Accurate Estimation of Program Error Rate for Timing-Speculative Processors

Omid Assare
University of California, San Diego
Department of Computer Science and Engineering
La Jolla, CA, USA
omid@ucsd.edu

Rajesh Gupta
University of California, San Diego
Department of Computer Science and Engineering
La Jolla, CA, USA
rgupta@ucsd.edu

ABSTRACT

We propose a framework that estimates the error rate experienced by an application as it runs on a timing-speculative processor. The framework uses an instruction error model that is comparable in accuracy to low-level simulations—as it considers the effects of operand values, preceding instructions, datapath configuration, and error correction scheme, as well as process variation, including its spatial correlation property—and yet efficient enough to allow its application in Monte Carlo experiments to characterize large program input datasets. We then use statistical limit theorems to estimate program error rate and quantify the effect of inter-instruction correlations.

1 INTRODUCTION

Timing-speculative (TS) processors [11] allow increasing the frequency or decreasing the voltage beyond the limits determined by static timing analysis (STA), thereby removing pessimistic safety margins that conventional processors require to prevent timing errors. 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 safety margins must outweigh the cost of detecting and correcting timing errors. While these costs have been an obstacle to adoption of timing speculation in commercial designs [24], they have been steadily decreasing over the past decade—from 44 additional transistors per flip-flop for detection and dozens of clock cycles for correction in the first Razor design [11], to only three additional transistors and as few as a single clock cycle in the latest version [24].

Motivating this work is another contributing factor hindering broader diffusion of TS processors: complexity of their timing analysis. Since each timing error incurs a penalty for error correction, 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 [12]. 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 correction 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 paper, we introduce a framework to estimate error rate of programs running on in-order TS processors. We make the following contributions:

(1) We develop a dynamic timing analysis (DTA) tool that accurately calculates DTS in Section 3, and an instruction error model that predicts the likelihood that the instruction will experience a timing error in Section 4. 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 correction scheme.
(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 5. We then use Stein's method [22] to establish bounds on the approximation error, including the inaccuracy caused by inter-instruction correlations.

2 RELATED WORK

Graph-Based DTA. Authors in [7] 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 [6] 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

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317758

the dynamic effect of timing errors on DTS. Our framework does not suffer from the long simulation times of other path-based techniques because it performs 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 [12] 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 [18], 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 [8, 15]. In addition to long simulation times necessary, because they rely on the simulator to perform DTA, they cannot use nondeterministic 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 [7] can be extended to do so.

3 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 3.1. An ordered set of gates in N is a *path* if

(1) the first gate is the only endpoint in the set,
(2) each gate, except the first one, is connected to the previous gate in the set, and
(3) the last gate is connected to an endpoint.

Definition 3.2. We say a gate is *activated* in a particular clock cycle if, were the clock period sufficiently long, the net connected to its output would eventually change its value.

Definition 3.3. 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) and computes DTS of a specified pipeline stage at a given 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 1 shows how the inputs to the algorithm are generated. 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

Table 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, $s = 0, 1, \ldots, S(N) - 1$
$E(N, s)$	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

Algorithm 1: Dynamic Timing Slack Algorithm

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

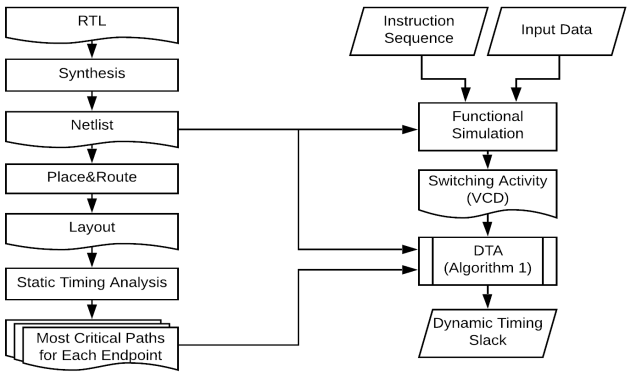

Figure 1: DTA Flowchart.

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 [21] that performs a sequence of pairwise minimum operations in an order that would minimize the approximation error.

4 INSTRUCTION ERROR MODEL

The DTA tool described in Section 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 2 uses the DTS of pipeline stages to calculate DTS of an instruction executed on an in-order processor. Figure 2 shows the three components of the instruction DTS estimation flow adopted from [3].

Algorithm 2: Instruction Dynamic Timing Slack

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

Control Network DTS Characterization. We start by forming a control flow graph (CFG) for the program. Each time a basic block is executed on an in-order processor, the control network, like the logic for fetching and decoding instructions, performs the same task. Therefore, in most cases, the same set of timing paths in the control network are activated every time. We divide the endpoints of the processor into two sets—the set of *data endpoints* includes endpoints that hold the operands and results of instructions, including condition codes and intermediate results like load/store addresses, and the set of *control endpoints* which includes the rest of the endpoints. We apply Algorithm 2 on the control endpoints as we execute each basic block separately, and record DTS of all instructions. Because the instructions of two basic blocks can share the pipeline at times, we characterize the control DTS of instructions along all incoming edges to the basic block.

Datapath DTS Characterization. Estimating DTS of the datapath is much simpler than the control network. This provides the opportunity to improve simulation time by using a higher-level timing model instead of the gate-level analysis we performed on the control endpoints. We use the timing model proposed in [2], which is trained by applying Algorithm 1 to measure DTS of data endpoints as the processor runs special instruction sequences and input data that selectively activate specific timing paths.

Datapath Activity Characterization. Because our datapath DTS model only requires values of architecturally visible registers, we can perform the analysis at the architecture level. To further improve efficiency, instead of a simulator, we implement the model by instrumenting the program with native instructions. We used the same technique as in [3], implemented on LLVM [19] platform.

4.1 Instruction Error Probability

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

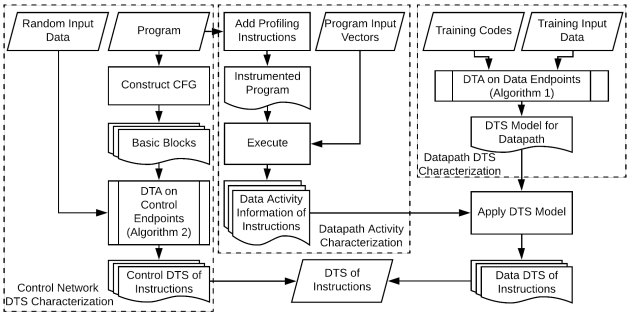

Figure 2: Instruction DTS Estimation Flow.

Effect of Error Correction Scheme. The error correction 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 [9] or flush the pipeline to resolve any complex bypass register issues [4]. 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 correction 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 correction 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*. We proceed by describing a procedure for computing the *marginal* error probabilities.

4.2 Marginal Error Probability

Problem Formulation. 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 jth 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 $j = 1, \ldots, d_i$.

*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.

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 $k = 2, \ldots, n_i$.

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

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,

$$p^{in}_i = \sum_{j=1}^{d_i} p^a_{ij} p^{out}_{t_i(j)}, \quad (2)$$

where $t_i(j)$ is the index of the basic block connected to the tail of the jth incoming edge to B_i. For cycles in the CFG, we construct a system of linear equations by writing Equations (1) and (2) for all the basic blocks in the cycle, in which edge activation probabilities form the coefficient matrix and instruction error probabilities are the unknowns. Using Tarjan's algorithm [23] to identify strongly connected components of the CFG and their topological ordering, we write and solve a system of linear equations for each component.

5 PROGRAM ERROR RATE

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. 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^{in}_i be a Bernoulli random variable representing the instruction executed just before entering B_i and let $p^{in}_i = Pr(I^{in}_i = 1)$. 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^{in}_i and p^{in}_i for $k = 1$[†]. 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 $\overline{N}_E$.

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 [17]. Consequently, approximation techniques targeting various distributions such as normal and Poisson have been widely developed and used [10]. The law of rare events provides the intuition (proof in [20]) 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 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 [22]. Chen [5] 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 5.1. *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*

$$b_1 = \sum_{\alpha \in I} \sum_{\beta \in B_\alpha} p_\alpha p_\beta, \quad (3)$$

$$b_2 = \sum_{\alpha \in I} \sum_{\alpha \neq \beta \in B_\alpha} p_{\alpha\beta}, \quad where \quad p_{\alpha\beta} = E[X_\alpha X_\beta]. \quad (4)$$

Then,

$$d_{TV}(W, Z) \leq min\left\{1, \lambda^{-1}\right\}(b_1 + b_2), \quad (5)$$

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 (6).

$$N_E = \sum_{i=1}^{m} \sum_{k=1}^{n_i} e_i I_{i_k} = \sum_{i=1}^{m} \sum_{k=1}^{n_i} \sum_{j=1}^{e_i} I_{i_k}. \quad (6)$$

[†]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 correction mechanism (see Section 4.1) 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.

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.

$$b_1 = \sum_{i=1}^{m}\sum_{j=1}^{e_i}(p_i^{in}p_{i_1} + \sum_{k=2}^{n_i} p_{i_{k-1}}p_{i_k}) \quad (7)$$

$$b_2 = \sum_{i=1}^{m}\sum_{j=1}^{e_i}(p_i^{in}p_{i_1}^{e} + \sum_{k=2}^{n_i} p_{i_{k-1}}p_{i_k}^{e}) \quad (8)$$

$$d_K(N_E, \overline{N}_E) \leq \frac{b_1 + b_2}{\lambda} \quad (9)$$

$$\lambda = \sum_{i=1}^{m}\sum_{k=1}^{n_i}\sum_{j=1}^{e_i} p_{i_k}, \quad (10)$$

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 (5) with the Kolmogorov metric because $d_K \leq d_{TV}$ (proof in [14]). 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).

To approximate the distribution of λ in Equation (10), 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 5.2 provides a simple description of the results of Stein's method as applicable to our problem.

Theorem 5.2. *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*

$$b_1 = \frac{D^2}{\sigma^3}\sum_{i=1}^{n} E|X_i|^3 \quad (11)$$

$$b_2 = \frac{\sqrt{28}D^{\frac{3}{2}}}{\sqrt{\pi}\sigma^2}\sqrt{\sum_{i=1}^{n} E[X_i^4]}. \quad (12)$$

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

$$d_K(W, Z) \leq (\tfrac{2}{\pi})^{\frac{1}{4}}(b_1 + b_2), \quad (13)$$

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 (11) and (12). 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 (14).

$$\overline{N}_E(k) = \int_0^{\infty} e^{-\lambda(x)} \sum_{i=0}^{\lfloor k \rfloor} \frac{\lambda^i(x)}{i!} dx \quad (14)$$

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.

6 EXPERIMENTAL RESULTS

6.1 Experimental Setup

We consider the integer unit of LEON3, an open-source in-order processor core that implements the SPARC V8 architecture [13].

Synthesis and Static Timing Analysis. 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).

Error Detection and Correction. We assume that error detection and correction circuits guarantee correct execution. Similar to a LEON3-based 45nm resilient Intel research processor [4], we adopt a conservative error correction mechanism known as instruction replay at half-frequency. When a timing error is detected, frequency is halved, the pipeline is flushed, and the errant instruction is reissued, resulting in a 24-cycle penalty for our 6-stage pipeline.

Power and Area Overheads. The error detection and correction schemes for our design have been shown to incur a power and area overhead of less than 0.9% and 3.8%, respectively [4].

6.2 Framework Runtime

We selected 12 benchmark programs, two from each of the six categories of MiBench [16]. 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 2.

6.3 Error Rate Distributions

Figure 3 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 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

Table 2: Results, Performance, and Accuracy of Our Framework

Benchmarks	Program Size		Runtime (s)			Program Error Rate (%)		Approximation Error	
	Instructions	Basic Blocks	Training	Simulation	Total	Mean	SD	$d_K(\lambda, \bar{\lambda})$	$d_K(R_E, \overline{R}_E)$
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				

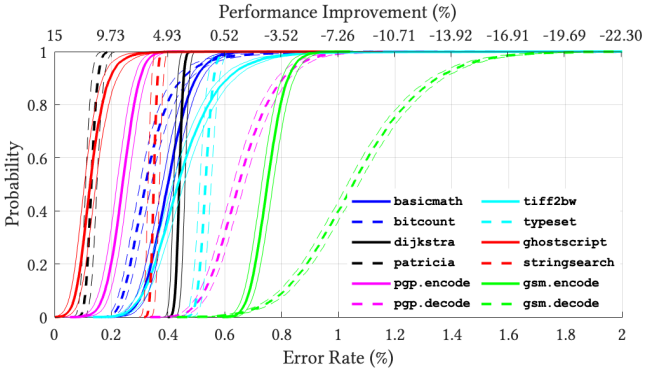

Figure 3: Cumulative Probability Distributions of Program Error Rate and Their Lower and Upper Bounds.

patricia to 1.068% (resulting in a 8.46% performance degradation) for gsm.decode. These results demonstrate that application-specific analysis is necessary for accurate evaluation of TS processors and to identify suitability of specific applications for timing speculation.

6.4 Approximation Error

In Section 5, 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 $d_K(\lambda, \bar{\lambda})$ we established in Equation (13) to/from both instances of λ in Equation (14). Then, we add/subtract the bound on $d_K(N_E, \overline{N}_E)$ we established in Equation (9) to/from the right-hand side of Equation (14). Table 2 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).

REFERENCES

[1] R. Arratia, L. Goldstein, and L. Gordon. Poisson approximation and the chen-stein method. *Statist. Sci.*, 5(4):403–424, 11 1990.

[2] O. Assare and R. Gupta. Timing analysis of erroneous systems. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, CODES '14, pages 7:1–7:10, New York, NY, USA, 2014. ACM.

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

[4] K. Bowman, J. Tschanz, S. Lu, P. Aseron, M. Khellah, A. Raychowdhury, B. Geuskens, C. Tokunaga, C. Wilkerson, T. Karnik, and V. De. A 45 nm resilient microprocessor core for dynamic variation tolerance. *Solid-State Circuits, IEEE Journal of*, 46(1):194–208, Jan 2011.

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

[6] H. Cherupalli, R. Kumar, and J. Sartori. 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)*, pages 671–681, June 2016.

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

[8] J. Constantin, L. Wang, G. Karakonstantis, A. Chattopadhyay, and A. Burg. 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)*, pages 381–386, March 2015.

[9] S. Das, C. Tokunaga, S. Pant, W.-H. Ma, S. Kalaiselvan, K. Lai, D. Bull, and D. Blaauw. Razorii: In situ error detection and correction for pvt and ser tolerance. *Solid-State Circuits, IEEE Journal of*, 44(1):32–48, Jan 2009.

[10] C. Daskalakis, I. Diakonikolas, and R. A. Servedio. Learning poisson binomial distributions. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 709–728, New York, NY, USA, 2012. ACM.

[11] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner. Razor: circuit-level correction of timing errors for low-power operation. *Micro, IEEE*, 24(6):10–20, Nov 2004.

[12] Y. Fan, T. Jia, J. Gu, S. Campanoni, and R. Joseph. Compiler-guided instruction-level clock scheduling for timing speculative processors. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 40:1–40:6, New York, NY, USA, 2018. ACM.

[13] A. Gaisler. Tsim erc32/leon simulator.

[14] A. L. Gibbs and F. E. Su. On choosing and bounding probability metrics. *International statistical review*, 70(3):419–435, 2002.

[15] B. Greskamp, L. Wan, U. R. Karpuzcu, J. J. Cook, J. Torrellas, D. Chen, and C. Zilles. Blueshift: Designing processors for timing speculation from the ground up. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 213–224, Feb 2009.

[16] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Dec 2001.

[17] Y. Hong. On computing the distribution function for the poisson binomial distribution. *Comp. Stat. Data Anal.*, 59:41–51, Mar. 2013.

[18] X. Jiao, A. Rahimi, Y. Jiang, J. Wang, H. Fatemi, J. P. de Gyvez, and R. K. Gupta. Clim: A cross-level workload-aware timing error prediction model for functional units. *IEEE Transactions on Computers*, 67(6):771–783, June 2018.

[19] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, March 2004.

[20] L. Le Cam. An approximation theorem for the Poisson binomial distribution. *Pac. J. Math.*, 10:1181–1197, 1960.

[21] D. Sinha, H. Zhou, and N. V. Shenoy. 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):1522–1533, Aug 2007.

[22] C. Stein. 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*, pages 583–602. University of California Press, 1972.

[23] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[24] Y. Zhang, M. Khayatzadeh, K. Yang, M. Saligane, N. Pinckney, M. Alioto, D. Blaauw, and D. Sylvester. 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):619–631, Feb 2018.