UNIVERSITY OF CALIFORNIA SAN DIEGO

Performance Analysis of Timing-Speculative Processors

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

in

Computer Science (Computer Engineering)

by

Omid Assare

Committee in charge:

Professor Rajesh Gupta, Chair
Professor Chung Kuan Cheng
Professor Farinaz Koushanfar
Professor Steven Swanson
Professor Dean Tullsen

2019

The Dissertation of Omid Assare is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2019

DEDICATION

To Mom.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Rajesh Gupta, for the technical guidance and the financial support that made this dissertation possible. I am grateful for the freedom I had to explore and change direction and the opportunity to learn and practice independent thinking. It has been an honor to work under his supervision.

I would like to thank members of my committee, Professor CK Cheng, Professor Farinaz Koushanfar, Professor Steven Swanson, and Professor Dean Tullsen for their constructive feedback and comments.

I also wish to thank all members of the Microelectronic Embedded Systems Laboratory. In particular, thanks to Atieh Lotfi for her help with the dissertation, Manish Gupta for the technical discussions—and all the jokes—and Abbas Rahimi for his kindness and help during my first few years in the group.

I must also thank Professor Maziar Goudarzi at Sharif University who first introduced me to the world of research and taught me about integrity and ethics by example. I also thank members of the Energy-Aware Systems Laboratory at Sharif University, and in particular Mahmoud Momtazpour, for their help during my time there.

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

Chapter 3, in full, is a reprint of Omid Assare and Rajesh Gupta, "Timing Analysis of Erroneous Systems," *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2014. The dissertation author was the primary investigator and author of this paper.

Chapter 4 is, in part, a reprint of Omid Assare and Rajesh Gupta, "Accurate Estimation of Program Error Rate for Timing-Speculative Processors," *IEEE/ACM Design Automation Conference (DAC)*, 2019, and, in part, currently being prepared for submission for publication of the material. Omid Assare and Rajesh Gupta, "Performance Analysis of Timing-Speculative Processors." The dissertation author was the primary investigator and author of these papers.

Chapter 5, in full, is a reprint of Omid Assare and Rajesh Gupta, "Strategies for Optimal

Operating Point Selection in Timing-Speculative Processors," *IEEE International Conference on Computer Design (ICCD)*, 2016. The dissertation author was the primary investigator and author of this paper.

VITA

2012	Bachelor of Science, Electrical Engineering, Sharif University of Technology
2015	Master of Science, Computer Science (Computer Engineering), University of California San Diego
2019	Doctor of Philosophy, Computer Science (Computer Engineering), University of California San Diego

PUBLICATIONS

Omid Assare and Rajesh Gupta, "Accurate Estimation of Program Error Rate for Timing-Speculative Processors," *IEEE/ACM Design Automation Conference (DAC)*, 2019.

Omid Assare and Rajesh Gupta, "Strategies for Optimal Operating Point Selection in Timing-Speculative Processors," *IEEE International Conference on Computer Design (ICCD)*, 2016.

Mahmoud Momtazpour, Omid Assare, Negar Rahmati, Amirali Boroumand, Saeed Barati, and Maziar Goudarzi, "Yield-Driven Design-Time Task Scheduling Techniques for MPSoCs under Process Variation: A Comparative Study," *IET Journal of Computers and Digital Techniques*, Volume 9, Issue 4, 2015.

Omid Assare and Rajesh Gupta, "Timing Analysis of Erroneous Systems," *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2014.

Omid Assare and Rajesh Gupta, "Clustered Timing Model: Statistical Modeling of Variability for Dynamic Estimation of Errors," *IEEE/ACM Design Automation Conference (DAC)*, 2014 (poster).

Omid Assare, Mahmoud Momtazpour, and Maziar Goudarzi, "Leak-Gauge: A Late-Mode Variability-Aware Leakage Power Estimation Framework," *Elsevier Journal of Microprocessors and Microsystems (MICPRO)*, Volume 37, Issue 8, Part A, 2013.

Omid Assare, Mahmoud Momtazpour, and Maziar Goudarzi, "Accurate Estimation of Leakage Power Variability in Sub-Micrometer CMOS Circuits," *The 15th Euromicro Conference on Digital System Design (DSD)*, 2012. Nominated for Best Paper Award.

Omid Assare and Maziar Goudarzi, "Opportunities for Embedded Software Power Reductions," *IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2011.

ABSTRACT OF THE DISSERTATION

Performance Analysis of Timing-Speculative Processors

by

Omid Assare

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2019

Professor Rajesh Gupta, Chair

Timing guardbands act as a barrier protecting conventional processors from circuit-level phenomena like timing errors. Timing-speculative (TS) processors replace these guardbands with timing error detection and recovery circuits to guarantee correct execution. For timing speculation to be effective, the performance and/or energy improvements gained from eliminating the guardbands must outweigh the costs of detecting and recovering from timing errors. The high costs and limited benefits that have been an obstacle to adoption of timing speculation in commercial designs have been steadily improving over the past decade. Likewise, recent advances in design of ultra-fast on-chip voltage regulators and all-digital phase locked loops with sub-nanosecond response times have increased the potential benefits by enabling more

aggressive timing speculation schemes.

This dissertation is motivated by another contributing factor limiting broader adoption of TS processors—complexity of their performance analysis. The absence of timing guardbands complicates timing analysis of TS processors as circuit and architecture, and their interdependence, must be considered simultaneously. We present a cross-layer performance analysis framework for TS processors that spans the system stack from circuit to application, including dynamic timing analysis tools at the level of gates, microarchitecture, and architecture, an instruction-level timing error model, and a statistical program error rate estimation methodology.

We then use our framework to study the performance of a TS processor with an emphasis on characterizing the role of software. Our experiments show that the combination of running application and its input data can change the performance of a TS processor by as much as 25 percent, demonstrating that application-specific analysis is necessary for accurate evaluation of TS processors and should be used to inform design decisions and assess the suitability of applications for timing speculation.

Performance of TS processors also relies on accurate prediction of the optimal operating point. Our experiments show that, in a typical case, the most commonly used policy achieves only a fraction of the potential gains of timing speculation. Inspired by our modeling of timing errors, the improved timing speculation strategies we propose in this dissertation can realize a more than 50 percent throughput improvement compared to a guardbanded design.

Chapter 1

Introduction

As the semiconductor technology scales deep into the nanometer regime, its product, the integrated circuit, becomes increasingly sensitive to variations in manufacturing process and operating environment [31]. As shown in Table 1.1, the sources of variability can be categorized according to their spatial reach as well as their temporal rate of change. Spatial reach determines whether a source affects all transistors of a chip (global) or only a few transistors in close proximity (local). For instance, process variation has global or die-to-die (D2D) and local or within-die (WID) components [5]. An experimental Intel processor shows around 50% performance variation among its 80 cores when operated at $0.8V$ due to WID process variation alone [21].

Both D2D and WID components of process variation, however, are in the same category based on their temporal rate of change. Variations caused by these sources are called static variations since their magnitude remains constant during the lifetime of the chip. Dynamic sources of variation, on the other hand, cause variations that change during the operation of the chip. They include slow variations such as temperature hotspots that are spread over thousands of clock cycles as well as fast variations such as supply voltage fluctuations that occur over the course of several clock cycles. Ambient temperature variations can also be expected to be dynamic. International Technology Roadmap for Semiconductors (ITRS) projects supply power variation to be 10% while the operating temperature can vary from 30 to 175°C resulting in

Table 1.1. Sources of Variability for CMOS Integrated Circuits [9]

		Temporal Rate of Change		
		Static	Dynamic	
		Extremely Slow	*Slow-Changing*	*Fast-Changing*
Spatial Reach	Global	D2D Process Variation Lifetime Degradation	VDD Fluctuations Ambient Temp. Variation	PLL Jitter, IR Drop Ldi/dt, Input Data
	Local	WID Process Variation	Local IR Temperature Hotspots	Capacitive Decoupling Clock Tree Jitter

several tens of percent performance change [31].

Finally, it is useful to consider input data (running code and its input in case of processors) as variation sources. Studies show that changing input data to a single program can cause performance variations comparable in magnitude to those caused by other variability sources such as process variation [2].

1.1 Timing Guardbands

To ensure correct operation, conventional processors are designed pessimistically for the worst case, leading to large safety margins, also known as timing guardbands. Each source of variability is accounted for by adding more guardbands. While more sophisticated design-time analysis methods such as statistical static timing analysis (SSTA) have tried to reduce design pessimism, guardbands continue to increase steadily with each technology generation [31], leading to loss of performance and increasing cost due to over-design.

Other techniques try to reduce timing guardbands dynamically, by monitoring the state of the system and adjusting the operating point accordingly. For instance conventional dynamic voltage and frequency scaling (DVFS) adjusts the system's operating point based on pre-characterized safe values stored in a look-up table [60][56]. Canary circuits are a more sophisticated class of system state monitors that try to monitor the critical path's available slack directly [25][10][45][22][59]. One such technique, called active management of timing guard-

band [25] , is implemented in IBM POWER7 server where critical path monitors measure the available timing margin under thermal fluctuations, voltage skewing, workload-induced voltage and temperature variations, etc., and a control unit adjusts processor voltage and frequency to achieve error-free operation while reducing the guardbands. While these techniques provide a low-cost solution for reducing timing guardbands, they are unable to account for local and fast changing dynamic variations including delay variations caused by input data. Therefore, guardbands associated with these variations cannot be removed and the associated costs remain.

1.2 Timing Speculation

In this dissertation, we study a new class of processors that take a completely different approach. Timing-speculative (TS) processors allow increasing the frequency or decreasing the supply voltage beyond the limits determined by static timing analysis, thereby removing timing guardbands altogether. Instead, they rely on circuit- and microarchitecture-level techniques to detect and recover from potential timing errors. For timing speculation to be effective, the performance and/or energy improvements gained from eliminating the guardbands must outweigh the costs of detecting and recovering from timing errors.

The high cost-benefit ratio that has been an obstacle to adoption of timing speculation in commercial designs has been steadily improving over the past decade. Error detection and recovery costs have decreased from 44 additional transistors per flip-flop for detection and dozens of clock cycles for recovery in the first Razor design [23], to only 3 additional transistors and as few as a single clock cycle in the latest version [65]. Likewise, recent advances in the design of ultra-fast on-chip voltage regulators and all-digital phase locked loops with sub-nanosecond response times have increased the potential benefits by enabling more aggressive timing speculation schemes [24][37][8].

However, another contributing factor is still limiting broader diffusion of TS processors—complexity of their performance analysis. In a conventional processor, timing guardbands act as

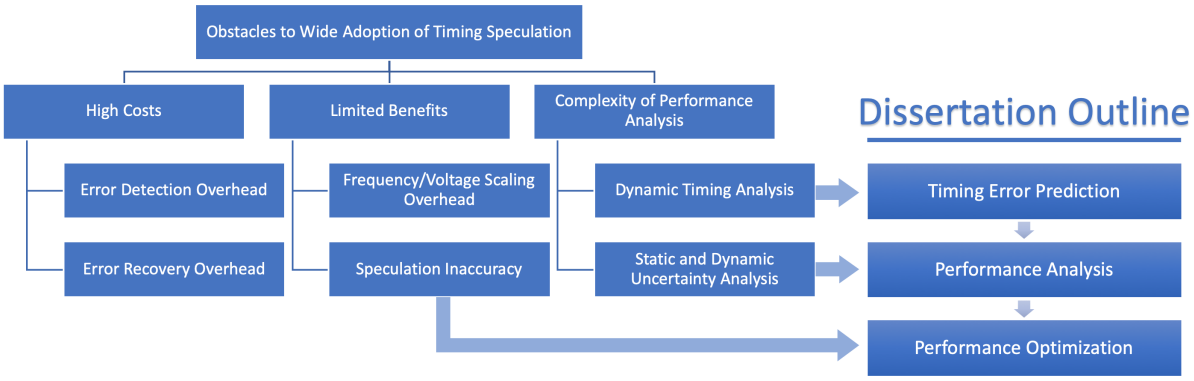


Figure 1.1. Timing Speculation Challenges and Dissertation Outline

a barrier between the circuit and the architecture. Guardbands for ambient and process variations provide the architecture with a consistent timing model of the circuit. For example, the number of clock cycles required to execute an instruction does not depend on the supply voltage, the temperature, or the location of the die in the wafer. In the opposite direction, guardbands for data variation allow timing analysis of the circuit to be performed statically, without considering the specific program and its input data. This independence greatly simplifies the analysis by allowing the circuit and the architecture to be analyzed in isolation.

Without these guardbands in a TS processor, the architecture is no longer protected from circuit-level phenomena and vice versa. For example, a voltage droop can, in effect, change the number of clock cycles required for an instruction to execute. Conversely, a change in the operands of an instruction can lead to timing errors in the circuit by sensitizing its critical timing paths. Consequently, accurate performance analysis of TS processors must consider the circuit and the architecture simultaneously and account for their interdependence through timing errors.

In this dissertation, we present a cross-layer performance analysis framework for TS processors that spans the system stack from circuit to application, including dynamic timing analysis (DTA) tools at the level of gates, microarchitecture, and architecture, an instruction-level timing error model, and a statistical program error rate estimation methodology. We then use our framework to study and improve performance of a TS processor.

1.3 Related Work

The need for a fast and accurate timing model for dynamic estimation of errors is most evident in experimental-results sections of papers describing variability management techniques of erroneous systems (i.e. systems that allow timing errors) where researchers face the long simulation times of variability effects at the architecture level [46], [51], [64], [53]. As a result, most works take one of the following approaches.

Inaccurate Models. Error rate models typically used for analyzing TS processors do not get more sophisticated than assigning different numbers to various hardware/software components and/or operating conditions, and the role of software in sensitizing timing paths and changing error rates is often ignored. It is even common to use the simplest possible model, a fixed number, for error rate of the system at all times [5], [21], [31].

Limited Analysis. In the absence of efficient error models and when more accuracy is needed, researchers are forced limit their analysis in both time (analyzing only small parts of the application software) and space (analyzing only a few components of the system), adversely affecting optimization opportunities and/or accuracy of evaluation. The following are some examples. Trifecta [46] focuses only on the ALU adder and selection logic. Roy and Chakarborty [51] only consider ALU errors and limit their simulations to 100 instructions that most frequently exercise the ALU. Xin and Joseph [64] focus on ALU, LSU, and Shift/Branch units. Hoang et al. [33] reduce the size of benchmark input sets due to extremely long run- times associated with gate-level simulations. Similarly, Sartori and Kumar [53] only study the LSU and IQ and do not simulate the benchmarks for their entire duration.

Simply stated, the lack of fast simulation platforms for these systems is becoming a bottleneck and a key challenge for timing speculation research [31], [64]. Earlier efforts produced limited success. Rehman et al. [50] assume that the area of functional units determine their error rate which, while acceptable in case of soft errors, is not extendable to errors caused by variability. Rahimi et al. propose error prediction for individual [47], [49] and sequences [48] of instructions,

but their models do not specifically take input data dependence into account. Others [51], [64] have proposed to use the Program Counter (PC) to predict errors, suggesting that various dynamic instances of an instruction behave similarly due to locality of input data. We will show in this dissertation that the assumption does not hold for all instructions, specifically those that exercise critical paths in processor datapath. Finally, an emulation testbed for variability-aware software called VarEMU is presented in [63]. While VarEMU provides a fast and easy-to-use platform for implementation of error models and may be useful for variability-aware software power analyses, it cannot handle the cycle-by-cycle nature of variability in causing errors due to the functional (and not cycle-true) operation of the underlying virtual machine.

More recently, performance analysis proposals that take one of the following approaches have been more successful.

Graph-Based DTA. Authors in [15] have proposed a graph-based DTA method that improves the efficiency of path-based techniques like ours. The approach is optimized for longer simulation times, as demonstrated in [14] where a safe, error-free operating point is determined for the entire runtime of an application. It is not as beneficial for TS processors where timing errors must be predicted on a cycle-by-cycle basis to determine the error rate and consider the dynamic effect of timing errors on DTS. The frameworks we propose in this dissertation do not suffer from the long simulation times of other path-based techniques because they perform the most time-consuming part of the analysis—calculating DTS of the control network—only once and only on short instruction sequences (basic blocks).

Machine Learning. There has been growing interest in using machine learning techniques to predict timing errors. Authors in [24] use decision trees to enable the compiler to predict timing errors. But since instruction operands are not available at compile time, their effect is ignored. In [38], random forest trees are used to construct timing error prediction models for functional units. Such models could be used in place of the datapath error model in our framework, but it is not clear if the methodology can be extended to the control network. Moreover, since, as classifiers, these methods predict timing errors directly, without estimating

DTS, they are not suitable for analysis of design-time uncertainty, like process variation, that precludes deterministic prediction of timing errors.

Timing Simulation. A number of other gate-level DTA techniques use timing simulations to predict timing errors [30, 17]. In addition to long simulation times necessary, because they rely on the simulator to perform DTA, they cannot use non-deterministic timing models that are necessary to analyze the effect of process variation. Our framework uses functional simulations coupled with STA to estimate DTS. That enables us to take process variation into account by replacing STA with statistical STA (SSTA). In fact, we are not aware of any existing DTA technique that includes process variation in the analysis, though the graph-based method in [15] can be extended to do so.

1.4 Dissertation Organization

The remainder of the dissertation is organized in five chapters.

We start in Chapter 2 with a comprehensive review of error detection and recovery techniques used by TS processors. We analyze the most important advances and challenges in designing error detection circuits and conclude that TS processors can be designed to be cost-effective.

Chapter 3 introduces our first DTA tool which uses a microarchitecture-level timing model based on clustering functionally similar timing paths of the processor, called clustered timing model (CTM). In the second part of this chapter, we use this CTM-based simulator to characterize error rate of instructions and programs. We propose inter- and intra-program variation as measures of error rate variability in different programs and among instructions of a program, respectively. We also characterize the error rate variation caused by the program input data and show that it is comparable to other sources of variability such as process variation. Finally, we present an analysis of the physical location of errors in hardware, identify regions in which most of the errors occur, and how different programs change the distribution of errors

among these regions.

The DTA technique in Chapter 3 balances the trade-off between performance and accuracy by performing the analysis at the microarchitecture level. In Chapter 4, we introduce another DTA technique that achieves better efficiency *and* accuracy by taking a hybrid approach. It combines an accurate gate-level DTA tool for the more complex but less frequent part of the analysis—the control network—with a fast architecture-level execution-driven simulator based on a high-level path activation model for the simpler part that needs to be performed repeatedly—the datapath. In the second part of the chapter, we propose an instruction-level error model that estimates the likelihood that an instruction experiences a timing error, capturing the effects of process and data variations as well as inter-instruction correlations caused by the error recovery scheme used by the TS processor. We then utilize two well-known laws of applied statistics, the law of small numbers and the law of large numbers, to estimate, with bounded inaccuracy, the number of timing errors the TS processor experiences while executing a specific application. The results demonstrate the significant effect of software in determining performance of TS processors.

In Chapter 5, we extend the DTA framework in Chapter 3 with the capability to implement dynamic frequency scaling to evaluate the speculation scheme and improve its efficiency by implementing some of the optimization techniques we used for the DTA tool in Chapter 4. Then, inspired by insights from the modeling of timing errors, we propose a new program-driven timing speculation scheme that improves the conventional control-based approach based on three strategies—selective local speculation, temporally-limited error sampling, and maximum throughput tracking. Our experiments show that, in a typical case, while the most commonly used timing speculation policy achieves only a 21.8% of the potential gains, our technique can realize up to 35.6% of the potential gains, improving throughput by 50.9% over a guardbanded design.

Finally, in Chapter 6, we discuss the results of the dissertation and summarize its conclusions.

Chapter 2

A Review of Timing Error Detection and Recovery Schemes

In this chapter, we present a review of the major error detection and recovery techniques used in TS processors in recent years. While both detection and recovery of timing errors can be performed at various levels of the hardware/software stack, in this chapter we focus on circuit-level detection and microarchitecture-level recovery techniques.

The chapter is organized in two sections. Section 2.1 presents a review of new circuits designed for detecting timing errors. These circuits which are referred to as Error Detection Sequential (EDS) circuits are designed to add timing error detection capabilities to the conventional sequential circuits (i.e. latches and flip-flops). Then in Section 2.2, we review a number of microarchitectural error recovery techniques that are designed to restore the correct system state after a timing error is detected by the EDS circuits.

2.1 Error Detection

In this section, we present a review of the major EDS circuits. These circuits are designed to add timing error detection capabilities to the conventional sequential circuits (i.e. latches and flip-flops) of digital systems. If the input data signal arrives late (i.e. after the clock edge) at their input pins, EDS circuits raise an error signal which is used by the error recovery logic to restore the correct system state. Error recovery schemes are reviewed in Section 2.2. The

major challenges in the design of EDS circuits are energy overhead and possible occurrence of metastability. An ideal EDS circuit would detect a timing error if and only if the input signal arrives after the clock edge while (i) enabling fast, low-overhead error recovery, (ii) incurring little energy overhead, and (iii) minimizing the probability of metastability. We introduce these issues in describing the first EDS circuit proposal, known as *Razor* [18] and explain how subsequent methods have tried to address them.

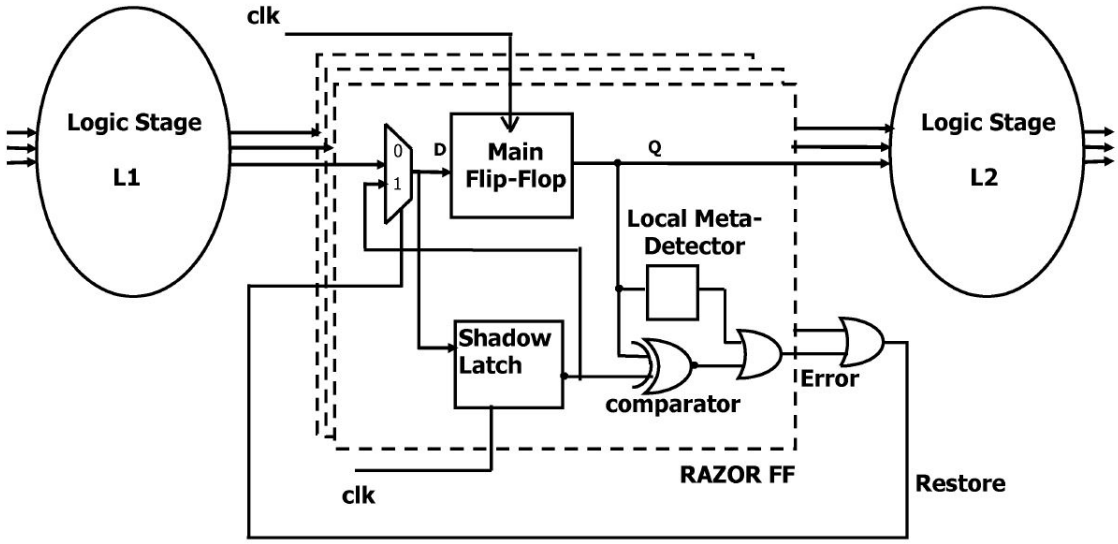
In general, EDS circuits take one of the following approaches to detect timing errors:

Double Sampling: In addition to the conventional flip-flop that samples the input data at the edge of the clock, these EDS circuits sample the data a second time *after* the clock edge. The second sample is guaranteed to be correct using design-time worst-case timing analysis. Consequently, the two samples are compared and a timing error is declared in case of a mismatch. EDS circuits described in Sections 2.1.1 and 2.1.2 are of this type.

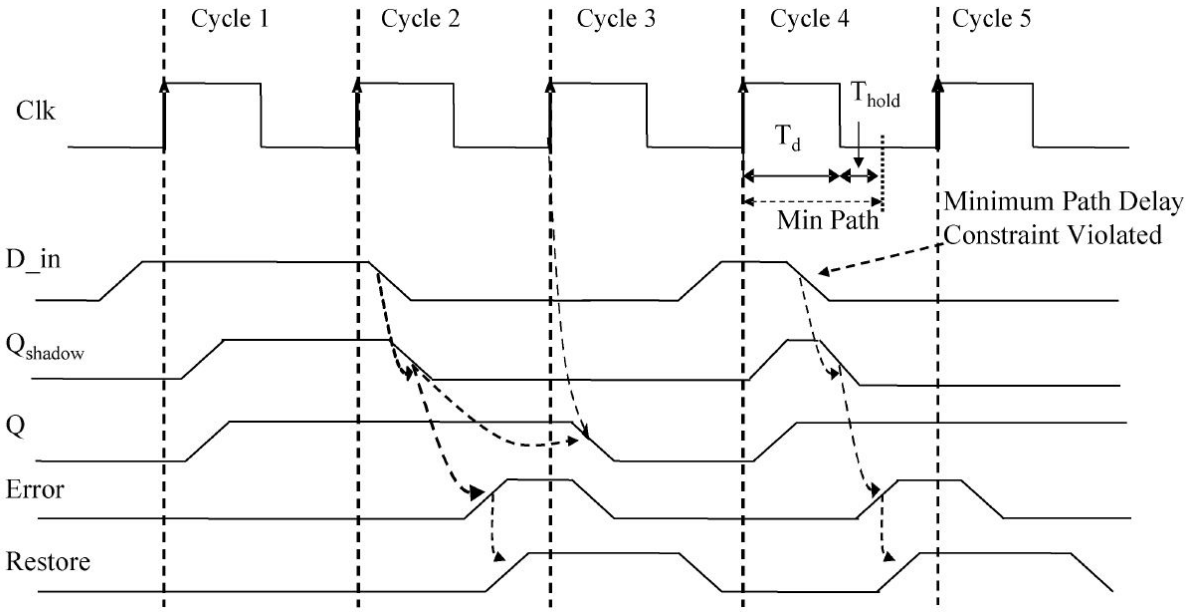
Transition Detection: Instead of a second delayed sampling, these EDS circuits dynamically monitor the input data or some internal node *after* the clock edge. A transition during this period is indicative of a late arriving input signal and a timing error is declared if such a transition is detected. EDS circuits described in Sections 2.1.4, 2.1.3, and 2.1.5 take this approach.

2.1.1 Razor

Figure 2.1a shows a block diagram of a Razor flip-flop (henceforth referred to as RFF). In addition to the standard edge triggered D-flip-flop (DFF), the RFF includes a *shadow latch*. Here the main flip-flop samples the input data at the rising edge of the clock while the shadow latch is transparent throughout the high clock phase. Therefore, a signal arriving at the input pin of RFF after the rising edge of the clock (cycle 2 in Figure 2.1b) causes different values to be captured by the main flip-flop and the shadow latch. This raises the *error* signal which then initiates the recovery mechanism. Note that the shadow latch always captures the correct value as long as the input data arrives no later than the falling clock edge. For this reason, the high clock phase is referred to as the *detection*, *sampling* or *speculation window* of the RFF. The



(a) Conceptual Representation



(b) Timing Diagrams

Figure 2.1. Design and Operation of Razor Flip-Flop [18] © 2006 IEEE

length of the detection window determines the maximum allowable delay of the incoming timing paths, and consequently the maximum amount of increase in frequency or decrease in supply voltage that the RFF can tolerate. The maximum path delay constraint is therefore defined as [6]

$$T_{max} \leq T_{cycle} + T_w - T_{setup} \tag{2.1}$$

where T_{max} is the maximum path delay, T_{cycle} is the clock cycle time, T_w is the detection window length, and T_{setup} is the setup time of the shadow latch. Equation 2.1 illustrates how a Razor flip-flop *relaxes* the maximum path delay constraint of a conventional DFF by the amount equal to the length of the detection window.

But, what if the input signal toggles during the high clock phase not due to a *slow* path from last cycle but because of a *fast* path in the current cycle? The RFF has no way of differentiating between these two cases and would indicate a false error if the former occurs. This is illustrated in Figure 2.1b where the input signal toggles too soon during the high clock phase of cycle 4. In fact, the length of the detection window lies at the heart of a fundamental trade-off in the design of EDS circuits. While a wide detection window is desirable to achieve maximum throughput and energy improvement, its length is limited by the minimum delay of fast paths. In order to eliminate the possibility of fast paths incurring false errors, Razor requires all paths to have best-case delays larger than the length of the detection window plus the hold time of the shadow latch. The minimum path delay constraint is therefore written as [6]

$$T_{min} \geq T_w + T_{hold} \tag{2.2}$$

where T_{min} is the minimum path delay, T_w is the detection window length, and T_{hold} is the hold time of the shadow latch. This constraint is met by inserting delay buffers in fast paths to increase their propagation delay. This incurs a power overhead to the design, working against the original goal of reducing it. Hence, the length of the detection window should be configured to maximize

power savings through voltage reduction while minimizing the power overhead from the insertion of delay buffers.

Equation 2.2 illustrates two important facts about a Razor flip-flop. First, RFF *restricts* the minimum path delay constraint of a conventional DFF by the amount equal to the length of the detection window. This is in contrast to the maximum path delay constraint of RFF which is *relaxed* by the same amount (See Equation 2.1). The maximum and minimum path delay constraints of an RFF are, therefore, linked together by the detection window. Energy efficiency improvements which are achieved by relaxing the maximum path delay constraint come at the cost of additional restriction on the minimum path delay constraint, which is met by adding delay buffers leading to diminishing energy efficiency gains. Second, the minimum path delay constraint only limits the maximum length of the high clock phase and imposes no restrictions on the clock frequency. Hence, the operating frequency can be chosen arbitrarily while the duty cycle of the clock is tuned for the minimum path delay constraint to be met.

Another important issue in the design of EDS circuits arises from the fact that by allowing the input signal to arrive after the rising edge of the clock, setup and hold time constraints of the main flip-flop are not respected. Therefore, if the input signal is only slightly late and toggles ”too close” to the rising clock edge, there is a possibility that the main flip-flop becomes metastable. Razor handles this issue by taking two measures. First, a local metastability detector is placed at the output of the main flip-flop and the RFF raises its error signal in the case it becomes metastable. This ensures that metastability-causing errors do not go undetected while, again, incurring additional energy overhead. Second, Razor requires at least two successive non-critical pipeline stages immediately before storage to eliminate the possibility of metastable signals being committed to memory. Guardbanding two pipeline stages runs contrary to the design motivation of Razor and can limit its ability to achieve optimal energy and/or throughput improvements.

For a timing error to go undetected, the resolution time (i.e. the time it takes for the RFF to resolve its state) must satisfy two timing constraints. First, the data-path signal must become

stable early enough so that a low-logic is sampled as the error signal. Therefore, the resolution time must satisfy

$$T_r < T_{cycle} - T_{error-path} - T_{setup} \tag{2.3}$$

Second, the additional delay incurred on the data-path by the resolution time must cause the subsequent RFF to fail its maximum path delay timing constraint. In other words

$$T_r + T_{datapath} > T_{cycle} + T_w - T_{setup} \tag{2.4}$$

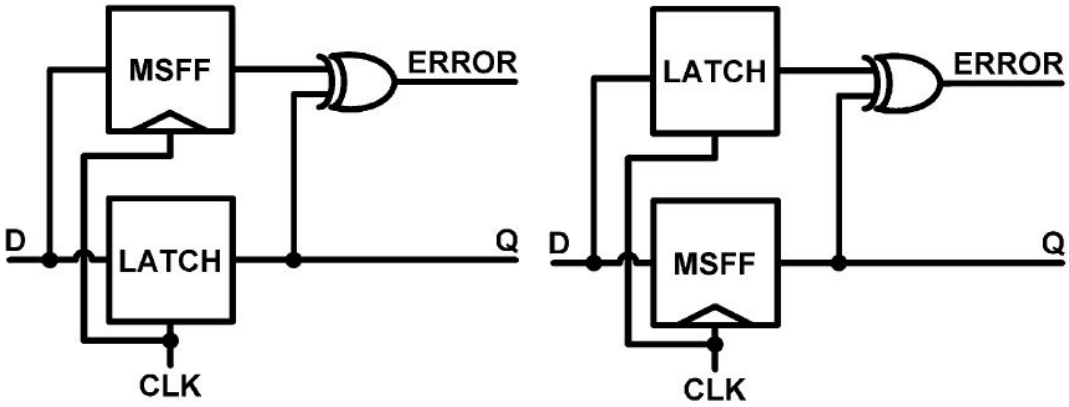
where $T_{datapath}$ is the propagation delay of the data-path. Hence, for these conditions to be met, the data-path propagation delay must approximately fall in the following range

$$T_{min} + T_{error-path} - T_{hold} < T_{datapath} < T_{max} \tag{2.5}$$

Since propagation delay of the error path ($T_{error-path}$) is typically a small fraction of the cycle time, a large number of data-paths would satisfy these conditions and result in undetected timing errors. As a result, it is imperative for an RFF to include a metastability detector in its data-path to prevent such an event.

2.1.2 Double Sampling with Time Borrowing (DSTB)

The second EDS circuit discussed is called Double Sampling with Time Borrowing (DSTB). The design of DSTB is very similar to that of the RFF. Figure 2.2a shows a conceptual view of a DSTB next to that of an RFF in Figure 2.2b. In DSTB, relative to RFF, the main flip-flop and the shadow latch have swapped places. Recall that since the setup and hold time constraints of the main flip-flop is not respected in RFF, there is always the possibility of it becoming metastable. The shadow latch, in contrast, would never be metastable as long as the maximum path delay constraint of the RFF (Equation 2.1) is satisfied. In RFF, the problematic DFF feeds both the data path and the error path. Hence, signals on both paths can become



(a) DSTB **(b)** RFF

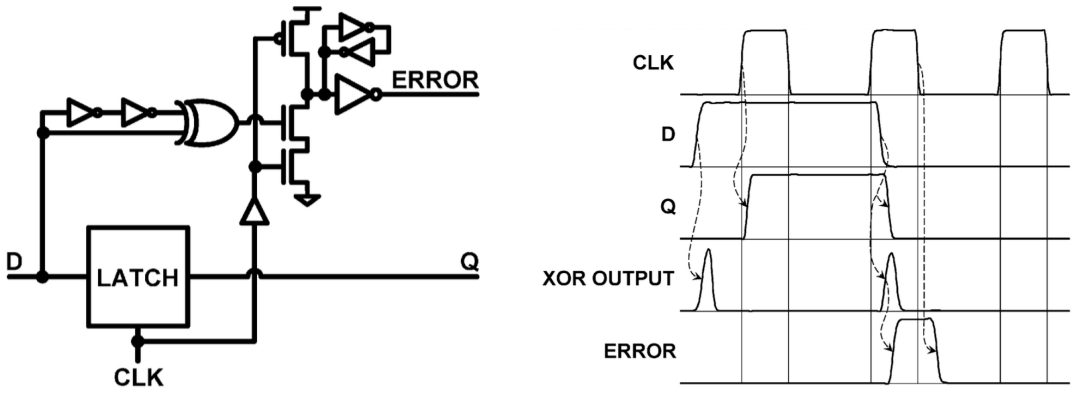
Figure 2.2. Conceptual Representation of DSTB and Razor Flip-Flop [6] © 2009 IEEE

metastable.

As discussed in Section 2.1.1, Razor requires a metastability detector to prevent undetected errors as well as two successive non-critical pipeline stages immediately before the memory to ensure that metastable signals do not corrupt the state of the processor, requirements that limit the benefits of timing speculation offered by Razor. In contrast, in DSTB, the data-path is driven by the shadow latch and the metastability issue is limited to the error path. An analysis of the metastability in DSTB (See [6] for more details) reveals that limiting metastability to error path causes the *Mean Time Between Failures* (MTBF) to be orders of magnitude larger for DSTB than for RFF, to the point that the metastability detector can be safely omitted from the EDS circuit, reducing the energy overhead of timing speculation.

2.1.3 Transition Detector with Time Borrowing (TDTB)

The next EDS circuit is Transition Detection with Time Borrowing (TDTB) [6]. Figure 2.3a and 2.3b show the circuit-level implementation and sample timing diagrams of the TDTB EDS circuit. In contrast to previous EDS circuits, TDTB implements a *dynamic* error detection scheme where timing errors are identified not by delayed re-sampling, but by dynamic monitoring of the input data using a transition detector. The XOR gate continuously compares

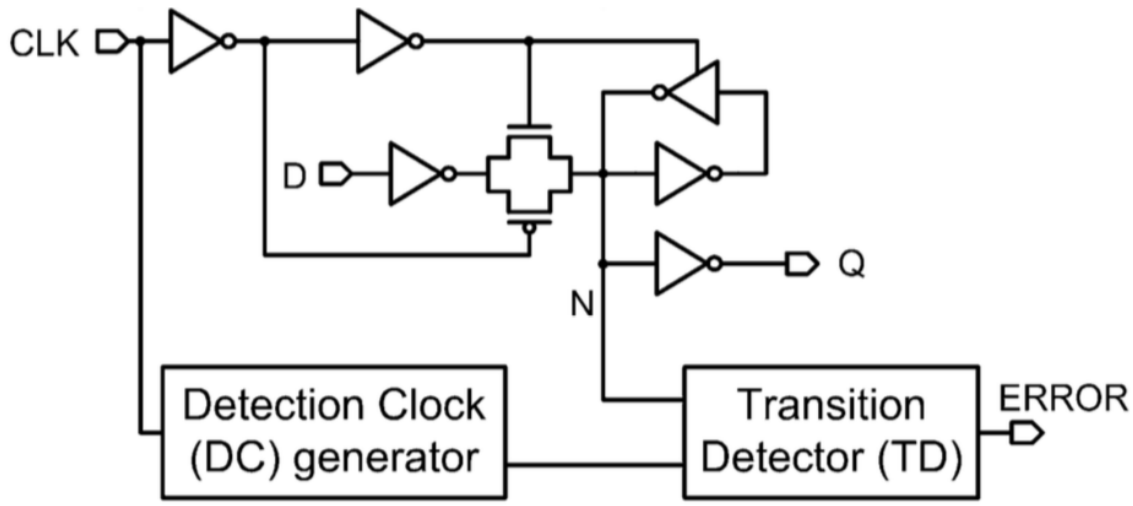


(a) Conceptual Representation **(b)** Timing Diagrams

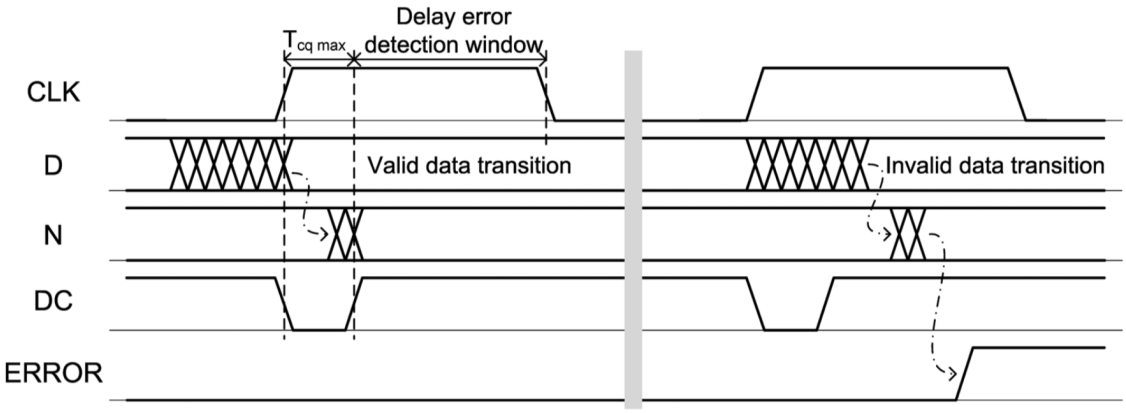
Figure 2.3. Design and Operation of TDTB EDS Circuit [6] © 2009 IEEE

the input data with its delayed version and produces a pulse when they are not equal, effectively detecting input data transitions. During the low phase of the clock, the *error* signal is disconnected from the transition detector and is driven by the top transistor to a logic low. As the data-path latch is opaque during this period, the EDS circuit behaves like a conventional flip-flop. However, if a late-arriving signal causes the input data to transition during the high clock phase, the *error* signal transitions to logic high and remains in this state until the falling clock edge brings it to a logic low again. Similar to DSTB EDS circuit, while time borrowing is potentially enabled by employing a level-sensitive latch in the data-path, it is suppressed by the activated *error* signal, effectively enforcing conventional edge-triggered flip-flop operation.

The above analysis shows that a TDTB design implements the same functionality as double sampling EDS circuits, including limiting the metastability issue to the error path. At the same time, TDTB improves the performance and energy efficiency of DSTB EDS circuits in the following ways. First, both size and clock energy of TDTB is significantly smaller than that of DSTB, and even the conventional master-slave flip-flop. Granted, conventional master-slave flip-flops can always be replaced with pulse-latches at the expense of additional design complexity to achieve lower clock energy. Second, both the propagation delay (CLK-to-Q) and setup time (defined as the minimum D-to-CLK delay prior to the rising clock edge such that an error signal is not generated) are improved in a TDTB EDS circuit, resulting in potential



(a) Conceptual Representation



(b) Timing Diagrams

Figure 2.4. Design and Operation of RazorII Flip-Flop [19] © 2009 IEEE

performance improvements. These benefits, however, come at the price of increased design complexity as well as sensitivity to within-die process variation. Moreover, the issues of error path metastability and minimum path delay constraint remain in TDTB.

2.1.4 RazorII

The next version of Razor flip-flop is called RazorII [19]. Figure 2.4a and 2.4b show the block diagram and sample timing diagrams of the RazorII EDS circuit. Similar to the TDTB design, RazorII uses a single latch and employs a transition detector to monitor dynamic behavior of the input signal and detect timing errors. Unlike the TDTB transition detector that directly

monitors the input data pin, the transition detector in RazorII is connected to the internal latch node and essentially detects transitions on the latch output rather than the latch input. This design decision trades off some of EDS circuit timing error coverage to achieve full soft error coverage. TDTB EDS circuit has a detection window equal to the high phase of clock for both timing and soft errors. RazorII, on the other hand, can detect soft errors during the entire clock cycle [1], while its detection window for timing errors is reduced. The reason for this will be clear after an analysis of RazorII's operation which follows.

During the low phase of the clock, the latch is opaque and no transitions happen at the internal latch node denoted by N in Figure 2.4a. The EDS circuit, therefore, operates similar to a conventional flip-flop during this time. A transition caused by a late-arriving signal during the high clock phase, on the other hand, toggles N as the latch is transparent at this time. The transition is detected by the transition detector and flagged as a timing error. However, even a legitimate transition at the EDS circuit input before the rising edge of the clock takes the time equal to the CLK-to-Q delay of the latch to appear at its output after the rising edge of the clock. Therefore, the transition detector must be disabled at the beginning of the clock cycle for a time greater than this delay to prevent flagging such transitions as timing errors. This is accomplished by the detection clock generator block that produces a negative pulse which deactivates the transition detector. In order to ensure correct operation, the length of this pulse must be guaranteed to be greater than the CLK-to-Q delay of the latch across all PVT corners. Hence, it is required that the *minimum* width of the negative pulse is greater than the *maximum* CLK-to-Q delay of the latch. While post-manufacturing tuning of the detection clock pulse width can be used to account for process variation, the width of the pulse should still be suitably margined. This causes the detection pulse to be longer than the CLK-to-Q delay of the latch. As a result, the transition detector remains disabled for an additional period equal to the difference between the pulse width and CLK-to-Q delay. During this time, transitions on the internal latch node are not detected and no timing errors are declared, thereby reducing the detection window

[1] Analysis of soft error tolerance is beyond the scope of this dissertation. See [19] for details.

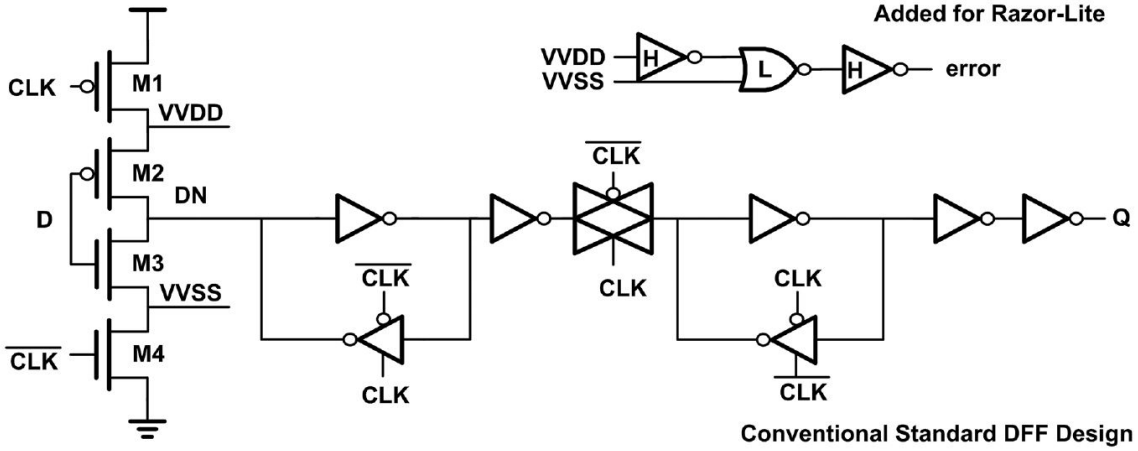


Figure 2.5. Circuit-Level Implementation of a Conventional Flip-Flop and Added Razor-Lite EDS Circuit [41] © 2014 IEEE

width of the EDS circuit. If the adjoining path in the next pipeline stage has ample timing slack, correct operation is maintained through time borrowing, resulting in even more performance improvements. However, there is always the possibility of system failure if multiple stage time borrowing is accumulated beyond the ability of RazorII flip-flops error tolerance. Complex design-time timing analysis of time-borrowing is, therefore, required to guarantee correct system operation.

Similar to DSTB and TDTB EDS circuits, RazorII successfully limits possible metastability to the error path while the minimum path delay constraint problem remains unresolved. It accomplishes improved energy efficiency compared to double sampling EDS circuits at the expense of additional design complexity and sensitivity to process variation. Compared to TDTB, soft error coverage is extended to the entire clock cycle, but timing error detection window is shortened and sensitivity to process variation is increased.

2.1.5 Razor-Lite

All EDS circuits discussed so far implement error detection capabilities at the expense of, among other things, significantly increasing the energy consumption of conventional sequential circuits. Razor-Lite [41] is a more recent EDS circuit designed to mitigate this problem.

Figure 2.5 shows circuit-level implementation of a conventional flip-flop along with the error detection circuit added by Razor-Lite. The detection circuit uses two internal nodes of the flip-flop's input buffer, called *virtual rails* and denoted by VVDD and VVSS in Figure 2.5, to detect timing errors. To understand how this is accomplished, it is useful to analyze how these two nodes (VVDD and VVSS) behave during the absence and presence of timing errors.

During the low phase of the clock, M1 and M4 are both on and VVSS and VVDD are driven (by ground and VDD) to logic-low and logic-high, respectively. DN is connected to one of the virtual rails based on the value of the input data D by turning on either M2 or M3. During this time, transitions on D change the value of DN but VVDD and VVSS remain constant. At the rising edge of the clock, both M1 and M4 are turned off and the virtual rails start floating. Note, however, that either VVDD or VVSS remains connected to DN through the M2 or M3. If no timing errors occur and D does not transition during the high clock phase, VVDD and VVSS remain high and low, respectively. However, a transition on D caused by a late-arriving signal changes the states of M2 and M3, disconnecting the previously connected virtual rail from and connecting the other one to DN. Since the newly connected virtual rails has the opposite state as DN, the feedback inverter of the master latch pull it toward the value of DN. In other words, an input data transition during the high clock phase causes either a low-to-high transition on VVSS or a high-to-low transition on VVDD. Such transitions do not occur in the absence of timing errors, and can, therefore, be used to detect timing errors.

The detection circuit consist of two high-skewed inverters and a low-skewed OR gate. Skewed logic is used to take advantage of the unidirectionality of the transitions to speed up the error path. An important property of these error-indicating transitions in Razor-Lite is that they never occur during error-free operation, neither in the low nor in the high phase of the clock. This is in contrast to the error-indicating transitions monitored by previous EDS circuits with transition detection that indicated timing errors only during the detection window. As a result, unlike previous transition detectors that needed to employ the clock signal, Razor-Lite EDS circuit implements a static transition detector, incurring no additional clock load. Extra

data-path loading is also avoided. As a bonus, setup and hold times of the circuit are slightly reduced. Energy overhead is consequently limited to less than 3% as only 8 transistors are used to implement error detection. As a point of comparison, the most light-weight previous EDS circuit, TDTB, uses 15 transistors and incurs around 10% of energy overhead for error detection while also incurring extra clock and data-path loadings that add larger overheads to the performance (CLK-to-Q) of the EDS circuit.

In addition to the energy-efficiency of the EDS circuit, Razor-Lite is inherently more resilient to metastability. While the possibility of metastability in data-path is present similar to Razor, the use of skewed logic for error detection ensures with a high confidence that long-term metastable events are identified as timing errors. Short-term metastable events are either detected as timing errors (if they resolve to the correct state) or add a small delay to the CLK-to-Q delay of the EDS circuit. It is reasonable to assume that this small additional propagation delay can be absorbed by the next pipeline stage.

Finally, Razor-Lite mitigates the problem of minimum path delay constraints by implementing a duty cycle controller to dynamically adjust the duty cycle of the clock. An algorithm for initial and run-time calibration of duty cycle is proposed that minimizes false error detections due to fast paths. Initial calibration is performed at half-frequency and by reducing the duty cycle from 50% until no fast path errors are detected. At run-time, duty cycle is tuned during error recovery where again the processor works at half-frequency and replays the errant instruction. If another timing error is detected during this time, both errors are assumed to be the result of a fast path violation and the duty cycle is reduced to suppress them. Duty cycle is increased when too many slow path violations (i.e. true timing errors) are detected. This strategy mitigates the energy overhead incurred by the delay buffers added to fast paths to satisfy minimum path delay constraints of EDS circuits and further improves the energy efficiency of Razor-Lite.

2.2 Error Recovery

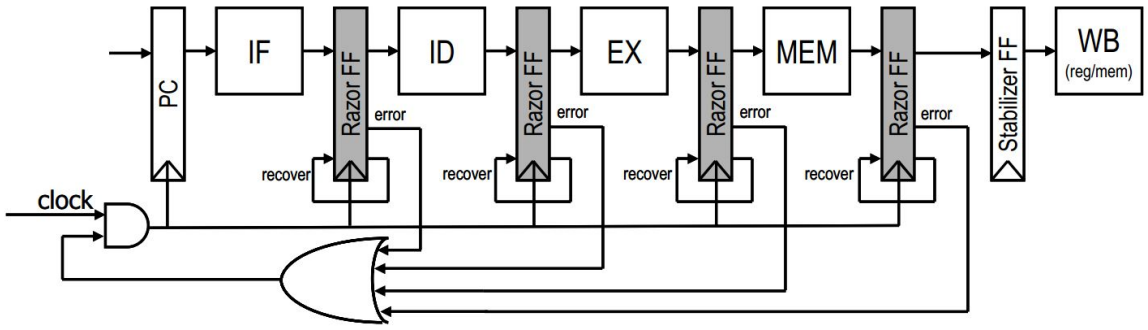
When a timing error is detected by EDS circuits, they raise an *error* signal that initiates error recovery. The error recovery logic is responsible for restoring the correct system state by (i) preventing the propagation of the error and (ii) maintaining the correct order of instruction executions. Recovery penalty is defined as the average additional time spent for an errant instruction to correctly finish execution compared with the error-free case. A large recovery penalty significantly limits the potential energy/performance improvements that timing speculation can achieve. Various error recovery mechanisms can be characterized, among other things, by how they choose to balance the trade off between the recovery penalty and the energy overheads of EDS circuits and the recovery logic. Other desirable properties include manageable design complexity including architecture independence, un-intrusiveness, and automatic design and analysis using CAD tools.

The error recovery techniques discussed in this section take one of the following approaches to accomplish these tasks.

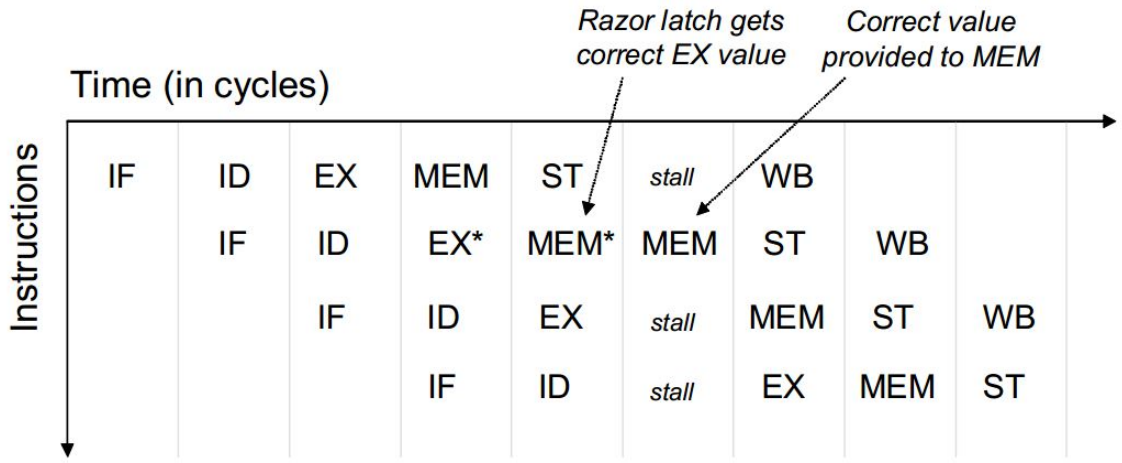
Local Error Correction: This recovery technique relies on the ability of the error detection circuit to correct its own state. Therefore, this approach cannot be realized using the EDS circuits discussed in Section 2.1 except first Razor EDS circuit reviewed in Section 2.1.1 which implements local error correction. The techniques discussed in Sections 2.2.1 and 2.2.2 are of this type.

Instruction Replay: It is not strictly necessary to correct the timing error as long as it is prevented from corrupting the architectural state of the processor. A number of recovery techniques take advantage of this observation to enable the use of simpler EDS circuits that have smaller energy overheads. The errant instruction is simply replayed until it completes without experiencing errors. This, of course, results in a larger recovery penalty. These methods are discussed in Section 2.2.3.

Error Masking: Data corruption can be avoided by using EDS circuits that have a latch



(a) Microarchitecture Design



(b) Pipeline Timing during Error Recovery

Figure 2.6. Design and Operation of Clock Gating [23] © 2003 IEEE

in their data-path, by taking advantage of their ability to mask timing errors using time borrowing. This approach incurs some additional design complexity but achieves a smaller recovery penalty. Techniques discussed in Sections 2.2.4 and 2.2.5 take this approach.

2.2.1 Clock Gating

Figure 2.6a and 2.6b show the microarchitecture design of clock gating and a timing diagram of the pipeline during the recovery of a timing error in the EX stage. This technique relies on the capability of the EDS circuits to perform *local* or *in situ* error correction. The Razor EDS circuit reviewed in Section 2.1.1 (RFF) implements this functionality. Error signals of all RFFs are simply OR-ed together and produce a *recover* signal which is used as the global clock gating control signal. When a timing error is detected, the entire pipeline is stalled for one cycle

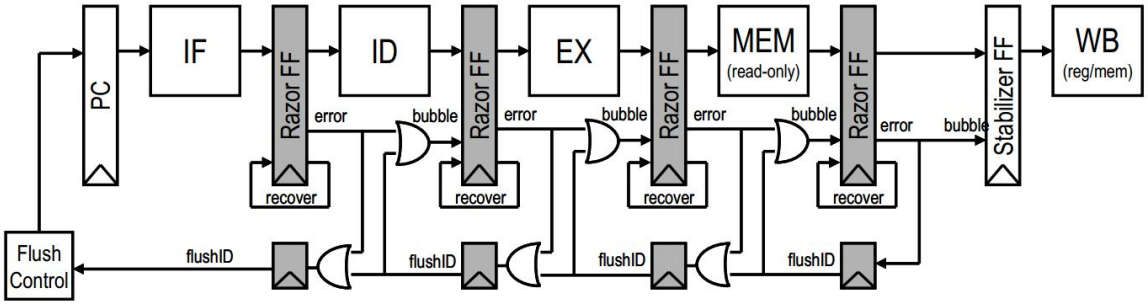
during which all RFFs reload their main flip-flops with the value in their shadow latches. Normal execution continues with the next clock cycle. The recovery penalty is only one clock cycle, even when multiple errors occur during one clock cycle.

Other than reliance on an EDS circuit design with local error correction capability, this scheme meets all design goals of error recovery. Recovery penalty is only one clock cycle and the recovery logic incurs small area and energy overheads. Moreover, it can be implemented independent of the processor architecture using existing CAD tools. The main drawback arises from the strict requirement on the error path delay. For correct operation, the timing error must be detected, translated into the recover signal, and routed to all flip-flops, in less than a clock cycle. This is impractical in high-performance processors where the chip-wide wire delays alone are more than one clock cycle.

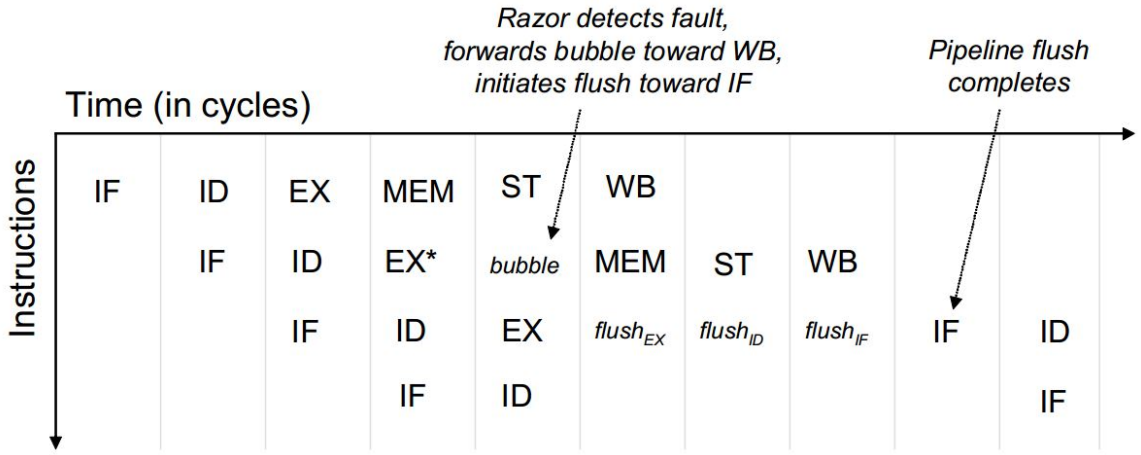
2.2.2 Counterflow Pipelining

Figure 2.7a and 2.7b show the microarchitecture design of counterflow pipelining and a timing diagram of the pipeline during the recovery of a timing error in the EX stage. Similar to clock gating, this technique relies on EDS circuits with local error correction capability. Using the counterflow pipelining concept [55] of instruction results (error signals in this case) moving backwards in the pipeline, this technique eliminates the problematic timing constraint of the error path in clock gating at the expense of increasing the recovery penalty.

The detection of a timing error initiates two actions. First, a *bubble* is sent to downstream pipeline stages nullifying the computation in the stage following the errant stage. This prevents the propagation of the error. Second, a *flush* train carrying the stage identifier of the failing stage is launched backwards. The flush train inserts a bubble as it passes through upstream stages nullifying the succeeding instructions. The errant instruction is corrected by the EDS circuits in the following clock cycle and continues execution. Once the flush train reaches the start of the pipeline, execution at the instruction following the errant one. The recovery penalty consists of the time for the flush train to reach the start of the pipeline and the time for the re-



(a) Microarchitecture Design



(b) Pipeline Timing during Error Recovery

Figure 2.7. Design and Operation of Counterflow Pipelining [23] © 2003 IEEE

executed instructions to return to their corresponding stages before the timing error. Counterflow pipelining, therefore, incurs a recovery penalty of $2N$ cycles where N is the maximum (in case of multiple errors) depth of the errant stages in the pipeline. This value ranges from 2 to $2S$ with an average value of $S+1$ (assuming uniform error probabilities across the stages), where S is the number of pipeline stages.

Counterflow pipelining remains reliant on the error correction ability of the EDS circuits while it successfully eliminates the error path delay constraint of the clock gating scheme at the expense of increased recovery penalty, area and energy overheads, and design complexity.

2.2.3 Instruction Replay

The design of instruction replay error recovery techniques is motivated by the observation that energy and/or performance gains from aggressive voltage scaling or overclocking is mitigated by the exponential increase in the rate at which timing errors occur. Therefore, the processor should be operated near the first point of failure where error rates are low. At this point, the energy of the EDS circuits and recovery logic rather than the recovery penalty is the main contributor to the energy/performance overhead of timing speculation. As a result, the recovery mechanism is designed to trade off the recovery penalty for the energy overhead of EDS circuits and recovery logic. This is accomplished in two steps. First, no error correction mechanism is implemented, neither by the EDS circuit nor the microarchitecture. Error signals are simply passed along the pipeline and serve in the write-back stage as write enable controls for the register file and memory to prevent the corruption of the architectural state. Second, the recovery mechanism is very similar to the logic already present in most processors for branch miss-prediction and can share resources with it to decrease the energy overhead.

In order to ensure that replaying instructions resolves the timing errors, one of the following approaches, or a combination of them is taken. For a timing error to occur on a path, the clock cycle should be smaller than its propagation delay and transitions at the start of the clock cycle should sensitize the path (i.e. toggle all its nets). The two approaches correspond to

these two requirements and seek to reduce or eliminate their probability.

Slow Execution: In this approach, clock frequency is substantially reduced (typically halved) such that the clock cycle is longer than worst case delay of all the paths, thereby eliminating the possibility of timing errors. Minimum path delay constraints are guaranteed to be satisfied by maintaining the high phase delay of the clock. In addition to doubling the penalty for flushing the pipeline from S cycles to $2S$ cycles where S is the number of stages in the pipeline, this approach incurs an instruction replay penalty of $2S$ cycles.

Path Sensitization Reduction: This approach reduces and ultimately eliminates the probability of the failing paths being sensitized. It is accomplished by N back-to-back executions of the errant instruction. In a pipeline with S stages, the probability of failing paths being sensitized by the Nth execution reduces as N is increased and reaches zero for $N = S+1$. The first $N-1$ executions set register input values without changing the architectural state and the last execution is the only valid one. Each increment of N sets more register input values and reduces path sensitization probabilities, eventually preventing all signal transitions at $N = S+1$. While the worst-case total recovery penalty of this approach which equals $3S$ cycles ($2S$ cycles for instruction replay plus S cycles for flushing the pipeline) is smaller than the penalty of halving the frequency which equals $4S$ cycles ($2S$ cycles for instruction replay plus $2S$ cycles for flushing the pipeline), careful selection of N can further reduce the total penalty to $2S+N-1$. If the selected N is too small, however, the effective recovery penalty would be larger than the first approach as the entire recovery mechanism must be repeated to ensure correct execution.

A RazorII implementation [19] uses a combination of the above techniques to recover from a timing error as follows. First, the recovery mechanism is initiated by the error signals. Next, the entire pipeline is flushed. Then, the errant instruction is replayed for a maximum of N_{max} times, called the replay limit. If the error persists through all the replays, a final replay is issued at half frequency to guarantee correct execution. Experimental results indicate that around 60% of the errant instructions do not require frequency reduction when $N_{max} = 2$. With $N_{max} = 2$, errant instructions are replayed once at the current frequency and, in case of a repeated

error, once at the half frequency.

An Intel research processor [7] implements two slightly different policies. The first one, called instruction replay at half frequency, essentially implements the above technique for $N_{max} = 1$. This policy has also been implemented in an ARM processor using RazorII EDS circuits [9]. The second policy proposed in [7] does not reduce the frequency and relies on the second approach only. First the errant instruction is replayed N times with only the Nth execution allowed to change the architectural state. If the error persists even in the Nth execution, the instruction is replayed with $N = S + 1$ to ensure correct execution.

Selecting the optimal policy requires a comprehensive analysis of the system's error behavior while the effect of the typical workload is taken into account. This highlights the need for fast simulation/emulation platforms that enable extensive design space exploration.

2.2.4 Bubble Razor

With the exception of clock gating which is architecture-independent, all the recovery mechanisms discussed so far must be implemented during the design of the microarchitecture at the register transfer level. This increased design complexity substantially limits their scalability to large high-performance processors. Bubble Razor [26] proposes a distributed and architecture-independent error recovery mechanism to provide improved scalability. The basic idea is to convert a conventional flip-flop based design into a two-phase latch based design. The flip-flops are broken into their constituent master and slave latches and the master latch is moved to backwards in the data-path to achieve a new balanced pipeline with twice the number of stages as the original pipeline. Any of the EDS circuits discussed in Section 2.1 with a latch in its data-path can be used. A latch clustering scheme is used to mitigate the overhead incurred by the additional latches.

A key property of the new two-phase latch based design is that consecutive latches operate out of phase. In other words, when a latch is transparent, all its neighbors are opaque and vice versa. This property is extremely beneficial to a TS system by producing two main consequences.

First, it restores the minimum path delay constraints of the design to their conventional forms by breaking their link to the detection window length and minimum path delay constraints. Recall that the minimum path delay constraints of EDS circuits is substantially more restricted than conventional flip-flops due to the requirement of differentiating between slow paths from the last clock cycle and fast paths from the current clock cycle. This problem does not exist in a two-phase latch based pipeline. An input signal arriving during the transparency of period of a latch is guaranteed to be a slow path from the last clock cycle because the neighboring latches are closed during this time and do not launch new signals. Recall that the minimum path delay constraints of EDS circuits substantially limit the energy and performance benefits of timing speculation. Bubble Razor eliminates this limitation.

Second, since latches operate out of phase, they can be stalled one after the other without losing data. In flip-flop based pipelines, flip-flops must be stalled simultaneously to avoid the loss of data as launching and capturing operations are performed at the same time. A two-phase latch based pipeline, on the other hand, interleaves the launch and capture operations. Therefore, when a latch stalls, data is not launched towards it for a period of one clock phase. This time difference can be used to communicate the stall signal to all neighbors, causing them to stall one clock phase later as necessary. Since neighbors are by definition less than one clock phase apart (otherwise normal data communication would have been impossible), stall signals are guaranteed to reach their destinations in time. Therefore, it is possible and practical to implement a scalable clock gating scheme to achieve a one-cycle error recovery penalty.

Figure 2.8 illustrates the error recovery mechanism of Bubble Razor. Once a timing error is detected by a latch, it communicates the error to the next stage, causing it to stall by skipping its next transparent phase. This provides additional time for the late-arriving signal to reach the next latch. This mechanism essentially implements a *controlled* time borrowing scheme where the amount of borrowed time is always equal to one clock cycle. While a latch based design such as the one used by Bubble Razor can mask timing errors by *continuous* time borrowing without paying any recovery penalty, a failure is possible if the time borrowing compounds through

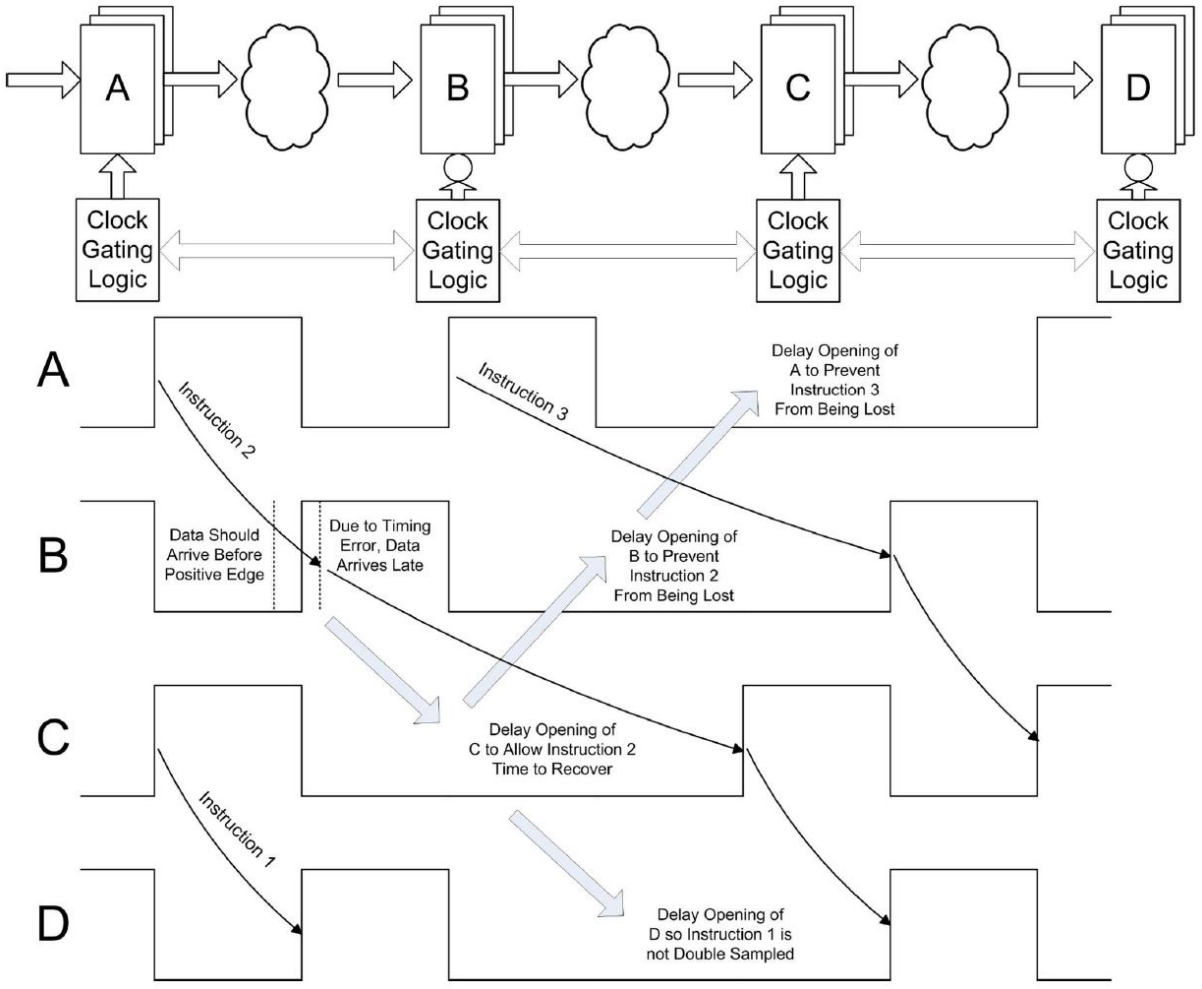


Figure 2.8. Two-Phase Latch Based Pipeline Used for Error Recovery in Bubble Razor [26] © 2013 IEEE

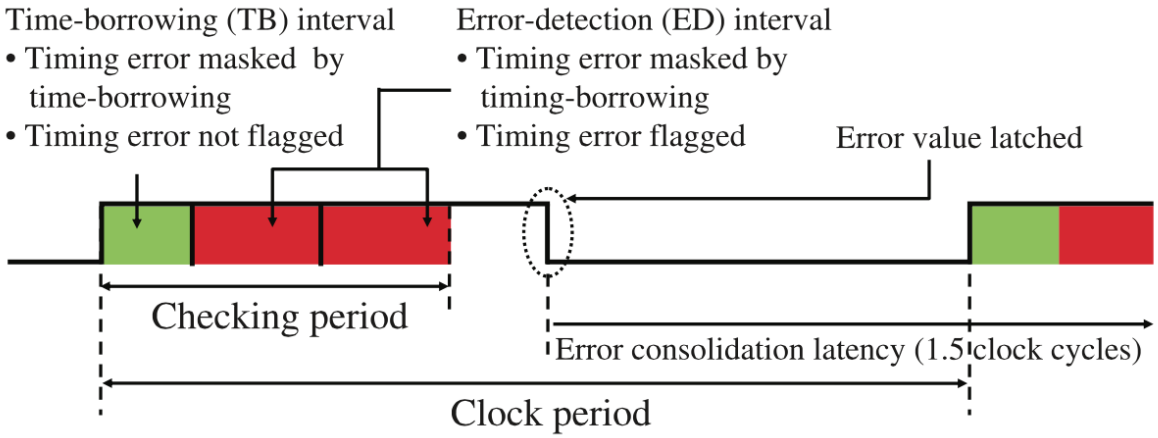


Figure 2.9. Timing Diagram of TIMBER Operation [16] © 2014 IEEE

multiple consecutive failing stages. The discrete time borrowing scheme implemented by Bubble Razor avoids the complex analysis required for verification against this effect at the expense of paying a one cycle recovery penalty for all timing errors including the ones that would not induce failures. Going back to the recovery mechanism, when a stage receives an error signal, it starts the bubbling process by sending bubbles to all its neighbors. A latch receiving bubbles from one or more of its neighbors stalls and sends bubbles to the neighbors it did not receive bubbles from.

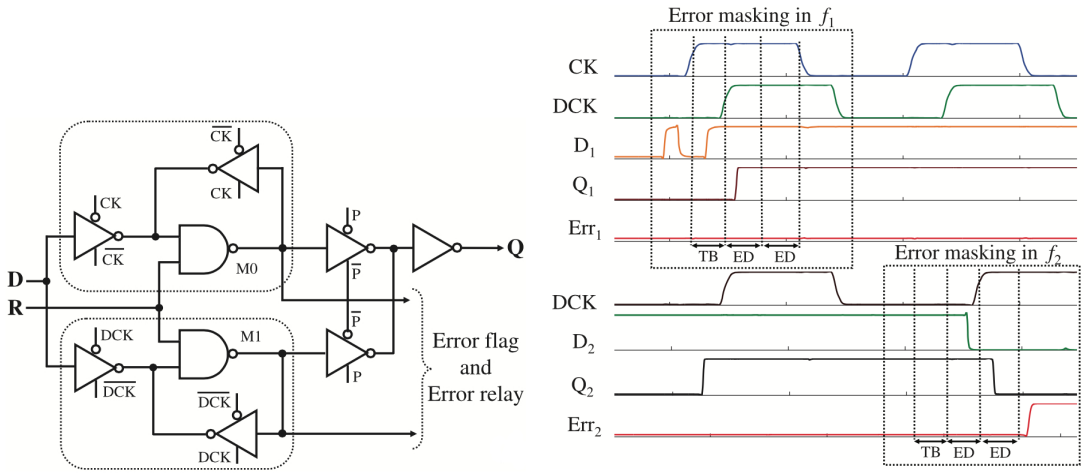
This recovery mechanism (with a small modification not discussed here) guarantees error recovery and forward progress even in the face of multiple timing errors by paying a constant one-cycle recovery penalty. This comes at the expense of an increased number of latches compared to a conventional latch-based design that is required with instruction replay recovery scheme. The increased energy overhead incurred by these extra latches is compensated by the low-overhead recovery mechanism and the elimination of large energy overheads as a result of more relaxed minimum path delay constraints.

2.2.5 TIMBER

TIMBER [16] introduced a pair of new EDS circuits (TIMBER flip-flop and TIMBER latch) based on the concept of double sampling. Unlike other EDS circuits that are designed

to be accompanied by some error correction scheme to recover from errors, TIMBER EDS circuits enable a different recovery mechanism based on *time borrowing* and *error relaying*. Design of TIMBER is motivated by two observations. First, as frequency is increased (or voltage is decreased) past the point of first failure, the rate at which timing errors occur increases exponentially, and benefits of aggressive overclocking (or voltage scaling) are exceeded by the large cost of error recovery. As a result, a large fraction of wide detection windows remain unused by the error-rate-aware DVFS operation of the processor. Accordingly, TIMBER is designed with a narrow detection window. Second, while a processor may have a large number of critical paths, only a small fraction them are connected together by flip-flops. In other words, most critical paths are preceded and followed by non-critical paths. The narrow detection window of TIMBER allows it to *mask* timing errors in a pipeline stage by borrowing time from the next stage. Indeed, this comes at the expense of large recovery costs for multiple-stage timing errors (i.e. errors spanning multiple successive pipeline stages).

Figure 2.9 illustrates design concept of TIMBER. Checking period is the time after the rising clock edge during which possible late signals may arrive at the EDS circuits. This period is divided into a time borrowing and two error detection intervals.The length of the time borrowing interval determines the amount of overclocking allowed such that a single-stage timing error is guaranteed to arrive during the time borrowing period. This timing error is masked by borrowing the time borrowing interval of the next pipeline stage. An error signal is relayed to the next stage so that incoming signal is sampled later, but no errors are flagged to the central control unit. If a timing error occurs in the next stage as well (i.e. a two-stage error), the signal to the endpoint of the second path is similarly guaranteed to arrive in the first error detection window. This error is similarly masked by borrowing the first error detection interval of the next stage. The number of error detection intervals determines the number of additional successive errors after the first timing error that can be tolerated. In order to prevent borrowed times from accumulating, in addition to the error relay signal, an error flag is raised after detection of a timing error in the first error detection interval that causes a reduction in clock frequency. Because no error correction



(a) Circuit-Level Implementation **(b)** Timing Diagrams

Figure 2.10. Design and Operation of TIMBER Flip-Flop [16] © 2014 IEEE

mechanism is present, the frequency must be reduced to a safe level so that a non-maskable error does not occur. The second error detection interval ensures that a possible additional error in the next stage can also be masked to account for the latency in error consolidation and frequency reduction.

Figure 2.10a and shows the circuit-level implementation of the TIMBER flip-flop. Design concepts of the TIMBER latch is similar and are not discussed here (see [16] for details). Latches M0 and M1 take turns in driving the inputs to the next stage by setting signal P such that inputs are driven by M0 during the time borrowing interval and by M1 during the rest of the clock cycle. Each TIMBER flip-flop includes additional logic (not shown) for producing the delayed clock (DCK) and relaying error signals to the next stage. Incoming error relay signals are used to generate DCK by delaying the clock signal appropriately. If the previous stage has not relayed error signals, the delay is equal to the length of time borrowing interval. It is increased by one or two error detection interval lengths if one or two previous stages have experienced errors. Figure 2.10b the timing diagrams for a two-stage error occurring at flip-flops f_1 and f_2. M0 samples the input data on the rising edge of the clock and drives next stage input signals during the time borrowing interval. M1 samples the input data again on the rising edge of the delayed

clock, guaranteeing that the correct value is stored. If no timing error occurs, M1 starts driving next stage input signals, starting at the end of the time borrowing period for the rest of the clock cycle, with the same value as in M0 and the EDS circuit operates similar to a conventional flip-flop. In case of a timing error, the correct value is launched a time borrowing interval late. An error signal is generated at the falling edge of the clock and relayed to the next stage flip-flops to inform them of the late-arriving signal.

Minimum path delay constraints are improved in TIMBER due to the smaller detection window. However, metastability remains an issue and a metastability detector must monitor the output of M0 to detect metastable signals in case of a multiple-stage timing error. Since correct operation relies on accurate generation of the delayed clock, guardbands are required for the clock control logic. Moreover, the complex design of this EDS circuit suffers from high sensitivity to process variation. While TIMBER flip-flop incurs a large energy overhead compared to previous EDS circuits, accurate comparison of their relative efficiency requires detailed information about single as well as multiple-stage error rates.

Chapter 3

Performance Analysis at the Microarchitecture Level

This chapter is organized in two parts. First, we introduce a process-variation-aware microarchitecture-level timing model based on clustering functionally similar timing paths of the processor, called clustered timing model (CTM), and verify its accuracy across a range of voltage-temperature corners. We then implement the model in an architecture-level simulator—although the timing analysis is effectively performed at the microarchitecture level—that estimates the likelihood of each executed instruction experiencing a timing error.

In the second part of this chapter, we use this CTM-based simulator to characterize error rate of instructions and programs. We propose Inter- and Intra-Program Variation as measures of error rate variability in different programs and among instructions of a program, respectively. We also characterize the error rate variation caused by the program input data and show that it is comparable to other sources of variability such as process variation. Finally, we present an analysis of the physical location of errors in hardware, identify regions in which most of the errors occur, and how different programs change the distribution of errors among these regions.

3.1 Introduction and Problem Definition

In this chapter, we seek to answer the following questions:

1. Do different programs behave similarly on the same processor and cause similar error

rates? If not, by how much do these differ?

2. Will the error rates remain unchanged when the program is rerun with different input data? If not, can we quantify the effect of input data? Is the variability in this case as significant as other sources such as process variation?

3. Which instructions/parts of the program cause more errors (how is the error distribution like among instructions)? Do these instructions/parts of the program have something in common?

4. Where do errors happen in hardware (how is error distribution like among different modules of the processor)? Does this distribution change with different programs?

In trying to find answers to these questions, we make three main contributions:

1. We start by constructing an accurate timing analysis framework to enable variability-aware analysis of the error behavior of small pieces of code. The framework takes advantage of industry-standard timing analysis methods and provides accurate dynamic delay distributions of an arbitrary circuit block while considering environmental conditions (i.e. voltage and temperature), process variation including its within-die spatial correlation property, and the timing paths sensitized by the specific instruction sequence and input data (Section 3.3).

2. We introduce *Clustered Timing Model* (CTM) as a high-level timing model for dynamic estimation of errors. In order to enable fast implementations, CTM relies on grouping functionally similar timing paths and modeling their timing behavior as a function of their specific operation. We develop a CTM for LEON3 as a representative in-order RISC processor and use our timing analysis framework to verify the accuracy of the model and demonstrate its robustness across a wide range of voltage-temperature corners with an average error of 3.9% (max. 6.7%). Moreover, we discuss important properties of

the model such as modularity and hierarchy which enable its easy use and re-use during different stages of system design (Section 3.4).

3. We present an analysis of representative software error behavior by introducing four aspects of the error behavior of erroneous systems. *Inter-* and *Intra-Program Variation* which represent the error rate variability in different programs and among instructions of a program, respectively, are discussed. We also characterize the error rate variation caused by the program input data and show that it is comparable to other sources of variability such as process variation. Finally, we present an analysis of the physical location of errors in hardware, identify regions in which most of the errors occur, and how different programs change the distribution of errors among these regions (Section 3.5).

3.2 Background

In this section, we present a basic overview of how errors occur and why it is difficult to model the error behavior of a processor. We would like to note that this description does not aim for absolute accuracy and makes some assumptions for a clearer explanation.

A sequential circuit contains a set of combinational blocks, each enclosed within two sets of registers that save the state of the circuit during each clock cycle. Each combinational block has a set of inputs and outputs and is composed of paths of logic gates connecting the inputs to the outputs. Each path starts from an input, goes through a set of logic gates and terminates at an output. Assuming a constant propagation delay for each gate, the propagation delay of a path is the sum of the propagation delays of all its gates. Static Timing Analysis (STA) computes the propagation delay of each combinational block as the delay of its longest path and the minimum clock period of the sequential circuit as the maximum of the delays of all its combinational blocks which is called the *static minimum clock period*. If the circuit operates at a higher clock frequency (i.e. lower clock period), at least one of its paths with a propagation delay larger than the clock period fails the timing requirement of the circuit. If a transition on the inputs

of combinational blocks needs a failing path to reach the outputs, an incorrect value will be registered at the destination register and cause an error. However, other transitions that need non-failing paths to propagate their input transitions to the outputs continue to operate correctly.

At each clock cycle, the *dynamic minimum clock period* can be calculated as the delay of the longest *sensitized path*. A path is called sensitized or *activated* when all its composing gate outputs toggle their values. Since only a subset of paths are sensitized during each clock cycle, the circuit can still operate correctly at this frequency even though it is higher than the static maximum frequency. The calculation of the dynamic maximum frequency requires the identification of the sensitized paths. While dynamic simulations can be used to achieve this, their high computation complexity renders this solution prohibitively time consuming. We will explain a framework that takes this approach in Section 3.3 in detail.

The within-die component of process variation and its spatial correlation property further exacerbates this problem by non-uniformly affecting gate propagation delays. While the set of activated paths is determined solely by input transitions, the propagation delay of paths, and hence, the dynamic maximum frequency of the circuit cannot be deterministically calculated. When process variation is considered, the path delays become statistical distributions rather than deterministic numbers. Moreover, these distributions are statically correlated due to spatial correlation of process variation. A path that would nominally pass or fail the timing requirements may now do otherwise with a certain probability. While Statistical Static Timing Analysis (SSTA) computes the static delay distribution of the circuit, in this chapter, we are interested in approximating the dynamic maximum frequency distribution of a circuit when process variation is taken into account. Similar to SSTA, we assume all delay distributions to be normal Gaussian distributions. Given the dynamic maximum frequency distribution (or dynamic delay distribution) and the actual working frequency, we can derive the probability of the occurrence of errors.

3.3 Gate-Level Dynamic Timing Analysis

Consider the following problem: *Given a gate-level implementation of a processor and a piece of code, compute the dynamic delay distribution of the processor at a given clock cycle during the execution*. In this section, we describe a first attempt to solve this problem, in which we will aim for maximum accuracy, and relax the concerns of computation time. This will provide us with an analysis framework suitable for use as a baseline (ground truth) in accuracy evaluation and for model training.

The basic idea is to perform SSTA only on the set of paths sensitized during the desired clock cycle, which will give the dynamic delay distribution at that point in time. The framework, therefore, consists of (i) performing functional simulation and extracting sensitized paths, (ii) performing SSTA over the set of sensitized paths to find their delay distributions and their correlations, and (iii) applying a statistical *MAX* operation to achieve the processor delay distribution. A detailed description of the framework, as shown in Figure3.1, follows.

First, the design is synthesized using its RTL description and a gate-level netlist is obtained. The netlist is then used, along with the sequence of instructions given as input, to perform a functional simulation (a timing simulation is not necessary). Switching activity of all circuit nets (i.e. toggling times) obtained from the simulation and the gate-level connectivity information in the netlist are used to extract the activated paths during the clock cycle of interest (recall that a path is activated when all its nets have toggled). Finally, using variation-aware standard cell libraries, SSTA is performed on the set of activated paths and their delay distributions are calculated. For the paths originating from SRAM-based memory structures, the access time distribution of the memory (obtained from a variability-aware SRAM timing model) is added to the delay distribution of combinational paths. Using correlations of each activated path delay pair from the results of SSTA, a statistical *MAX* operation is applied to achieve the dynamic delay distribution of the processor.

Inside the experimental infrastructure (Figure 3.1), ASIC implementation is performed

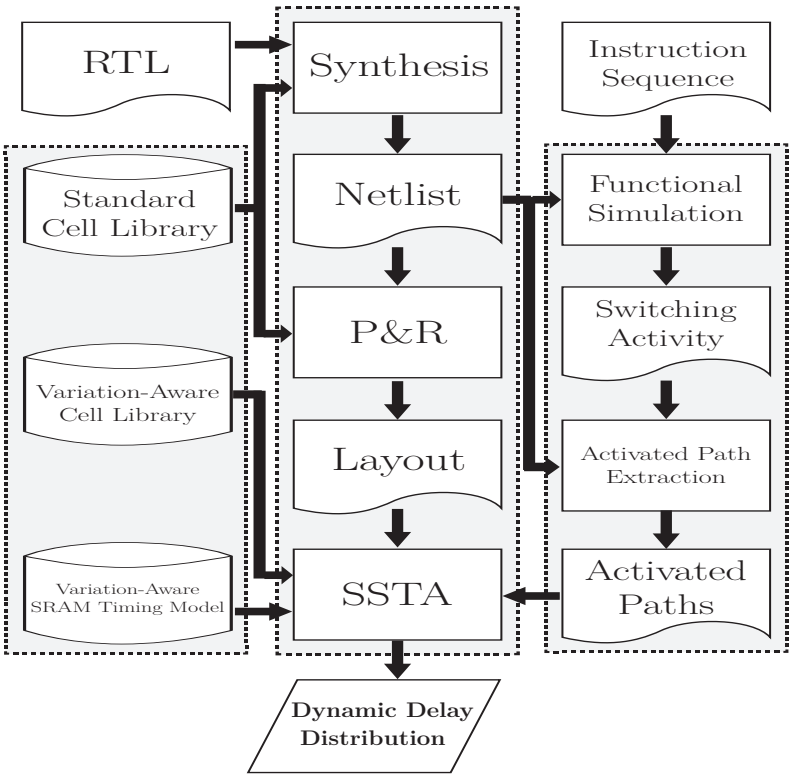


Figure 3.1. Variation-Aware Timing Analysis Framework

using *Synopsys Design Compiler* and *Synopsys IC Compiler* targeting a *TSMC 45nm* technology [61]. SSTA is performed with the variation-aware timing analysis engine of *Synopsys PrimeTime* using the variation-aware TSMC libraries. This flow is commonly used in the industry and is widely considered as a reliable methodology for chip implementation and timing analysis. Functional simulation is performed using the *Mentor Graphics Modelsim*. Variation-aware timing models of SRAM structures (i.e. register file and caches) are developed using VAR-TX [52], a hybrid analytical-empirical model that provides SRAM access times in presence of process variation. Finally, the manual statistical maximum operation inside the SSTA block is performed using the algorithm in [54]. This greedy algorithm combines the normal distributions in pairs in a sequence that would minimize the approximation error.

The framework described above introduces little additional inaccuracy to the conventional SSTA, including SRAM timing models and statistical maximum operation. Therefore, its results can be considered almost as accurate as the SSTA procedures used. Additionally, the analysis

can be done for different parts of the design separately, resulting in more detailed results. For example, we can find the delay distribution of each pipeline stage and determine the faulty stage(s). However, the long run time of this approach makes it unsuitable for analysis of actual applications which typically consist of millions of instructions. Relying on the high levels of accuracy and resolution of this framework, we will use it to train and evaluate the accuracy of a timing model presented next.

3.4 Clustered Timing Model

To enable timing analysis of actual applications, we need a method that is not only nearly as accurate and detailed as the framework described in Section 3.3, but also nearly as fast as a microarchitecture-level simulator. To achieve this, we employ a methodology consisting of (i) high-level modeling of path delays and (ii) utilizing runtime architectural information of the processor. In this section, we present a high-level timing model that simplifies the timing analysis while maintaining similar accuracy levels as the one used in the framework we developed earlier. The basic idea is to take advantage of the functional similarity of timing paths and cluster them into a few microarchitecture-level objects that determine the delay of the processor.

3.4.1 Preliminaries

The *state* of a sequential circuit consists of the contents of all its flip-flops and its primary input/outputs which together we call its *registers* (memory components are considered as delayed input/output ports). *Register clustering* defines an equivalence relation on the state of the circuit, partitioning its registers into a set of *Register Clusters* (RC) and the paths into *hyperpaths*. There is a hyperpath between two RCs when there is at least one timing path connecting a register output in the *origin* RC to a register input in the *destination* RC. Therefore, there can be zero, one, or two hyperpaths between two RCs. The resulting model is called a Clustered Timing Model (CTM). (Figure 3.4 shows a CTM of a typical in-order pipelined processor. More details in Section 3.4.4).

Now, consider an RC with N registers. At clock cycle i, the *value* of the RC is a vector of size N denoted by $V(i)$ containing the binary values of its registers and a *transition* on the RC is defined as a vector $T(i)$ such that $T(i) = V(i-1) \oplus V(i)$. The *delay* of a hyperpath is a random variable representing its propagation delay distribution in the presence of process variation and a hyperpath *delay function* is a function that maps a transition of its origin RC to this random variable.

When register clustering is performed according to functional similarities of registers, hyperpaths tend to be formed as collections of timing paths that jointly perform a specific function. For example, the timing paths forming the WB-RF hyperpath in Figure 3.4 work together to transfer the 32-bit result of an instruction to be written back to the register file. Depending on the hyperpath functionality, a transition resulting in an operation performed by a hyperpath can be constructed as the combination of a set of *primary transitions*. In the example of WB-RF hyperpath, this set can be the set of all single bit transitions. While a hyperpath is essentially a cluster of timing paths, we can also equivalently consider it as a collection of *functional paths* such that a functional path is activated as a result of a primary transition. Therefore, an operation performed by a hyperpath can be thought of as a combination of the activation of some of its functional paths, each activated by a primary transition. The correlation among timing paths is abstracted into correlations among functional paths and the delay of a hyperpath is calculated as the maximum of the delays of its activated functional paths rather than the activated timing paths.

3.4.2 Training and Application

With these set of abstractions in place, a Clustered Timing Model is trained and used in two steps:

Model Training

In this step, functional path delays and their correlations are characterized. This can be achieved by measuring the hyperpath delays corresponding to primary transitions and some

selectively chosen combinations of them. In order to estimate the delay correlation (ρ) of functional paths A and B, we measure the hyperpath delay when only A is activated, only B is activated, and both A and B are activated, and call them D_A, D_B, and D_{AB}, respectively, where each is a pair of the mean (μ) and standard deviation (σ) of the delay distribution. The important observation here is that $D_{AB} = MAX(D_A, D_B)$, where MAX is the statistical maximum operation and returns μ_{AB} and σ_{AB} [44]:

$$\mu_{AB} = \mu_A \Phi(\tfrac{\mu_A - \mu_B}{\theta}) + \mu_B \Phi(\tfrac{\mu_B - \mu_A}{\theta}) + \theta \phi(\tfrac{\mu_A - \mu_B}{\theta}) \tag{3.1}$$

$$\sigma_{AB} = [(\sigma_A^2 + \mu_A^2)\Phi(\tfrac{\mu_A - \mu_B}{\theta}) + (\sigma_B^2 + \mu_B^2)\Phi(\tfrac{\mu_B - \mu_A}{\theta}) - (\mu_A + \mu_B)\theta\phi(\tfrac{\mu_A - \mu_B}{\theta})]^{\frac{1}{2}} \tag{3.2}$$

where $\phi(.)$ and $\Phi(.)$ are probability density function (pdf) and cumulative distribution function (cdf) of the standard normal distribution, and $\theta = \sqrt{\sigma_A^2 + \sigma_B^2 - 2\rho\sigma_A\sigma_B}$. The correlation coefficient ρ can be derived from either Equations 3.1 and 3.2, or as their average to reduce estimation error.

Model Usage

Given an arbitrary transition, we split it into a set of primary transitions. The hyperpath delay corresponding to this transition is then computed as the maximum of the delays of the functional paths activated by each constructing primary transition two at a time, according to Equations 3.1 and 3.2 which only require delay distributions and correlations of the activated functional paths obtained in the training step.

Finally, a *Probability of Error* (PoE) is assigned to each hyperpath by replacing the circuits maximum delay (i.e. $\frac{1}{F}$ where F is the working frequency) in the cdf of the hyperpath delay:

$$PoE_H = \frac{1}{2}\left[1 - \text{erf}\left(\frac{\frac{1}{F} - \mu_H}{\sqrt{2\sigma_H^2}}\right)\right] \tag{3.3}$$

where F is the working frequency and μ_H and σ_H are the hyperpath delay mean and standard

deviation, respectively. Since hyperpath delays are considered uncorrelated, PoE of a processor with N *activated* hyperpaths H_1 through H_N, can be obtained using Equation 3.4.

$$PoE = 1 - \prod_{i=1}^{N} (1 - PoE_{H_i}) \tag{3.4}$$

As will be explained in Section 3.5, activated hyperpaths can be identified using microarchitecture-level information available during simulation. In LEON3 CTM (Figure 3.4), for instance, EXE-D$ hyperpath is only activated by load and store instruction while EXE-EXE, MEM-EXE, WB-EXE hyperpaths are activated when back-to-back dependencies are present.

3.4.3 Modularity and Hierarchy

The timing abstraction described above fits well into a modular and hierarchical timing model providing easy integration of previously developed models and configurable levels of accuracy-speed trade-off for different components of the design.

Hierarchy

A higher-level CTM can be constructed in a bottom-up fashion from an existing CTM by clustering its RCs. The clustering may be done step by step merging two RCs in each step. When two RCs are *merged*, two hyperpaths that connect both the RCs to a single other one are replaced with a new hyperpath. The delay function of the new hyperpath would then be the *maximum* of the delay functions of the 'merged' hyperpaths. Since this clustering often involves approximations in the merging and maximum operations, it can be used to construct simpler higher-level models from existing CTMs to obtain faster analysis speeds. This is shown in Figure 3.2 where the red RCs and hyperpaths in the left CTM are merged and replaced by the blue ones in the left CTM.

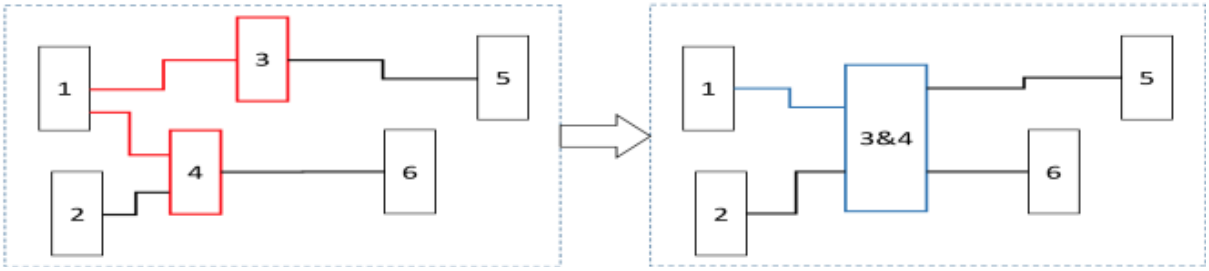


Figure 3.2. CTM Hierarchy: Merging RCs to Generate Higher-Level CTM

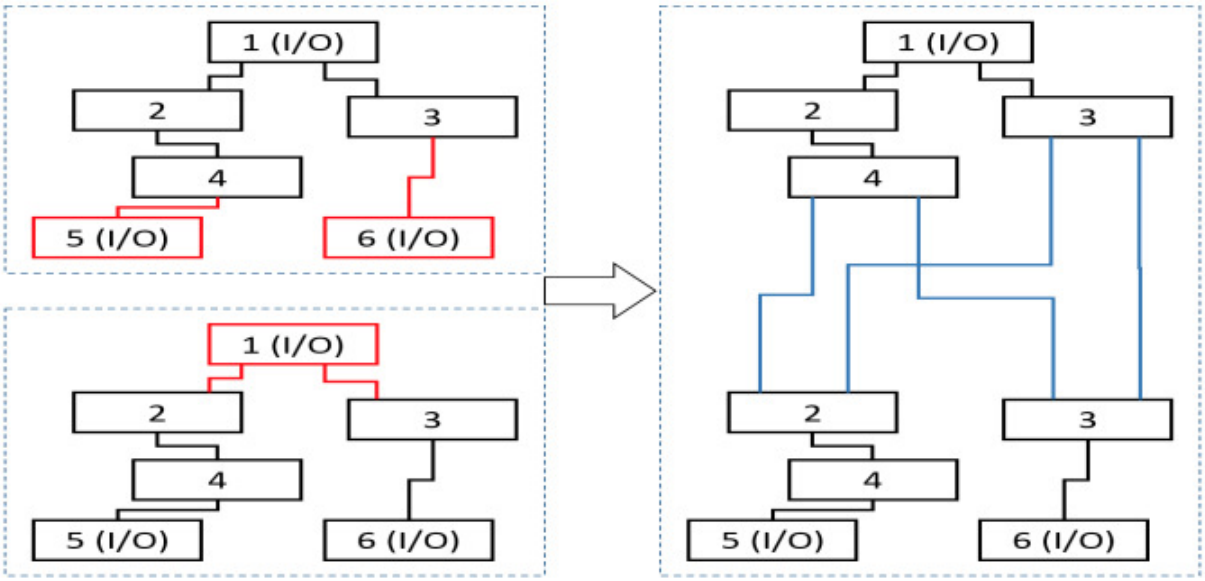


Figure 3.3. CTM Modularity: Connecting Two CTMs

Modularity

Multiple existing CTMs can be *connected* to produce a new unified CTM in two steps: first, all connecting input/output RCs are removed from the CTMs. Next, each hyperpath pair in the two CTMs that was previously connected to the removed RCs is replaced with a single hyperpath connecting the two internal RCs of the two CTMs. The delay function of each new hyperpath is obtained by *summing* the delay functions of the two hyperpaths it has replaced. This is shown in Figure 3.3 where the two identical CTMs in the left are serially connected to obtain the right CTM. For example, previously developed CTMs for Integer Unit, Floating-Point Unit, and Co-Processors of a processor could be combined to obtain a new unified CTM.

3.4.4 A CTM for In-Order RISC Processors

We now describe a method for developing a CTM for in-order RISC processors. We will focus on LEON3 processor as a publicly available representative of such processors, but our methodology is extendable to other similar RISC cores.

LEON3 Processor Core Overview

LEON3 [27] is a 32-bit processor core conforming to the IEEE-1754 (SPARC V8) architecture [36], and can easily be used in a multiprocessor system. It has a 7-stage pipeline with separate instruction and data caches and support for static branch prediction. Instructions are fetched from the Instruction cache (I$) in Fetch (F) stage and decoded in the Decode (D) stage. Call and branch target addresses are also generated in the Decode stage. In Register-Access (RA) stage, operands are read from the register file or from internal data bypasses. ALU, logical, and shift operations are performed in the Execution (EXE) stage. The address for memory operations as well as `jmpl` and `rett` instructions are also generated in this stage. Data cache (D$) is accessed in the Memory (M) stage, exceptions are handled in the Exception (X) stage, and the result of any ALU, logical, shift, or cache operations are written back to the register file in the Write-Back (WB) stage.

Register Clustering and Model Training

The register clustering (defined in Section 3.4.4) is the most important step in the development of a CTM. The clustering should be chosen such that simple and accurate delay functions can be associated with the resulting hyperpaths. We propose a functionality-based clustering for in-order RISC processors such as LEON3 incorporating a functional equivalence relation for clustering the registers. Our clustering scheme assigns an RC to all registers of a pipeline stage, an RC to the PC, an RC to the register file ports, and two RCs to the instruction and data cache ports, resulting in an abstraction depicted in Figure 3.4, where boxes are RCs and lines are hyperpaths (shaded boxes represent I/O RCs). To simplify the illustration, we have

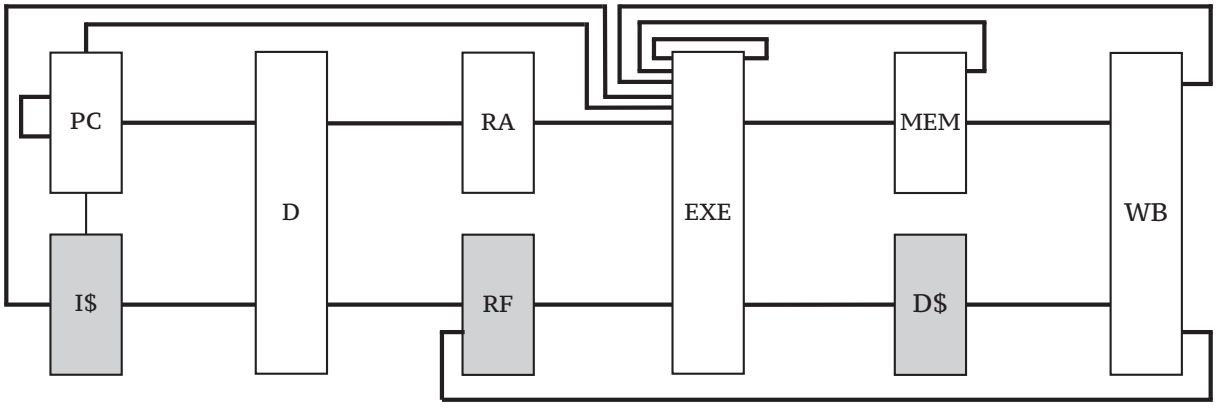


Figure 3.4. Clustered Timing Model for LEON3 Pipeline

not included X stage. That does not affect our analysis because the paths in X stage are not timing-critical. With this register clustering applied, we are able to divide the hyperpaths of an in-order RISC processor such as LEON3 into four types, which will be explained later. To measure the hyperpath delays, we identify the primary transitions and provide *training code* examples for each type. Training codes are special pieces of code aimed at a specific hyperpath and enable controlled activation of its different functional paths.

An example pseudocode template is given in Figure 3.5. Note that this code template is only suitable for some of the hyperpaths and serves as an example of how training codes should be designed. Training codes for other hyperpaths can be designed in a similar manner with some changes. Lines 1-3 initialize three registers and lines 4-11 repeatedly execute the training sequence. Cache misses are avoided by placing the training instructions in a loop. The training framework (Section 3.3) is then configured to record the timing distribution of the desired hyperpath at some intermediate iteration of the loop. Note that the model would still be able to handle exceptions such as cache misses and avoiding them in the training process is done merely for accurate modeling of hyperpaths without external influences. Inside the loop, two sets of `nop` instructions are executed before and after the training instructions to initialize the pipeline to a clear state and prevent any back-to-back dependencies. In lines 7-9, three instances of a suitable instruction are executed where the first and third ones set the control network of

```
clr %l0
mov data1, %l1
mov data2, %l2
while (1){
    nop     /* repeated 7 times */
    inst    %l0, %l0, %l3
    inst    %l1, %l2, %l3
    inst    %l0, %l0, %l3
    nop     /* repeated 7 times */
}
```

Figure 3.5. Example Pseudocode Template for Training the Model

the pipeline to the same state the target instruction (line 8) would induce. This limits back-end pipeline activity to the data flow activities of training instructions. By setting `inst`, `data1`, and `data2` variables to suitable values, we can configure the code to induce only the primary transitions needed for each hyperpath type.

Next, we introduce the four types of hyperpaths and give examples of how setting these variables for each type provides controlled activation of its primary transitions.

I. Data transfer hyperpaths: These hyperpaths perform little computation on the contents of their origin RC. Their function can mainly be described as transferring the contents of one RC to another. At times, the transfer also involves bit masking, sign extension, and shift operation. In LEON3 pipeline, hyperpaths PC-I$, PC-D, I$-D, EXE-EXE, MEM-EXE, WB-EXE, MEM-WB, RF-EXE, D$-WB, WB-RF, and EXE-MEM (when the active instruction is shift) are data transfer hyperpaths. The set of primary transitions consists of all single-bit transitions of the origin RC. Therefore, the training step involves measuring the delay of these hyperpaths when single and two bit transitions occur on the origin RCs. For the WB-RF hyperpath as an example, this can be achieved by setting the `inst` variable in Figure 3.5 to `and`, and simultaneously setting `data1` and `data2` to values in which only one and two bits are set.

II. Addition hyperpaths: These hyperpaths perform an addition operation on the contents of their origin RC. In LEON3 pipeline, hyperpaths EXE-PC, EXE-I$, EXE-D$, and EXE-MEM (when the active instruction in the Execution stage is `add` or `sub`) are addition

hyperpaths. In a multi-bit addition, a bit in the result may be changed due to either a local path activation (i.e. a 1-0 or 0-1 situation at that bit position in addition inputs) or a carry-chain activation from any lower order bit. These paths, in fact, constitute the set of hyperpath functional paths, and the set of primary transitions consists of one transition for each local path activation and one for each carry-chain activation path. For the EXE-MEM hyperpath as an example, this can be achieved by setting the `inst` variable in Figure 3.5 to `add`, and setting `data1` and `data2` to values that would induce local and carry-chain path activations. For example, a carry-chain path from bit position 3 to bit position 7 can be induced by setting `data1` and `data2` to 0x00000008 and 0x00000078, respectively.

II. Increment hyperpaths*: A simpler form of addition hyperpaths that only adds one unit to the contents of the origin RC. In LEON3 pipeline, hyperpath PC-PC is of this type. The functional paths set for this type includes only carry-chain paths from the LSB.

III. Logic hyperpaths: These hyperpaths perform a logical operation on the contents of their origin RC. In LEON3 pipeline, hyperpath EXE-MEM (when the active instruction in the Execution stage is a logic instruction) is a logic hyperpath. In a multi-bit logic operation, a bit in the result is changed depending on the specific operation (i.e. AND, OR, etc.), and the value of the operands at the same bit position. Therefore, the set of primary transitions for the AND operation as an example include single bit zero to one transitions of both operands. For the EXE-MEM hyperpath executing AND operation as an example, this can be achieved by setting the `inst` variable in Figure 3.5 to `and`, and simultaneously setting `data1` and `data2` to values in which only one and two bits are set.

IV. Hybrid hyperpaths: These hyperpaths are combinations of the previous types and can be characterized by splitting them into their constructing parts. In LEON3, hyperpaths D-RA, D-RF, and RA-EXE are hybrid hyperpaths.

3.4.5 Accuracy Evaluation

In this section, we present an evaluation of the accuracy of the timing model developed for LEON3 processor. In order to achieve a reliable evaluation, we perform the analysis on each hyperpath separately, comparing the model with detailed gate-level experiments using an industry-standard timing analysis flow and a 45*nm* TSMC library with a nominal voltage of $0.9V$. For each hyperpath in the LEON3 pipeline, we perform 100 experiments using random input data and measure the difference in the PoE calculated by the analysis framework described in Section 3.3 assuming a working frequency of 1.15 times the nominal frequency. These PoE values are then compared against the values estimated by our model and the level of inaccuracy is determined for each hyperpath. To demonstrate the robustness of our timing model to changes in voltage and temperature, we constructed separate model versions and performed the evaluation at all TSMC-characterized corners. Table3.1 tab shows the relative errors in the calculated PoE for each hyperpath at each voltage-temperature corner. The model is highly accurate both across different hyperpaths and across corners with an overall average error of 3.9% and maximum error of 6.7%.

3.5 Fast Timing Analysis with CTM

Having verified the accuracy of our timing model, we now proceed to utilize it for enabling fast timing simulation of large programs. In order to use the CTM we developed for LEON3 in Section 3.4, we used TSIM [28], an instruction-level simulator that enables accurate and cycle-true emulation of LEON3. A pipeline analysis module was added on top of the simulator to provide stage-level transition information of the running code. The timing model was implemented in a separate module which takes an instruction trace and stage-level transition information from the simulator and produces PoEs for each hyperpath every clock cycle. These hyperpath PoEs are then combined with microarchitecture-level information that determines the activated hyperpaths of instructions to derive instruction error rates at different pipeline

Table 3.1. CTM Estimation Relative Error Across LEON3 Hyperpaths and Voltage-Temperature Corners

		Hyperpaths																		
	Origin	**PC**	**PC**	**PC**	**I$**	**D**	**D**	**RA**	**RF**	**EXE**	**EXE**	**EXE**	**EXE**	**EXE**	**MEM**	**MEM**	**D$**	**WB**	**WB**	**Avg**
	Destination	***PC***	**I$**	**D**	**D**	**RA**	**RF**	**EXE**	**EXE**	**I$**	**PC**	**EXE**	**MEM**	**D$**	**EXE**	**WB**	**WB**	**EXE**	**RF**	
(V,T)	***(0.99V,-40°C)***	4.3	2.2	2.3	1.7	5.2	5.8	3.9	2.6	4.1	4.3	4.6	4.7	4.5	4.8	3.1	2.9	4.9	2.1	**3.8**
	(0.81V,125°C)	5.2	2.0	2.7	1.9	5.5	5.0	3.8	2.5	4.9	4.3	4.8	6.4	4.4	4.2	3.5	2.6	6.1	2.1	**4.0**
	(0.99V,0°C)	4.5	2.1	2.5	1.8	5.8	5.2	3.5	2.8	4.7	3.9	4.5	5.6	4.0	4.5	3.4	2.8	6.7	2.0	**3.9**
	(0.81V,0°C)	4.3	2.1	2.2	1.7	5.5	5.9	3.8	2.4	4.2	4.9	4.3	4.5	4.3	4.5	3.1	3.1	4.5	2.3	**3.8**
	(0.81V,-40°C)	4.3	2.3	2.4	1.9	6.0	5.7	3.4	2.4	4.8	4.2	5.5	4.3	4.9	4.9	3.4	3.2	6.4	2.2	**4.0**
	Avg	**4.5**	**2.1**	**2.4**	**1.8**	**5.6**	**5.5**	**3.7**	**2.5**	**4.5**	**4.3**	**4.7**	**5.1**	**4.4**	**4.6**	**3.3**	**2.9**	**5.7**	**2.1**	**3.9**

Table 3.2. Inter-Program Variation

Benchmarks	Program Error Rate (%)
`basicmath`	1.728
`bitcount`	0.578
`qsort`	4.704
`dijkstra`	2.076
`stringsearch`	0.593

stages. The resulting variation-aware simulation platform is much faster than the timing analysis framework we developed in Section 3.3 and enables the simulation of actual programs. We selected five benchmarks (shown in Table 3.2) from MiBench [32] to study the error behavior of representative programs. In our experiments, we assumed a fixed frequency of 1.15 times the nominal frequency. Errors are assumed to occur when a hyperpath is activated and its PoE exceeds 0.9.

3.5.1 Inter-Program Variation

Inter-Program Variation denotes the variability in the error rates of different programs when run on a fixed processor. In order to evaluate the variation in the error rates of different programs, we ran the analysis on the five benchmarks with their default input data. The *Program Error Rate* (PER) is defined as the number of faulty instruction executions divided by the total number of executed instructions. Note that an instruction may be executed multiple times and cause errors in some of them. This is accounted for by considering each dynamic instance of the instruction rather than its static representation as an executed instruction. Table 3.2 summarizes the results. Across our small experiment set, the error rate covers a range from around 0.6% to around 4.7%, illustrating the large potential variation due to the execution of different pieces of code.

Table 3.3. Variation in PER Due to Input Data Variability

Benchmarks	Program Error Rate		
	Mean(%)	Standard Deviation (%)	$\frac{\mu}{\sigma}$
`basicmath`	2.214	0.512	0.23
`bitcount`	0.754	0.254	0.34
`qsort`	3.124	0.784	0.25
`dijkstra`	2.854	0.412	0.14
`stringsearch`	0.943	0.267	0.28

3.5.2 Input Data Variability

What happens if we run a single program with different sets of input data? In order to evaluate the amount of variation caused by the input data, we performed the analysis on each benchmark with 100 randomly generated input data sets. Table 3.3 shows the mean and standard deviation of the 100 runs for each program, and the $\frac{\mu}{\sigma}$ ratio as a measure of the variation caused by the input data. To put the results into perspective, we note that other sources of variability cause tens of percent variations in chip performance [31], which is similar to the values obtained for input data variability. An interesting observation is that different programs demonstrate different sensitivities to changes in their input data (ranging from $\frac{\mu}{\sigma}$ values of 0.14 to 0.34), pointing to potential opportunities for more in-depth analysis of this effect.

3.5.3 Intra-Program Variation

Intra-Program Variation denotes the variability in the error rates of the instructions of a program. The *Instruction Error Rate* (IER) of an instruction is defined as the number of its faulty executions divided by the total number of its executed dynamic instances. For example, IER of an instruction which is executed 1000 times and fails 300 of them is 0.3. Less faulty instructions have IERs closer to zero, while more vulnerable ones have IERs closer to one. The IER distribution of a program determines the extent to which some of its instructions are

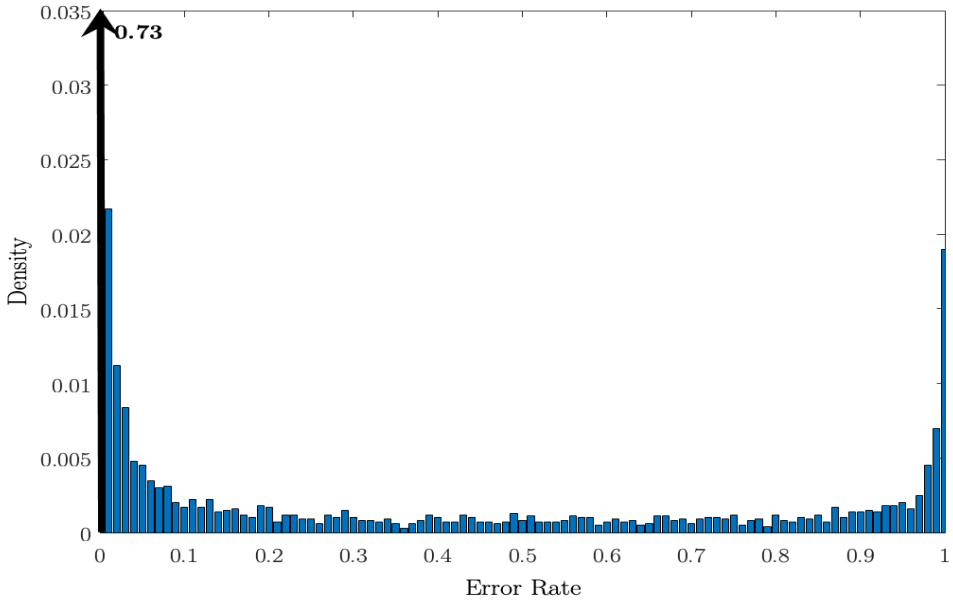


Figure 3.6. Instruction Error Rate Distribution for `basicmath`

more or less faulty than others. A sufficiently positively skewed distribution indicates potential opportunities for reducing the PER by focusing on the more faulty instructions.

Figures 3.6-3.10 demonstrates the IER distributions of the five benchmarks normalized by the total number of instructions. Since most of the instructions never cause errors (i.e. $IER = 0$), for better visuality, we have zoomed into error rate values larger than zero and shown the densities at $IER = 0$ next to an arrow representing the truncated Y axis. All distributions start with a higher density at IERs closer to zero, then significantly drop and remain rather uniform until they rise again at IERs closer to one. The higher density of the distributions at IERs close to one has been previously observed and is referred to as *instruction error locality* [64]. The sharp spike at $IER = 1$ is caused by instructions that fail their only execution. Excluding very small and very large IERs, the distributions are neither negatively nor positively skewed and are more or less uniform.

The shape of IER distributions provides insight into the efficacy of a large class of error-management techniques. These techniques that are based on providing more vulnerable

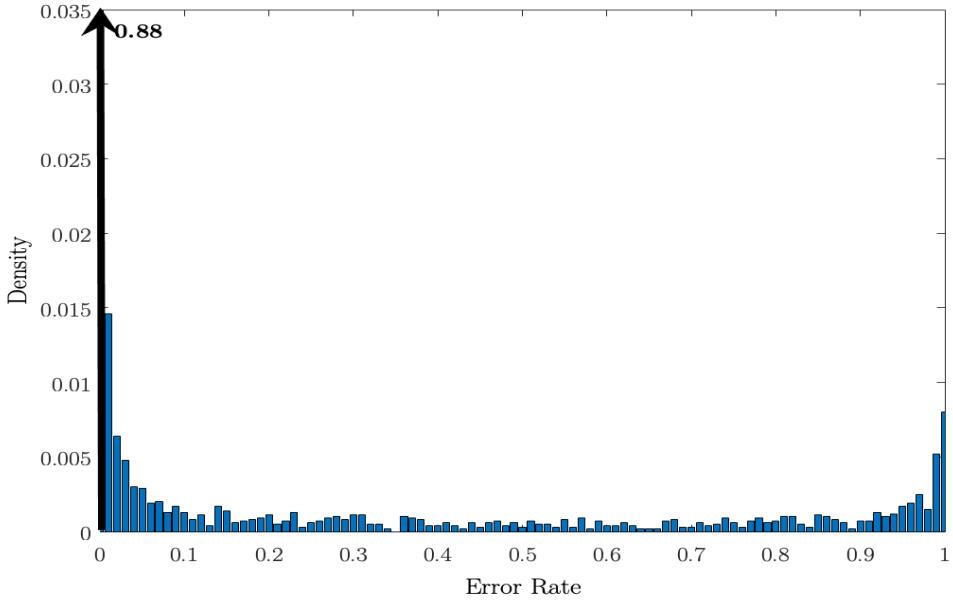


Figure 3.7. Instruction Error Rate Distribution for `bitcount`

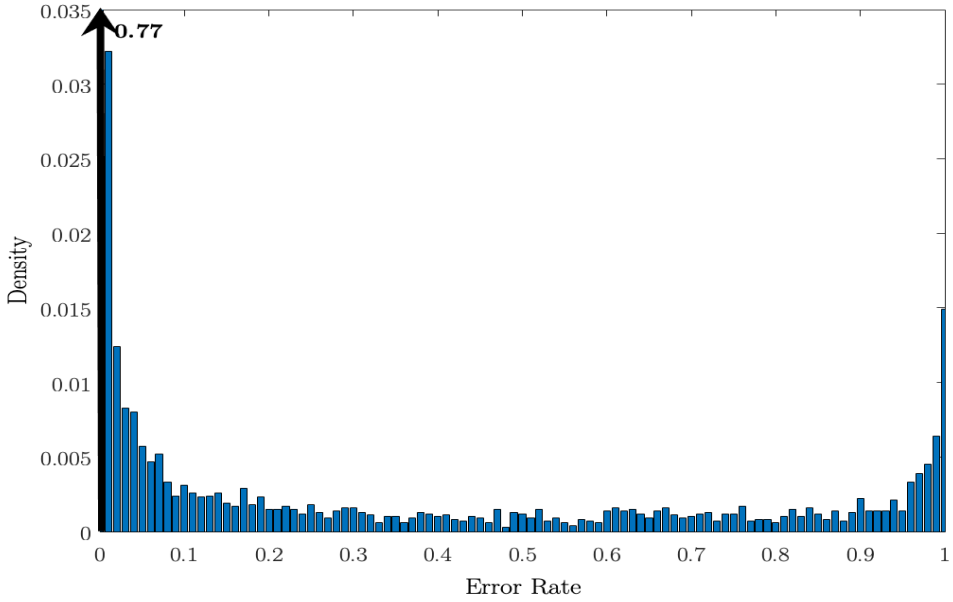


Figure 3.8. Instruction Error Rate Distribution for `qsort`

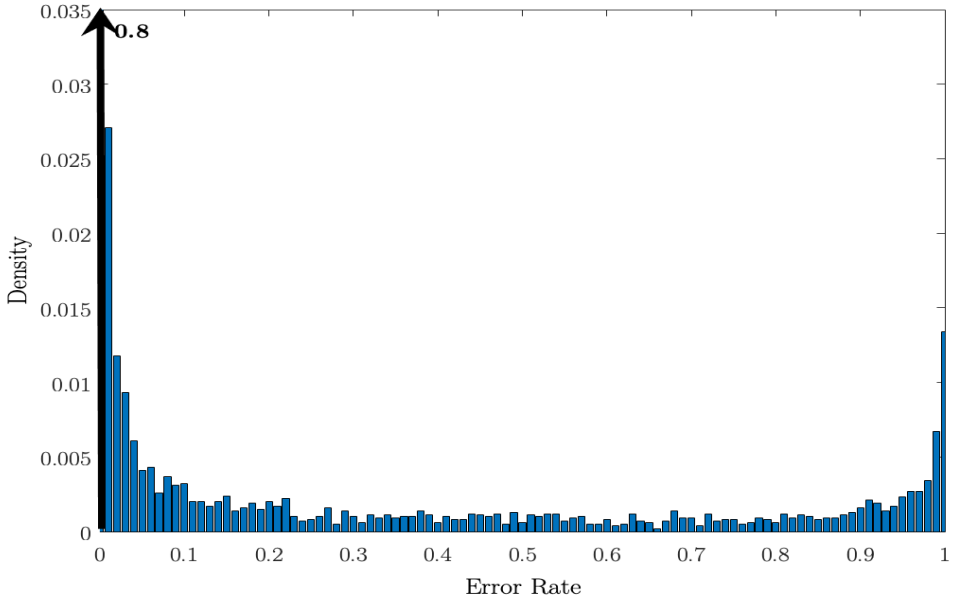


Figure 3.9. Instruction Error Rate Distribution for `dijkstra`

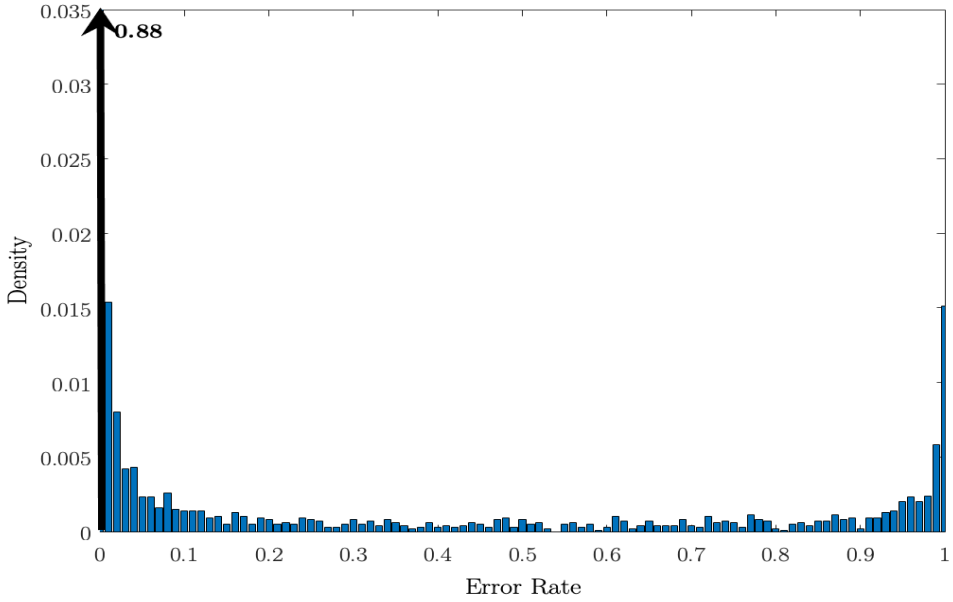


Figure 3.10. Instruction Error Rate Distribution for `stringsearch`

instructions with more relaxed timing requirements, use circuit- and microarchitecture-level techniques such as time borrowing or increase timing guardbands by decreasing frequency or increasing voltage [51][64][12]. The main idea behind such methods is to reduce the PER by paying a small penalty for the most faulty instructions with the assumption that there are not too many such instructions and that they are not executed too many times that the error recovery penalty would nullify faster execution of less faulty instructions. The IER threshold chosen to identify the target instructions determines the efficiency of such strategies and IER distributions, along with instruction execution counts can be used in finding the sweet spot in selecting the IER threshold. A more positively skewed IER distribution means that such techniques could be more beneficial to that program.

3.5.4 Physical Location of Errors

Another important aspect of the error behavior of a program is the physical location of errors in hardware. How reasonable is it to assume that errors occur in more critical-path-populated regions of the processor? Furthermore, do different programs cause similar error distributions among these regions or do some programs cause significantly more errors in specific regions? CTMs are immensely useful in this kind of analysis as they provide a high-level view into the error behavior of various networks in hardware. Such analysis is also important due to different sensitivities of the processors to errors in their different modules. For instance, an error in the fetch or decode stage will most probably crash the system, while an error in the execution stage might vanish completely or merely present some inaccuracy in the result.

Table 3.4 presents the error rate in hyperpaths and functional networks of LEON3. Error rate is the fraction of instructions using the hyperpath—for example, only load and store instructions use the hyperpaths to and from data cache—that induce at least one error in its timing paths. The results show that instruction decode, address generation, ALU, and instruction result (also containing the exception handling) networks produce the majority of errors inside the processor, while instruction fetch, operand read, and bypass networks account for a smaller

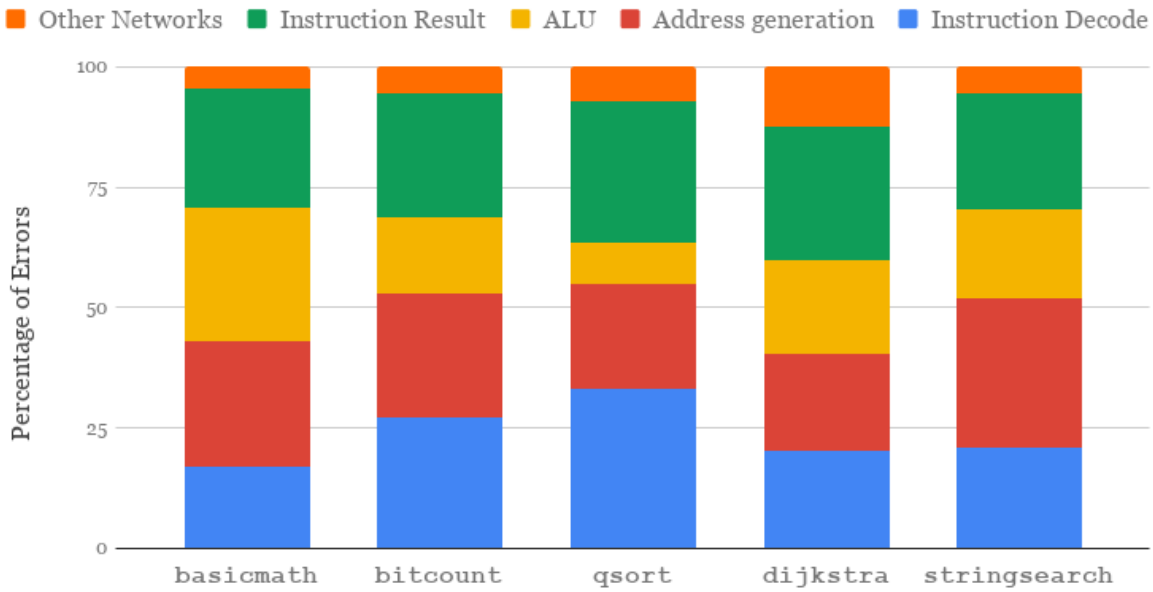


Figure 3.11. Percentage of Errors in LEON3 Networks

portion.

For better visuality, Figure 3.11 shows the percentage of errors in each network. It is interesting to note that while most of the processor critical paths lie in the ALU causing most of the timing failures, errors may occur in ALU less often than in the less critical-path-populated networks of instruction decode and address generation. We speculate that this is due to the fact that many instructions do not activate the highly critical paths of the ALU resulting in many healthy ALU operations, while instruction decode and address generation networks perform more similar operations which activate the same critical paths most of the time. This observation stresses the importance of a comprehensive analysis in the evaluation of software error management techniques. An imperfect assumption that the arithmetic execution network produces the majority of errors may lead to techniques that reduce ALU errors by, for example, spacing out instructions that heavily activate its critical paths, while neglecting or even giving rise to errors occurring in other vulnerable networks.

Along the axis of programs, we observe a significant variation in the percentage of

Table 3.4. Error Rate (%) in Hyperpaths and Functional Networks of LEON3

		Hyperpaths																	
	Org	**PC**	**PC**	**PC**	**I$**	**D**	**D**	**RA**	**RF**	**EXE**	**EXE**	**EXE**	**EXE**	**EXE**	**MEM**	**WB**	**MEM**	**D$**	**WB**
	Dest	**PC**	**I$**	**D**	**D**	**RA**	**RF**	**EXE**	**EXE**	**PC**	**I$**	**D$**	**MEM**	**EXE**	**EXE**	**EXE**	**WB**	**WB**	**RF**
	Networks	**Inst. Fetch**				**Decode**		**Op. Read**		**Address Gen.**			**ALU**	**Bypass**			**Inst. Result**		
Benchmarks	`basicmath`	2.4	0.6	0.9	0.3	26.4	8.4	0.6	1.2	16.2	12.6	24.9	56.7	2.1	0.3	0.9	9.3	15.3	26.1
	`bitcount`	0.9	0.6	0.5	0.2	12.1	4.9	0.6	0.1	3.8	3.1	9.2	10.1	0.3	0.2	0.1	1.1	5.7	9.2
	`qsort`	11.4	4.2	0.6	1.2	79.2	71.4	6.0	3.0	38.4	30.6	29.4	39.4	1.8	0.6	3.6	58.8	37.2	36.6
	`dijkstra`	13.6	4.8	0.4	1.2	21.6	24.8	0.8	2.4	16.8	15.2	13.6	44.7	1.6	2.0	1.6	12.4	24.4	26.8
	`stringsearch`	1.2	0.3	0.1	0.1	6.8	7.4	0.8	0.2	9.1	6.4	5.2	12.6	0.2	0.7	0.1	6.4	1.9	7.9
	Avg	**5.9**	**2.1**	**0.5**	**0.6**	**29.2**	**23.4**	**1.8**	**1.4**	**16.9**	**13.6**	**16.5**	**32.7**	**1.2**	**0.8**	**1.3**	**17.6**	**16.9**	**21.3**

errors occurring in different networks. For instance, `basicmath` which contains more complex mathematical operations induces almost three times more errors in the ALU than the control-intensive `qsort` that has most of its errors in the decoding network, while the memory intensive `stringsearch` produces more errors in the address generation and instruction result networks which also handle cache misses. This observation stresses the potential efficacy of workload-aware methods that consider both error rates and error locations based on the running code.

Chapter 3 is based upon the work supported by the National Science Foundations Variability Expedition in Computing under Award No. 1029783. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Chapter 3, in full, is a reprint of Omid Assare and Rajesh Gupta, "Timing Analysis of Erroneous Systems," *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2014. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Performance Analysis at the Architecture Level

In Chapter 3, we analyzed TS processor performance using a microarchitecture-level timing model. The performance analysis framework we introduce in this chapter achieves more accurate results in less time.

4.1 Introduction

This chapter is organized in two parts. First, we propose techniques to predict timing errors more efficiently by breaking up the analysis of both hardware and software and taking advantage of our knowledge about their structure. For hardware, we analyze the control network and datapath of the processor separately and at different levels. For software, we break up the program into basic blocks and analyze them separately. Carefully matching the components of hardware and software allows us to analyze the control network only once, which, in turn, provides the opportunity to perform that analysis at the gate level, thereby improving the accuracy. For the datapath, we develop high-level models that identify the activated paths using only the architecturally-visible state of the processor. Finally, because the analysis can be performed at the architecture level, we use execution-driven simulations to further improve simulation times by developing a source code instrumentation technique.

Second, we develop a statistical instruction error model that estimates the likelihood that

each instruction in the program experiences a timing error, capturing the uncertainty caused by both input data dependence and process variation. We then propose an approximation technique to estimate program error rate using statistical limit theorems and utilize two well-known tools of applied statistics to establish bounds on the approximation error.

4.1.1 Dynamic Timing Analysis

Since each timing error incurs a penalty for error recovery, performance of a TS processor is a function of the number of timing errors it experiences. But conventional timing analysis techniques are not designed to predict timing errors, which requires one to determine *dynamic timing slack* (DTS) on a cycle-by-cycle basis. DTS is the unused portion of the clock cycle where all signals have already propagated through logic paths and are waiting to be captured by flip-flops at the end of the clock cycle [24]. A positive DTS indicates that the critical paths of the processor have not been exercised and there will be no timing errors. A negative DTS means that at least one flip-flop will capture the wrong value and at least one timing error will occur. Once a timing error has been predicted, its dynamic effect needs to be considered as well, because the occurrence of a timing error can itself affect DTS by triggering the error recovery mechanism. The analysis gets more complicated when process variation is taken into account. The variation in propagation delay of gates transforms DTS from a fixed number to a random variable. So before chips are manufactured, we might not be able to decide whether DTS is negative or positive—and if there will be a timing error—particularly when DTS is close to zero.

Previous work has shown that DTS is a function of the sequence of instructions being executed. As a result, applications experience different DTS and, consequently, different number of timing errors. Even a single application could see different error counts when it is run with different input vectors because DTS is a function of instruction operands as well. To quantify the vulnerability of an application to timing errors and its sensitivity to data and process variations, we define *error rate* as the fraction of executed instructions that experience timing errors. In this chapter, we introduce a framework to estimate error rate of programs running on in-order TS

processors.

4.1.2 Contributions

We make the following contributions:

1. We develop a dynamic timing analysis (DTA) tool that accurately calculates DTS in Section 4.3, and an instruction error model that predicts the likelihood that the instruction will experience a timing error in Section 4.6. To the best of our knowledge, we are the first to simultaneously take into account the effects of process variation, instruction sequence and operands, datapath configuration, and error recovery scheme. By taking advantage of the fact that sequence of executed instructions is mostly fixed in in-order processors we are able to improve the efficiency of our framework to allow comprehensive analysis of the effect of application input data by using large datasets.

2. We propose a statistical approach for estimating error rate of programs based on two well-known laws of applied statistics, central limit theorem and Poisson limit theorem, to produce distributions that capture the effects of data and process variations in Section 4.7. We then use Stein's method [57] to establish bounds on the approximation error, including the inaccuracy caused by inter-instruction correlations.

4.2 Experimental Setup

We adopt an Intel research TS processor [7] based on LEON3 [27], an open-source 32-bit, RISC, in-order pipeline with seven stages that implements the SPARC V8 architecture [36]. The processor does not include a floating-point unit or hardware multiplier and only supports integer operations.

Error Detection. Error detection circuits protect the first five stages of the pipeline, instruction fetch (IF), decode (DE), register access (RA), execute (EX), and memory (MEM). Because timing errors are detected with a one-cycle delay, timing errors in the last two stages,

exception (X) and write-back (WB), cannot be detected before the errant instruction has already started writing to the register file. To prevent this, X and WB stages are implemented with extra timing guardbands so that no timing errors can occur even in worst-case conditions. Extra guardbands are also used to prevent timing errors in the error recovery logic.

Error Recovery. A one-bit register is added to each stage in the pipeline indicating if the instruction in that stage has experienced a timing error. This error register is used in WB stage to prevent errant instructions and subsequent instructions in the pipeline from writing to the register file and initiate the recovery process. Once an errant instruction reaches the WB stage, frequency is reduced by half, the pipeline is flushed and the errant instruction is replayed. This roll-back mechanism already exists in most processors for branch mispredictions. For the seven-stage pipeline, this process (pipeline flush and instruction replay) takes 14 clock cycles, but because frequency is halved, the effective recovery penalty is 28 clock cycles.

Power and Area Overheads. Silicon measurements have shown that the error detection and recovery logic incur a power and area overhead of less than 0.9% and 3.8%, respectively [7].

Synthesis and Static Timing Analysis. We perform our simulations for DTA using an unmodified LEON3 core. The design was synthesized, placed, and routed on the 45nm TSMC technology targeting the typical-case corner (TT,0.9V,25C) using Synopsys Design Compiler and Synopsys IC Compiler. Synopsys PrimeTime calculated maximum (non-speculative) frequency at 718MHz using SSTA performed at (0.81V,25C), guardbanding for a 10% voltage droop. The point of first failure was measured at 810MHz (1.13x baseline) and we assumed a working frequency of 825MHz (1.15x baseline).

4.3 Gate-Level Dynamic Timing Analysis

Suppose that N is a graph of the processor netlist where vertices are gates and edges are nets of the netlist. We include flip-flops and I/O ports in the set of gates and call them *endpoints*.

Definition 4.3.1. *An ordered set of gates in N is a path if (i) the first gate is the only endpoint*

Table 4.1. Symbols and Definitions

Symbols	Definitions
N	Netlist of the processor pipeline
$S(N)$	Number of pipeline stages in N
s	A stage in the pipeline, $s = 0, 1, ..., S(N) - 1$
$E(N,s)$	Set of all endpoints in pipeline stage s of N
E_i	A particular set of endpoints
e_i	A particular endpoint
$P(e_i)$	Set of all paths ending in endpoint e_i
P_i	A particular set of paths
p_i	A particular path
$G(p_i)$	Set of all gates in path p_i
G_i	A particular set of gates
g_i	A particular gate
$SL(p_i)$	Timing slack of path p_i
$CP(P_i)$	Most critical (minimum slack) path in P_i
$VCD(t)$	Set of all activated gates in cycle t
$AP(N,s,t)$	Set of the most critical activated paths in stage s of N at clock cycle t

in the set, (ii) each gate, except the first one, is connected to the previous gate in the set, and (iii) the last gate is connected to an endpoint.

Definition 4.3.2. *We say a net is activated in a particular clock cycle if, were the clock period sufficiently long, it would eventually toggle, i.e., change its value.*

Definition 4.3.3. *We say a gate is activated in a particular clock cycle if the net connected to its output is activated.*

Definition 4.3.4. *We say a path is activated in a particular clock cycle if all of its gates are activated in that clock cycle.*

Algorithm 1 takes the processor netlist and signal activity information (VCD), a list of activated nets in each clock cycle, and computes DTS of a specified pipeline stage at a given

Algorithm 1: Gate-Level Dynamic Timing Analysis

```
1  Function DTS(N, s, t, VCD):
2      AP(N, s, t) ← ∅ foreach e_i ∈ E(N, s) do
3          P_i ← P(e_i)
4          while P_i ≠ ∅ do
5              p_i ← CP(P_i)
6              G_i ← G(p_i)
7              Activated ← true
8              foreach g_i ∈ G_i do
9                  if g_i ∉ VCD(t) then
10                     Activated ← false
11                     break
12             P_i ← P_i − p_i
13             if Activated = true then
14                 AP(N, s, t) ← AP(N, s, t) ∪ p_i
15                 break
16     return SL(CP(AP(N, s, t)))
```

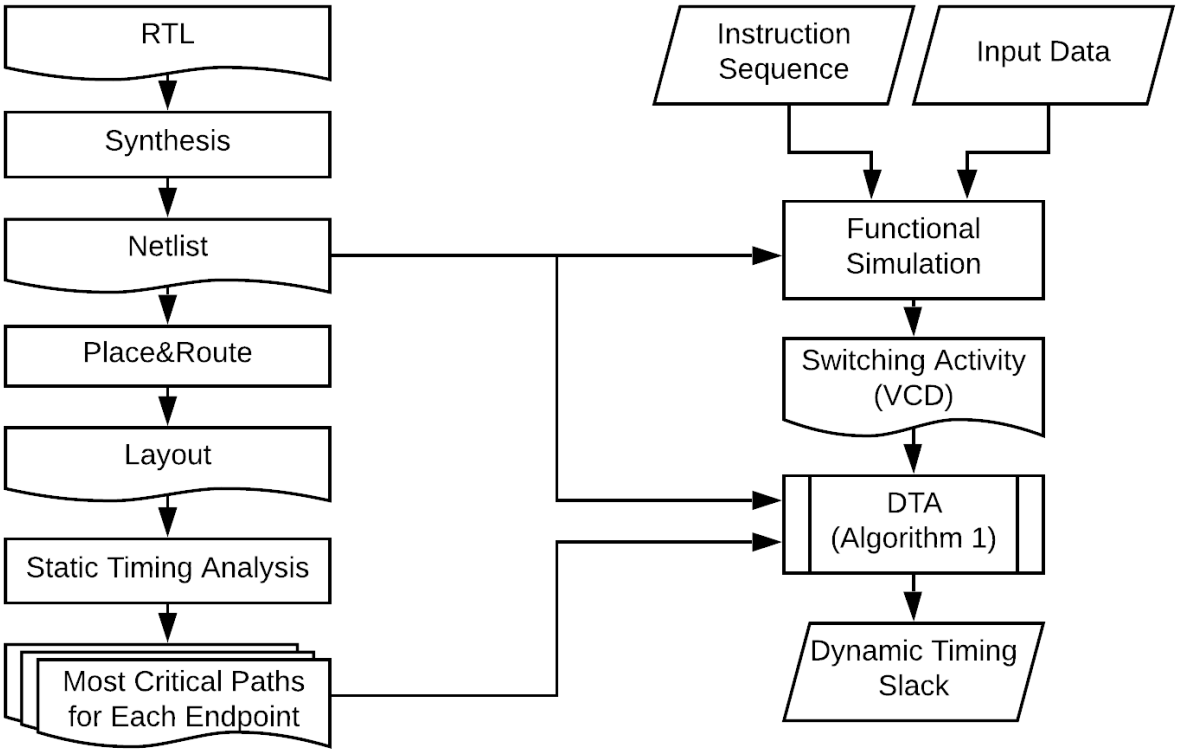


Figure 4.1. Dynamic Timing Analysis Flow

clock cycle as *timing slack* of the longest activated path in that stage. Timing slack of a path is the maximum reduction in clock period that would not violate setup time constraint of the endpoint connected to its last gate.

Figure 4.1 shows how the inputs to the algorithm are generated using STA and functional simulations on a given instruction sequence and input data. It assumes STA has been performed and the results – a list of the most critical paths for each endpoint – are available. The algorithm creates a list that includes the most critical path of each endpoint in the specified stage that has been activated in the given clock cycle. It then finds the longest path in the list and returns its timing slack. For each endpoint, it goes over the list of critical paths starting with the longest one and checks if the path was activated by searching the list of activated gates in the given clock cycle. Unless all the gates in the path are found on the list, it discards the path and moves to the next one, repeating the process until it finds an activated path.

To include process variation in the analysis, we replace STA with SSTA. That complicates Algorithm 1 because all timing slacks turn into random variables. So the path at the top of the list of critical paths returned by function *CP* in lines 6 and 22 might not be the true most-critical path. To ensure that *AP* includes all activated paths that could become critical, we run the while-loop (lines 5-20) twice where *CP* selects the most-critical path based on worst-case (1st percentile) timing slacks in one and best-case (99th percentile) timing slacks in the other. Then in line 22, instead of *CP* selecting the most critical path and *SL* returning its timing slack, we combine the two functions and return the statistical minimum of timing slacks of all paths in *AP* using a greedy algorithm [54] that performs a sequence of pairwise minimum operations in an order that would minimize the approximation error.

While accurate, the algorithm is too slow to be used for a comprehensive analysis. We employ three optimization techniques (discussed below) to improve its runtime and enable analysis of real-world applications. We use the algorithm as the ground truth to verify the accuracy of our optimizations and to train the models they are based on.

4.4 Offline Control Network Analysis

The two main factors that determine DTS are the sequence of executed instructions and their input data. The first optimization technique takes advantage of the fact that the sequence of executed instructions is mostly fixed in in-order processors [4], only affected by the program control flow. Starting with the control flow graph (CFG) of the program, we remove that factor by limiting the analysis to basic blocks, straight-line sequences of instructions ending with a control transfer instruction. Each time a basic block is executed, because the sequence of instructions is fixed, the control network, like the logic for fetching and decoding instructions, performs the same task [39]. Therefore, in most cases, the same set of timing paths in the control network are activated every time. To take advantage of this observation, we partition the set of processor endpoints into two sets—the set of *data endpoints* includes endpoints whose contents could change each time an instruction is executed and the set of *control endpoints* includes the rest of the endpoints.

We then pre-characterize DTS of the control endpoints using Algorithm 11 for each instruction in the basic block. We refer to this value as the instruction's control DTS and DTS of data endpoints as data DTS. Note that the characterization is a one-time process and is performed offline. Later during the main simulations, we estimate DTS of an instruction as the minimum of its control DTS and data DTS. Because instructions of two basic blocks can share the pipeline at times, we characterize the control DTS of instructions for each incoming edge to the basic block separately. We use a combination of fuzzing (libFuzzer [35]) and concolic execution (KLEE [11]) to find a minimum set of program input vectors that cover all edges of the CFG and perform gate-level simulations using those input vectors[1].

Table 4.2 shows a list of data endpoints in LEON3 and the pipeline stage in which they are updated. Note that this list does not include data endpoints that are not timing critical and

[1] Characterizing basic blocks in isolation might produce inaccurate results because the processor's state, including instruction cache addresses, would not match its state during program execution

cannot have timing errors. It also does not include endpoints whose contents could change as a result of branch outcomes such as the program counter (PC). We measure DTS of the terminating branch instructions for both scenarios and use the appropriate value during simulations when the branch outcome is known.

To verify our assumption that control DTS is largely unchanged for different instances of an instruction, we measured control DTS every time an instruction was executed during pre-characterization. With the set of data endpoints consisting of endpoints listed in Table 4.2, we were able to achieve an average normalised root mean square (RMS) error of 3% across all instructions. That indicates that almost all variation in DTS is caused by variation in data DTS. In practice, when the assumption has been verified, we only need to measure control DTS once (or a few times for more confidence) for each instruction. When we report the framework's runtime in Section 4.8, we are referring to the latter scenario.

Cache Misses. Some instances of the same instruction can behave differently due to an instruction or data cache miss, but that does not affect our analysis because no timing errors can occur while the cache miss is being resolved because the pipeline is stalled.

Branch Mispredictions. Some instances of a branch instruction could behave differently due to a branch misprediction, but that does not affect our analysis because the recovery penalty of any resulting timing errors is masked by the misprediction penalty.

Traps. Some instances of an instruction could behave differently due to a trap such as an exception or external interrupt request, but that does not affect our analysis because the exception stage where exceptions are resolved is designed with extra guardbands and cannot have timing errors. But the supervisor trap code can be analyzed like the user application program.

4.5 High-Level Modeling of Datapath

To identify the activated paths of a circuit, Algorithm 1 requires activity information of every net. This allows the analysis of any arbitrary circuit, including the control network of a

Table 4.2. Data Endpoints of LEON3 Integer Pipeline

Endpoint	Stage	Description
Upper bits of register addresses	D	Calculated using the current window pointer
Current window pointer	D	Updated by `save` and `restore` instructions
Window overflow/underflow flags	D	Set if the new current window flag is invalid
ALU operands	RA	From register file, register bypasses, or immediate values
ALU input carry	RA	Carry-in of the ALU adder
Shift count	RA	Number of bits to shift
Shift MSB input	RA	Most significant bit of arithmetic right shift result
ALU result	E	Result of arithmetic, logical, and shift instructions
Integer condition codes	E	Calculated based on ALU result
Data cache address	E	Effective address of load and store instructions
Data cache data output	M	Result of load instructions

processor, but requires the functional simulations to be performed at the gate level.

In contrast to the control network, circuits in the processor datapath perform operations that can be described mathematically. This creates an opportunity to simplify the DTA algorithm by utilizing our knowledge about the structure of datapath circuits. The second optimization technique does that by creating models of datapath components that identify their activated paths using activity information of only a subset of nets, allowing the simulations to be performed at a higher level. We develop separate path activation models for adders, shifters, and Boolean operators (AND, OR, etc.).

The most common component used in the datapath is a multi-bit adder. An examination of the data endpoints listed in Table 4.2 shows that, except for the shift operands and data cache output, an addition operation is used for all other data endpoints. Upper bits of register addresses are determined by adding the most significant bit of the register index to the current window pointer. The current window pointer itself is updated by incrementing or decrementing its previous value and the result is used to detect window overflow and underflow traps. The ALU adder is used when an `add` or `sub` instruction, or other instructions that perform addition such as `save`, `restore`, and `jmpl`, is in EX stage to find the result and input carry of the ALU, its operands when a dependent instruction is in RA stage, and to calculate data cache address of a `load` or `store` instruction. Here, we use a n-bit adder, shown in Figure 4.2, to describe our modeling technique. We use a ripple-carry adder for simplicity but the analysis is not dependent on the adder implementation and applies, without any changes, to other types of adders because it makes no assumptions about timing slack of paths as previous research has shown that the length of the carry-chain is not a reliable predictor of DTS in adders [62]. Other datapath components, including logical operators (AND, OR, etc.) and shifters, have significantly simpler DTS models, mainly because of their parallel paths from inputs to outputs resulting in almost constant timing slacks. In the remainder of this section, we focus on modeling DTS for multi-bit adders.

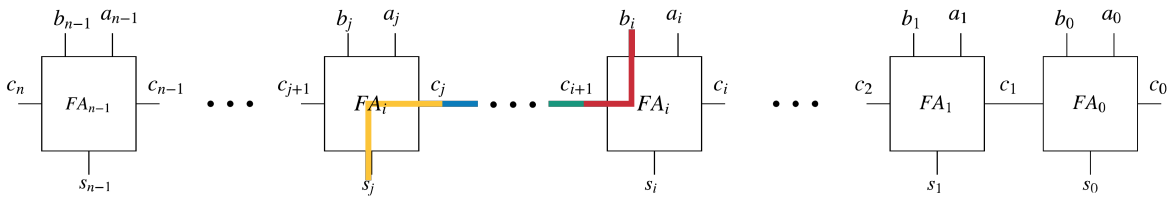


Figure 4.2. Example Segmentation for a Path from b_i to s_j of an n-Bit Adder

4.5.1 Formulation

We start by defining a few terms and symbols. Consider a path p with n gates $g_1, \ldots, g_n$ where g_1 is an endpoint and for all $1 \leq i < n$, g_i is connected to g_{i+1}.

Definition 4.5.1. *For any pair of gates $g_i, g_j \in p$ with $i \leq j$, we call the set $p(g_i, g_j) = \{g_k | i \leq k \leq j\}$ a path segment between g_i and g_j through p. Note that $p = p(g_1, g_n)$. We say a path segment is activated in a particular clock cycle if all its gates are activated in that clock cycle.*

Definition 4.5.2. *Any m-element set $q \subset p$ defines a segmentation of p, creating $m+1$ path segments defined as $p(q) = \{p(g_i, g_j) | g_i, g_j \in q \wedge g_k \notin q, i \leq k \leq j\}$.*

Combining these with definitions of activated gates and paths in Section 4.3, it can be shown that the path p is activated in a particular clock cycle if and only if all path segments in $p(q)$ are activated in that clock cycle, where q is a segmentation of p.

Notation. We identify a gate and the net connected to its output with the same lower-case letter. Binary variables x^p and x^c represent the value of net x in the previous and current clock cycles, respectively, and the corresponding upper-case letter, X, denotes a binary variable representing the activation of x. $X = 1$ indicates that the net/gate x is activated, i.e., $x^p \neq x^c$, and $X = 0$ indicates that it has not, i.e., $x^p = x^c$. We can, therefore, write $X = x^p \oplus x^c$, where $\oplus$ is the exclusive-or operator.

4.5.2 Overview

Path Segmentation. Consider a path p starting from an input port connected to FA_i and ending in an output port connected to FA_j where $i < j$. We determine if p is activated by checking if all path segments in $p(q)$ are activated where q is a segmentation of p. Letting $q = \{c_{i+1}, \ldots, c_j\}$ limits each path segment to a single full-adder. Therefore, it is sufficient for our model to identify path segments of a single full-adder. An example is shown in Figure 4.2 with path segments in different colors.

Nets. Figure 4.3 shows the gate-level implementation of a full-adder. It has three inputs a, b, and c_{in}, and two outputs s and c_{out}[2]. In addition, we have marked three internal nets with labels g, p, and h. To simplify the analysis, we use a single label op to represent both operand inputs a and b such that op is activated if either a or b is activated. So if we find that a path from op is activated, we can conclude that at least one of the corresponding paths from a and b is activated.

Paths. There are two paths ending in s, $c_{in} \rightarrow s$ and $op \rightarrow s$. We call these paths *operand-to-sum* (OS) and *carry-to-sum* (CS), respectively. Similarly, there are three paths ending in c_{out}, $c_{in} \rightarrow h \rightarrow c_{out}$, $op \rightarrow g \rightarrow c_{out}$, and $op \rightarrow p \rightarrow h \rightarrow c_{out}$. We call these paths *carry-chain* (CC), *carry-generate* (CG), and *carry-propagate* (CP), respectively.

4.5.3 Theorems

We use Lemma 4.5.1 to help identify the activated path to the sum output of a full-adder.

Lemma 4.5.1. *A 2-input XOR gate is activated if and only if exactly one of its input nets is activated.*

Proof. The output z of an XOR gate with inputs a and b is 0 if $a = b$ and 1 if $a \neq b$.

If $a^p = b^p$, $z^p = 0$. If either a or b, but not both, is activated, $a^c \neq b^c$ and so $z^c = 1$. That means z is activated (because $z^p \neq z^c$) and so the gate is activated as well. Similarly, if $a^p \neq b^p$,

[2]In our analysis of a single full-adder, we assume that the inputs and outputs are endpoints, so we refer to the sequence of gates that connects an input to an output as a path rather than a path segment.

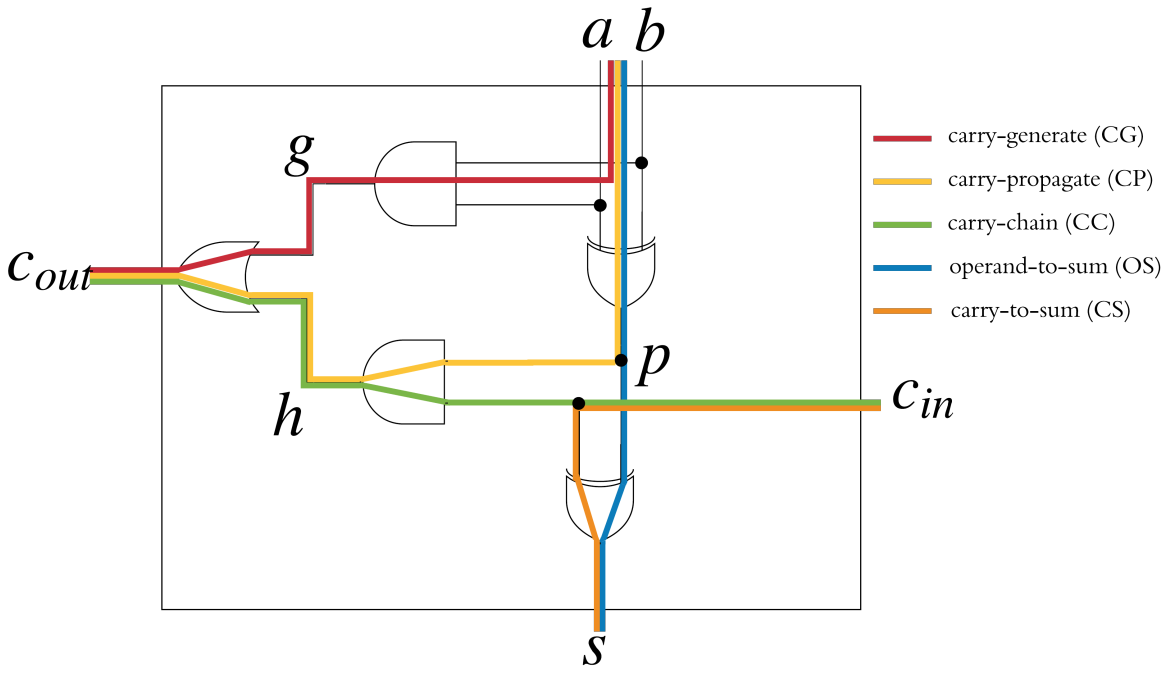


Figure 4.3. Gate-Level Implementation of a Full Adder and Its Paths

$z^p = 1$. If either a or b, but not both, is activated, $a^c = b^c$ and so $z^c = 0$. That means z is activated (because $z^p \neq z^c$) and so the gate is activated as well.

Conversely, if the gate is activated, z is activated, i.e., $z^p \neq z^c$. If $z^p = 0$ and $z^c = 1$, $a^p = b^p$ and $a^c \neq b^c$. That can only happen when either a or b, but not both, is activated. Similarly, if $z^p = 1$ and $z^c = 0$, $a^p \neq b^p$ and $a^c = b^c$. That can only happen when either a or b, but not both, is activated. □

Theorem 4.5.1 allows us to determine if a path to the sum output of a full-adder is activated based on whether or not the other path to that output is activated.

Theorem 4.5.1. *In a full-adder, OS and CS paths cannot be activated at the same time (i.e., in the same clock cycle).*

Proof. We prove the theorem by contradiction. Assume OS and CS paths are both activated. So all their gates, in particular, the endpoint c_{in} and both XOR gates, are activated, and so both c_{in}

and p are activated. That is a contradiction because c_{in} and p are inputs of an activated XOR gate and, according to Lemma 2, cannot both be activated. □

We use Lemma 4.5.2 to help identify the activated paths to the carry-out output of a full-adder.

Lemma 4.5.2. *An OR gate with inputs a and b and output z cannot be activated unless either (1) $a^p = b^p = 0$ or (2) $a^c = b^c = 0$, but not both, is true.*

Proof. Since $z^p = a^p + b^p$, $z^p = 0$ is true if and only if $a^p = b^p = 0$. Similarly, since $z^c = a^c + b^c$, $z^c = 0$ is true if and only if $a^c = b^c = 0$. By definition, the gate is activated if and only if its output z is activated, i.e., $z^p \neq z^c$. If $z^p = 0$ and $z^c = 1$, (1) is true and (2) is false and if $z^p = 1$ and $z^c = 0$, (1) is false and (2) is true. □

Theorem 4.5.2 allows us to determine the source of an activated path to the carry-out output of a full-adder.

Theorem 4.5.2. *In a full-adder, (1) CG and CP paths cannot be activated at the same time, and (2) CG and CC paths cannot be activated at the same time.*

Proof. We prove the theorem by contradiction. (1) Assume CG and CP are both activated. So all their gates, in particular, the OR gate and both AND gates, g and h, are activated. According to Lemma 1, since the OR gate is activated, either $g^p = h^p = 0$ or $g^c = h^c = 0$. Since g and h are activated, if $g^p = h^p = 0$, $g^c = h^c = 1$ and if $g^c = h^c = 0$, $g^p = h^p = 1$. In either case, because g and h are driven by AND gates, the case where both are 1 requires that all inputs of the AND gates, in particular, a, b, and p, are 1. That is a contradiction because a, b, and p are connected to an XOR gate and, by definition, cannot all be 1. This proof can be used, without any changes except replacing CP with CC, to prove (2) as well. □

4.5.4 Identifying Activated Paths

Based on Theorems 4.5.1 and 4.5.2, we can determine if each of the five paths of a full-adder is activated using Equations 4.1-4.5.

$$CG_i = C_i.G_i \tag{4.1}$$

$$CP_i = C_i.\overline{G_i}.P_i \tag{4.2}$$

$$CC_i = C_i.\overline{G_i}.C_{i-1} \tag{4.3}$$

$$OS_i = S_i.P_i \tag{4.4}$$

$$CS_i = S_i.\overline{P_i}, \tag{4.5}$$

where C_{i-1} and C_i are C_{in} and C_{out} of FA_i and CG_i, CP_i, CC_i, OS_i, and CS_i are binary variables representing activation of its paths. For example, $CG_i = 1$ indicates that the CG path of FA_i is activated while $CG_i = 0$ indicates that it is not.

With activated path segments of each full-adder identified, activated paths of the n-bit adder can be found using Equations 4.6-4.12.

$$a_i/b_i \rightarrow s_i = OS_i \tag{4.6}$$

$$c_{in} \rightarrow s_i = CS_i.CC_{i-1}\ldots CC_0 \tag{4.7}$$

$$a_i/b_i \rightarrow g_i \rightarrow s_j = CS_j.CC_{j-1>i}\ldots CC_{i+1<j}.CG_i \tag{4.8}$$

$$a_i/b_i \rightarrow p_i \rightarrow s_j = CS_j.CC_{j-1>i}\ldots CC_{i+1<j}.CP_i \tag{4.9}$$

$$a_i/b_i \rightarrow g_i \rightarrow c_{out} = CC_{n-1}\ldots CC_{i+1<n}.CG_i \tag{4.10}$$

$$a_i/b_i \rightarrow p_i \rightarrow c_{out} = CC_{n-1}\ldots CC_{i+1<n}.CP_i \tag{4.11}$$

$$c_{in} \rightarrow c_{out} = CC_{n-1}\ldots CC_0, \tag{4.12}$$

where $in \rightarrow (net) \rightarrow out = 1$ indicates that the path from input *in* to output *out* (through net *net*) is activated and the binary variables on the right-hand-side represent the activation of full-adder

path segments as defined in Equations 4.1-4.5.

Finally, we use Algorithms 2-10 to identify activated paths of the multi-bit adder. Algorithm 2 identifies the first (from the LSB) activated carry-chain of an n-bit adder. It uses two calls to a function that returns the number of leading zeros of an n-bit binary vector. Many architectures have dedicated instructions that implement this function efficiently. In SPARC Oracle Architecture 2011 and later, that instruction is called `lzcnt`.

Algorithm 2: First Activated Carry-Chain

Input : An n-bit binary vector CC
Output : A pair of integers (i, j) where the first chain of 1's (from LSB) starts at index i and ends at index j.

```
1 Function (i, j) = FirstActivatedCarryChain(CC):
2     if CC = 0 then
3         return (0,0)
4     i ← lzcnt(CC) /* number of leading zeros          */
5     CCC ← CC ≫ i /* right-shift by i bits            */
6     CCC ← not(CCC) /* bitwise NOT                    */
7     j ← lzcnt(CCC) /* number of leading zeros        */
8     return (i, j)
```

Algorithms 3-9 describe how Equations 4.6-4.12 can be used to find all activated paths of a multi-bit adder. Note that these algorithms do not introduce any inaccuracy in calculation of DTS and are as accurate as the static timing analysis tool used to train the model. As mentioned earlier, they make no assumptions about timing slack of the paths. They do not assume, for example, that a path from a_i/b_i to s_j is slower, i.e., has a smaller timing slack, than a path from a_k/b_k to s_l even if the carry-chain it uses is longer, i.e, $j-i > l-k$. The only assumption they make is that a path from a_i/b_i to s_j is faster than a path from a_i/b_i to s_k if $j < k$ and a path from a_i/b_i to s_j is faster than a path from a_k/b_k to s_j if $i > k$.

Finally, during the simulation, the function described in Algorithm 10 is called for each datapath adder used by an instruction after its operands and results are identified.

Algorithm 3: Local Operand-to-Sum Paths

Input : An n-bit binary vector OS

Output : An n-bit binary vector $OSLV$ where $OSLV[i] = 1$ if $a_i/b_i \rightarrow s_i$ is activated, $OSLV[i] = 0$ otherwise.

```
1 Function OSLV = OSLActivatedPaths(OS):
2     return OS
```

Algorithm 4: Carry-in-to-Sum Paths

Input : Two n-bit binary vectors CS and CC

Output : Largest integer k for which $c_{in} \rightarrow s_k$ is activated, -1 if no such path is activated.

```
1  Function k = CISActivatedPaths(CS,CC):
2      (i, j) ← FirstActivatedCarryChain(CC)
3      if i ≠ 0 then
4          return −1
5      ks ← j
6      while ks ≥ 0 do
7          if CS[ks] = 1 then
8              return ks
9          ks ← ks − 1
10     return −1
```

Algorithm 5: Operand-to-Sum Paths via Generate Net

Input : Three n-bit binary vectors CG, CC, and CS

Output : An $n \times n$ binary matrix $OGSM$ where entry (i, j) is 1 if $a_i/b_i \to g_i \to s_j$ is activated.

```
Function OGSM = OGSActivatedPaths(CG,CC,CS):
    CCC ← CC
    OGSM ← 0 /* set all entries to zero                    */
    while CCC ≠ 0 do
        (i, j) ← FirstActivatedCarryChain(CCC)
        CCC ← CCC ≫ (j+1) /* right-shift by j+1 bits       */
        k ← i − 1
        while k ≤ j do
            if CG[k] = 1 then
                break
            k ← k + 1
        if k > j then
            break
        l ← j + 1
        while l ≥ i do
            if CS[l] = 1 then
                break
            l ← l − 1
        if l < i then
            break
        OGSM(k, l) = 1
    return OGSM
```

Algorithm 6: Operand-to-Sum Paths via Propagate Net

Input : Three n-bit binary vectors CP, CC, and CS

Output : An $n \times n$ binary matrix $OPSM$ where entry (i, j) is 1 if $a_i/b_i \to p_i \to s_j$ is activated.

```
1  Function OPSM = OPSActivatedPaths(CP,CC,CS):
2      CCC ← CC
3      OPSM ← 0 /* set all entries to zero */
4      while CCC ≠ 0 do
5          (i, j) ← FirstActivatedCarryChain(CCC)
6          CCC ← CCC ≫ (j+1) /* right-shift by j+1 bits */
7          k ← i − 1
8          while k ≤ j do
9              if CP[k] = 1 then
10                 break
11             k ← k + 1
12         if k > j then
13             break
14         l ← j + 1
15         while l ≥ i do
16             if CS[l] = 1 then
17                 break
18             l ← l − 1
19         if l < i then
20             break
21         OPSM(k, l) = 1
22     return OPSM
```

Algorithm 7: Operand-to-Carry-out Paths via Generate Net

Input : Two n-bit binary vectors CG and CC

Output : Smallest integer k which is 1 if $a_k/b_k \rightarrow g_k \rightarrow c_{out}$ is activated, -1 if no such path is activated.

```
1  Function k = OGCActivatedPaths(CG,CC):
2      CCR ← reverse(CC) /* reverse vector (MSB and LSB
         swapped)                                          */
3      (i, j) ← FirstActivatedCarryChain(CCR)
4      if i ≠ 0 then
5          return −1
6      k ← n − j − 1 /* n is width of the adder             */
7      while k ≤ n − 1 do
8          if CG[k] = 1 then
9              return k
10         k ← k + 1
11     return −1
```

Algorithm 8: Operand-to-Carry-out Paths via Propagate Net

Input : Two n-bit binary vectors CP and CC

Output : Smallest integer k for which $a_k/b_k \rightarrow p_k \rightarrow c_{out}$ is activated, -1 if no such path is activated.

```
1  Function k = OPCActivatedPaths(CP,CC):
2      CCR ← reverse(CC) /* reverse vector (MSB and LSB
         swapped)                                          */
3      (i, j) ← FirstActivatedCarryChain(CCR)
4      if i ≠ 0 then
5          return −1
6      k ← n − j − 1 /* n is width of the adder             */
7      while k ≤ n − 1 do
8          if CP[k] = 1 then
9              return k
10         k ← k + 1
11     return −1
```

Algorithm 9: Carry-in-to-Carry-out Path

Input : An n-bit binary vector CC

Output : A binary variable $CICO$ where $CICO = 1$ if $c_{in} \rightarrow c_{out}$ is activated.

```
1  Function CICO = CICOActivated(CC):
2      return (CC = 2^n − 1)
```

Algorithm 10: Dynamic Timing Slack of Multi-Bit Adder

1 **Procedure** `AdderDTS()`**:**

`/* current value of nets */`

2 $g^c \leftarrow op_1^c \wedge op_2^c$

3 $p^c \leftarrow op_1^c \oplus op_2^c$

4 $s^c \leftarrow op_1^c + op_2^c + c_{in}^c$

5 $c^c \leftarrow p^c \oplus s^c$

`/* previous value of nets */`

6 $g^p \leftarrow op_1^p \wedge op_2^p$

7 $p^p \leftarrow op_1^p \oplus op_2^p$

8 $s^p \leftarrow op_1^p + op_2^p + c_{in}^p$

9 $c^p \leftarrow p^p \oplus s^p$

`/* net activation variables */`

10 $G \leftarrow g^p \oplus g^c$

11 $P \leftarrow p^p \oplus p^c$

12 $S \leftarrow s^p \oplus s^c$

13 $C_{out} \leftarrow c^p \oplus c^c$

14 $C_{in} \leftarrow (C_{out} \ll 1) \vee (c_{in}^p \oplus c_{in}^c)$

`/* activated path segments of full-adders */`

15 $CG \leftarrow C_{out} \wedge G$

16 $CP \leftarrow C_{out} \wedge \overline{G} \wedge P$

17 $CC \leftarrow C_{out} \wedge \overline{G} \wedge C_{in}$

18 $OS \leftarrow S \wedge P$

19 $CS \leftarrow S \wedge \overline{P}$

`/* activated paths of the adder */`

20 *OSLV* ← `OSLActivatedPaths(`*OS*`)`

21 *OGSM* ← `OGSActivatedPaths(`*CG, CC, CS*`)`

22 *OPSM* ← `OPSActivatedPaths(`*CP, CC, CS*`)`

23 *CICO* ← `CICOActivated(`*CC*`)`

24 *kg* ← `OGCActivatedPaths(`*CG, CC*`)`

25 *kp* ← `OPCActivatedPaths(`*CP, CC*`)`

26 *kc* ← `CISActivatedPaths(`*CC, CS*`)`

`/* find dynamic timing slack */`

27 `FindDTS(`*CICO, OSLV, OGSM, OPSM, kg, kp, kc*`)`

Instrumenting addx rs1, rs2, rd

```
/* save operands (and carry-in) in local registers %l0-%l2 */
mov     rs1, %l0 /* first operand */
mov     rs2, %l1 /* second operand */
addx    %g0, 0, %l2 /* input carry */
/* find DTS (operands of previous instruction are in %o3-%o5) */
mov     %l0, %o0 /* first parameter */
mov     %l1, %o1 /* second parameter */
mov     %l2, %o2 /* third parameter */
call    adderDTS
nop
/* copy operands to %o3-%o5 for the next instruction */
mov     %l0, %o3
mov     %l1, %o4
mov     %l2, %o5
/* execute the instruction */
addx     rs1, rs2, rd
```

Figure 4.4. Example Instrumentation Code for Add-with-Carry Instruction

4.5.5 Execution-Driven-Simulation

Because our datapath DTS model only requires values of architecturally-visible registers, we can perform the analysis at the architecture level. Similar to [3], to further improve efficiency, instead of a simulator, we implement the model by instrumenting the program with native instructions, implemented in LLVM [42] back-end for SPARC. Figure 4.4 shows an example template for `addx` (add-with-carry) instruction. The instrumentation code extracts the operands and input-carry of the the `addx` instruction and those of the previous instruction—as if it is also an `addx` instruction—and passes them to `AdderDTS` (Algorithm 10) to find data DTS. The instrumentation technique is described in more detail in Chapter 5.

4.5.6 Training and Application

We use the gate-level DTA algorithm introduced in Section 4.3 to train our datapath DTS model. We perform functional simulations with special training code and input vectors similar to the ones we used to train CTM in Chapter 3 to selectively activate the set of paths we identified in Section 4.4 for each datapath component. For example, we use `add` instructions for the ALU

adder and `save` and `restore` instructions for the current window pointer adder and select a set of input vectors that activate the paths listed in Equations 4.6-4.12. During the simulations, we use the model to find the activated paths of datapath components for each instruction and estimate DTS as the minimum DTS of all data endpoints.

4.6 Instruction Error Model

4.6.1 Instruction Error Probability

The DTA tool described in Section 4.3 calculates DTS of a pipeline stage. We define DTS of an instruction as the minimum DTS of all pipeline stages in the clock cycle that the instruction is in that stage. An instruction with a negative DTS will experience at least one timing error as it moves through the pipeline. Algorithm 11 uses the DTS of pipeline stages to calculate DTS of an instruction executed on an in-order processor.

Algorithm 11: Instruction Dynamic Timing Slack

```
1 Function InstDTS(N,t,VCD):
2     return min_{s=0:S(N)-1}( DTS(N, s, t + s, VCD) )
```

Figure 4.5 shows the three components of the instruction DTS estimation flow. The three components, control network DTS characterization, datapath DTS characterization, and datapath activity characterization, implement the optimizations discussed in Sections 4.4, 4.5, and 4.5.5, respectively.

As explained in Section 4.3, when process variation is considered in the analysis, DTS is a random variable rather than a fixed number. Therefore, it is not possible to deterministically predict whether or not some instructions with near-zero DTS will experience timing errors. Instead, we can assign a *probability* of error to each instruction. As the program is executed with different input vectors, we record error probability of all dynamic instances of each instruction and form a probability distribution of them that captures the effect of data variation. We also measure, for each basic block, the *activation probability* of each incoming edge as the fraction

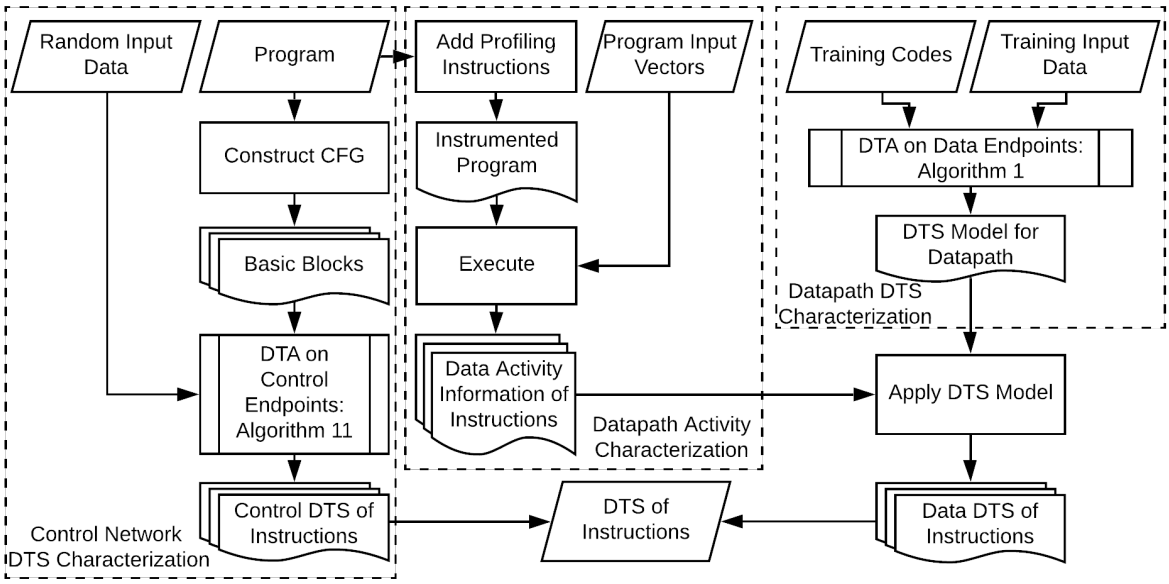


Figure 4.5. Instruction DTS Estimation Flow

of basic block executions in which the edge was used to transfer the control to the basic block.

4.6.2 Inter-Instruction Correlation

The error recovery mechanism used by the TS processor can have a dynamic effect on instruction error probabilities. For example, when a timing error is detected, the processor might insert bubbles into the pipeline to keep the errant instruction and the ones that follow from changing the architectural state [19] or flush the pipeline to resolve any complex bypass register issues [7]. Consequently, the next instruction has to change the processor state, i.e., the contents of registers, not from the state induced by the errant instruction, but from the state induced by the recovery mechanism, to the state it induces itself, thereby activating a different set of timing paths. In other words, when we simulate the program, the instruction error probabilities we find are in fact *conditional* probabilities assuming correct execution of the previous instruction. So we must also find the other set of conditional error probabilities—assuming the previous instruction experienced a timing error. We emulate the error recovery scheme by instrumenting the program with instructions that mimic its effect. For example, we insert a `nop` instruction

before every instruction in the program to mimic the effect of a pipeline flush[3]. We proceed by describing a procedure for computing the *marginal* error probabilities.

Problem Formulation 1. *Suppose that the program has been divided into m basic blocks* $B_1, \ldots, B_m$. *Let* d_i *and* n_i *be the number of incoming edges (indegree) and instructions of* B_i, *respectively. For all* $j = 1, \ldots, d_i$ *and* $k = 1, \ldots, n_i$, $p^a_{i_j}$ *is the activation probability of the* j*th incoming edge to* B_i *such that* $\sum_{j=1}^{d_i} p^a_{i_j} = 1$ *while random variables* $p^c_{i_k}$ *and* $p^e_{i_k}$ *are conditional error probabilities of its kth instruction given the previous instruction has executed correctly or incorrectly, respectively. Let* p_{i_k} *be a random variable representing the (marginal) error probability of the kth instruction in the ith basic block. Find* p_{i_k} *using* $p^c_{i_k}$, $p^e_{i_k}$, *and* $p^a_{i_j}$ *for all* $i = 1, \ldots, m$, $k = 1, \ldots, n_i$, *and* $j = 1, \ldots, d_i$.

If the marginal error probability of the first instruction of a basic block is known, marginal error probabilities of all others can be computed using a recurrence relation. For all $k = 2, \ldots, n_i$.

$$p_{i_k} = p^e_{i_k} p_{i_{k-1}} + p^c_{i_k}(1 - p_{i_{k-1}}) \tag{4.13}$$

For basic blocks with more than one incoming edge, define *input error probability* of B_i as a new random variable p^{in}_i that represents the error probability of the instruction executed just before entering B_i. In addition, let $p^{out}_i = p_{i_{n_i}}$ be the *output error probability* of B_i. To model the assumption that the processor is in a flushed state when it starts executing the program, we assume $p^{in}_1 = 1$. Then,

$$p^{in}_i = \sum_{j=1}^{d_i} p^a_{i_j} p^{out}_{t_i(j)}, \tag{4.14}$$

where $t_i(j)$ is the index of the basic block connected to the tail of the jth incoming edge to B_i. If the program's CFG is acyclic, applying Equations 4.13 and 4.14 (with $p_{i_0} = p^{in}_i$) to the basic blocks sequentially determines unconditional error probabilities of all instructions. However, a non-trivial program almost always contain loops and its CFG is, therefore, cyclic. If some basic

[3]The added instructions are only used to find the conditional error probabilities and the phrase "previous instruction" still refers to the previous instruction in the original program.

blocks form a cycle in the CFG, their input and output error probabilities would depend on each other in a cyclic manner. So instruction error probabilities cannot be obtained consecutively.

For cycles in the CFG, we construct a system of linear equations by writing Equations 4.13 and 4.14 for all the basic blocks in the cycle, in which edge activation probabilities form the coefficient matrix and instruction error probabilities are the unknowns. In order to implement this, we employ Tarjan's algorithm [58] to identify the strongly connected components of the CFG and find their topological ordering. Tarjan's algorithm takes a directed graph as input and produces, in linear time, a partition of the graph's vertices into the graph's strongly connected components. The order in which the strongly connected components are identified constitutes a reverse topological sort of the acyclic graph formed by the strongly connected components. We can, therefore, write and solve the system of linear equations for each component in the topological order of components.

4.7 Program Error Rate

4.7.1 Overview

In this section, we propose a methodology for estimating a program's error rate distribution. To simplify the equations, we estimate the number of timing errors, *error count*, rather than error rate. Our approach is inspired by the fact that real-world programs typically execute a very large number (up to trillions) of dynamic instructions. This observation, along with the fact that each instruction fails with a small probability, hints at effective use of limit theorems for estimating program error count. Specifically, we use the *law of rare events*, also known as the Poisson limit theorem, to approximate the program error count with a Poisson distribution and the *law of large numbers*, also known as the central limit theorem, to approximate the parameter of this Poisson distribution with a Gaussian one. To verify the accuracy of our approximations, we cannot use Monte Carlo experiments because our baseline simulator is too slow to handle large input datasets. Instead, we use *Stein's method* and its application, *Chen-Stein method*, to

obtain bounds on the approximation error of the normal and Poisson distributions, respectively.

Problem Formulation 2. *Suppose that the program has been divided into m basic blocks* $B_1, \ldots, B_m$. *Let* n_i *and* e_i *be the number of instructions and executions of basic block* B_i, *respectively. For all* $i = 1, \ldots, m$ *and* $k = 1, \ldots, n_i$, *let* I_{i_k} *be a set of Bernoulli random variables corresponding to the instructions such that* $Pr(I_{i_k} = 1)$ *is equal to the error probability of the kth instruction in the ith basic block, denoted by* p_{i_k}. *Moreover, let* I_i^{in} *be a Bernoulli random variable representing the instruction executed just before entering* B_i *and let* $p_i^{in} = Pr(I_i^{in} = 1)$. *Assume that* I_{i_k} *and* p_{i_k} *are only dependent on* $I_{i_{k-1}}$ *and* $p_{i_{k-1}}$, *respectively, for all* $i = 1, \ldots, m$ *and* $k = 2, \ldots, n_i$ *and on* I_i^{in} *and* p_i^{in} *for* $k = 1$[4]. *The number of errors in the program, a random variable denoted by* N_E, *can then be calculated as a weighted sum of the Bernoulli random variables* $N_E = \sum_{i=1}^{m} \sum_{k=1}^{n_i} e_i I_{i_k}$. *Estimate the program error count distribution as an approximation of* N_E *denoted by* $\overline{N}_E$.

4.7.2 The Law of Rare Events

The distribution of the sum of independent, non-identically distributed Bernoulli indicators is called a *Poisson binomial distribution* (PBD). Computing PBD, however, becomes prohibitively complex when there are more than a few indicators [34]. Consequently, approximation techniques targeting various distributions such as normal and Poisson have been widely developed and used [20]. The law of rare events provides the intuition (proof in [43]) that when there are a large number of indicators, each with a small success probability, PBD is approximately a Poisson distribution. Even in the case where the indicators are not independent, if the dependence can be somehow confined, their sum should still approximately follow a Poisson distribution. Accordingly, since programs typically execute a large number of instructions, each with a very small error probability, the total number of errors could effectively be approximated

[4]Note that the dependence of I_{i_k} on $I_{i_{k-1}}$ (whether or not the instructions fail) and that of p_{i_k} on $p_{i_{k-1}}$ (the probability that the instruction fails) stem from different roots. The former is caused by the error recovery mechanism (see Section 4.6) while the latter is a result of the correlation between DTS of the two instructions due to their activated paths including the same gates and/or nearby gates affected by the spatial correlation property of process variation.

by a Poisson distribution. But for this approximation to be reliably used, it is necessary to determine how much error it could potentially incur. A method for establishing bounds on normal approximation of the sum of dependent random indicators was introduced by Stein [57]. Chen [13] later used this methodology in the Poisson setting and obtained error bounds for Poisson approximation as well. Here, we use the Stein and Chen-Stein methods to evaluate the reliability of using Poisson and normal approximations for estimating the distribution of a program's error count. We start by a formal formulation of the results of the Chen-Stein method as given in [1].

Theorem 4.7.1 (Chen-Stein method)**.** *Let I be an index set. For each $\alpha \in I$, let X_α be a Bernoulli random variable with $p_\alpha = Pr(X_\alpha = 1) > 0$. Let $W = \sum_{\alpha \in I} X_\alpha$, and let Z be a Poisson random variable with $EZ = EW = \lambda < \infty$. For each $\alpha \in I$, let $B_\alpha \subset I$ with $\alpha \in B_\alpha$ be a neighborhood of α consisting of the set of indices β such that X_α and X_β are dependent. Define*

$$b_1 = \sum_{\alpha \in I} \sum_{\beta \in B_\alpha} p_\alpha p_\beta, \tag{4.15}$$

$$b_2 = \sum_{\alpha \in I} \sum_{\alpha \neq \beta \in B_\alpha} p_{\alpha\beta}, \quad \textit{where} \quad p_{\alpha\beta} = E[X_\alpha X_\beta]. \tag{4.16}$$

Then,

$$d_{TV}(W,Z) \leq min\left\{1, \lambda^{-1}\right\}(b_1 + b_2), \tag{4.17}$$

where $d_{TV}(W,Z)$ is the total variation distance between the distributions of W and Z.

While in our problem, the number of errors is a weighted sum of the Bernoulli indicators, because the indicators can be dependent and the weights are integers, we can simply assume multiple identical indicators for each instruction as reflected in Equation 4.18.

$$N_E = \sum_{i=1}^{m} \sum_{k=1}^{n_i} e_i I_{i_k} = \sum_{i=1}^{m} \sum_{k=1}^{n_i} \sum_{j=1}^{e_i} I_{i_k}. \tag{4.18}$$

Dependency neighborhood of each instruction consists of itself and the previous instruction.

Therefore, we can calculate the parameters b_1 and b_2 to obtain the error bound.

$$b_1 = \sum_{i=1}^{m} \sum_{j=1}^{e_i} (p_i^{in} p_{i_1} + \sum_{k=2}^{n_i} p_{i_{k-1}} p_{i_k}) \tag{4.19}$$

$$b_2 = \sum_{i=1}^{m} \sum_{j=1}^{e_i} (p_i^{in} p_{i_1}^{e} + \sum_{k=2}^{n_i} p_{i_{k-1}} p_{i_k}^{e}) \tag{4.20}$$

We can then write,

$$d_K(N_E, \overline{N}_E) \leq \frac{b_1 + b_2}{\lambda} \tag{4.21}$$

and

$$\lambda = \sum_{i=1}^{m} \sum_{k=1}^{n_i} \sum_{j=1}^{e_i} p_{i_k}, \tag{4.22}$$

where $\overline{N}_E$ is a Poisson random variable with mean (and variance) $E[\overline{N}_E] = E[N_E] = \lambda > 1$ and $d_K(N_E, \overline{N}_E)$ is the *Kolmogorov metric*, the maximum distance between distributions of N_E and $\overline{N}_E$. We could replace the total variation distance in Equation 4.17 with the Kolmogorov metric because $d_K \leq d_{TV}$ (proof in [29]). Also, note that b_1 and b_2 are random variables. However, for the purpose of bounding the approximation error, we will use their worst-case values (expected value plus 6 times standard deviation).

4.7.3 The Law of Large Numbers

To approximate the distribution of λ in Equation 4.22, which we call $\overline{\lambda}$, we turn to another limit theorem. The central limit theorem provides that the sum of a large number of random variables approximately follows a normal distribution. A bound on the approximation error can be found by applying the Stein's method. Theorem 4.7.2 provides a simple description of the results of Stein's method as applicable to our problem.

Theorem 4.7.2 (Stein's method)**.** *Let $X_1, \ldots, X_n$ be random variables such that $E[X_i^4] < \infty$, $E[X_i] = \mu_i$, and define $\mu = \sum_i \mu_i$, $\sigma^2 = Var(\sum_i X_i)$, and $W = \sum_i X_i$. Let the collection*

$(X_1,\ldots,X_n)$ have dependency neighborhoods N_i, $i = 1,\ldots,n$, and let $D = max_{1\leq i\leq n}|N_i|$. Define

$$b_1 = \frac{D^2}{\sigma^3}\sum_{i=1}^{n} E|X_i|^3 \tag{4.23}$$

$$b_2 = \frac{\sqrt{28}D^{\frac{3}{2}}}{\sqrt{\pi}\sigma^2}\sqrt{\sum_{i=1}^{n} E[X_i^4]}. \tag{4.24}$$

Then, for a normal variable $Z = N(\mu,\sigma^2)$,

$$d_K(W,Z) \leq (\tfrac{2}{\pi})^{\frac{1}{4}}(b_1+b_2), \tag{4.25}$$

where $d_K(W,Z)$ is the Kolmogorov metric, the maximum distance between the two distributions.

Defining dependency neighborhoods as before, we have $D = 2$. With error probability distributions represented as discrete random variables, it is straightforward to compute their third and fourth moments to substitute in Equations 4.23 and 4.24. The result is a bound on $d_K(\lambda,\overline{\lambda})$, the maximum distance between distributions of λ and $\overline{\lambda}$.

Finally, the estimated cumulative distribution function of the total number of errors, denoted by $\overline{N}_E(k)$ is given by Equation 4.26.

$$\overline{N}_E(k) = \int_0^{\infty} e^{-\lambda(x)} \sum_{i=0}^{\lfloor k \rfloor} \frac{\lambda^i(x)}{i!} dx \tag{4.26}$$

where $\lambda(x)$ is the probability distribution function of λ. In simple words, $\overline{N}_E(k)$ returns the probability of the program experiencing less than, or exactly, k errors when it is run with a random input on a randomly chosen manufactured chip.

4.8 Experimental Results

4.8.1 Framework Runtime

We selected 12 benchmark programs, two from each of the six categories of MiBench [32]. We used the `small` and `large` input datasets for training and simulation, respectively. The training phase, which consists of characterizing DTS of the control network, was performed on a machine with a 3.40GHz Intel Core i7-3770 processor. We ran the simulations, i.e., executed the instrumented programs, on a Sun UltraSPARC IIIi running Solaris 10 at 1.36GHz and measured the runtime at around 4.6 million instructions (of the original program) per second. In total, it took us around 85 minutes to run all experiments—training the model for 1,240 basic blocks and simulating around 5.8 billion instructions—for the 12 programs. The runtimes for individual programs divided into training and simulation times are listed in Table 4.3.

4.8.2 Error Rate Distributions

Figure 4.6 shows the cumulative probability distributions our framework estimated for each program's error rate along with their lower and upper bounds. The top horizontal axis is labeled (not to scale) with performance improvements resulting from the corresponding error rate on the bottom axis. For example, an error rate of 0.4% results in a 4.93% improvement in performance of the TS processor we considered. Error rate distributions provide an estimate of how much a program would benefit from running on a TS processor, if at all, and how sensitive it is to variations in physical parameters and program input data. The programs exhibit varying degrees of vulnerability to timing errors, with the mean error rates ranging from 0.131% (resulting in a 11.9% performance improvement) in the case of `patricia` to 1.068% (resulting in a 8.46% performance degradation) for `gsm.decode`. The combination of running application and input data can change the performance of a TS processor by as much as 25%, demonstrating that application-specific analysis is necessary for accurate evaluation of TS processors and to identify suitability of specific applications for timing speculation.

Table 4.3. Results, Performance, and Accuracy of Our Framework

Benchmarks	Program Size		Runtime (s)			Error Rate (%)		Approximation Error	
	Instructions	Basic Blocks	Training	Simulation	Total	Mean	SD	$d_K(\lambda, \overline{\lambda})$	$d_K(R_E, \overline{R}_E)$
`basicmath`	1,487,629,739	86	274	322	596	0.406	0.074	0.023	0.020
`bitcount`	589,809,283	72	221	128	349	0.339	0.102	0.035	0.037
`dijkstra`	254,491,123	70	211	55	266	0.441	0.012	0.022	0.020
`patricia`	1,167,201	184	555	0.2	556	0.131	0.017	0.007	0.005
`pgp.encode`	782,002,182	49	140	170	310	0.241	0.049	0.012	0.011
`pgp.decode`	212,201,598	56	160	46	206	0.661	0.110	0.042	0.039
`tiff2bw`	670,620,091	174	450	145	595	0.457	0.131	0.040	0.032
`typeset`	66,490,215	69	241	14	255	0.532	0.022	0.030	0.022
`ghostscript`	743,108,760	192	652	161	813	0.133	0.052	0.015	0.014
`stringsearch`	27,984,283	133	423	6	429	0.351	0.010	0.019	0.015
`gsm.encode`	473,017,210	75	238	102	340	0.753	0.053	0.036	0.032
`gsm.decode`	497,219,812	80	254	107	361	1.068	0.213	0.056	0.054
Total	5,805,741,497	1,240	3,825	1,259	5,084				

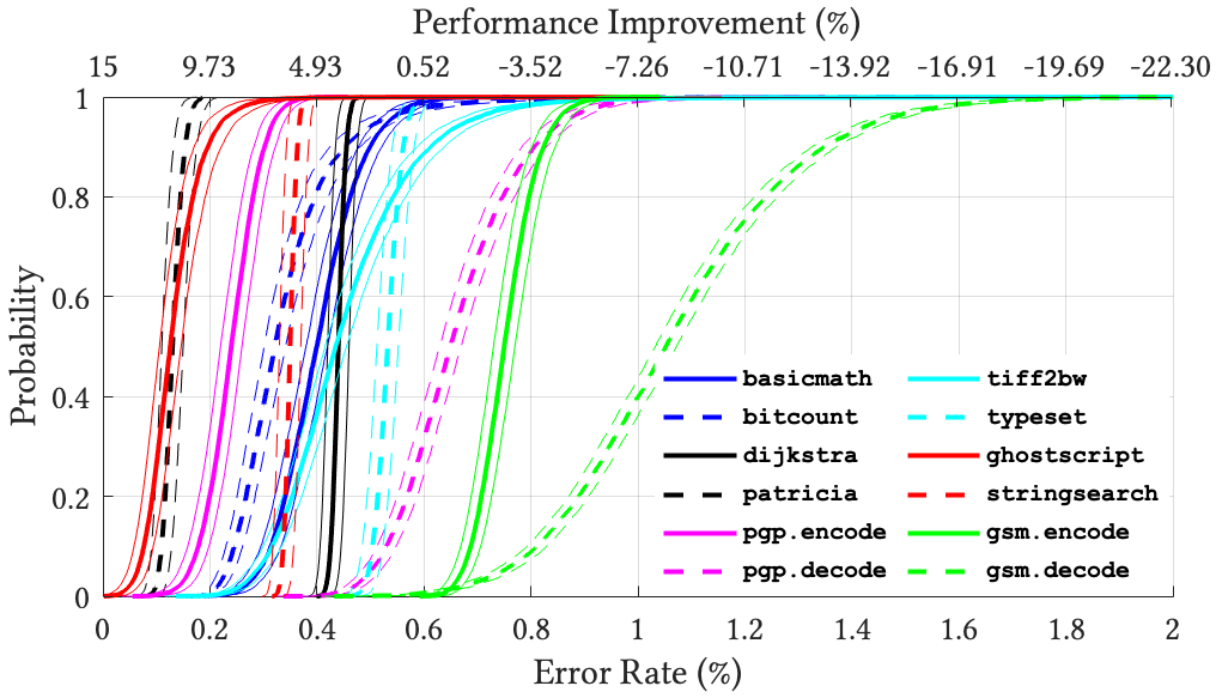


Figure 4.6. Cumulative Probability Distributions of Program Error Rate and Their Lower and Upper Bounds

4.8.3 Approximation Error

In Section 4.7, we identified two sources of inaccuracy in our error rate model—Poisson approximation of program error count and normal approximation of program error count mean. By combining these errors, we form two additional distributions for program error count, a lower bound distribution and an upper bound distribution. First, we add/subtract the bound on $d_K(\lambda, \overline{\lambda})$ we established in Equation 4.25 to/from both instances of λ in Equation 4.26. Then, we add/subtract the bound on $d_K(N_E, \overline{N}_E)$ we established in Equation 4.21 to/from the right-hand side of Equation 4.26. Table 4.3 lists the results for each program. According to these results, our framework can approximate the probability that a program experiences a certain error rate with a maximum error of 5.4%. Note that the last column shows the bounds on the approximation error of program error rate (R_E), not error count (N_E).

Chapter 4 is, in part, a reprint of Omid Assare and Rajesh Gupta, "Accurate Estimation of Program Error Rate for Timing-Speculative Processors," *IEEE/ACM Design Automation*

Conference (DAC), 2019, and, in part, currently being prepared for submission for publication of the material. Omid Assare and Rajesh Gupta, “Performance Analysis of Timing-Speculative Processors.” The dissertation author was the primary investigator and author of these papers.

Chapter 5

Timing Speculation Strategies for Performance Improvement

Performance of TS processors relies on strategies for accurate prediction of optimal operating points. In this chapter, we extend the framework introduced in Chapter 3 to include dynamic frequency tuning and evaluate a range of such timing speculation strategies. We also improve the efficiency of the framework by incorporating two optimization techniques we proposed in Chapter 4.

Our experiments on a TS processor running applications from the MiBench benchmark suite show that, in a typical case, while a perfect timing speculation strategy can improve throughput by up to 143% over a guardbanded design, the most commonly used approach in the literature achieves only a 21.8% of the potential gains. By improving the speculation accuracy, the new strategies we propose in this chapter can realize up to 35.6% of the potential gains, a throughput improvement of 50.9% over a guardbanded design.

5.1 Introduction

Performance of these processors is determined by two competing mechanisms. While increasing the frequency improves processor throughput[1] by fitting more clock cycles into a fixed amount of time, it also increases the rate of timing errors because more paths fail timing

[1] In this chapter, we focus on TS processors that use frequency to tune their operating point, but our method is orthogonal to dynamic voltage scaling.

requirements. The goal of dynamic frequency tuning is finding the frequency that balances these effects such that processor throughput is maximized. Accordingly, TS processors track the error rate during the execution and use it as a feedback mechanism for tuning their frequency. For instance, the conventional approach used in most related proposals periodically samples the processor error rate and tries to keep the long-term error rate close to a pre-specified threshold by increasing (decreasing) the frequency when the error rate is below (above) the threshold. In this chapter, we examine a range of strategies that a TS processor can adopt to dynamically select the optimal operating point.

Contributions of this chapter have been summarized below:

1. We introduce and analyze three timing speculation strategies. First, we argue that frequency tuning should be directed by software and performed at the basic block level where instruction sequence is fixed and predictions are likely to be more accurate. Second, we show that error rate sampling should be temporally limited because the most recent history of errors is often a better predictor of timing behavior. Third, we propose a more robust scheme for dynamic frequency tuning by relying on an optimization algorithm instead of threshold-based control. Finally, we describe the design of a new timing speculation scheme based on these strategies.

2. We extend and improve the simulation framework introduced in Chapter 3 for evaluating the performance of TS processors. The framework creates an instrumented version of the program that simultaneously implements (i) a process-variation-aware instruction-level error model to predict timing errors and estimate error rates as well as (ii) the dynamic frequency tuning mechanism necessary to realize potential gains of timing speculation. This method results in faster simulations because instead of using a microarchitecture-level simulator, they are performed by running the instrumented program on a machine that implements ISA of the target processor.

3. Using our simulation framework, we tune the design parameters of our timing speculation

scheme and evaluate its performance. We show that our method improves processor throughput by more than 50% over a conventional guard- banded design while incurring little power overhead. To put this improvement in perspective, we evaluate the potential gains of an ideal timing speculation scheme that can perfectly track the timing behavior of the system as well as the most popular method in the literature. We find that while the conventional approach can only realize around a fifth of the potential gains of timing speculation, even our efficient method leaves almost two-thirds of potential gains untapped.

5.2 Timing Speculation Strategies

In this section, we describe the design of our timing speculation scheme while analyzing the three main speculation strategies it adopts.

5.2.1 Selective Local Speculation

A number of recent works have documented the concept of spatial timing error locality where static instructions exhibit consistent error behavior over a period of program execution [33]. To take advantage of this phenomenon, in selective local speculation, decisions for changing the frequency are made separately for some basic blocks. This is in contrast to the conventional approach where the processor only tracks and controls the global error rate. We expect that a speculation strategy that works at the basic block level, where the instruction sequence is fixed, can make more accurate predictions.

This scheme can be realized by maintaining a table of basic block error rates and frequencies tagged by the PC address of the first instruction in the basic block. We refer to this table as timing speculation table. At the basic block entry when PC points to the first instruction, the frequency is set to the value previously stored for the basic block. This value should then be updated with a new frequency prediction for the next execution at the basic block exit. To track when execution is exiting the basic block, the table also includes a counter initiated with

the number of basic block instructions, N_I, at the entry of the basic block. This counter is decremented each time an instruction finishes execution, reaching zero at the basic block exit.

The additional costs over the conventional approach include the power and area of the timing speculation table as well as potentially more frequent frequency changes. To control these costs, we introduce two design parameters. The first parameter, N_B, limits the speculation instances spatially to the N_B most frequently executed basic blocks. These basic blocks are selected for local speculation because the accuracy of predicting their frequency affects running time more significantly than others. N_B determines the number of entries in timing speculation table. Similar to [64], we assume that the table is implemented as a SRAM structure and incurs a negligible power overhead as long as $N_B \leq 128$.

The second parameter, N_S, limits the speculation instances temporally by specifying how many times we skip prediction and reuse the previous frequency for a basic block before a new prediction is made. For example, $N_S = 4$ means that a predicted frequency will be reused for the next 4 executions of the basic block. This effectively limits the number of frequency changes for single basic block loops because the predicted frequency does not change for the next N_S iterations/executions. Similar to N_I, this can be implemented with a counter decremented with each basic block execution. New predictions are kept from updating the frequency field of the timing speculation table unless the counter value is 0. We will explore the design space created by these parameters in Section 5.5.

5.2.2 Limited Error Sampling

Selective local speculation is based on the assumption that error rate of previous executions of a basic block is a better predictor of its future error rate than the global error rate. This local strategy raises the question of the appropriate depth of error rate sampling. To predict the future error rate of a basic block, should we implement a long-term sampling scheme using error rates of all previous executions of the basic block, or rely only on its most recent history and use, for instance, the only last n executions?

To answer this question, we designed and performed a simple experiment. In this experiment, we explored how the effective delay of an instruction defined as the propagation delay of the slowest path it sensitizes changes as it is executed multiple times. We selected the 5 most time consuming basic blocks of the programs in Mibench benchmark suite [32] and tracked the effective delays of their instructions as they were iteratively executed in loops. We then measured the distance between effective delays of dynamic instances of each instruction. Figure 5.1 shows the average distance of the instruction effective delays as a function of their execution distance. Execution distance of two dynamic instances of an instruction executed in the kth and jth iterations of the loop is defined to be $|k - j|$. For example, the execution distance between two dynamic instances of an instruction executed in the first and second iterations of the loop is 1 while the first and third instances have an execution distance of 2. Instruction delay distances were measured as the Hellinger distance between the distributions. As Figure 5.1 shows, there is a generally direct relationship between delay and execution distances. This implies that the most recent execution of a basic block is likely a better predictor of its next execution in terms of timing errors. Accordingly, limited error sampling uses the most recent error rate of a basic block to predict its next frequency.

5.2.3 Maximum Throughput Tracking

In the conventional approach, frequency is adjusted so that the error rate remains close to a pre-specified error rate threshold. Consequently, the performance of this method is highly dependent on the selection of error rate threshold(s). In addition to the difficulty of finding optimal threshold values, this approach cannot capture the highly dynamic relationship of frequency and error rate. Maximum throughput tracking is a more robust strategy where error rate threshold is eliminated and frequency adjustments are made based on the dynamic changes of throughput rather than the raw error rate. Similar to the well-known hill climbing algorithm, the direction of frequency change in each iteration is determined based on the effect of the previous change on the throughput. Frequency is increased when a previous frequency increase (decrease)

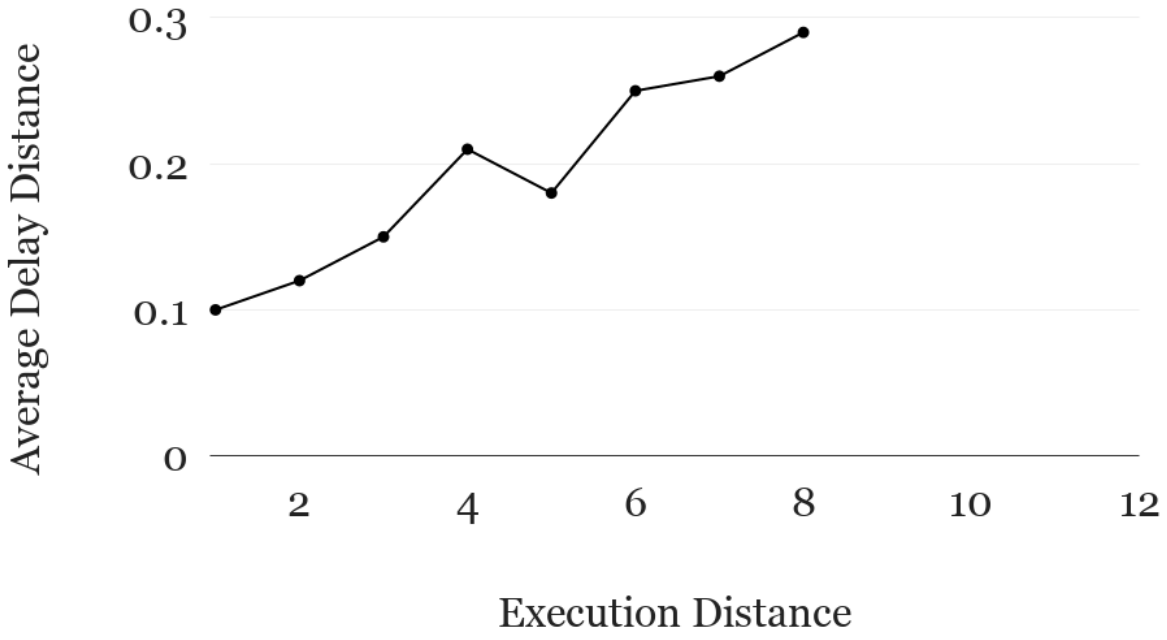


Figure 5.1. Delay Distance as a Function of Execution Distance

has resulted in an increase (decrease) in throughput. Conversely, frequency is decreased when a previous frequency increase (decrease) has led to a decrease (increase) in throughput. Similar to [7], we assume the hardware cost of implementing this algorithm is negligible.

5.3 Error Model

We use a functional timing model called Clustered Timing Model (CTM) [2] that enables dynamic timing analysis by grouping functionally similar timing paths of the processor and modeling their collective propagation delay as a function of their specific operation. Accuracy of CTM has been verified with a maximum error of 6.7% across a wide range of voltage-temperature corners [2]. Our approach in estimating effective delay of instructions is motivated by the observation that typical applications spend most of their runtime in loops, executing a few basic blocks over and over again. In order to take advantage of this, we develop timing models for each basic block in a pre-characterization phase where the most time consuming parts of the

timing analysis are performed offline. In the next section, we will show how this approach allows the simulation to be performed at the architecture level, significantly improving the simulation time.

5.3.1 Clustered Timing Model

Overview. CTM partitions the endpoints of a digital circuit into a set of Register Clusters (RC) and the timing paths into a set of hyperpaths that connect the RCs together. We consider the integer unit of LEON3, an open-source in-order processor core that implements the SPARC V8 architecture [36]. As instructions go through the pipeline, they change the RC values. The model then includes a function for each hyperpath that predicts its effective delay (i.e. maximum propagation delay of its sensitized paths) based on its origin and destination RC value transitions. Finally, an instruction is predicted to cause a timing error when at least one of the hyperpaths it uses has an effective delay larger than the clock cycle.

Functional Paths In order to map RC value transitions to hyperpath effective delays, CTM models each hyperpath, which is essentially a collection of timing paths, as a set of functional paths. The operation performed by a hyperpath is then viewed as a combination of the activation of some of its functional paths. As an example, consider the execution stage hyperpath when an add instruction is being performed. This hyperpath consists of the timing paths of the multi-bit adder in the execution stage. Roughly speaking, each bit in the output of the adder can go high using a carry chain that starts from a lower order position (we ignored the local activation when input carry is zero because carry chains are typically slower). Accordingly, CTM considers a functional path for every possible carry chain in the adder, from every bit position to all higher order ones. Functional paths of all hyperpaths are identified similarly based on their specific operation.

Training and Use. Training of a CTM involves characterizing the delay of functional paths. This is achieved by measuring the hyperpath delay when running special training codes designed to selectively activate specific functional paths. When process variation is considered,

all delay values turn into random variables and the correlation between timing paths is abstracted into correlations between functional paths. To use the model, the delay of a hyperpath is calculated as the maximum of the delays of its activated functional paths rather than the activated timing paths.

Implementation. We implemented a CTM for LEON3 in a micro-architectural simulator, similar to the one described in [2], that takes a sequence of instructions and produces their effective delays. Since the model needs RC values at every clock cycle, the simulation cannot be performed at the architecture level and is, therefore, too slow for typical programs with large data sets. Throughout this chapter, measuring instruction probabilities refers to using this CTM-enabled simulator to estimate them.

5.3.2 Control Delay Characterization

Distinguishing the data and control planes of the processor, our approach is based on the intuition that while the sensitized paths in the processor datapath vary each time a basic block is executed with a different input, control network paths go through similar activation patterns. Using CTM terminology, we propose to classify the hyperpaths into two types: (i) control hyperpaths that together constitute the control network of the processor, and (ii) data hyperpaths that together form the datapath. Then, the effective delay of an instruction, D, is estimated as,

$$D = MAX(D_{control}, D_{data}), \tag{5.1}$$

where MAX represents a statistical maximum operation and $D_{control}$ and D_{data} are the effective delay of the control and data hyperpaths used by the instruction, respectively, hereafter referred to as the instruction control and data delays. Note that all delays are assumed to follow a Gaussian distribution.

By moving estimation of control hyperpath delay to an offline pre-characterization phase, we expect to achieve significant simulation time improvements for the following two reasons.

First, while data hyperpaths perform operations that can be concisely described mathematically, control hyperpaths have a more irregular functional path structure due to the bit-level computations they perform on control signals. More importantly, we can now perform the simulation at the architecture level because data hyperpath delays can be modeled using only architecturally visible registers whereas estimating control hyperpath delays requires values of internal pipeline registers, referred to as RCs in CTM terminology.

Therefore, in the pre-characterization phase, we measure the control delays of all instructions for each basic block. A complicating issue is the effect of the program control flow. Instructions of two neighboring basic blocks usually share the pipeline during their execution. As a result, control delay of an instruction could be different depending on the previous executed basic block. To account for this effect, we measure instruction control delays once for each possible previous basic block in the Control Flow Graph (CFG). Later during the simulation when the previous basic block is known, we use the appropriate control delay when evaluating Equation 5.1.

Suppose that we want to characterize control delays of instructions in basic block B along one of its incoming edges e from basic block B. We implemented a simple symbolic execution tool that derives the branch condition of B in terms of its input (i.e. registers and/or memory locations accessed by instructions in B). This condition is then used to ensure that the randomly generated input executes e. Finally, the execution is started at the top of B and control delays of instructions in B are measured. This process is repeated multiple times and the mean of all measured control delays of each instruction is used during the simulation.

To evaluate the accuracy of our model, we randomly selected 100 basic blocks from the applications in MiBench [32] benchmark suite. These basic blocks contained an average of around 11 instructions. We characterized the basic blocks using 10 measurements for each control delay estimation using the method described above. Finally, using 100 randomly generated input vectors for each basic block, we compared the estimated and measured effective delays of all instructions. To quantify the comparison, we use squared Hellinger distance as a measure of the

difference between instruction delay distributions. It takes values between 0 and 1 where smaller values indicate more accuracy. The squared Hellinger distance for two Gaussian distributions, $PN(\mu_1, \sigma_1)$ and $QN(\mu_2, \sigma_2)$, is given by,

$$H^2(P,Q) = 1 - \sqrt{\frac{2\sigma_1\sigma_2}{\sigma_1^2\sigma_2^2}} e^{-\frac{(\mu_1-\mu_2)^2}{4(\sigma_1^2+\sigma_2^2)}} \tag{5.2}$$

We found that the distance was smaller than 0.1 in 97.3% of the experiments with an average value of 0.027, illustrating the reliability of the model.

5.3.3 Error Rate Estimation

During the simulation when frequency is known, instruction delays estimated by the error model must be converted into basic block error rates so that the simulator can implement frequency tuning. Since instruction delays are estimated as Gaussian distributions, the probability of an instruction experiencing a timing error, referred to as its error probability, is given by,

$$P = \frac{1}{2}[1 + erf(\frac{\frac{1}{F} - \mu}{\sqrt{2\sigma^2}})], \tag{5.3}$$

where F is the working frequency, μ and σ are the mean and standard deviation of the instruction delay, and $erf()$ is the error function.

Now, consider a basic block containing n instructions with error probabilities $p_1, \ldots, p_n$. To estimate the error rate, let $I = (I_1, \ldots, I_n)$ be a set of Bernoulli random variables corresponding to the instructions such that $Pr(I_i = 1) = p_i$. Clearly, the number of errors can be expressed as the sum of instruction random variables, $n_e = \sum_{i=1}^{n} I_i$. Therefore, the expected number of errors can be calculated by summing instruction error probabilities and the expected error rate is given by,

$$r_e = \frac{1}{n}\sum_{i=1}^{n} p_i. \tag{5.4}$$

5.4 Simulation Framework

In this section we describe a framework for evaluation of various timing speculation strategies. The framework creates an instrumented version of the program that simultaneously implements the error model described in Section 5.3 to predict timing errors and estimate error rates as well as the timing speculation strategy described in Section 5.2 to perform dynamic frequency tuning. To perform the simulation, the instrumented program can be run on any machine that implements the instruction set architecture (ISA), resulting in very fast simulations. The operation of the instrumented program can be summarized in the following steps:

Before executing a basic block, error rate is read, frequency is set and pre-characterized instruction control delays corresponding to the executed incoming edge are loaded. During the execution, transition signatures of instructions are extracted and saved. Transition signatures, which will be explained shortly, are used to identify the functional paths activated by an instruction. After executing the basic block, activated functional paths are identified from transition signatures and used to estimate data delay of instructions. Then, effective delays of instructions are estimated using data and control delays and are used to find error probabilities of instructions using current frequency. Finally, the expected error rate of the basic block is computed, recorded and used to calculate speculation speedup.

5.4.1 Transition Signatures

Suppose that we are interested in finding the functional paths activated by the current instruction, i^c, with operands op_1^c and op_2^c and result r^c, which is executed immediately after the previous instruction, i^p, with operands op_1^p and op_2^p and result r^p. Below, we define a set of transition signatures derived from these architecturally visible parameters that uniquely identify the functional paths activated by i^c.

Register Access. In the register access stage, two sets of functional paths simply transfer the instruction operands from the register file. Activated functional paths are those used by

exactly one of the instructions and can be readily identified from the two transition signatures $ts_1^{r_a} = op_1^c \oplus op_1^p$ and $ts_2^{r_a} = op_2^c \oplus op_2^p$, where $\oplus$ is the exclusive-or operation.

Execute. In the execution stage of LEON3, all functional units share a single register for input operands. As a result, each functional unit performs its operation on the shared operands although the result of only one is registered. There- fore, the previous state of the active functional unit which, together with its current state, determines its sensitized paths, is the state induced by performing its operation on the operands of the previous instruction, even if it was a different type of instruction. To emulate this shared register scheme, we assume that the previous instruction i^p performs the same type of operation on its operands as the current instruction i^c. We will later show that this assumption can easily be implemented by inserting another instruction with source and destination registers of i^p and the opcode of i^c between the two instructions. Transition signatures are then defined based on the type of the instruction.

For logical instructions (`and`, `or`, etc.), each functional path is used if the corresponding bit in the result is 1 and the activated functional paths are those used by exactly one of the instructions, readily identified from the transition signature $ts^{exe} = r^c \oplus r^p$. Arithmetic instructions (`add`, `sub`, etc.) and memory access instructions (`ld` and `st`, which behave similar to `add`) have functional paths corresponding the carry chains of the addition they perform in the execute stage. Consider the multi-bit addition, $s = a + b$ in which $s = a \oplus b \oplus c$ where c_i denotes the input carry to ith bit. We can find carry bits by rewriting this as $c_i = a_i \oplus b_i \oplus s_i$. A carry chain from bit i to bit j ($i < j$) is used when $c_i = 0$, $c_{i+1} = \cdots = c_j = 1$, Therefore, we define two transition signatures $ts_1^{exe} = op_1^p \oplus op_2^p \oplus r^p$ and $ts_2^{exe} = op_1^c \oplus op_2^c \oplus r^c$. The activated functional paths are those used by exactly one of the additions.

Memory Access and Write-Back. The functional paths used in the memory access and write-back stages are determined by instruction results. Therefore, activated functional paths can be readily identified from the two transition signatures $ts^{mem} = ts^{wb} = r^c \oplus r^p$. Note that while the activation patterns are the same, functional paths of `ld` and `st` instructions in the memory stage are different from those of other instructions.

5.4.2 Source Code Instrumentation

In this section, we describe a source code instrumentation technique that extracts and stores the transition signatures and implements our timing speculation scheme. We explain the details using the example in Figure 5.2 which shows how each basic block in the CFG is instrumented.

Incoming Edge Basic Block. A basic block is added along each incoming edge. Lines 1-2 call the function `bb_in` with the parameter `bb_id` that identifies the basic block. This function reads the error rate and is responsible for setting the frequency for the basic block. It also loads the control delays corresponding to the incoming edge into a pre-specified array to be used for estimating instruction delays. In lines 3- 5, `r1`, `r2`, and `r3` represent any three regular registers not read or written in the basic block. These registers which are called working registers are copied into three Ancillary State Registers (ASRs). ASRs are a set of 16 registers provided by SPARC architecture for profiling and testing purposes.

Outgoing Edge Basic Block. A basic block is added along each outgoing edge. In lines 27-29, the working registers are restored to their original values. Lines 30-31 call the function `bb_out` with the basic block index which (i) identifies the activated functional paths using extracted transition signatures, (ii) uses them to estimate data delays, (iii) reads control delays and calculates instruction delays, and (iv) computes instruction error probabilities, the expected error rate, and speculation speedup.

Instrumented Basic Block. The original basic block is replaced by another basic block that identifies and stores the transition signatures of all its instructions. Lines 6-26 show the instructions that replace a `ld [%l1+%l2], %l3` instruction. We chose a load instruction to explain the instrumentation technique as it is the most complex type of instruction for transition signature extraction and shows all instrumentation code details . Lines 6-7 move operands of the previous instruction into `r1` and `r2`. Instrumentation codes of all instructions store the operands and the result of the the current instruction in `asr1`, `asr2`, and `asr3` (lines 16, 17, and 23).

Incoming Edge Basic Block

```
set %o0, bb_id
call  bb_in; nop
wr  %r1, %asr4
wr  %r2, %asr5
wr  %r3, %asr6
```

Instrumented Basic Block

```
...
rd  %asr1, %r1
rd  %asr2, %r2
xor %l1, %r1, %o0
call  save_sig; nop
xor %l2, %r2, %o0
call  save_sig; nop
add %r1, %r2, %r3
xor %r1, %r2, %o0
xor %r3, %o0, %o0
call  save_sig; nop
wr  %l1, %asr1
wr  %l2, %asr2
add %l1, %l2, %r3
xor %r1, %r2, %o0
xor %r3, %o0, %o0
call  save_sig; nop
ld  %l1, %l2, %l3
rd  %asr3, %r3
xor %l3, %r3, %o0
call  save_sig; nop
wr  %l3, %asr3
...
```

Outgoing Edge Basic Block

```
rd  %asr4, %r1
rd  %asr5, %r2
rd  %asr6, %r3
set %o0, bb_id
call  bb_out; nop
```

Figure 5.2. Example of Basic Block and `load` Instruction Instrumentation

Lines 8-11 extract and store transition signatures for the register access stage (i.e. ts_1^{ra} and ts_2^{ra}). Line 12 emulates the shared operand register scheme by performing the operation of the current instruction in the execute stage on the operands of the previous instruction. Lines 13-15 then identify and store one of the execution stage transition signatures ts_1^{exe}. Operands of the current instruction are stored in `asr1`, `asr2` in lines 16-17 before the second transition signature of the execute stage, ts_2^{exe}, is extracted in lines 18-21. Line 22 is 2 the original load instruction executed to ensure that behavior of the instrumented program does not change. Finally, the last pair of transition signatures, ts^{mem} and ts^{wb} are extracted and stored in lines 23-25 and the result of the instruction is stored in `asr3` in line 26. All arrays used for storing variables including frequencies, error rates, etc. are maintained as global variables as are functions `bb_in`, `save_sig`, and `bb_out` which are written in C and linked with the instrumented program. We implement the instrumentation technique in C++ and recompile the instrumented codes to produce executables. To perform the simulation, the instrumented program can be run on any machine that implements the ISA.

5.5 Experimental Results

In this section, we evaluate the timing speculation strategies proposed in this chapter. Note that experiments for validating our error model were presented in Section 5.3.

5.5.1 Experimental Setup

Synthesis and Static Timing Analysis. The design was synthesized on the 45nm TSMC technology targeting the typical-case corner (TT,$0.9V$,$25°C$). We set the frequency of our baseline system to $718MHz$ using SSTA at ($0.81V$,$25°C$), guardbanding for a 10% voltage droop. For a fair comparison, we assume a fixed supply voltage of $0.81V$ for the TS systems studied. This ensures that performance improvements accurately reflect the ability of the speculation strategies to track data variations.

Error Detection and Recovery. We assume that error detection and recovery circuits

guarantee correct execution. We adopt a conservative error recovery mechanism known as instruction replay at half-frequency. When a timing error is detected, the frequency is halved, the pipeline is flushed, and the errant instruction is reissued, resulting in a 24 cycle recovery penalty for our 6-stage pipeline.

Dynamic Frequency Tuning. Similar to a LEON3-based 45nm resilient Intel research processor [7], we consider a clock generator that uses phase-locked loop (PLL) based on the one in the 45nm Intel Core i7 microprocessor [40] which provides fine-grain frequency tuning in less than 2 cycles, which we consider as the penalty of each frequency change.

Power and Area Overheads. Implementing these adaptive clocking and error detection and recovery schemes on a processor similar to LEON3 has been shown to incur a power and area overhead of less than 0.9% and 3.8%, respectively [7].

5.5.2 Speculation Strategies

Before we can evaluate the performance of our speculation scheme, we need to tune the design parameters described in Section 5.2. We selected 12 applications from MiBench benchmark suite for our study. We used the small datasets of benchmark applications for tuning and the large datasets for performance evaluation. In all experiments, throughput values have been normalized to the throughput of the error-free guardbanded design.

1. *Tuning N_B:* This parameter which specifies the number of basic blocks for which timing speculation is performed determines the number of entries in the timing speculation table. For this evaluation, the other parameter is set to its default value, $N_S = 0$. Figure 5.3 shows normalized throughput as N_B is increased by powers of two from 2 to 128. From the figure, it can be seen that maximum throughput is achieved for $N_B = 32$ or $N_B = 64$ depending on the application. A larger N_B increases throughput by improving the accuracy of frequency predictions. However, it also increases the number of frequency changes which incur a 2-cycle penalty each and limit throughput increase. Based on these results, we consider $N_B = 32$ or $N_B = 64$ for now.

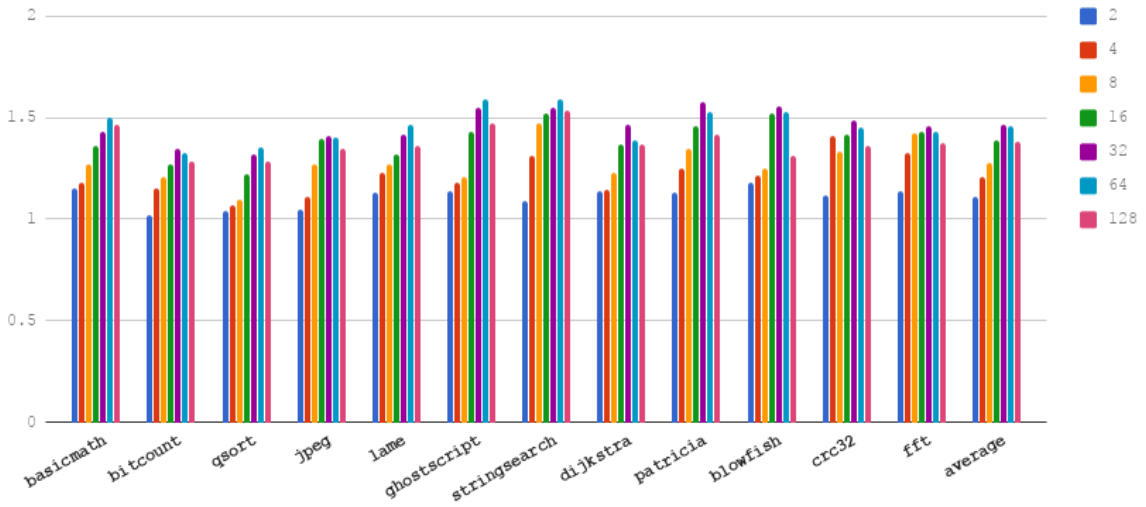
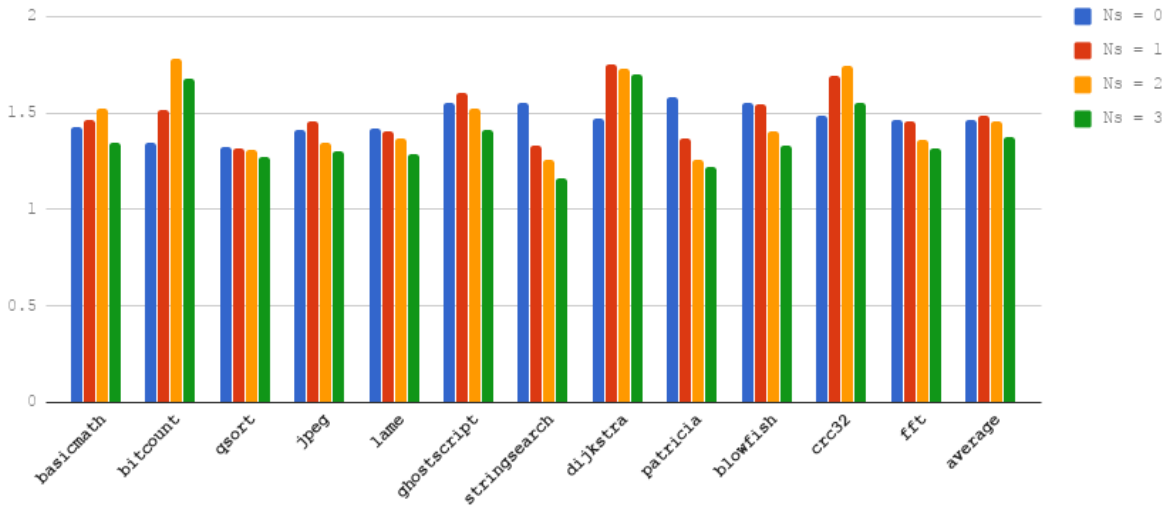


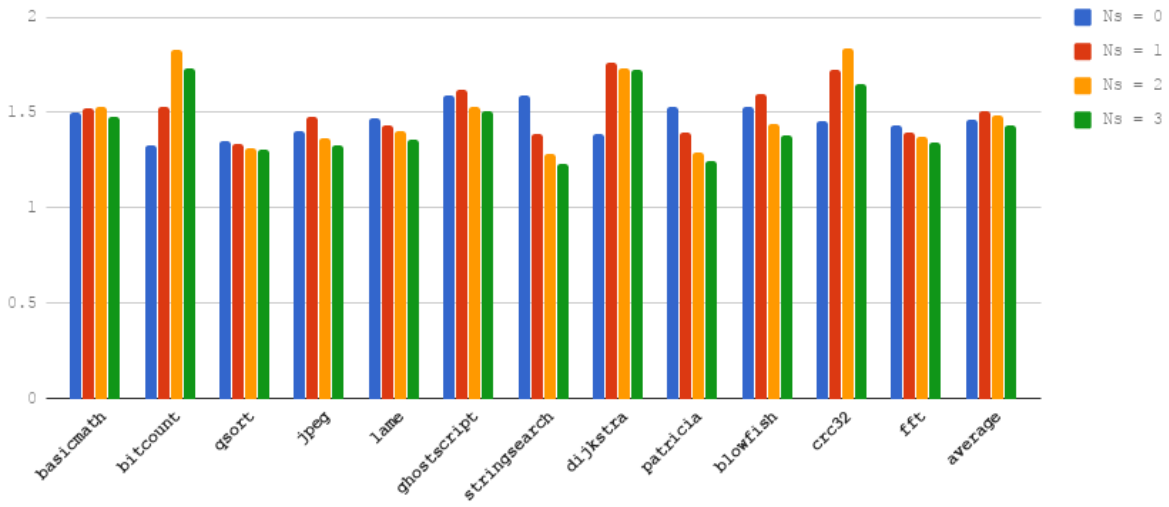
Figure 5.3. Tuning N_B. Normalized Throughput as N_B Is Increased from 2 to 128

2. *Tuning N_S:* This parameter specifies how many times a basic block skips frequency prediction and reuses the previous frequency before a new prediction is made. Since a larger N_S reduces the number of frequency changes, it could allow for a larger N_B as well. We then consider both $N_B = 32$ or $N_B = 64$ for this experiment. Figure 5.4 shows normalized throughput as NS is increased from 0 to 3. From the figure, it can be seen that the best value for N_S is highly dependent on the application. For example, the performance of `patricia` and `stringsearch` is significantly better for $N_S = 0$. This indicates a highly variable timing behavior which requires more frequent predictions to achieve high accuracy. In contrast, `bitcount`, `crc32`, and `dijkstra` exhibit highly predictable timing behaviors which allows for less predictions. It is interesting to note that when N_S is not 0, $N_B = 64$ performs better than $N_B = 32$ by limiting the number of frequency changes. Based on these experiments, we select the first two parameters, $N_B = 64$ and $N_S = 1$, for the next experiments.

3. *Performance Evaluation:* Using the tuned parameters, we evaluated the performance of our speculation scheme. Figure 5.5 shows the results of our experiments for three TS systems. Our speculation scheme is denoted by **Proposed**. The conventional approach, denoted by



(a) $N_B = 32$



(b) $N_B = 64$

Figure 5.4. Tuning N_S. Normalized Throughput as N_S Is Increased from 0 to 3

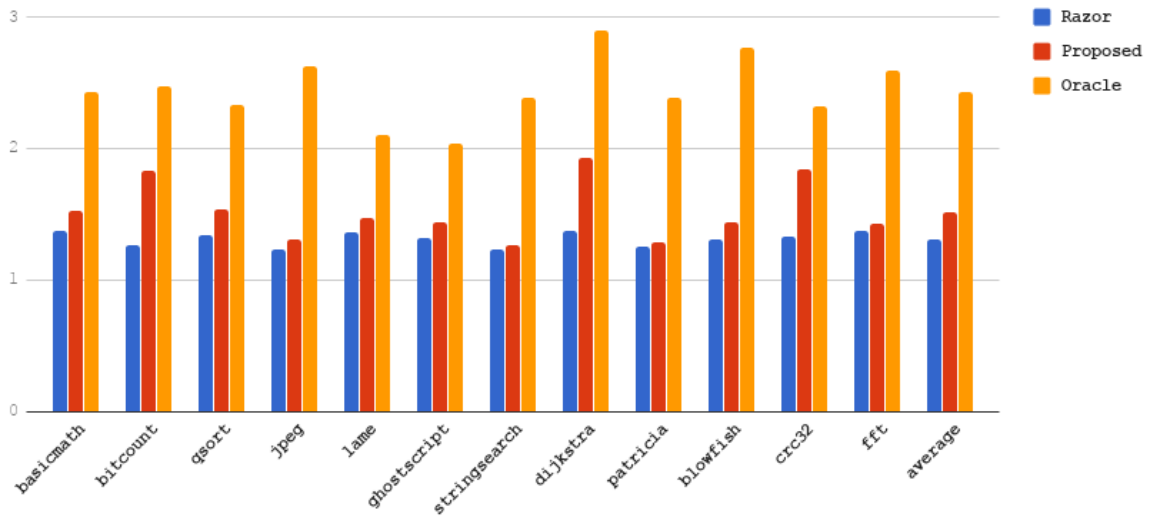


Figure 5.5. Normalized Throughput of Our Timing Speculation Scheme

Razor in the figure, periodically records the global error rate and compares it to a threshold value. For a more conservative comparison, we chose the sampling frequency and error rate threshold values that maximized the performance on average. The **Oracle** strategy represents an ideal system that precisely predicts all instruction delays and instantly sets the frequency to the largest value that causes no timing errors for each basic block. We obtained the throughput of this system by simply summing the effective delays of all executed basic blocks. Effective delay of a basic block is the maximum of its instruction delays.

The figure shows that our proposed scheme consistently outperforms the conventional approach. On average, **Oracle** improves throughput of the guardbanded design by 143%. But the conventional approach achieves a 31.1% improvement, realizing less than 22% of the potential gains. While the strategies introduced in this chapter achieves a throughput improvement of 50.9%, more than 64% of the potential gains remains untapped. This points to the significant opportunities for improving system performance with timing speculation.

Chapter 5 is based upon the work supported by the National Science Foundations Variability Expedition in Computing under Award No. 1029783. Any opinions, findings, and conclusions

or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Chapter 5, in full, is a reprint of Omid Assare and Rajesh Gupta, “Strategies for Optimal Operating Point Selection in Timing-Speculative Processors,” *IEEE International Conference on Computer Design (ICCD)*, 2016. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Summary and Conclusions

6.1 Cross-Layer Performance Analysis

In this dissertation, we described two dynamic timing analysis tools to efficiently analyze performance of TS processors. Our microarchitecture-level DTA tool relies on a high-level process-variation-aware timing model based grouping functionally similar timing paths and modeling their timing behavior as a function of their specific operation. Our architecture-level DTA tool accurately calculates DTS by simultaneously taking into account the effects of process variation, instruction sequence and operands, datapath configuration, and error recovery scheme. To facilitate the analysis, we developed an instruction-level error model that estimates the likelihood that an instruction experiences a timing error, capturing the uncertainty caused by process and data variations and the dynamic effect of timing errors in the form of inter-instruction correlations caused by the error recovery scheme used by the TS processor. Based on our instruction error model, we proposed a statistical approach for estimating error rate of programs using statistical limit theorems and established bounds on the approximation error using Stein's method.

Conclusion: Modeling interdependence of circuit and architecture is necessary for accurate performance analysis of TS processors, but a cross-layer approach that considers hardware and software at the same time provides opportunities for improving the efficiency of the analysis as well.

6.2 Impact of Software on Performance

We also presented results of using our tools to analyze performance of TS processors with an emphasis on characterizing the role of software. We introduced and characterized four aspects of how the error behavior is affected by the running software. We proposed inter- and intra-program variation as measures of error rate variability in different programs and among instructions of a program. We also demonstrated that input data can cause performance variations comparable to other sources of variability such as process variation. Finally, an analysis of the physical location of errors in hardware was presented. We identified the regions in which most errors occur and how different programs change the distribution of errors among them.

Conclusion: Not all applications benefit from running on a TS processor. Applications vary significantly in frequency, sensitivity to data and process variations, and even physical location of the timing errors they experience. Application-specific analysis is necessary for accurate evaluation of TS processors and should be used to inform design decisions and assess the suitability of the application for timing speculation. The combination of program and input data can change performance of a TS processor by as much as 25%.

6.3 Timing Speculation Policy

Using the DTA tools described in this dissertation, we studied the opportunities provided by timing speculation for improving system performance and found that current methods realize only a fraction of the potential speedup. We proposed a timing speculation scheme that attains more performance gains by improving the quality of frequency predictions. Our timing speculation method limits the scope of speculation both in time and space. Spatially, we argued that frequency tuning should be directed by software and performed at the basic block level where instruction sequence is fixed and predictions are likely to be more accurate. Temporally, we showed that the most recent history of errors is a better predictor of timing behavior than a long-term average. Finally, we proposed to dynamically tune the frequency using an optimization

algorithm instead of a controller. Together these strategies achieved a throughput improvement of 50.9%.

Conclusion: Current dynamic frequency or voltage scaling schemes leave most of the potential benefits of timing speculation untapped. A speculation policy directed by the program control flow that selects the operating point locally, i.e., for each basic block separately and based on its own timing error history, can achieve a larger portion of the potential benefits than current methods that perform the frequency or voltage scaling periodically and based on the global history of timing errors.

Bibliography

[1] ARRATIA, R., GOLDSTEIN, L., AND GORDON, L. Poisson approximation and the chen-stein method. *Statist. Sci. 5*, 4 (11 1990), 403–424.

[2] ASSARE, O., AND GUPTA, R. Timing analysis of erroneous systems. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2014 International Conference on* (Oct 2014), pp. 1–10.

[3] ASSARE, O., AND GUPTA, R. Strategies for optimal operating point selection in timing speculative processors. In *2016 IEEE 34th International Conference on Computer Design (ICCD)* (Oct 2016), pp. 584–591.

[4] ASSARE, O., AND RAJESH, G. Accurate estimation of program error rate for timing-speculative processors. In *The 56th Annual Design Automation Conference* (2019), DAC '19, ACM.

[5] BLAAUW, D., CHOPRA, K., SRIVASTAVA, A., AND SCHEFFER, L. Statistical timing analysis: From basic principles to state of the art. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 27*, 4 (April 2008), 589–607.

[6] BOWMAN, K., TSCHANZ, J., KIM, N. S., LEE, J., WILKERSON, C., LU, S., KARNIK, T., AND DE, V. Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance. *Solid-State Circuits, IEEE Journal of 44*, 1 (Jan 2009), 49–63.

[7] BOWMAN, K., TSCHANZ, J., LU, S., ASERON, P., KHELLAH, M., RAYCHOWDHURY, A., GEUSKENS, B., TOKUNAGA, C., WILKERSON, C., KARNIK, T., AND DE, V. A 45 nm resilient microprocessor core for dynamic variation tolerance. *Solid-State Circuits, IEEE Journal of 46*, 1 (Jan 2011), 194–208.

[8] BOWMAN, K. A., RAINA, S., BRIDGES, J. T., YINGLING, D. J., NGUYEN, H. H., APPEL, B. R., KOLLA, Y. N., JEONG, J., ATALLAH, F. I., AND HANSQUINE, D. W. A 16 nm all-digital auto-calibrating adaptive clock distribution for supply voltage droop tolerance across a wide operating range. *IEEE Journal of Solid-State Circuits 51*, 1 (Jan 2016), 8–17.

[9] BULL, D., DAS, S., SHIVASHANKAR, K., DASIKA, G., FLAUTNER, K., AND BLAAUW, D. A power-efficient 32 bit arm processor using timing-error detection and correction for

transient-error tolerance and adaptation to pvt variation. *Solid-State Circuits, IEEE Journal of 46*, 1 (Jan 2011), 18–31.

[10] BURD, T., PERING, T., STRATAKOS, A., AND BRODERSEN, R. A dynamic voltage scaled microprocessor system. In *Solid-State Circuits Conference, 2000. Digest of Technical Papers. ISSCC. 2000 IEEE International* (Feb 2000), pp. 294–295.

[11] CADAR, C., DUNBAR, D., AND ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 209–224.

[12] CHAE, K., MUKHOPADHYAY, S., LEE, C.-H., AND LASKAR, J. A dynamic timing control technique utilizing time borrowing and clock stretching. In *Custom Integrated Circuits Conference (CICC), 2010 IEEE* (Sept 2010), pp. 1–4.

[13] CHEN, L. H. Y. Poisson approximation for dependent trials. *Ann. Probab. 3*, 3 (06 1975), 534–545.

[14] CHERUPALLI, H., KUMAR, R., AND SARTORI, J. Exploiting dynamic timing slack for energy efficiency in ultra-low-power embedded systems. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (June 2016), pp. 671–681.

[15] CHERUPALLI, H., AND SARTORI, J. Scalable n-worst algorithms for dynamic timing and activity analysis. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (Nov 2017), pp. 585–592.

[16] CHOUDHURY, M., CHANDRA, V., AITKEN, R., AND MOHANRAM, K. Time-borrowing circuit designs and hardware prototyping for timing error resilience. *Computers, IEEE Transactions on 63*, 2 (Feb 2014), 497–509.

[17] CONSTANTIN, J., WANG, L., KARAKONSTANTIS, G., CHATTOPADHYAY, A., AND BURG, A. Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)* (March 2015), pp. 381–386.

[18] DAS, S., ROBERTS, D., LEE, S., PANT, S., BLAAUW, D., AUSTIN, T., FLAUTNER, K., AND MUDGE, T. A self-tuning dvs processor using delay-error detection and correction. *Solid-State Circuits, IEEE Journal of 41*, 4 (April 2006), 792–804.

[19] DAS, S., TOKUNAGA, C., PANT, S., MA, W.-H., KALAISELVAN, S., LAI, K., BULL, D., AND BLAAUW, D. Razorii: In situ error detection and correction for pvt and ser tolerance. *Solid-State Circuits, IEEE Journal of 44*, 1 (Jan 2009), 32–48.

[20] DASKALAKIS, C., DIAKONIKOLAS, I., AND SERVEDIO, R. A. Learning poisson binomial distributions. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2012), STOC '12, ACM, pp. 709–728.

[21] DIGHE, S., VANGAL, S., ASERON, P., KUMAR, S., JACOB, T., BOWMAN, K., HOWARD, J., TSCHANZ, J., ERRAGUNTLA, V., BORKAR, N., DE, V., AND BORKAR, S. 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 (Jan 2011), 184–193.

[22] DRAKE, A., SENGER, R., DEOGUN, H., CARPENTER, G., GHIASI, S., NGUYEN, T., JAMES, N., FLOYD, M., AND POKALA, V. A distributed critical-path timing monitor for a 65nm high-performance microprocessor. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International* (Feb 2007), pp. 398–399.

[23] ERNST, D., KIM, N. S., DAS, S., PANT, S., RAO, R., PHAM, T., ZIESLER, C., BLAAUW, D., AUSTIN, T., FLAUTNER, K., AND MUDGE, T. Razor: a low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on* (Dec 2003), pp. 7–18.

[24] FAN, Y., JIA, T., GU, J., CAMPANONI, S., AND JOSEPH, R. Compiler-guided instruction-level clock scheduling for timing speculative processors. In *Proceedings of the 55th Annual Design Automation Conference* (New York, NY, USA, 2018), DAC '18, ACM, pp. 40:1–40:6.

[25] FLOYD, M., ALLEN-WARE, M., RAJAMANI, K., BROCK, B., LEFURGY, C., DRAKE, A., PESANTEZ, L., GLOEKLER, T., TIERNO, J., BOSE, P., AND BUYUKTOSUNOGLU, A. Introducing the adaptive energy management features of the power7 chip. *Micro, IEEE 31*, 2 (March 2011), 60–75.

[26] FOJTIK, M., FICK, D., KIM, Y., PINCKNEY, N., HARRIS, D., BLAAUW, D., AND SYLVESTER, D. Bubble razor: Eliminating timing margins in an arm cortex-m3 processor in 45 nm cmos using architecturally independent error detection and correction. *Solid-State Circuits, IEEE Journal of 48*, 1 (Jan 2013), 66–81.

[27] GAISLER, A. Leon3 processor.

[28] GAISLER, A. Tsim erc32/leon simulator.

[29] GIBBS, A. L., AND SU, F. E. On choosing and bounding probability metrics. *International statistical review 70*, 3 (2002), 419–435.

[30] GRESKAMP, B., WAN, L., KARPUZCU, U. R., COOK, J. J., TORRELLAS, J., CHEN, D., AND ZILLES, C. Blueshift: Designing processors for timing speculation from the ground up. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture* (Feb 2009), pp. 213–224.

[31] GUPTA, P., AGARWAL, Y., DOLECEK, L., DUTT, N., GUPTA, R., KUMAR, R., MITRA, S., NICOLAU, A., ROSING, T., SRIVASTAVA, M., SWANSON, S., AND SYLVESTER, D. Underdesigned and opportunistic computing in presence of hardware variability. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 32*, 1 (Jan 2013), 8–23.

[32] GUTHAUS, M., RINGENBERG, J., ERNST, D., AUSTIN, T., MUDGE, T., AND BROWN, R. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on* (Dec 2001), pp. 3–14.

[33] HOANG, G., FINDLER, R. B., AND JOSEPH, R. Exploring circuit timing-aware language and compilation. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 345–356.

[34] HONG, Y. On computing the distribution function for the poisson binomial distribution. *Comp. Stat. Data Anal. 59* (Mar. 2013), 41–51.

[35] INFRASTRUCTURE, L. C. libfuzzer a library for coverage-guided fuzz testing.

[36] INTERNATIONAL, S. *The SPARC architecture manual: Version 8*. Prentice Hall, 1992.

[37] JIA, T., JOSEPH, R., AND JIE GU. Greybox design methodology: A program driven hardware co-optimization with ultra-dynamic clock management. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)* (June 2017), pp. 1–6.

[38] JIAO, X., RAHIMI, A., JIANG, Y., WANG, J., FATEMI, H., DE GYVEZ, J. P., AND GUPTA, R. K. Clim: A cross-level workload-aware timing error prediction model for functional units. *IEEE Transactions on Computers 67*, 6 (June 2018), 771–783.

[39] KLEEBERGER, V. B., MAIER, P. R., AND SCHLICHTMANN, U. Workload- and instruction-aware timing analysis: The missing link between technology and system-level resilience. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference* (New York, NY, USA, 2014), DAC '14, ACM, pp. 49:1–49:6.

[40] KURD, N., MOSALIKANTI, P., NEIDENGARD, M., DOUGLAS, J., AND KUMAR, R. Next generation intel core micro-architecture (nehalem) clocking. *IEEE Journal of Solid-State Circuits 44*, 4 (April 2009), 1121–1129.

[41] KWON, I., KIM, S., FICK, D., KIM, M., CHEN, Y.-P., AND SYLVESTER, D. Razor-lite: A light-weight register for error detection by observing virtual supply rails. *Solid-State Circuits, IEEE Journal of 49*, 9 (Sept 2014), 2054–2066.

[42] LATTNER, C., AND ADVE, V. Llvm: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (March 2004), pp. 75–86.

[43] LE CAM, L. An approximation theorem for the Poisson binomial distribution. *Pac. J. Math. 10* (1960), 1181–1197.

[44] NADARAJAH, S., AND KOTZ, S. Exact distribution of the max/min of two gaussian random variables. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 16*, 2 (Feb 2008), 210–212.

[45] NAKAI, M., AKUI, S., SENO, K., MEGURO, T., SEKI, T., KONDO, T., HASHIGUCHI, A., KAWAHARA, H., KUMANO, K., AND SHIMURA, M. Dynamic voltage and frequency management for a low-power embedded microprocessor. *Solid-State Circuits, IEEE Journal of 40*, 1 (Jan 2005), 28–35.

[46] NDAI, P., RAFIQUE, N., THOTTETHODI, M., GHOSH, S., BHUNIA, S., AND ROY, K. Trifecta: A nonspeculative scheme to exploit common, data-dependent subcritical paths. *Very Large Scale Integration (VLSI) Systems, IEEE Trans. on 18*, 1 (Jan 2010), 53–65.

[47] RAHIMI, A., BENINI, L., AND GUPTA, R. K. Analysis of instruction-level vulnerability to dynamic voltage and temperature variations. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012* (March 2012), pp. 1102–1105.

[48] RAHIMI, A., BENINI, L., AND GUPTA, R. K. Application-adaptive guardbanding to mitigate static and dynamic variability. *Computers, IEEE Transactions on* (2013).

[49] RAHIMI, A., BENINI, L., AND GUPTA, R. K. Hierarchically focused guardbanding: An adaptive approach to mitigate pvt variations and aging. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013* (March 2013), pp. 1695–1700.

[50] REHMAN, S., SHAFIQUE, M., KRIEBEL, F., AND HENKEL, J. Reliable software for unreliable hardware: Embedded code generation aiming at reliability. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2011 Proceedings of the 9th International Conference on* (Oct 2011), pp. 237–246.

[51] ROY, S., AND CHAKRABORTY, K. Predicting timing violations through instruction-level path sensitization analysis. In *Proceedings of the 49th Annual Design Automation Conference* (New York, NY, USA, 2012), DAC '12, ACM, pp. 1074–1081.

[52] SAMANDARI-RAD, J., GUTHAUS, M., AND HUGHEY, R. Var-tx: A variability-aware sram model for predicting the optimum architecture to achieve minimum access-time for yield enhancement in nano-scaled cmos. In *Thirteenth International Symposium on Quality Electronic Design (ISQED)* (March 2012), pp. 506–515.

[53] SARTORI, J., AND KUMAR, R. Architecting processors to allow voltage/reliability trade-offs. In *Compilers, Architectures and Synthesis for Embedded Systems (CASES), 2011 Proceedings of the 14th International Conference on* (Oct 2011), pp. 115–124.

[54] SINHA, D., ZHOU, H., AND SHENOY, N. V. Advances in computation of the maximum of a set of gaussian random variables. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 26*, 8 (Aug 2007), 1522–1533.

[55] SPROULL, R., SUTHERLAND, I., AND MOLNAR, C. The counterflow pipeline processor architecture. *Design Test of Computers, IEEE 11*, 3 (Autumn 1994), 48–.

[56] STACKHOUSE, B., BHIMJI, S., BOSTAK, C., BRADLEY, D., CHERKAUER, B., DESAI, J., FRANCOM, E., GOWAN, M., GRONOWSKI, P., KRUEGER, D., MORGANTI, C., AND

TROYER, S. A 65 nm 2-billion transistor quad-core itanium processor. *Solid-State Circuits, IEEE Journal of 44*, 1 (Jan 2009), 18–31.

[57] STEIN, C. A bound for the error in the normal approximation to the distribution of a sum of dependent random variables. In *Proceedings of the Sixth Berkeley Symposium on Mathematical Statistics and Probability, Volume 2: Probability Theory* (1972), University of California Press, pp. 583–602.

[58] TARJAN, R. Depth first search and linear graph algorithms. *SIAM Journal on Computing* (1972).

[59] TSCHANZ, J., BOWMAN, K., WALSTRA, S., AGOSTINELLI, M., KARNIK, T., AND DE, V. Tunable replica circuits and adaptive voltage-frequency techniques for dynamic voltage, temperature, and aging variation tolerance. In *VLSI Circuits, 2009 Symposium on* (June 2009), pp. 112–113.

[60] TSCHANZ, J., KIM, N. S., DIGHE, S., HOWARD, J., RUHL, G., VANGAL, S., NARENDRA, S., HOSKOTE, Y., WILSON, H., LAM, C., SHUMAN, M., TOKUNAGA, C., SOMASEKHAR, D., TANG, S., FINAN, D., KARNIK, T., BORKAR, N., KURD, N., AND DE, V. 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* (Feb 2007), pp. 292–604.

[61] TSMC. Tsmc 45nm standard cell library release note, tcbn45gsbwp, version 120a.

[62] TZIANTZIOULIS, G., GOK, A. M., FAISAL, S. M., HARDAVELLAS, N., OGRENCI-MEMIK, S., AND PARTHASARATHY, S. b-hive: A bit-level history-based error model with value correlation for voltage-scaled integer and floating point units. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)* (June 2015), pp. 1–6.

[63] WANNER, L., ELMALAKI, S., LAI, L., GUPTA, P., AND SRIVASTAVA, M. Varemu: An emulation testbed for variability-aware software. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on* (Sept 2013), pp. 1–10.

[64] XIN, J., AND JOSEPH, R. Identifying and predicting timing-critical instructions to boost timing speculation. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2011), MICRO-44, ACM, pp. 128–139.

[65] ZHANG, Y., KHAYATZADEH, M., YANG, K., SALIGANE, M., PINCKNEY, N., ALIOTO, M., BLAAUW, D., AND SYLVESTER, D. irazor: Current-based error detection and correction scheme for pvt variation in 40-nm arm cortex-r4 processor. *IEEE Journal of Solid-State Circuits 53*, 2 (Feb 2018), 619–631.