical instructions, the pipeline is stalled one or two cycles to allow enough timing margin to finish the instruction execution. However, they are not able to consume all available timing slack as they lack in knowledge of absolute value of instruction timing requirements. It also overlooks the effect of input operands on the path sensitization behavior, leading to a less efficient or pessimistic modeling of hardware timing.

There is, of course, a correlation between the input workload and timing violations because of its direct effect on dynamic path sensitization. During execution, the sensitized paths strongly vary with different input workload [54]. This is seen in different instruction-level timing delay as a function of the operands to instructions.

Proposed approach: In this chapter, we explore the use of a workload-dependent predictive model for instruction-level timing management. We propose **`WILD`**, a supervised learning model to predict the dynamic delay of functional units based on input workload (operands). First, we extracted useful input features from input workload by analyzing the variations of circuit timing delay, measured dynamically each cycle under different input operands. We employ a switching activity file obtained from a post-layout gate-level simulation on TSMC 45nm technology. Then, we use supervised learning methods to construct and train the model, evaluate and compare with the baseline modeling in terms of prediction accuracy. We evaluate various commonly used machine learning techniques and chose logistic regression as our modeling method due to its high prediction accuracy and efficient computing time. Furthermore, we have applied our prediction model to three different datasets, random data, Sobel filter and Gaussian filter, and achieve prediction accuracy ranging between 96.2–99.8%. Using **`WILD`**-directed dynamic frequency scaling (DFS), the average instruction-level timing delay could be reduced for three different instructions (Int_ADD, Int_MUL and FP_MUL). In addition, our model executes 60X faster compared with gate-level simulation to compute dynamic delay for 200k data.

Contributions: We make the following contributions:

- We develop an accurate dynamic timing delay measurement methodology based on switching activity derived through gate-level simulation. We analyze the sources of delay variations and the effect of input workload on timing delay.

- We propose **`WILD`**, a supervised learning model to predict the dynamic delay of functional units based on input workload (operands) using supervised learning methods.

- The evaluation results demonstrate the robustness and accuracy of **`WILD`** in prediction and its effectiveness in system performance increase. We profile the test input workload from real-world applications and achieve 96.2–99.8% prediction accuracy, which is 3.6X and 1.5X higher compared to two baseline models. Further, by using **WILD**-directed dynamic frequency scaling, the instructions can achieve 13–44% operation speed up compared with the state-of-art instruction-level timing model directed dynamic frequency scaling [25, 67].

4.2 Instruction-Delay Prediction Model (WILD)

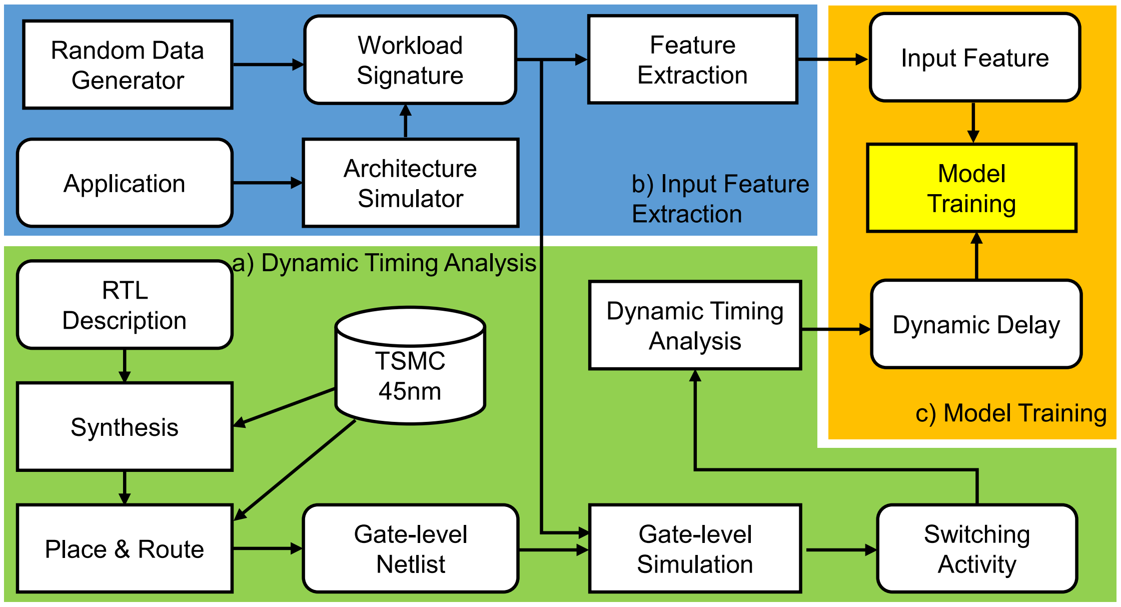


Figure 4.1: **WILD** model: a) Dynamic Timing Analysis to measure the dynamic delay; b) Input Feature Extraction to extract useful "features"; c) Model Training to use supervised learning to train the model.

It is comprised of three phases as shown in Fig. 4.1: *Dynamic Timing Analysis, Input Feature Extraction* and *Model Training*. a) The *Dynamic Timing Analysis* phase implements the standard ASIC flow and uses gate-level simulation to generate switching activity file. Then, our Python-written dynamic timing analysis script will analyze the switching activity file to generate the dynamic delay under different input workload. b) In the *Input Feature Extraction*, we generate the input training data in two ways: using a random data generator and profiling of functional unit input operands in real-world applications using architectural-level simulator. Then, the workload signature of training data is pre-processesed and useful features are extracted from the training data, such as bit locations and input history, which are then incorporated into model training. c) In the *Model Training* phase, the model is trained with the previous collected data using different supervised learning algorithms. We classify the output dynamic delay into different classes and the model will predict the class to which the output delay belongs for a given input data. More details about the three phases are illustrated as follows.

4.2.1 Dynamic Timing Analysis

We focus on three different types of functional units, 32-bit integer adder and multiplier, and 32-bit single-precision floating point multiplier. The floating point units (FPUs) are compatible with IEEE-754 standard, and can provide more complex circuit structures compared to their integer counterparts. We vary the circuit structures not only by function types but also by data types to assess the robustness of our model.

We use FloPoCo [28] to generate the synthesizable VHDL codes of functional units with wrapper at input and output ports. *Synopsys Design Compiler* is used to synthesize the VHDL codes and *Synopsys IC Compiler* is used to do place&route to generate post-layout netlist in TSMC 45nm technology. *Synopsys PrimeTime* is used to do static timing analysis to generate Standard Delay Format (SDF) file. Then, we use *Mentor Graphics Modelsim* to do SDF-back-annotation gate-level simulation to generate value change dump (VCD) file as a switching activity file. The stimuli input comes from two sources: random data generator script written in Python and the application input data profiled using *Multi2Sim* [75], a cycle-accurate CPU-GPU heterogeneous architectural simulator.

Next, unlike static timing analysis which can only give us the static timing of path delay, we use the switching activity file to do the dynamic timing analysis. The VCD file records the toggled nets at each cycle thus giving us the dynamic path sensitization information. To extract the dynamic delay based on sensitized critical path, we are only interested in the endpoints of every timing path. We run the simulation at a relatively slow clock period to make sure there is no timing violation. For each clock cycle, we use the last toggle event time of the input pin of all sequential elements (flip flop, registers, etc) to subtract the last positive clock edge arrival time to get the maximum delay at that cycle. For example, at cycle N the positive clock edge occurs at time t, and the very last toggled event at the data input pin of all sequential elements occurs at time t', then the dynamic delay at this cycle is $t' - t$. We run a large simulation and probe all toggled events at data input pins of sequential elements, and then parse the VCD file using our dynamic timing analysis tool that can provide us the dynamic delay at each cycle under different input workload. Note that when the input operands are the same for two consecutive cycles, there is no toggled nets, resulting a zero dynamic delay.

4.2.2 Input Feature Extraction

Having said before, there are extremely high number of possible input combinations. Given two 32-bit operands, there are 2^{64} different combinations. Thus, it is not feasible to apply all 2^{64} input patterns for training. To cover a large range of input space, we use the homogeneous distribution of two operands over 2D input space used in [74]. By applying these training input to the *dynamic timing analysis* module, we obtain the dynamic delay corresponding to each input workload. Then, for the training purpose, we need to find out the useful input features, i.e., the source factors which determine the dynamic delay. Intuitively, the current input workload directly affects the path sensitization. However, the preceding history input might also affect path sensitization of current cycle because the preceding input will set a state of the circuit and affect the signal transition between two cycles. In order to investigate the effect of history input workload, we use a trial-and-error process to iteratively vary the history input workload while keeping the current input fixed. We set the experiments as follows:

- Scenario 1: we only fix the current input while varying the immediate preceding input. We use this to evaluate the effect of immediately preceding input.
- Scenario 2: we fix both current and immediately preceding input while varying the preceding input of the immediately preceding one. We use this to evaluate effect of deeper history.

We perform 100K gate-level simulation to assess the effect of workload history and it turns out the dynamic delay of scenario 1 varies without irregularly while the scenario 2 results in constant dynamic delay. Therefore, we conclude that only the immediately preceding input will have effect on the dynamic delay of current cycle. This is expected as only the immediately preceding input and current input determine the signal transition which determines the path sensitization.

Next, we need to do data preprocessing to clean the training data. First of all, the decimal format of input data needs to be converted into binary vector representation. The reason behind this conversion is that, the circuit uses 32-bit vectors as input format and the 0/1 value at each bit location could affect different paths thus inferring different path sensitization behaviors. Meanwhile, the decimal format cannot precisely reflect the significance of each bit position. Therefore, we convert the decimal format to binary format. The next step is to clean the training data by removing the repetitive data patterns resulting the same delay, and excluding the cycles with a zero dynamic delay which could happen when both preceding cycle and current cycle have same input operands and no nets are toggled. These two scenarios need to be excluded to save meaningless training efforts. After preprocessing the training data, we need to extract the useful features of input vectors: history input and bit location, to train the model accurately and efficiently.

4.2.3 Model Training

Table 4.1: Five classes of dynamic delay (ps).

$500 > \text{delay} \geq 0$	$1000 > \text{delay} \geq 500$	$1500 > \text{delay} \geq 1000$	$2000 > \text{delay} \geq 1500$	$2500 > \text{delay} \geq 2000$
C_{ex_low}	C_{low}	C_{med}	C_{high}	C_{ex_high}

First, the output dynamic delay is classified into five different classes. Since 2.5ns is the clean clock period under which no timing violations occur for all of our designs, we use 500ps as step size, resulting in five different classes: C_{ex_low}, C_{low}, C_{med}, C_{high} and C_{ex_high} based on their delay range as shown in Table. 4.1. The reason of using five classes is that clock controller circuit (CGU) in DFS can only use a limited number of *phase locked loops* (PLLs) [73], each with a pre-configured fixed frequency. Empirically, we assume five PLLs are used in CGU to balance the tradeoff between frequency resolution and hardware overhead.

Then, we set the previously extracted input features, $\{x[t], x[t-1]\}$ as input feature and $C[t]$ as output labeled class, where x[t] and x[t-1] are input binary vectors at cycle t and $t-1$, and $C[t] \in \{C_{ex\_low}, C_{low}, C_{med}, C_{high}, C_{ex\_high}\}$. The input training data comes from two sources: one is from random generated data and the other is from the input operands profiled from real world applications. The input workload is then applied to gate-level simulation and dynamic timing analysis to obtain the dynamic delay value for each input workload, which is then classified into $C[t]$. With the input features and output labels, the four supervised learning methods are applied to construct a multi-classification model.

Table 4.2: Prediction accuracy, and total training and testing time of four learning methods.

method	Accuracy	Time (s)
KNN	0.932	233.35
SVM	0.975	2478.38
LR	0.974	1.82
DT	0.952	4.08

Finally, we evaluate the aforementioned four different supervised learning methods: k-nearest neighbor (**k-NN**), support vector machine (**SVM**), logistic regression (**LR**) and decision tree (**DT**) classifiers by using 50K random training data and 10k random testing data across three functional units. As shown in Table 2.1, we observe that LR is the fastest method with high prediction accuracy. DT is also fast but achieves low prediction accuracy. k-NN takes several minutes to finish with this small scale of training and testing data. Actually, when the training data size becomes 100k, k-NN takes several hours to perform classification, due to that for every given test data, k-NN needs to calculate the distance with respect to each training vector and find the nearest neighbors from the entire training space. This implies that the training size affects the classification time of k-NN. Meanwhile in LR, the size of training data does not affect classification time because LR model outputs only the weight vectors, which are then used to operate with test features. That is, the training size only affects the value of weight vectors hence the classification result, but not the classification time. Although SVM achieves highest prediction accuracy, its training and testing takes more than half an hour, which is highest among four methods. Therefore, we finally choose LR due to its high prediction accuracy and better computing efficiency. The machine learning modules are provided by Scikit-learn package written in Python [62].

4.2.4 Model Evaluation

For a given input workload, our model will predict the class to which it belongs among the five classes. We use prediction accuracy as our evaluation metric and compare this with two baseline models.

Evaluation Metric

We evaluate the prediction accuracy of prediction model by comparing prediction result with golden output generated by gate-level simulation:

$$prediction\_accuracy = \frac{\#matched\_cycles}{\#total\_cycles} \tag{4.1}$$

where $\#total\_cycles$ is the number of total simulation cycles, and $\#matched\_cycles$ is the number of cycles at which predicted result equals to golden result.

Comparison Methods

Since there are no previous works on predicting dynamic delay of functional units, we compare **`WILD`** against following baseline methods which can help us evaluate the true performance of our model:

- **rand:** predict the dynamic delay class among the non-empty classes which contain at least one instance randomly. Some classes might have no instance , for example, C_{ex_low} in Fig. 4.4, thus we ignore those empty classes.
- **naive:** use a naive class to always predict the class which contains most instances. If the dataset is heavily biased, e.g., 99% of the data belongs to one class, then even a trivial class can achieve 99% prediction accuracy by always predicting that class.

4.3 Experimental Results

In this section, we present the dynamic delay distribution of three functional units under three different input datasets. Then, we present the prediction accuracy of LR-directed model and compare with the baseline model using particular evaluation metric. Finally, we utilize LR model-directed dynamic frequency scaling (DFS) to adjust instruction-level operating frequency to achieve instruction execution speedup.

4.3.1 Experimental Setup

We choose two image processing applications from AMD APP SDK v2.5 [2], Sobel filter and Gaussian filter. The OpenCL codes of these applications are simulated by our modified version of *Multi2Sim* to profile input workloads of interested functional units. We choose 10 images in Caltech-UCSD Birds 200 vision dataset [78] as input image for these applications to profile test data. We select the operating voltage to be 0.85V and temperature to be 50°C.

4.3.2 Delay Distribution of Functional Units

We use the dynamic timing analysis described in Section 4.2.1 to investigate the dynamic timing delay of the three functional units under the datasets generated from: random data described in Section 4.2.2, Sobel filter and Gaussian filter described in Section 4.3.1. Fig. 4.2 – Fig. 4.4 present the delay distribution of functional units under three different input datasets, from which we observe several important facts.

First, for all figures, it is clearly seen that functional units exhibit noticeably different dynamic delay under different input workload. In particular, we observe up to 5X difference of delay in INT_ADD and INT_MUL. Hence, all prior works on functional units and instruction-level timing modeling that ignore the effect of input workloads suffer from inaccuracy. Second, we observe that some functional units experiences a large deviation of dynamic delay. In particular, the INT_MUL presents a non-regular delay distribution for Sobel filter and Gaussian filter while the others all present Guassian-like distribution. This is because the critical paths in INT_MUL sensitized by these two applications exhibit vastly different timing delay. Third, by observing delay behavior

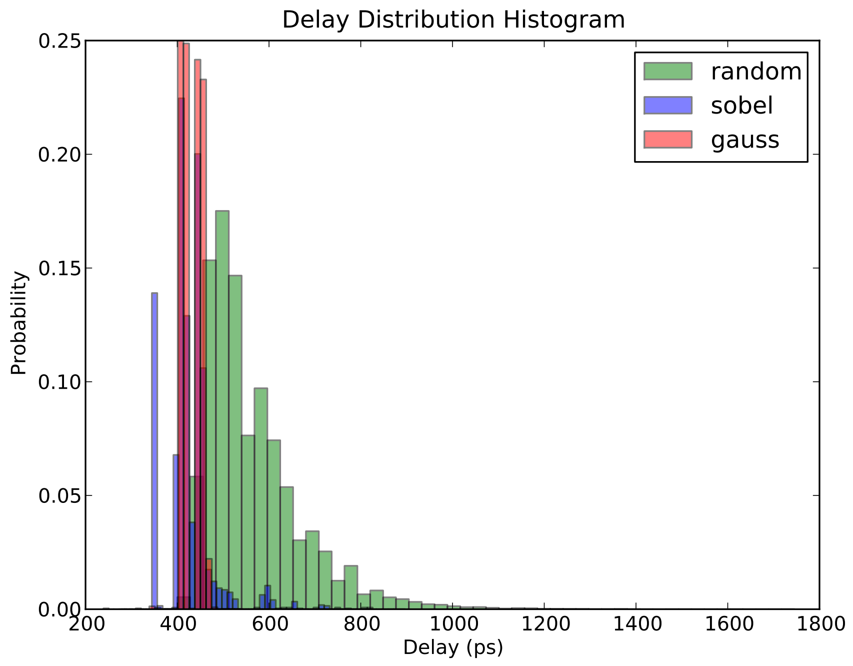


Figure 4.2: Delay distribution of INT_ADD under three different input workload sets.

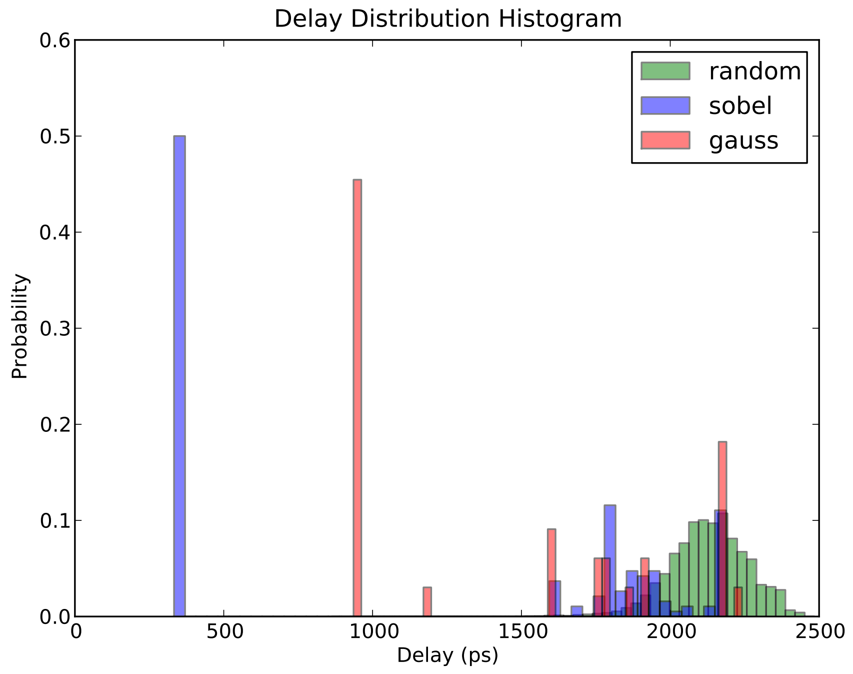


Figure 4.3: Delay distribution of INT_MUL under three different input workload sets.

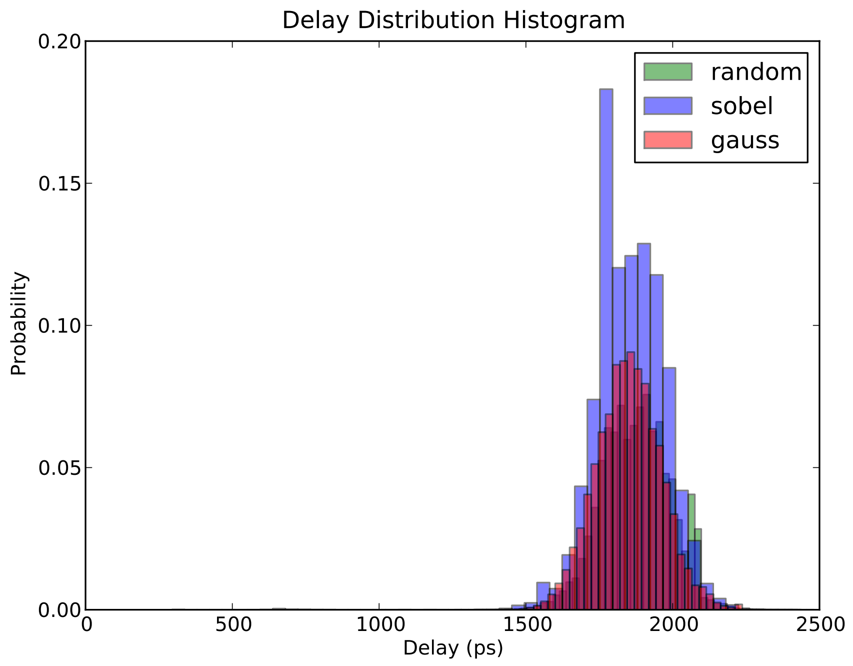


Figure 4.4: Delay distribution of FP_MUL under three different input workload sets.

resulted from each dataset individually, we find majority of them exhibits a Gaussian-like distribution. This is because some critical paths are more frequently sensitized by input workload, for that the conventional design strategies tend to produce a so-called wall of slack [51], which contains a large number of near-critical paths. In particular, the INT_ADD, for example, the sum of two highest bins are accounted nearly 50% of the delay data. Fourth, comparing with random data, we can see the delay from Sobel filter and Gaussian filter exhibit a more dense distribution. This is because there is a data locality phenomena in these applications, resulting in a high commonality in path sensitization [67]. In addition, the mean value of delay of Sobel filter and Gaussian filter are less than the one derived from random data. This is because a large number of real-world input operands for these functional units are small size operands, resulting small delays. Thus, the random dataset produced the largest delay variance and hence can be used as the most representative dataset to evaluate prediction accuracy.

4.3.3 Model Prediction Accuracy

Table. 4.3 presents the prediction accuracy of dynamic delay of three functional units under three different datasets using three models: **`WILD`**, **rand** and **naive**.

Table 4.3: Prediction accuracy of three different classifier models.

	random dataset			Sobel filter dataset			Gaussian filter dataset		
	WILD	rand	naive	**WILD**	rand	naive	**WILD**	rand	naive
Int_Add	0.974	0.333	0.319	0.966	0.330	0.954	0.998	0.334	0.999
Int_Mul	0.962	0.201	0.146	0.976	0.163	0.363	0.992	0.091	0.303
FP_Mul	0.983	0.332	0.841	0.985	0.332	0.890	0.993	0.333	0.914
Average	0.973	0.288	0.435	0.975	0.275	0.736	0.994	0.253	0.739

WILD exhibits the prediction accuracy ranging between 96.2–99.8% and achieves average prediction accuracy 98.0% over all functional units under all datasets. The **rand** model achieves average prediction accuracy at 27.2% while **naive** can achieve 63.6% on average. Thus, compared to these baseline models, **WILD** exhibits 3.6X and 1.5X higher prediction accuracy. We notice that, the prediction accuracy is affected by the delay distribution of different datasets. The prediction accuracy of Gaussian filter dataset is higher than that of random dataset. This is because the data in Gaussian filter dataset is so biased that its delay distribution is among a very small range then even a **naive** classifier can have a high prediction accuracy.

Table 4.4: Average instruction-level timing delay(ps) using **WILD** compared to existing instruction-level timing model [25].

	Sobel filter dataset			Gaussian filter dataset		
	WILD	existing	reduction	**WILD**	existing	reduction
Int_Add	521	826	33%	502	897	44%
Int_Mul	1370	2187	37%	1654	2241	26%
FP_Mul	2112	2438	13%	2057	2482	17%

4.3.4 Instruction-level Timing Margin Reduction

The existing instruction-level timing models [67] [25] measure the instruction delay under the worst case assumption of input operands. However, the instruction-level timing delay during runtime depends on the actual input workload and it can be changed from time to time. Thus, the dynamic frequency scaling (DFS) enabled by the existing instruction-level models leads to pessimistic operating timing margin. DFS is an architectural technique that adjusts operating frequency on-the-fly to improve performance. To address such problem, we use **WILD**-directed DFS to enable a finer-grained frequency adjustment as it can provide an specific delay value for a given input workload. As a result, the circuit can run at a higher frequency compared to existing model-directed DFS.

For a given instruction, the existing models uses the worst case instruction-level timing delay measured to set the operating frequency which will be used in DFS, e.g., 826ps, 2187ps and 2438ps for Int_ADD, Int_MUL and FP_MUL in Sobel filter as shown in Fig. 4.2–Fig. 4.4. On the other hand, **WILD** incorporates the input workload with the instruction and predicts the class of the resulted dynamic delay and uses the upper bound of that class to set the operating frequency to provide a safe operating frequency. For example, if the predicted class is C_{ex_low}, then the DFS will use 500ps as clock period to execute the instruction. However, the predicted class could be wrong, which could fall into one of the two categories:

- False positive: this will still ensure the circuit safety but incurs performance penalty because it uses larger clock period than needed.
- False negative: this will result in timing violation in the circuit. We assume the *detection-and-correction* is used here to correct the timing violations by using *instruction replay* [27], which will flush the pipeline.

We calculate the average timing delay of an instruction as:

$$average\_delay_{inst} = \frac{total\_running\_time_{inst}}{\#inst\_running} \tag{4.2}$$

where the $total\_running\_time_{inst}$ is the sum of running clock period of each instruction instance in the application execution, including the false positive and false negative induced cycle penalty, and $\#inst\_running$ is the total number of instruction instances executed in the application. We compare the **WILD**-directed DFS with the DFS directed by existing instruction-level model [25] in Table. 4.4. We observe that by using **WILD** directed DFS, the average instruction-level timing delay can be reduced 13–44%

compared to the existing model. This timing margin reduction can be further utilized online with DFS to accelerate the program execution.

4.4 Discussion

Implementation of DFS: Dynamic frequency scaling (DFS) has been used to adjust operating frequency according to real-time circuit delay during runtime to improve performance [25, 63, 64], implemented by a clock generation unit (CGU). CGU is a fast adaptive clock controller circuit that can be designed using a couple of phase-locked loop (PLL) circuits, each of which is running at a fixed frequency independently, and a multiplexer is used to select one specific frequency within a single cycle [73]. Moreover, a compact all-digital phase-locked loop (ADPLL) clock generator has been proposed in [43] with a ultra-low overhead in power and area, which can provide frequency switching arbitrarily in a wide range within a single clock cycle. However in reality, the frequency can only scale in discrete steps, so the operating frequency can only be selected within a fixed number of options. We consider five different classes of dynamic delay of a circuit to adjust the operating frequency, which can then be implemented by a 5-PLL CGU design. More details of DFS implementation can refer to [64, 73, 81].

Overhead of `WILD`: Although the on-chip model can aid the DFS to achieve better performance, the hardware overhead of implementing the learning model does need special care. An on-chip Gaussian-kernel based SVM is proposed using an analog circuit [52]. Recently, a voltage-droop induced circuit delay prediction model has been implemented using SVM to augment online DFS, whose hardware overhead is 1.5% for today's processor design [81]. We expect the overhead of **`WILD`** model is less than SVM since LR is less complex than SVM. In our experiment, the SVM classification time is more than 100X of LR model. Besides, the **`WILD`** model runs remarkably faster than gate-level simulation. To compute 200k test data for dynamic delay, **`WILD`** is 60X faster than gate-level simulation. Our future work focuses on developing machine learning model that has more efficient computing time and higher prediction accuracy.

4.5 Chapter Summary

This chapter presents an approach to proactively prevent timing errors using dynamic frequency scaling. We construct **`WILD`**, a workload-based supervised learning model to predict the dynamic delay of functional units for a given input workload. To calibrate its effectiveness, we perform dynamic timing analysis on a post-layout netlist and extract useful "features" that affect the circuit dynamic path sensitization, and hence the dynamic delay. The model is trained using logistic regression with training data from random generation and application profiling, and tested using unseen data from three different datasets. Across three functional unit types and three datasets, **`WILD`** exhibits high prediction accuracy up to 99.8% (average 98.0%) and achieves a prediction accuracy of 3.6X and 1.5X higher compared to two baseline models. Compared with the state-of-art instruction-level timing model enabled dynamic frequency scaling (DFS), with **`WILD`**-directed DFS, the average instruction-level timing delay could be reduced by 13%–44%.

Chapter 4 contains reprints of Xun Jiao, Yu Jiang, Abbas Rahimi, and Rajesh Gupta, "WILD: A Workload-Based Learning Model to Predict Dynamic Delay of Functional Units", *Proc. IEEE International Conference on Computer Design (ICCD)*, 2016. The dissertation author is the primary author of the paper.

Chapter 5

Application Vulnerability Assessment to Timing Errors

In Chapter 2, Chapter 3, and Chapter 4, we propose methods to combat the challenges associated with the **error prevention** and **error correction** approaches. Beginning with this chapter, we focus on challenges associated with **error acceptance** approaches. In particular, we consider methods to combat the **unacceptable application quality**. The error acceptance approaches typically utilize the intrinsic error tolerance of applications to allow occasional error occurrences in the system to trade for improved operational efficiency. Although applications such as neural networks exhibit intrinsic error tolerance, we need to examine carefully if the resulting application quality is acceptable against competing less expensive algorithmic solutions compared to gains from designing approximate hardware or relaxing the operating constraints. It is thus important to expose the hardware errors due to approximation to software execution. In this chapter, for the applications of neural networks, we propose a cross-layer approach to examine the hardware errors under relaxed operating conditions jointly with the quality of inference by the neural networks to assess their vulnerability to timing errors.

5.1 Introduction

As a problem solving method, neural network algorithms have found use in a wide range of applications such as medical diagnostics [80], image classification [55], speech recognition [42], and natural language processing [24]. This versatility has led to their implementation on a variety of hardware platforms: GPU [19], FPGA [41], and ASIC [18].

Due to the ability to adapt their learning parameters, neural networks have an inherent resilience to errors. Thus, one would expect that the quality of results produced by hardware neural networks (HNNs) to be relatively insensitive to the rising timing error rates caused by increased variation, thus opening doors for opportunistic reduction of guardbands to increase the operational efficiency of hardware. There is a need for a quantitative assessment here to explore the extent to which guardbands can be reduced in HNNs. In this chapter, we investigate this question as to whether and how much accuracy of HNNs could be affected by dynamic variations. To do this, we capture and represent variations from low-level hardware, and then expose them to neural networks inferences.

Approach and Contributions: We propose a cross-layer approach to assess the vulnerability of HNNs to dynamic voltage and temperature variations, in which we extract the timing errors from hardware layer using gate-level simulations and examine their effects in the software layer using error injections. To evaluate the soundness of this approach, we measure the timing errors using gate-level simulations of post-layout circuits in TSMC 45nm technology. We vary the voltage and temperature in a wide range to examine the effects of variations. Then, we represent and inject these timing errors to neural networks during their inference. Finally, we examine the resilience of two neural networks, MLP and CNN, by testing them on MNIST dataset [56].

We make the following contributions:

- We extract the circuit level timing errors caused by voltage and temperature variations from twenty different operating conditions using gate-level simulations.
- We inject such timing errors back into neural network inference and evaluate the accuracy on MNIST dataset at different conditions.
- Using two frequently used neural networks (MLP and CNN), our results show that variations can significantly affect the inference accuracy.

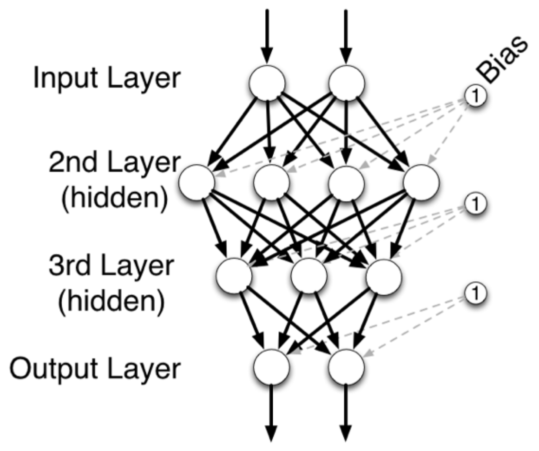


Figure 5.1: An example of 4-layer multi-layer perceptron neural network.

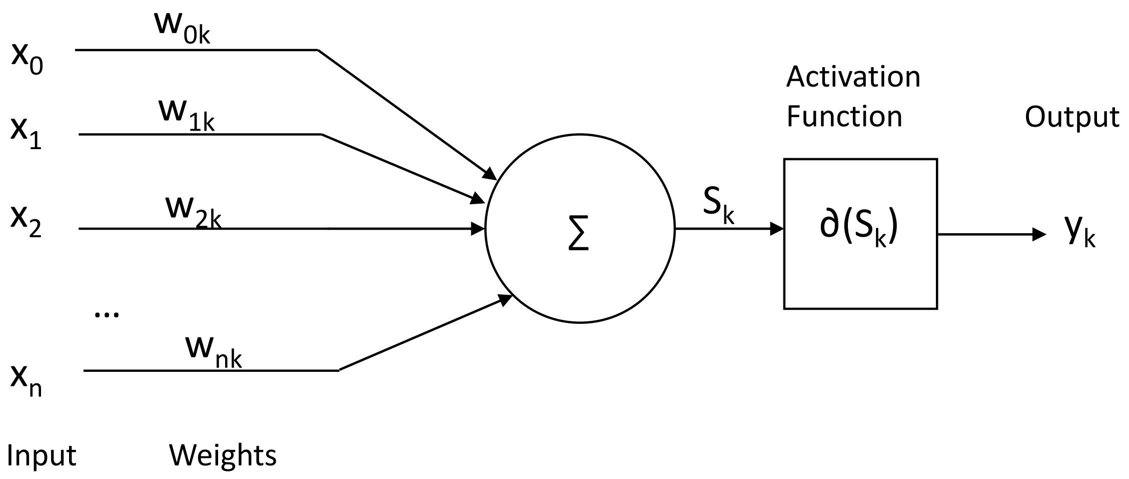


Figure 5.2: The computation processes of an artificial neuron.

5.2 Background

Modeled after biological neuronal processing, neural networks are a family of problem-solving methods in machine learning. Among different types of neural networks, two of them are most widely used: multi-layer perceptron (MLP) and convolutional neural network (CNN).

5.2.1 MLP Architecture

Fig. 5.1 depicts a typical MLP architecture. As one of the simplest neural network model, MLP has one input layer, one output layer, and several hidden layers. All

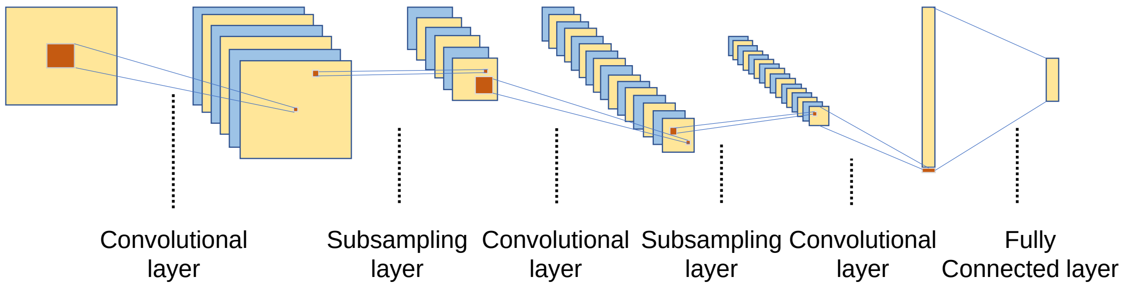


Figure 5.3: An illustration of a convolutional neural network.

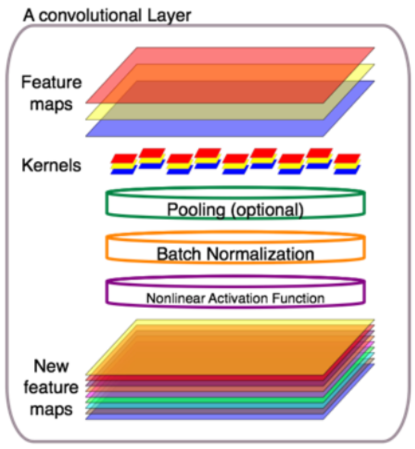


Figure 5.4: The processes among a convolutional layer.

layers except the input layer are composed of artificial neurons that perform the basic computations as illustrated in Fig. 5.2. A typical neuron performs a linear processing part followed by a non-linear processing part. In the linear processing part, inputs are multiplied with corresponding weights, and then all products are accumulated. In the non-linear processing part, an activation function is applied to the weighted sum. Common activation functions include logistic sigmoid, hyperbolic tangent, or rectilinear unit, whose purpose is to enable a neural network to be a universal function approximator [39]. The activation function is usually implemented by a lookup table in hardware [31]. Finally, the output y_k of neuron k is computed as $y_k = \delta(\sum_{j=1}^{n} x_j w_{jk} - \theta)$, where x_j is the j^{th} input, w_{jk} is the synaptic weight connecting j^{th} input and neuron k, θ is the bias, and δ is the activation function.

5.2.2 CNN Architecture

Recently, CNNs have grown in popularity in various applications such as image/video recognition due to its better performance. A CNN applies convolution operations to a restricted part of the input data for each neuron in the convolutional layer. A typical CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers can be either convolutional, pooling or fully connected. Fig. 5.3 depicts a typical CNN architecture that consists of six layers, where the first, third, and fifth layer are convolutional, while the second, fourth are pooling layers, and the sixth layer is a fully connected layer. Fig. 5.4 depicts the internal processes in a convolutional layer with 9 kernels, each of which consists of three filters. The convolution operation models the hardwired bonding between the neurons on adjacent layers. It uses a sliding filter to perform dot-products of the filter and uses a portion of the input image to generate an output image, namely the feature map. Since the convolution operations are differentiable, the filters can be trained to capture the features of the input images with backward propagation [68]. *Pooling* is used to reduce the size of a feature map by selecting the maximum pixel strength or averaging several pixel strengths. It benefits the transformation invariance because it drops unnecessary minor information and preserves the most dominant features for the overall classification task.

5.2.3 Error Tolerance of Neural Networks

The robustness of a neural network comes from many aspects. From a higher level point of view, the training process of a neural network model is simply an ensemble of multiple linear or logistic regressions working in parallel. The regression itself ignores minor noises of the data and yields a model for the most likely distribution of the given data. The regularization process inside a neural network also contributes to robustness because no matter how deterministically penalties on weights are added or how stochastically certain partials of the model are dropped, the weights are trained to accommodate the majority of the data with a simplest probable distribution.

Hardware variations could impact HNNs through timing errors in both computation logic and control logic. The errors in control logic could lead to catastrophic results but fortunately, most critical paths lie in computation logic, which is mainly composed of additions and multiplications, two of the most frequently used operations. Both the forward and backward propagation require intensive additions and multiplications. Thus, we mainly focus on the timing errors that occur in addition and multiplication.

5.3 Cross-layer Vulnerability Assessment

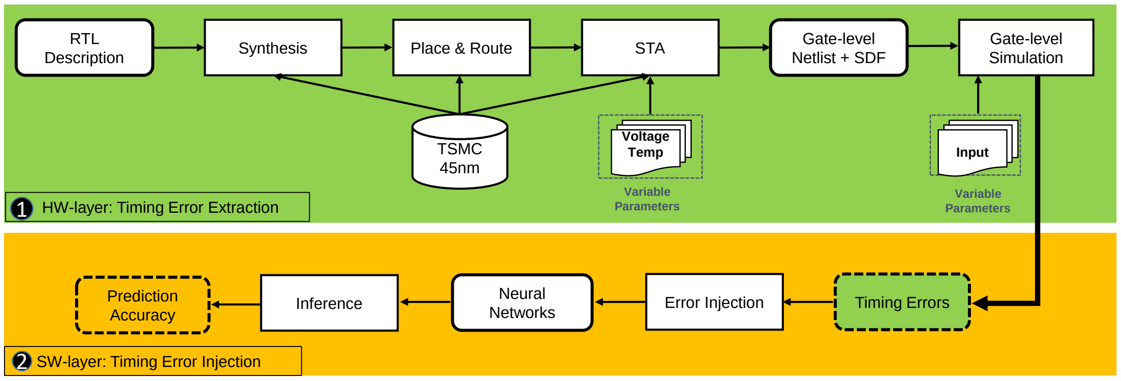


Figure 5.5: Cross-layer assessment flow with two stages: a) HW-layer: Timing Error Extraction to extract the timing errors under different operating conditions; b) SW-layer: Timing Error injection into neural network and perform inference.

The cross-layer vulnerability assessment is comprised of two phases as shown in Fig. 5.5: *Timing Error Extraction and Timing Error Injection*. a) The *Timing Error Extraction* phase implements the standard ASIC flow and uses gate-level simulation to generate timing errors at each operating condition. b) In the *Timing Error Injection* phase, we inject the timing errors into neural networks and then perform inference. We vary the neural network genres and operating conditions to examine the resulted accuracy. More details about the two phases are illustrated as follows.

5.3.1 HW-layer: Timing Error Extraction

We extract the timing errors through *Timing Error Extraction* module as illustrated in Fig. 5.5, which is divided into several steps. Note that we focus on dynamic variation-induced timing errors of computation units. We extract timing errors from the adder and the multiplier, which are the two most frequently used computation units in neural networks computation. We use FloPoCo [28] to generate the synthesizable VHDL codes of floating point units. We use *Synopsys Design Compiler* to synthesize the Verilog codes and use *Synopsys IC Compiler* to generate post place-and-route netlist in TSMC 45nm technology. Next, we use *Synopsys PrimeTime* to perform static timing analysis, generating Standard Delay Format (SDF) files at different operating conditions. To do this, we use the voltage temperature scaling features of Synopsys PrimeTime for the composite current source approach of modeling cell behavior. We consider twenty operating conditions as shown in Fig. 5.9, which could introduce both mild and aggressive timing errors. Then, we use *Mentor Graphics ModelSim* to do SDF back-annotation gate-level simulations under nominal frequency to generate output data at different operating conditions. To extract timing errors, we compare the gate-level simulation output $y[t]$ with a pure-RTL simulation result $y\_gold[t]$, which is free from timing errors because there is no delay annotation. If there is a mismatch, then we define it as a timing error.

5.3.2 SW-layer: Timing Error Injection

We inject the timing errors extracted by the *Timing Error Extraction* phase to the neural networks by using second phase *Timing Error Injection*. During the forward propagation in the neural network inference, we inject the errors into the computations (addition and multiplication). For a circuit, different input could excite different paths, resulting in an input-specific timing error behavior. To mimic this, an exhaustive look-up table containing the entire input space for each bit position of each computation unit under all operating conditions needs to be implemented. Then, the computations need to look up the table to check whether it has a match on any input operands in the input space. This makes the inference process prohibitively slow. To approximate the situation, we inject the timing errors as [69]: let the computation units return a random value each time they have timing errors. We inject the error into the computation with a TER extracted from the *Timing Error Extraction* phase to mimic the time error behavior. For example, if adder has a TER at 10%, we inject errors to 10% of the total additions. This probability is determined by operating conditions and computation logic (addition or multiplication), which can represent the impact of timing errors on computation logic.

We vary the error injection probability for each operating condition.

5.4 Experimental Results

In this section, we measure timing errors under twenty operating conditions. Then, we measure HNNs accuracy as a function of varying timing error rates. Finally, we characterize the HNNs accuracy under dynamic variations using MLP and CNN.

5.4.1 Experimental Setups

We use tiny-dnn [4], a header only, dependency free deep learning library written in C++, as our deep learning platform. This platform is light weighted, and is designed for deep learning on limited computational resource, such as embedded systems and IoT devices. For CNN, we use LeNet-5 like architecture and replace LeNet-5's RBF layer with normal fully-connected layer. For MLP, we use 3-layer MLP with a hidden layer of 60 neurons. We use MNIST (Mixed National Institute of Standards and Technology) database of handwritten numbers [56] as our dataset to evaluate the neural network accuracy. This dataset is a well-known dataset for evaluating the performance of neural network classifiers. The dataset is split into training set and test set with 60,000 and 10,000 28×28 images. We vary the voltage from 0.81V to 0.90V with a step at 0.01V and the temperature from 50°C to 100°C.

5.4.2 Accuracy under Timing Errors

We assess the accuracy for both MLP and CNN under the TER at 0, 0.00001, 0.0001, 0.001, 0.01, 0.1, 0.5, and 0.9 at three configurations as shown in Fig. 5.6 and Fig. 5.7; *add_only* means we only inject timing errors to adder, *mul_only* means we only inject timing errors to multiplier and *both* means we inject errors to adder and multiplier at the same time. We observe that for both MLP and CNN, as the TER increases, the accuracy drops monotonically. When the TER is 0.00001, the HNN can still deliver a decent accuracy close to original accuracy. Once the TER of adder reaches 0.0001, the accuracy drops to around 90% and continue dropping to 60% when the TER of adder

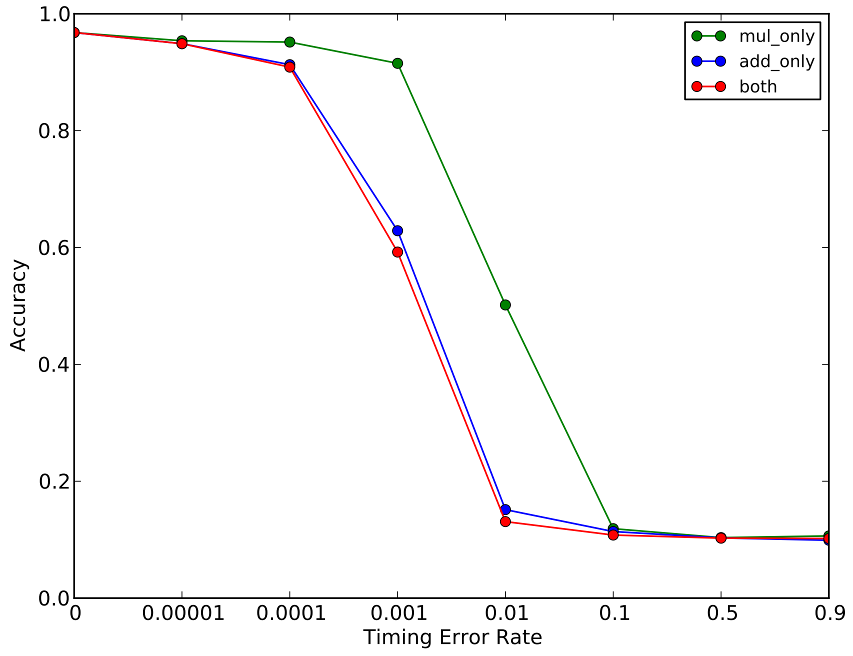


Figure 5.6: MLP accuracy as a function of TER.

reaches 0.001. In contrast, the multiplier exhibits much less significant impact on HNN accuracy: the HNN can still deliver 90% accuracy even when the TER of multiplier reaches 0.001. In fact, for all examined TERs, the *mul_only* resulted accuracy is always higher than that of *add_only*. Moreover, the accuracy under *both* configuration is almost identical to that of *add_only* configuration, suggesting that adders-induced errors contribute to most of the accuracy drop. One main reason for this is that the accumulated convolution sum or dot-product sum are fed into a nonlinear activation function. The errors from multipliers will be averaged, but the errors from adders directly impact the input of the activation function. This suggests that more hardware design effort should be made on the adder to ensure its low TER. On the other hand, the worst accuracy of both NN genres is around 10%, when either *add_only* or *mul_only* is 0.1. We can observe that such an accuracy drop starts saturating at 0.1 TER, almost identical to a random guess, and stays almost the same when TER continues increasing. Such observations show that even though neural networks have inherent error resilience, the timing errors still can significantly affect neural network accuracy and motivate this approach.

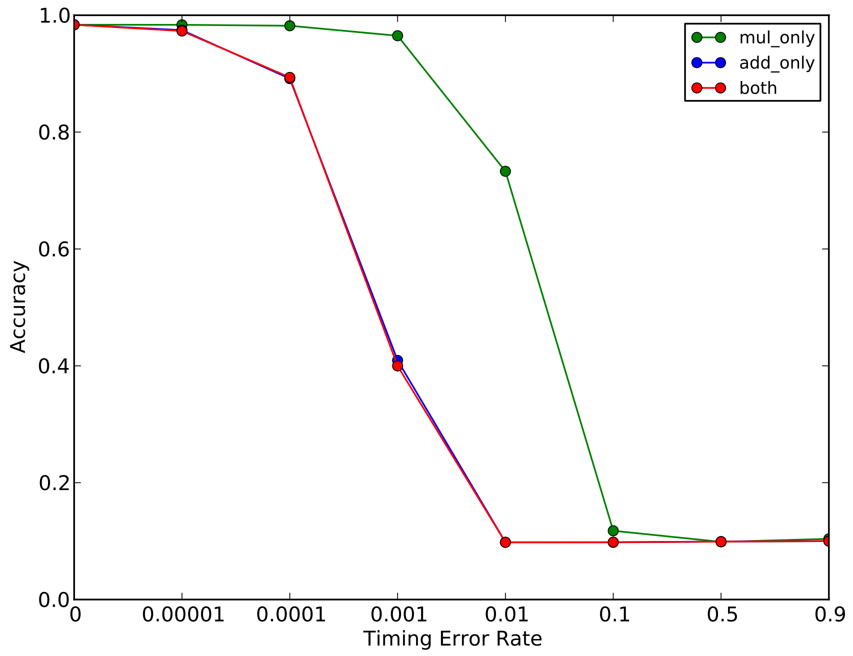


Figure 5.7: CNN accuracy as a function of TER.

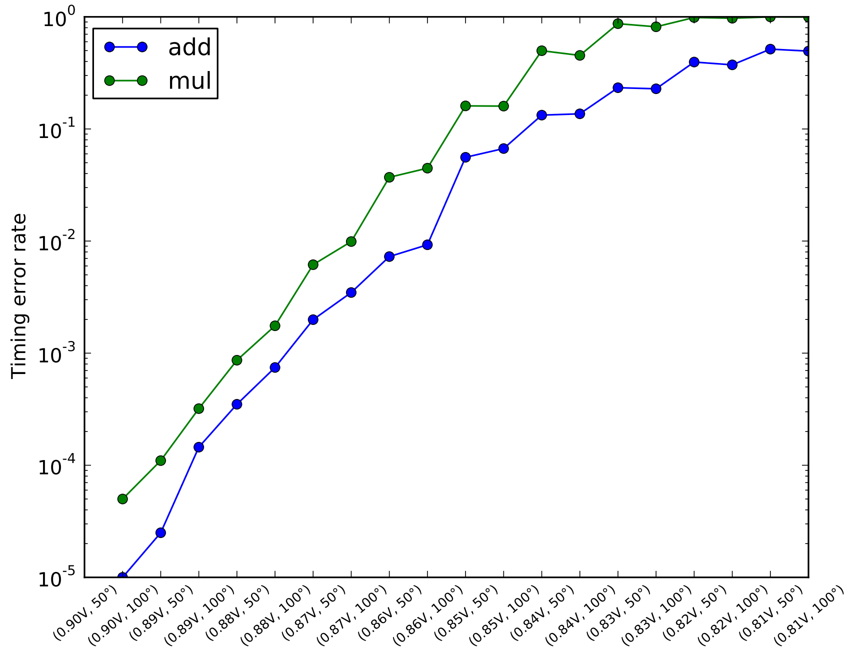


Figure 5.8: TER of adder and multiplier under different operating conditions.

Table 5.1: HNN accuracy under dynamic variations.

HNN	(0.90V, 50°C)	(0.90V, 100°C)	(0.89V, 50°C)	(0.89V, 100°C)	(0.88V, 50°C)	(0.88V, 100°C)
MLP	96.79%	96.03%	94.90%	87.93%	75.56%	57.76%
CNN	98.37%	97.31%	95.87%	85.15%	70.34%	48.64%
HNN	(0.87V, 50°C)	(0.87V, 100°C)	(0.86V, 50°C)	(0.86V, 100°C)	(0.85V, 50°C)	(0.86V, 100°C)
MLP	25.67%	15.89%	10.45%	10.33%	9.42%	9.91%
CNN	18.85%	11.13%	9.81%	9.80%	9.81%	9.81%
HNN	(0.85V, 50°C)	(0.85V, 100°C)	(0.84V, 50°C)	(0.84V, 100°C)	(0.83V, 50°C)	(0.83V, 100°C)
MLP	9.89%	9.80%	9.72%	9.60%	10.15%	9.60%
CNN	9.75%	9.81%	9.89%	9.80%	9.91%	9.84%

5.4.3 Accuracy Versus Dynamic Variations

We then use the real dynamic operating conditions to obtain realistic timing error rates and thereby characterize the vulnerability of HNNs to dynamic variations.

First, we use the *Timing Error Extraction* described in Section 5.3.1 to characterize the timing error behavior of 32-bit floating point adder and multiplier under different operating conditions as shown in Fig. 5.8. The selected operating conditions cover a wide range of TERs: at the best condition (0.90V, 50°C), no timing errors are injected for both computations; at the worst condition (0.81V, 50°C), 50% and 100% TER are found in adders and multipliers respectively. The TER of adder reaches 0.01 when the operating condition is around 0.86V. Based on Fig. 5.6 and Fig. 5.7, the accuracy drop starts to saturate when the TER of adder reaches 0.01, thus we expect to see a worst accuracy starting at around 0.86V.

We then present the accuracy of both MLP and CNN under twenty operating conditions, as shown in Fig. 5.9 and Table. 5.1, where we observe several important facts. First, the lowest accuracy under worst-case operating conditions is around 10%. By looking into the prediction results, we found CNN is able to identify more than 90% of the *0* digits even under worst condition.

Second, the 10% accuracy has been observed across multiple conditions from (0.86V, 50°C) to (0.81V, 100°C). (For better space utilization, we do not present the accuracy under 0.81V and 0.82V in Table. 5.1, where the accuracy of both is around 10%.) This is expected as we can see from Fig. 5.6 and Fig. 5.7 where the accuracy drops to 10% when the TER of either unit reaches 0.1.

Third, Table. 5.1 shows that under the condition between (0.86V, 50°C) and (0.90V, 100°C), where the TER of adder is less than 0.01, the accuracy drop of MLP to its original accuracy is less than that of CNN, indicating MLP might be more resilient than CNN within a certain TER. Part of the reason for this is that given the same TER, the amount of errors in CNN is larger than MLP because CNN has more arithmetic operations.

Last but not least, we find the voltage and temperature both play an important role in determining the inference accuracy. By fixing the temperature at 100°C, reducing the voltage by 0.01V from 0.89V to 0.88V results an accuracy drop from 85.15% to 48.64%; by fixing the voltage as 0.88V, increasing the temperature by 50°C results an accuracy drop from 70.34% to 48.64%. By comparing the accuracy at (0.90V, 50°C) and (0.86V, 50°C), we find the accuracy drops to worst case at around 10% from best case at around 98% by a voltage reduction of 0.04V. Such observations indicate there is a huge impact of dynamic variations on hardware neural networks accuracy and motivate the necessity of protecting HNN against variations.

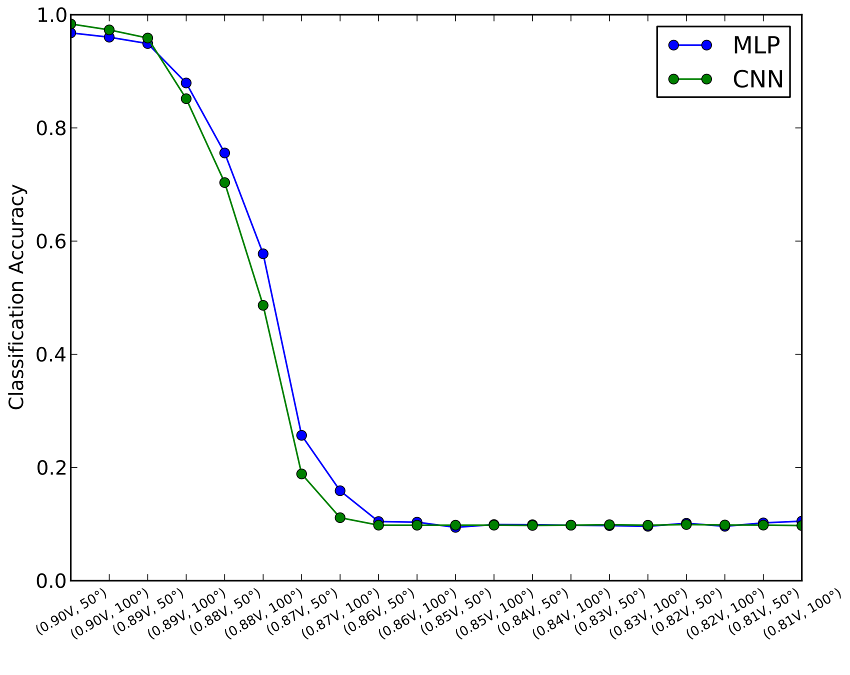


Figure 5.9: HNN accuracy as a function of dynamic variations.

5.5 Discussion

Threats to Validity: We mainly focus on variation-induced timing errors in computation logic. However, the timing errors could also occur in control logic, which might lead to more severe accuracy drop or malfunction. Fortunately, it was observed that control logic only contributes a small set of critical paths [76], making it less vulnerable to timing errors.

Future Work: We focus on assessing the effects of hardware variations on neural network performance. The next question is how we can mitigate such timing errors. For the future work, we focus on integrating the timing errors as a vector for backpropagation to enable an adaptive training method. Moreover, we plan to design a reconfigurable architecture that can automatically select suitable weights for a given voltage and temperature from a set of pre-stored weights.

5.6 Chapter Summary

This chapter presents a cross-layer approach to assess the effects of dynamic voltage and temperature variations on the performance of hardware neural networks. We first extract the timing errors of post place-and-route computation units under twenty different operating conditions through gate-level simulations. We then inject such errors to neural network inference phase and evaluate the resulted accuracy. Using two frequently used neural networks, MLP and CNN, we demonstrate that dynamic voltage and temperature variations can cause significant drop in inference accuracy.

Chapter 5 contains reprints of Xun Jiao, Mulong Luo, Jeng-Hau Lin, and Rajesh Gupta, "An Assessment of Vulnerability of Hardware Neural Networks to Dynamic Voltage and Temperature Variations", *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017. The dissertation author is the primary author of the paper.

Chapter 6

Enhancing Efficiency via Approximate Computation Reuse

In the last chapter, to deliver an **acceptable application quality** associated with the **error acceptance** approaches, we assess the quality of neural networks by exposing the timing errors caused by relaxed operating condition to software execution. The results show that it is unrealistic to save energy by lowering the voltage because small voltage drop can lead to the unacceptable quality of results. In this chapter, for the applications of neural networks, we propose a controllable and reconfigurable hardware approximation approach that can guarantee acceptable application quality while achieving significant energy savings.

6.1 Introduction

Energy consumption is an important metric for neural networks implementation in an increasingly broad range of computing platforms. Neural network computations are dominated by additions and multiplications. Due to their cost and latency, multiplications have been a natural target for optimization in hardware. **Approximate computing** has been used in recent literature to address the intensive computational workloads thus creating a tradeoff between accuracy and energy [31, 60]. Approximate computation units have been shown to have better energy efficiency than the exact ones [46]. For instance, in [31], the authors substitute the normal multipliers with inexact multipliers that provide inexact logic but with less hardware cost. Mrazek *et al.* further optimize approximate multiplier design with a uniform structure suitable for hardware implementation [60].

While the adaptability of neural networks in its applications is naturally suited to use approximation [48], in practice it also requires *retraining* the network to mitigate accuracy loss caused by logic errors from inexact design. Moreover, once the design has been physically implemented in hardware, it is not possible to reconfigure the design to control the approximation level entirely in hardware. Thus, it is hard to guarantee the quality of results after applying such hardware approximation design.

To overcome these limitations, we propose using a reconfigurable and controllable approximation technique in neural networks by exploiting the computation reuse opportunities. Computation reuse has been adopted in various applications where *value locality* and *similarity* are observed [65]. To enable computation reuse, we rely on tight integration of Bloom filters (BFs) with the computation units in hardware, a data structure that supports approximate set membership queries with a tunable rate of errors to store frequent computation patterns and return the results without actual execution of energy-intensive float point units (FPUs).

To ensure the effectiveness of computation reuse using Bloom Filters, we use a set of techniques. First, we perform approximate pattern matching instead of exact pattern matching in neural networks. This is done in the context of arithmetic operations on floating point numbers. We thus explore matching operations under limited precision of operands. This is done via a reconfigurable BF architecture that can do approximate pattern matching with hashing for data items that feature varying bit width. Second, we perform layer-based pattern matching instead of global pattern matching. That is, we detect and store different set of input patterns for each layer separately. The reason is that in neural networks, each layer has its own set of functions thus may experience different input workloads. Accordingly, we configure BFs for each layer separately. Third, we implement the BFs with resistive memory elements to provide energy efficient storage for saving the frequently used patterns [5].

We make the following contributions:

- We explore and use computation reuse opportunities in neural networks and enhance them with layer-based approximate pattern matching.

- We design a reconfigurable Bloom filter unit that can perform approximate pattern matching, increasing the computation reuse opportunities while leading to a controllable approximation level for neural networks.

- We demonstrate the effectiveness of the approximate BFs by reducing 47.5% energy consumption of multiplication operations in 45nm technology while incurring only 1% accuracy degradation.

6.2 Approximate Computation Reuse

In this section, we explore the computation reuse opportunities in neural networks that can avoid the energy overhead due to re-execution, and propose optimization techniques that improve the reuse opportunities. Based on this, we propose a novel architecture that can enable flexible control over computation reuse.

6.2.1 Layer-based Pattern Matching

To maximize the energy savings, we need to maximize the computation reuse opportunities. Since we need to store a set of pre-calculated computations, we aim to store most frequent input patterns to maximize the computation reuse opportunities. To do this, we use several steps. First, we profile the input operands of multiplications using some training input. Second, in the profiled input, we look for the most frequent input patterns and calculate their results.

In this process, we use two strategies to look for the most frequent input patterns: *global-based* and *layer-based*. *Global-based* means we look for the most frequent input patterns from all the multiplication operations in neural network inferences, regardless of their locations. *Layer-based* means we look for the frequent input patterns for each layer separately. That is, for each layer, we find the most frequent patterns from the input operands of multiplications profiled from that specific layer. For example, to find the most frequent patterns for the third convolutional layer, we profile all input operands

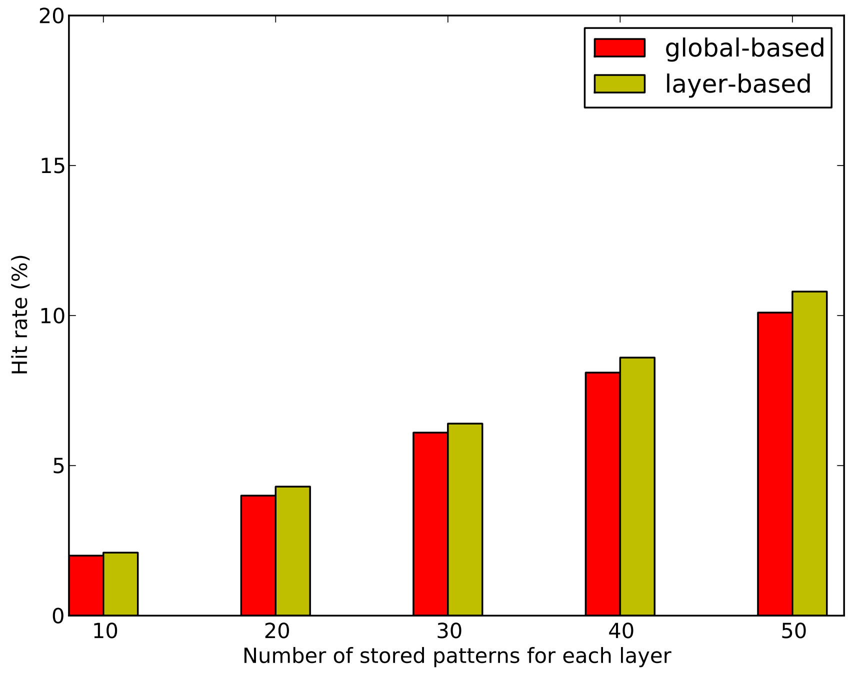


Figure 6.1: The hit rate of exact pattern matching.

of multiplications in that layer and find the most frequent patterns in this set of input operands.

Third, we then check the hit rate of the chosen frequent patterns using another set of data. We also vary the number of stored input patterns for each layer. Note that, for the sake of simplicity, we always use the same number of stored patterns for each layer. As shown in Fig. 6.1, we can see that layer-based matching leads to higher hit rate than global-based matching. From now on, we conduct all of our experiments using *layer-based* approach. We also observe that as the number of stored patterns increases, the hit rate also increases. However, the hit rate still remains low, at around 10%, even if we store 50 patterns for each layer. Thus, we improve the hit rate by developing approximation techniques as described in the next section.

6.2.2 Approximate Pattern Matching

As shown in Fig. 6.1, even if we use layer-based pattern matching, the hit rate is still low. Thus, we propose the use of approximate pattern matching for floating point numbers instead of exact matching, i.e., we only match for limited bit width. For example, there are two floating point numbers 0.45 and 0.451, with their IEEE 754 format as 00111110111001100110011001100110 and 00111110111001101110100101111001. If we use exact matching, then 0.451 would not match 0.45. However, if we use 9-bit matching, then 0.451 would match 0.45-because their first 9 bits (sign bit and exponential bits) match. In this case, their first 16 bits (sign bit, exponential bits and 7 mantissa bits) are identical so they will match even under *16-bit matching* mode. We use four different approximate matching modes to measure the hit rate: 9-bit, 10-bit, 11-bit, and exact matching, as illustrated in Fig. 6.2. We can see that as we increase the approximation level, the hit rate also increases significantly even by 1 bit. For example, by storing 50 patterns (for each layer), a 10-bit approximation can have hit rate at 57.1% while 9-bit approximation can have hit rate at 82.6%, which is 56% higher.

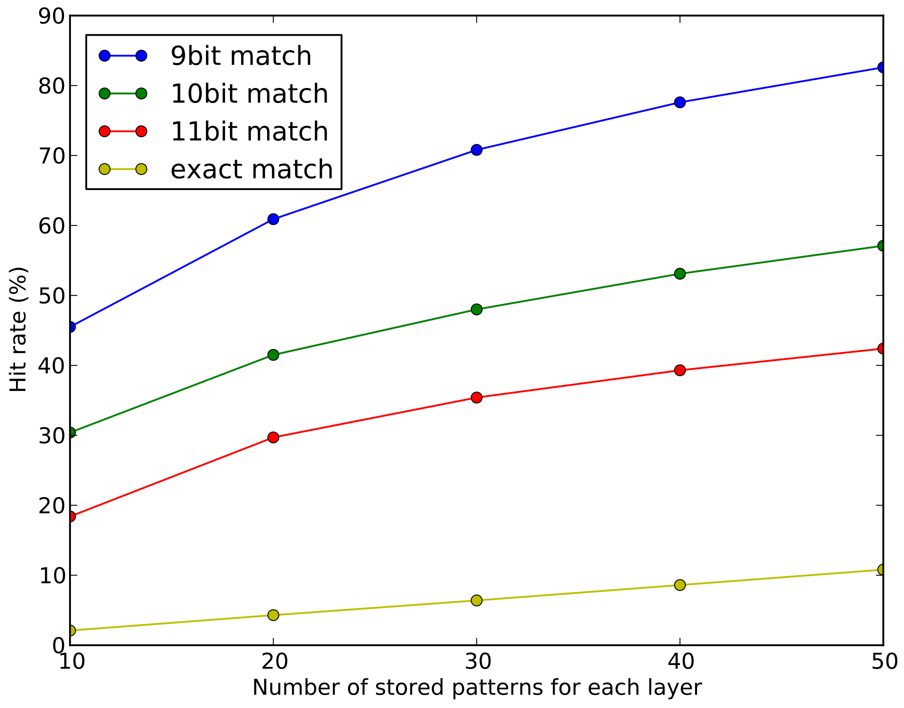


Figure 6.2: The hit rate of approximate pattern matching.

But note that the increased hit rate does come with a cost. Rather than returning the exact computation result, the approximate pattern matching will return an inexact result. And as we increase the approximation level, the extent of inaccuracy will also increase. We explore several different approximate matching modes in Section 6.4.2 by varying the matching bit width and number of stored patterns to maximize the energy savings while keeping the accuracy loss minimal.

6.3 Bloom Filters

To implement approximate pattern matching at the hardware level, we employ BFs. The detected frequent patterns are stored in a set of BFs, and the BFs are integrated to the multiplier. The number of BFs equals to the number of distinct output values generated by the frequent patterns. Each BF stores the patterns corresponding to its assigned output value.

BFs are known as compact storage units that provide an approximate response to the membership queries. A BF consists of a number of hash functions and a Bloom vector (BV). To store a set of inputs in the BF, the hash functions are executed for each input generating addresses to the BV. For each input, the corresponding bits of the BV determined by the hash functions are set to 1. To search for a given input in the BF, the same hash functions generate addresses for an incoming input, and the corresponding bits are checked in the BV. If all the bits are 1, the input is stored in the BF. Otherwise, the input does not exist. The overall architecture of using BF for approximate pattern

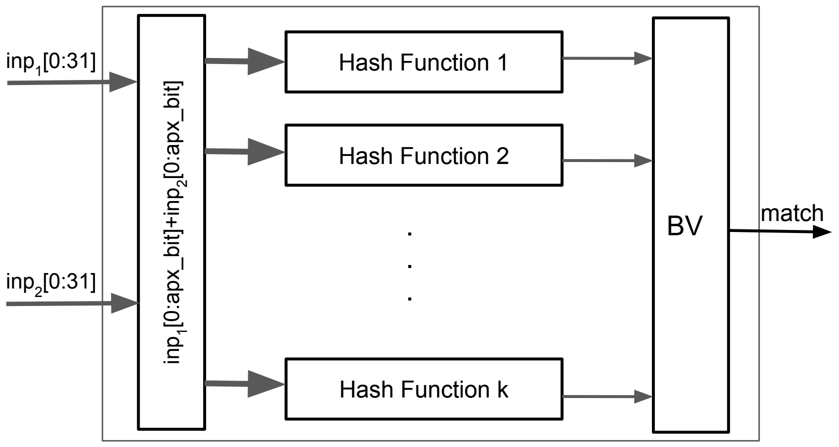


Figure 6.3: The implementation of approximate pattern matching.

matching is shown in figure 6.3. In order to enable the approximate pattern matching, we store approximated input patterns in the BF. We resize each input of the multiplier to *apx_bit* bits by selecting its *apx_bit* most significant bits and concatenate them into a single vector. The obtained vector forms an input to the hash functions which determine bits to be set in the BFs. Similarly, to investigate the approximate pattern matching of incoming inputs to the multiplier, the inputs are re-sized and concatenated before going to the hash functions. Then, the bits in the BV specified by the hash functions determine whether the incoming pattern of the multiplier is matched or not. In case of matching, the multiplier is clock-gated to avoid the re-execution and the output in the register corresponding to the BF is returned as the output of the multiplier.

Due to the characteristics of hash functions and the limited size of the BV, BF has a false positive (FP) error where BF wrongly confirms the presence of an input. However, the rate of the false positive, which is shown in equation 6.1, is dependent on several parameters such as the number of input operands stored in the BF (*n*), the number of hash functions (*k*) and the size of the BV (*m*) [29] [7]. Therefore, FP can be tuned by properly setting the mentioned parameters.

$$FP = (1 - e^{-\frac{nk}{m}})^k \tag{6.1}$$

Since most of today's applications such as neural networks demonstrate tolerance to the controlled imprecision in computations, BFs are adapted to implement approximate pattern matching and recall the computations in neural networks. To further improve the energy consumption of the computations, we employ resistive memory elements to implement Bloom vectors, which exhibit significant energy savings than its CMOS counterparts [5]. Moreover, resistive memory consumes little area overhead as it can be implemented on top of the chip [44].

6.4 Experimental Results

6.4.1 Experimental Setup

We use tiny-dnn [4], a header only, dependency free deep learning library written in C++, as our evaluation platform. For CNN, we use LeNet-like architecture as illustrated in Fig. 5.3. We use MNIST (Mixed National Institute of Standards and Technology) database of handwritten numbers [56] as our dataset to evaluate the accuracy. The dataset is split into a training set and a test set with 60,000 and 10,000 28×28

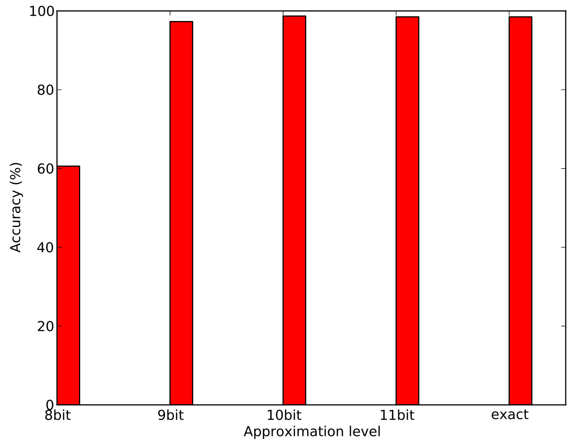


Figure 6.4: Neural network accuracy loss due to approximate pattern matching.

images. We randomly select 5% of the training input data to profile the frequent input operands. To estimate the energy consumption of the proposed design, we implement the hash functions using Verilog, and we extract the Verilog implementation of a six-stage pipelined floating point multiplier using FloPoCo [28]. Then, the implementations are synthesized using Synopsys Design Compiler, with 45 nm standard CMOS library. The operating voltage is set to 1.0V and the clock period is 1.5 ns. In addition, Bloom vectors (BV) are designed with resistive 1T1R cells using HSPICE, where RON is set to 1K Ω and ROFF to 1M Ω [82]. Bloom filters can be used in different hardware platforms, including CPU [35], FPGA [29], and GPU [5].

6.4.2 Accuracy Loss

As described in Section 6.2.2, the BF will return an inexact result due to approximate pattern matching. Thus, we investigate here on how the approximation level impacts the neural network accuracy. We vary the approximate pattern matching mode from 8-bit matching to 11-bit matching and store 10 most frequent patterns for each of the approximation modes with a FP rate of 0.001. Fig. 6.4 shows the accuracy under each configuration. The baseline accuracy is 98.5% without any approximations. The 8-bit matching introduces aggressive approximation because it does not cover the last bit in the exponent bits, which leads to only 60.6% accuracy. Starting from 9-bit matching, the accuracy loss is insignificant. Note that 9-bit matching covers the sign bit and exponent bits for floating point numbers. This indicates the high error-tolerance of neural networks to data imprecision.

According to Fig. 6.2, 9-bit matching gives us the highest hit rate among the approximation modes which introduces little drop on neural network accuracy. Thus, using 9-bit matching as our approximation mode, we then investigate how the accuracy will vary with the number of stored frequent patterns. As shown in Table 6.1, various number of stored patterns under 9-bit matching have little impact on neural network accuracy. The lowest accuracy is 97.2% under the (9, 50) configuration, meaning that we use 9-bit approximate pattern matching and store 50 input patterns.

Table 6.1: Energy savings and neural network accuracy across different BF settings.

Matching Mode	(BV size, #Hash_Fn, #inp_BF)	Hit Rate	NN Accuracy	E_{save}
(8, 10)	(64, 2, 1)	66.9%	60.6%	58.9%
(9, 5)	(64, 2, 1)	28.9%	97.5%	24.9%
(9, 10)	(64, 2, 1)	45.7%	97.4%	37.9%
(9, 20)	(64, 2, 1)	60.7%	97.9%	45.4%
(9, 30)	(64, 2, 1)	70.6%	97.4%	47.5%
(9, 30)	(32, 3, 1)	70.6%	97.4%	41.6%
(9, 40)	(64, 2, 1)	77.3%	97.3%	47.3%
(9, 50)	(64, 2, 1)	82.3%	97.2%	44.7%
(10, 10)	(64, 2, 1)	30.4%	98.3%	22.3%

6.4.3 Energy Savings

We use several different matching modes and BF configuration to compute the energy savings and the resulting neural network accuracy as shown in Table 6.1. The matching mode ($appx\_bit$,$\#inp$) refers to how many bits we use for approximate pattern matching and the number of patterns we store. The BF setting (BV size, #hash_Fn, #inp_BF) refers to BV size in bit length (m), number of hash functions (k) and number of input patterns stored in each BF (n). For example, the BF setting at (64, 2, 1) means that we set the BV size as 64 bits, use 2 hash functions and store 1 input pattern for each BF. To satisfy the FP rate which can lead to acceptable accuracy, we carefully select BF configurations.

Table. 6.1 exhibits several important facts. First, 9-bit matching is the optimal matching mode here. By comparing with 8-bit matching and 10-bit matching, we find that 8-bit matching achieves the most energy saving at 58.9%, but its resulted neural network accuracy is only 60.6%, a significant accuracy drop over baseline accuracy of 98.5%. 10-bit matching achieves higher accuracy than 9-bit because it introduces smaller approximation errors into the neural network than 9-bit matching but its resulting energy saving is only at 22.3%, which is less than the one obtained with 9-bit matching mode. Thus, 9-bit matching achieves the better balance between neural network accuracy and energy savings.

Second, after we fix 9-bit matching mode, we then look for the optimal number of patterns to store. We vary the number of stored patterns from 5 to 50. Note that all 9-bit matching modes, regardless of the number of stored patterns, achieve accuracy close to the baseline. Thus, we focus on locating the best energy saving setting. As shown in Fig. 6.5, we found that the energy saving increases as the number of stored patterns increases from 5 to 30 (we call it the first stage), however the energy saving starts to decrease as the number of stored patterns increases from 30 to 50 (second stage). This is because in the first stage, the hit rate increases as the number of stored patterns increases, which will reduce the use of multipliers. In the second stage, although the hit rate still increases, the energy consumption of BFs increases as the number of stored patterns increases, which dominates the energy consumption. Thus, we find that the optimal matching mode is (9, 30).

Third, we also try different settings of BV size and hash functions. To satisfy the FP error rate of 0.001, we use two realistic BF settings, (64, 2, 1) and (32, 3, 1). The BF setting at (32, 3, 1) consumes more energy than that of (64, 2, 1) because it uses 3 hash functions, which is the main source of energy consumption of BFs. The optimal configuration is (9, 30) as matching mode and (64, 2, 1) as BF setting. This leads to a neural network accuracy of 97.4% and energy saving of 47.5%.

6.5 Chapter Summary

This chapter presents a controllable and reconfigurable approximate computation reuse approach to improve the energy efficiency of hardware neural networks. We exploit the computation reuse opportunities in neural networks and enhance such opportunities by performing approximate pattern matching. We design an approximate Bloom

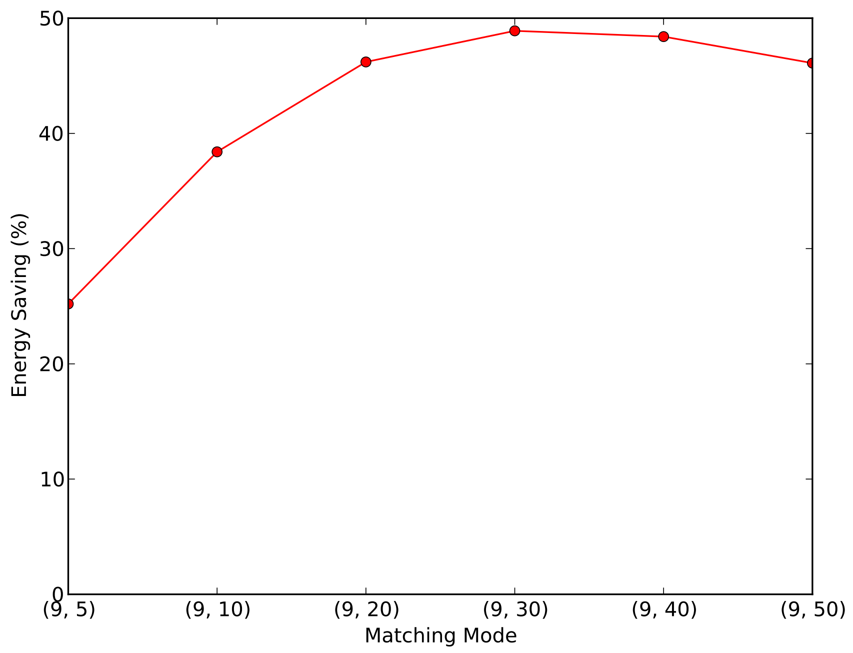


Figure 6.5: Energy Savings under different matching mode.

filter architecture to physically implement the approximate pattern matching function and tightly integrate it with computation units. By storing the frequent computation patterns, Bloom filters can recall the computation results to avoid the overhead due to redundant executions on computation units. The experimental results show 47.5% energy reductions of multiplication operation with classification accuracy degradation at 1% for convolutional neural networks. Our future works focus on investigating whether the variances of datasets and neural network types and architectures have an impact on the computation reuse opportunities. If so, then we may design a neural network that can maximize the computation reuse opportunities.

Chapter 6 contains reprints of Xun Jiao, Vahideh Akhlaghi, Yu Jiang, and Rajesh Gupta, "Energy-Efficient Neural Networks using Approximate Computation Reuse", *Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE)*, 2018. The dissertation author is the primary author of the paper.

Chapter 7

Conclusion and Future Directions

This dissertation focuses on improving timing error resilience of microelectronic computing systems. We have reviewed and classified existing approaches into three categories — **error prevention**, **error correction**, and **error acceptance**. We have then identified associated technical challenges: **inaccurate error modeling**, **high correction cost**, and **unacceptable application quality**. For each of these challenges, we proposed our corresponding solutions.

To combat the **inaccurate error modeling** challenge, Chapter 2 presents a method to integrate the impact of input operands in the error modeling using machine learning methods. To combat the **high correction cost** challenge, Chapter 3 presents an approach to efficiently correct timing errors using the structural errors of inexact (approximate) circuits. Chapter 4 presents an instruction-based timing error prevention approach to avoid the correction cost. To combat the **unacceptable application quality** challenge, Chapter 5 presents a cross-layer approach to expose the hardware errors under relaxed operating conditions to the software execution to assess application vulnerability to timing errors. Chapter 6 presents a controllable and reconfigurable hardware approximation approach that can guarantee acceptable application quality while achieving significant energy savings.

Looking beyond this dissertation, the following outlines our thinking for future work in the area:

Energy-efficient Hardware Neural Networks Artificial neural networks have shown broad success for medical applications, speech recognition, and natural language processing but their hardware implementation exhibit significant energy consumption. Such acceleration demands new methods that effectively combine computational efficiency with low power consumption. For instance, one can use voltage scaling on the computation units of hardware neural networks to save energy. Unfortunately, voltage scaling can save energy but increase the sensitivity to timing errors, which might affect the accuracy of neural networks. To address this, we can explore efficient timing error tolerance mechanisms at various layers. For instance, due to the adaptation of neural network parameters, error-aware retraining can be used to enable the self-healing of neural networks. Our work on CLIM [49] is a starting point for further work in the area.

Resilient Cyber-physical Systems A cyber-physical system (CPS) is a strongly interconnected embedded computing system with tight integration of computational and physical components. Given the proliferation of computing, sensing capabilities, these systems are beginning to be used critical infrastructures such as energy and transportation. The complexity and fragility of these systems due to ongoing interactions of computing with physical processes presents new challenges in system resilience and reliability. We believe these can only be addressed through a "cross-layer" perspective, that is, design and analytic methods that consider a deep (if not, full) stack of hardware and software from circuits to software services and protocols. To be specific, consider systems deployed in harsh environments of battlefield or space. Increased rate of soft errors or the impact of these errors on mission demands new models and methods for carrying out vulnerability analysis and devise robust protocols and runtime verification techniques to improve resilience. Such a work would require collaborations with experts from different fields including computer architecture and formal verification methods.

Energy-efficient Heterogeneous Microprocessors Multicore heterogeneous architecture has been a promising solution to enhance energy-efficiency of embedded systems. With the increasing complexity of applications, techniques for workload distribution on different cores and components are the key to achieving energy savings while satisfying performance and quality constraints [37,38]. I plan to explore energy-efficient multicore heterogeneous microprocessor architectures by exploring a suitable runtime environment that can assign the workload to the core that matches its resource needs. For example, we can design each core to be optimized at different process corners and assign different instructions or instruction sequences to different cores based on their timing slack requirements. We can also explore the tradeoff between reliability and energy efficiency in heterogeneous microprocessors.

Bibliography

[1] Amd app sdk v2.5. available: http://www.amd.com/stream.

[2] Amd app sdk v2.5. [online]. available: http://www.amd.com/stream.

[3] The itrs website: http://www.itrs.net/links/2011itrs/home2011.htm.

[4] tiny-dnn: https://github.com/tiny-dnn/tiny-dnn.

[5] Vahideh Akhlaghi, Abbas Rahimi, and Rajesh K. Gupta. Resistive bloom filters: from approximate membership to approximate computing with bounded errors. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, 2016.

[6] Christopher M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

[7] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrah, Sushil Singh, and George Varghese. Beyond bloom filters: from approximate membership checks to approximate state machines. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 315–326, 2006.

[8] Keith Bowman, Steven G. Duvall, and James D. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *Solid-State Circuits, IEEE Journal of*, 37(2):183–190, 2002.

[9] Keith Bowman, James W. Tschanz, Nam Sung Kim, Janice C. Lee, Chris B. Wilkerson, Shih-Lien L Lu, Tanay Karnik, and Vivek De. Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance. *IEEE Journal of Solid-State Circuits*, 44(1):49–63, 2009.

[10] Keith Bowman, James W. Tschanz, Shih-Lien L. Lu, Paolo Aseron, Muhammad M. Khellah, Arijit Raychowdhury, Bibiche M. Geuskens, Carlos Tokunaga, Chris B. Wilkerson, Tanay Karnik, and Vivek De. A 45 nm resilient microprocessor core for dynamic variation tolerance. *IEEE Journal of Solid-State Circuits*, 46(1):194–208, 2011.

[11] Michael Bushnell and Vishwani Agrawal. *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*, volume 17. Springer Science & Business Media, 2004.

[12] Vincent Camus, Jeremy Schlachter, and Christian Enz. Energy-efficient digital design through inexact and approximate arithmetic circuits. In *New Circuits and Systems Conference (NEWCAS), 2015 IEEE 13th International*, pages 1–4. IEEE, 2015.

[13] Vincent Camus, Jeremy Schlachter, and Christian Enz. Energy-efficient inexact speculative adder with high performance and accuracy control. In *Circuits and Systems (ISCAS), 2015 IEEE International Symposium on*, pages 45–48. IEEE, 2015.

[14] Vincent Camus, Jeremy Schlachter, and Christian Enz. A low-power carry cut-back approximate adder with fixed-point implementation and floating-point precision. In *Design Automation Conference (DAC), 2016 53rd ACM/EDAC/IEEE*, 2016.

[15] Vincent Camus, Jeremy Schlachter, Christian Enz, Michael Gautschi, and Frank K. Gurkaynak. Approximate 32-bit floating-point unit design with 53% power-area product reduction. In *European Solid-State Circuits Conference (ESSCIRC)*, pages 465–468, 2016.

[16] Michael Carbin, Sasa Misailovic, and Martin C Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *ACM SIGPLAN Notices*, volume 48, pages 33–52. ACM, 2013.

[17] Kwanyeob Chae, Saibal Mukhopadhyay, Chang-Ho Lee, and Joy Laskar. A dynamic timing control technique utilizing time borrowing and clock stretching. In *Custom Integrated Circuits Conference (CICC), 2010 IEEE*, pages 1–4. IEEE, 2010.

[18] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 269–284. ACM, 2014.

[19] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[20] Hari Cherupalli and John Sartori. Graph-based dynamic analysis: Efficient characterization of dynamic timing and activity distributions. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 729–735. IEEE Press, 2015.

[21] Hyungmin Cho, Larkhoon Leem, and Subhasish Mitra. Ersa: Error resilient system architecture for probabilistic applications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(4):546–558, 2012.

[22] Mihir R. Choudhury, Vishal Chandra, Robert Aitken, and Kartik Mohanram. Time-borrowing circuit designs and hardware prototyping for timing error resilience. *Computers, IEEE Transactions on*, 63(2):497–509, 2014.

[23] Mihir R. Choudhury and Kartik Mohanram. Masking timing errors on speed-paths in logic circuits. In *Design, Automation & Test in Europe Conference & Exhibition (DATE).*, pages 87–92. IEEE, 2009.

[24] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[25] Jeremy Constantin, Lai Wang, Georgios Karakonstantis, Anupam Chattopadhyay, and Andreas Burg. Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 381–386. EDA Consortium, 2015.

[26] Thomas M. Cover and Peter E. Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.

[27] Shidhartha Das, Carlos Tokunaga, Sanjay Pant, Wei-Hsiang Ma, Sudherssen Kalaiselvan, Kevin Lai, David M Bull, and David T Blaauw. Razorii: In situ error detection and correction for pvt and ser tolerance. *IEEE Journal of Solid-State Circuits*, 44(1):32–48, 2009.

[28] Florent De Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, 28(4):18–27, 2011.

[29] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John W. Lockwood. Deep packet inspection using parallel bloom filters. In *Proceedings of IEEE Symposium on High Performance Interconnects*, HotI03, pages 44–51, 2003.

[30] Saurabh Dighe, Sriram R. Vangal, Paolo Aseron, Shasi Kumar, Tiju Jacob, Keith Bowman, Jason Howard, James Tschanz, Vasantha Erraguntla, Nitin Borkar, and Vivek De. Within-die variation-aware dynamic-voltage-frequency-scaling with optimal core allocation and thread hopping for the 80-core teraflops processor. *Solid-State Circuits, IEEE Journal of*, 46(1):184–193, 2011.

[31] Zidong Du, Avinash Lingamneni, Yunji Chen, Krishna V Palem, Olivier Temam, and Chengyong Wu. Leveraging the error resilience of neural networks for designing highly energy efficient accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1223–1235, 2015.

[32] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Annual IEEE/ACM International Symposium on Microarchitecture, 2003.*, pages 7–18. IEEE, 2003.

[33] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460. IEEE Computer Society, 2012.

[34] Matthew Fojtik, David Fick, Yejoong Kim, Nathaniel Pinckney, David Harris, David Blaauw, and Dennis Sylvester. Bubble razor: An architecture-independent approach to timing-error detection and correction. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 488–490. IEEE, 2012.

[35] Masayoshi Fujii, Yuuki Sato, Tomoaki Tsumura, and Yasuhiko Nakashima. Exploiting bloom filters for saving power consumption of auto-memoization processor. In *Computing and Networking (CANDAR), 2016 Fourth International Symposium on*, pages 354–360. IEEE, 2016.

[36] Manish Gupta, Abbas Rahimi, Daniel Lowell, John Kalamatianos, Dean Tullsen, and Rajesh Gupta. Asar: Application-specific approximate recovery to mitigate hardware variability. *SELSE (Silicon Errors in Logic, System Effects)*, 2017.

[37] Manish Gupta, David Roberts, Mitesh Meswani, Vilas Sridharan, Dean Tullsen, and Rajesh Gupta. Reliability and performance trade-off study of heterogeneous memories. In *Proceedings of the Second International Symposium on Memory Systems*, pages 395–401. ACM, 2016.

[38] Manish Gupta, Vilas Sridharan, David Roberts, Andreas Prodromou, Ashish Venkat, Dean Tullsen, and Rajesh Gupta. Reliability-aware data placement for heterogeneous memory architecture. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 583–595. IEEE, 2018.

[39] G Gybenko. Approximation by superposition of sigmoidal functions. pages 303–314, 1989.

[40] Kai He, Andreas Gerstlauer, and Michael Orshansky. Circuit-level timing-error acceptance for design of energy-efficient dct/idct-based systems. *Circuits and Systems for Video Technology, IEEE Transactions on*, 23(6):961–974, 2013.

[41] S. Himavathi, D. Anitha, and A. Muthuramalingam. Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, 18(3):880–888, 2007.

[42] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[43] Sebastian Hoppner, Holger Eisenreich, Stephan Henker, Dennis Walter, Georg Ellguth, and René Schuffny. A compact clock generator for heterogeneous gals mpsocs in 65-nm cmos technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(3):566–570, 2013.

[44] Mohsen Imani, Shruti Patil, and Tajana Rosing. Approximate computing using multiple-access single-charge associative memory. *IEEE Transactions on Emerging Topics in Computing*, 2016.

[45] Kwangok Jeong, Andrew B Kahng, and Kambiz Samadi. Impact of guardband reduction on design outcomes: A quantitative approach. *Semiconductor Manufacturing, IEEE Transactions on*, 22(4):552–565, 2009.

[46] Honglan Jiang, Cong Liu, Leibo Liu, Fabrizio Lombardi, and Jie Han. A review, classification, and comparative evaluation of approximate arithmetic circuits. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(4):60, 2017.

[47] Xun Jiao, Vincent Camus, Mattia Cacciotti, Yu Jiang, Christian Enz, and Rajesh K Gupta. Combining structural and timing errors in overclocked inexact speculative adders. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 482–487. IEEE, 2017.

[48] Xun Jiao, Mulong Luo, Jeng-Hau Lin, and Rajesh K. Gupta. An assessment of vulnerability of hardware neural networks to dynamic voltage and temperature variations. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Irvine, USA*, 2017.

[49] Xun Jiao, Abbas Rahimi, Yu Jiang, Jianguo Wang, Hamed Fatemi, Jose Pineda de Gyvez, and Rajesh K. Gupta. Clim: A cross-level workload-aware timing error prediction model for functional units. *IEEE Transactions on Computers*, 2017.

[50] Andrew B. Kahng and Seokhyeong Kang. Accuracy-configurable adder for approximate arithmetic designs. In *Proceedings of the 49th Annual Design Automation Conference*, pages 820–825. ACM, 2012.

[51] Andrew B. Kahng, Seokhyeong Kang, Rakesh Kumar, and John Sartori. Slack redistribution for graceful degradation under voltage overscaling. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 825–831. IEEE, 2010.

[52] Kyunghee Kang and Tadashi Shibata. An on-chip-trainable gaussian-kernel analog support vector machine. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 57(7):1513–1524, 2010.

[53] Michael Kharitonov. Cryptographic hardness of distribution-specific learning. In *STOC*. ACM, 1993.

[54] Veit B. Kleeberger, Petra R. Maier, and Ulf Schlichtmann. Workload-and instruction-aware timing analysis: The missing link between technology and system-level resilience. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6. ACM, 2014.

[55] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[56] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[57] Xiaojun Li, Jin Qin, and Joseph B. Bernstein. Compact modeling of mosfet wearout mechanisms for circuit-reliability simulation. *IEEE Transactions on Device and Materials Reliability*, 8(1):98–121, 2008.

[58] Nathan Linial, Yishay Mansour, and Noam Nisan. Constant depth circuits, fourier transform, and learnability. *Journal of the ACM (JACM)*, 40(3):607–620, 1993.

[59] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin Rinard. Chisel: reliability-and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 309–328. ACM, 2014.

[60] Vojtech Mrazek, Syed Shakib Sarwar, Lukas Sekanina, Zdenek Vasicek, and Kaushik Roy. Design of power-efficient approximate multipliers for approximate artificial neural networks. In *International Conference On Computer Aided Design (ICCAD)*, 2016.

[61] Patrick Ndai, Nauman Rafique, Mithuna Thottethodi, Swaroop Ghosh, Swarup Bhunia, and Kaushik Roy. Trifecta: A nonspeculative scheme to exploit common, data-dependent subcritical paths. *IEEE Trans. VLSI Syst.*, 18(1):53–65, 2010.

[62] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, and Jake Vanderplas. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[63] Abbas Rahimi, Luca Benini, and Rajesh K. Gupta. Hierarchically focused guardbanding: an adaptive approach to mitigate pvt variations and aging. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 1695–1700. IEEE, 2013.

[64] Abbas Rahimi, Luca Benini, and Rajesh K. Gupta. Application-adaptive guardbanding to mitigate static and dynamic variability. *IEEE Transactions on Computers*, 63(9):2160–2173, 2014.

[65] Abbas Rahimi, Amirali Ghofrani, Miguel Angel Lastras-Montano, Kwang-Ting Cheng, Luca Benini, and Rajesh K. Gupta. Energy-efficient gpgpu architectures via collaborative compilation and memristive memory-based computing. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6. IEEE, 2014.

[66] Abbas Rahimi, Andrea Marongiu, Rajesh K. Gupta, and Luca Benini. A variability-aware openmp environment for efficient execution of accuracy-configurable computation on shared-fpu processor clusters. In *International Conference on Hardware/Software Codesign and System Synthesis*. IEEE, 2013.

[67] Sanghamitra Roy and Koushik Chakraborty. Predicting timing violations through instruction-level path sensitization analysis. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1074–1081. ACM, 2012.

[68] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. *CALIFORNIA UNIV SAN DIEGO LA JOLLA INST FOR COGNITIVE SCIENCE*, 1985.

[69] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[70] Jeremy Schlachter, Vincent Camus, and Christian Enz. Near/sub-threshold circuits and approximate computing: The perfect combination for ultra-low-power systems. In *VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on*, pages 476–480. IEEE, 2015.

[71] Jeremy Schlachter, Vincent Camus, Christian Enz, and Krishna V. Palem. Automatic generation of inexact digital circuits by gate-level pruning. In *Circuits and Systems (ISCAS), 2015 IEEE International Symposium on*, pages 173–176. IEEE, 2015.

[72] James Tschanz, Bowman, Steve Walstra, Marty Agostinelli, Tanay Karnik, and Vivek De. Tunable replica circuits and adaptive voltage-frequency techniques for dynamic voltage, temperature, and aging variation tolerance. In *2009 Symposium on VLSI Circuits*, 2009.

[73] James Tschanz, Nam Sung Kim, Saurabh Dighe, Jason Howard, Gregory Ruhl, Sriram Vangal, Siva Narendra, Yatin Hoskote, Howard Wilson, Carol Lam, Matthew Shuman, Carlos Tokunaga, Dinesh Somasekhar, Stephen Tang, David Finan, Tanay Karnik, Nitin Borkar, Nasser Kurd, and Vivek De. Adaptive frequency and biasing techniques for tolerance to dynamic temperature-voltage variations and aging. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 292–604. IEEE, 2007.

[74] G Tziantzioulis, AM Gok, SM Faisal, N Hardavellas, S Ogrenci-Memik, and S Parthasarathy. b-hive: a bit-level history-based error model with value correlation for voltage-scaled integer and floating point units. In *Proceedings of the 52nd Annual Design Automation Conference*, page 105. ACM, 2015.

[75] Rafael Ubal, Byunghyun Janscikitg, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: a simulation framework for cpu-gpu computing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 335–344. ACM, 2012.

[76] Ying Wang, Jiachao Deng, Yuntan Fang, Huawei Li, and Xiaowei Li. Resilience-aware frequency tuning for neural-network-based approximate computing chips. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[77] Lucas Wanner, Rahul Balani, Sadaf Zahedi, Charwak Apte, Puneet Gupta, and Mani Srivastava. Variability-aware duty cycle scheduling in long running embedded sensing systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[78] Peter Welinder, Steve Branson, Takeshi Mita, Catherine Wah, Florian Schroff, Serge Belongie, and Pietro Perona. Caltech-ucsd birds 200. 2010.

[79] Jing Xin and Russ Joseph. Identifying and predicting timing-critical instructions to boost timing speculation. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 128–139. ACM, 2011.

[80] Hongmei Yan, Yingtao Jiang, Jun Zheng, Chenglin Peng, and Qinghui Li. A multilayer perceptron-based medical decision support system for heart disease diagnosis. *Expert Systems with Applications*, 30(2):272–281, 2006.

[81] Fangming Ye, Farshad Firouzi, Yang Yang, Krishnendu Chakrabarty, and Mehdi B Tahoori. On-chip droop-induced circuit delay prediction based on support-vector machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(4):665–678, 2016.

[82] M. Zangeneh and A. Joshi. Design and optimization of nonvolatile multibit 1t1r resistive ram. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 22(8):1815–1828, Sept 2013.

[83] Ning Zhu, Wang Ling Goh, and Kiat Seng Yeo. An enhanced low-power high-speed adder for error-tolerant application. In *Integrated Circuits, ISIC'09. Proceedings of the 2009 12th International Symposium on*, pages 69–72. IEEE, 2009.