Timing Analysis of Erroneous Systems

Omid Assare and Rajesh Gupta

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0404
{omid, gupta}@ucsd.edu

ABSTRACT

Erroneous systems allow timing errors to occur during execution, but use measures to ensure continued operation through changes in operating parameters (voltage and frequency), error correction at various levels of the system, or ensuring controlled occurrence of errors to perform approximate computing. In this paper, we are interested in characterization of error behavior at the level 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. In order to enable reliable timing analysis of large programs, we propose *Clustered Timing Model* (CTM), a high level timing model based on clustering functionally similar timing paths of the processor, and develop a CTM for LEON3, a representative in-order RISC processor. The accuracy of the model is verified using our variation-aware timing analysis framework with an average error of 3.9% (max. 6.7%) across a wide range of voltage-temperature corners.

Categories and Subject Descriptors

B.8 [**Performance and Reliability**]; C.4 [**Performance of Systems**]: fault tolerance, measurement techniques, and modeling techniques

Keywords

erroneous systems, variability, process variation, delay faults, variation-aware timing analysis, dynamic error estimation, and software error behavior.

1. INTRODUCTION

The behavior of CMOS circuits is increasingly susceptible to variations in the manufacturing process and environmental conditions. There are two principal sources of variation [1]: *physical variations*, which cause structural parameter fluctuations in device and interconnect, and *environmental variations*, which include fluctuations in power supply and die temperature. Process variation has die-to-die (D2D) and within-die (WID) components [2]. A recent experimental Intel processor shows around 50% performance variation among its 80 cores when operated at $0.8V$ due to WID process variation alone [3]. As for the environmental variations, 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 several tens of percent performance change [4].

Circuit designers have addressed the growing uncertainty in the timing characteristics of variability-affected systems by continueous widening of timing guardbands. However, gaurdbands continue to increase steadily with each technology generation [4], leading to loss of performance and increased costs due to overdesign. In response, variability management techniques have seen significant attention in recent years. The majority of these techniques take one of the following approaches:

1. *Uncertainty modeling*: These methods try to develop timing models that would predict the effect of various sources of uncertainty on the timing behavior of systems. Statistical Static Timing Analysis (SSTA) is an example, where process parameters are treated as statistical distributions and their impact on performance is quantified using statistical methods. Using such models, designers are able to decrease timing gaurdbands caused by the variability in the manufacturing process of CMOS circuits.
2. *Dynamic adaptation*: These techniques build capabilities into the systems to enable runtime adaptation. One such technique, called Active Management of Timing Guardband [5], is implemented in IBM POWER7 server where Critical Path Monitors (CPM) 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.

A key shift in the way timing margins are viewed by circuit designers was introduced by the idea of *timing speculation* [6]. Traditional sequential circuit designs work under the strict condition that voltage and frequency of the circuit should be set such that no timing violations can occur. Timing speculation, on the other hand, allows timing errors by removing timing margins and relies on circuit- and architecture-level techniques to detect and recover from these errors. While static timing analysis methods assume an overly pessimistic worst-case condition on the input data to a circuit block (i.e. software in the case of processors), and impose an implicit timing margin, timing speculative systems view software as yet another source of uncertainty. A representative example is Razor [6] in which gaurdbands are reduced to the extent that represents onset of timing errors. However, the variability in how different instruction sequences cause errors leads to increased performance during the execution of less vulnerable code, while faulty instructions are detected and corrected using special hardware and by paying a recovery penalty.

Timing speculative systems are part of a larger class of systems called *erroneous systems* [3]. While timing speculative processors maintain correct execution by using hardware-based error recovery mechanisms, other erroneous systems rely upon application level error recovery, or even relax the correct execution criteria and perform *approximate computing*. A common characteristic of erroneous systems is the important role of software in both determining and managing the system's error behavior. However, to do this process well, we need models that can predict how errors occur as a function of the actual running code.

In this new paradigm, the model of hardware presented to software as the execution machine of the instructions is not a constant and deterministic model, but a variable and non-deterministic one. Moreover, the variability of this model is not only a function of physical and environmental parameters on which software has nearly no control, but also a strong function of the software itself. The execution of two different sequences of instructions could dramatically change the timing behavior of a fixed piece of hardware in a fixed environment. As such, the traditional hardware-to-software model that only consisted of a set of instructions with fixed and deterministic timing and functional behaviors is no longer sufficient. New models need to dynamically predict the next state of the system based on the software it is running and the environmental conditions it is placed in, and be aware of the non-determinism present due the physical variations. Moreover, for these models to be seamlessly utilized in the design flow of both hardware and software systems, they need to have vital characteristics of existing timing models such as modularity, hierarchy, and the capability of producing detailed information about the timing behavior of the system.

In this paper, we propose one such model and use it to explore questions such as:

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 4).
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 5).
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 6).

2. RELATED WORK

The need for a fast and accurate timing model for dynamic estimation of errors is most evident in the '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 [12-16]. The lack of fast simulation platforms for these systems is becoming a bottleneck and a key challenge for erroneous systems research [4], [14]. As a result, most works 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 of variability management techniques in which the analysis of dynamic error behavior has been limited. Trifecta [12] focuses only on the ALU adder and selection logic. Roy and Chakarborty [13] only consider ALU errors and limit their simulations to 100 instructions that most frequently exercise the ALU. Xin and Joseph [14] focus on ALU, LSU, and Shift/Branch units. Hoang et al. [15] reduce the size of benchmark input sets due to extremely long run- times associated with gate-level simulations. Similarly, Sartori and Kumar [16] only study the LSU and IQ and do not simulate the benchmarks for their entire duration.

At the same time, efforts specifically directed at variability-induced error prediction have also been limited. Rehman et al. [9] 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 Instruction [10] and Sequence [11] Level Vulnerability to predict the error probabilities, but their models do not specifically take input data dependence into account. A number of recent works [13], [14] 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. Such techniques, even if effective, do not result in timing models that can be used at various phases of the design of these systems. Finally, an emulation testbed for variability-aware software called VarEMU is presented in [7]. 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.

3. 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 4 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 paper, 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.

4. VARIATION-AWARE TIMING ANALYSIS: A FIRST ATTEMPT

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 Fig. 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 dis-

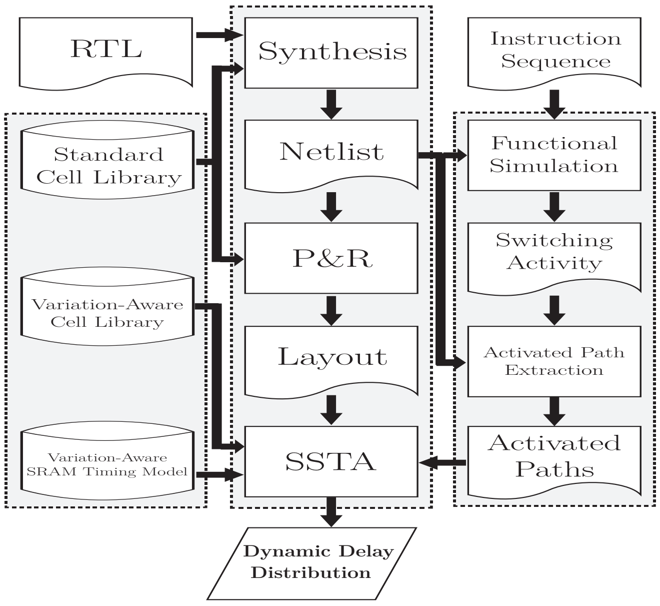

Figure 1: Variation-Aware Timing Analysis Framework

tribution 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 (Fig. 1), ASIC implementation is performed using *Synopsys Design Compiler* and *Synopsys IC Compiler* targeting a *TSMC 45nm* technology [21]. 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 VARTX [17], 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 [18]. 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.

5. 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 4, but also nearly as fast as an architecture 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 architecture level objects that determine the delay of the processor.

5.1 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). (Fig. 4 shows a CTM of a typical in-order pipelined processor. More details in section 5.4).

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 Fig. 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.

5.2 Training and Application

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

5.2.1 *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} [20]:

$$\mu_{AB} = \mu_A \Phi(\frac{\mu_A - \mu_B}{\theta}) + \mu_B \Phi(\frac{\mu_B - \mu_A}{\theta}) + \theta \phi(\frac{\mu_A - \mu_B}{\theta}) \quad (1)$$

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

where $\phi(.)$ and $\Phi(.)$ are, the 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 Eq. 1 or Eq. 2, or as their average to reduce estimation error.

5.2.2 *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 Eq. 1 and Eq. 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:

$$PoE_H = \frac{1}{2}\left[1 - \text{erf}\left(\frac{\frac{1}{F} - \mu_H}{\sqrt{2\sigma_H^2}}\right)\right] \quad (3)$$

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 Eq. 4.

$$PoE = 1 - \prod_{i=1}^{N}(1 - PoE_{H_i}) \quad (4)$$

As will be explained in section 6, activated hyperpaths can be identified using architecture level information available during simulation. In LEON3 CTM (Fig. 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.

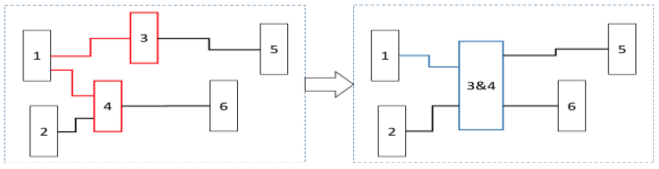

Figure 2: CTM hierarchy. SSCs 3 and 4 merge to obtain in a higher-level CTM

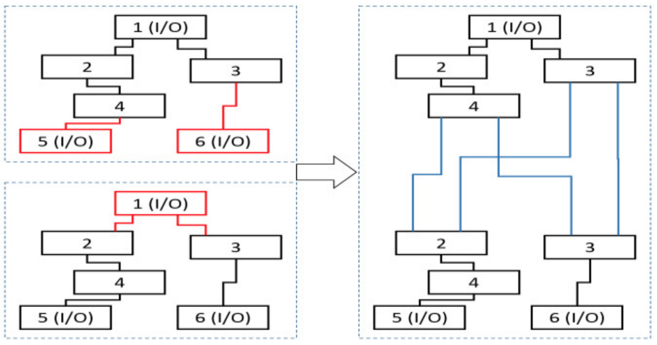

Figure 3: CTM modularity. CTMs on the left connect to form CTM on the right

5.3 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.

5.3.1 *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 Fig. 2 where the red RCs and hyperpaths in the left CTM are merged and replaced by the blue ones in the left CTM.

5.3.2 *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 Fig. 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.

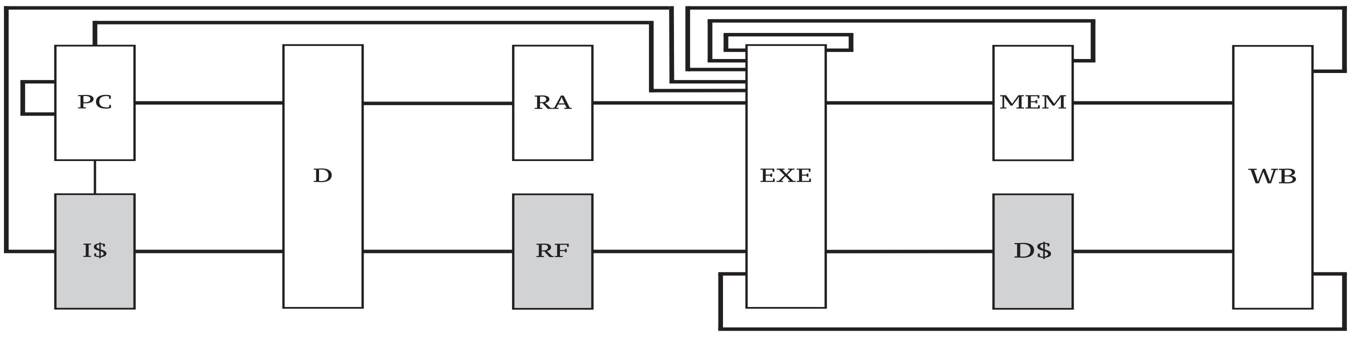

Figure 4: Clustered Timing Model for LEON3 pipeline.

5.4 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.

5.4.1 LEON3 Processor Core Overview

LEON3 is a 32-bit processor core conforming to the IEEE-1754 (SPARC V8) architecture, 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.

5.4.2 Register Clustering and Model Training

The register clustering (defined in section 5.1) 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 Fig. 4, where boxes are RCs and lines are hyperpaths (shaded boxes represent I/O RCs). 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.

```
 1 clr %l0
 2 mov data1, %l1
 3 mov data2, %l2
 4 while (1)
 5 {
 6     nop   ; 7 times
 7     inst  %l0, %l0, %l3
 8     inst  %l1, %l2, %l3
 9     inst  %l0, %l0, %l3
10     nop   ; 7 times
11 }
```

Figure 5: Example pseudocode template for model training

An example pseudocode template is given in Fig. 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 4) 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 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-

Table 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**

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 Fig. 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 Fig. 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 Fig. 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.

5.5 Accuracy Evaluation

In this subsection, 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 $45nm$ 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 4 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. Table 1 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%.

6. 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 5, we used TSIM [19], 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 architecture level information that determines the activated hyperpaths of instructions to derive instruction error rates at different pipeline stages. The resulting variation-aware simulation platform is much faster than the timing analysis framework we developed in Section 4 and enables the simulation of actual programs. We selected five benchmarks (shown in Table 2) from MiBench [8] 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.

6.1 Inter-Program Variation

Inter-Program Variation denotes the variability in the error rates of different programs when run on a fixed proces-

Table 2: Inter-Program Variation

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

sor. 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 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.

6.2 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 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 [4], 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, which is beyond the scope of this paper.

6.3 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 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.

Table 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

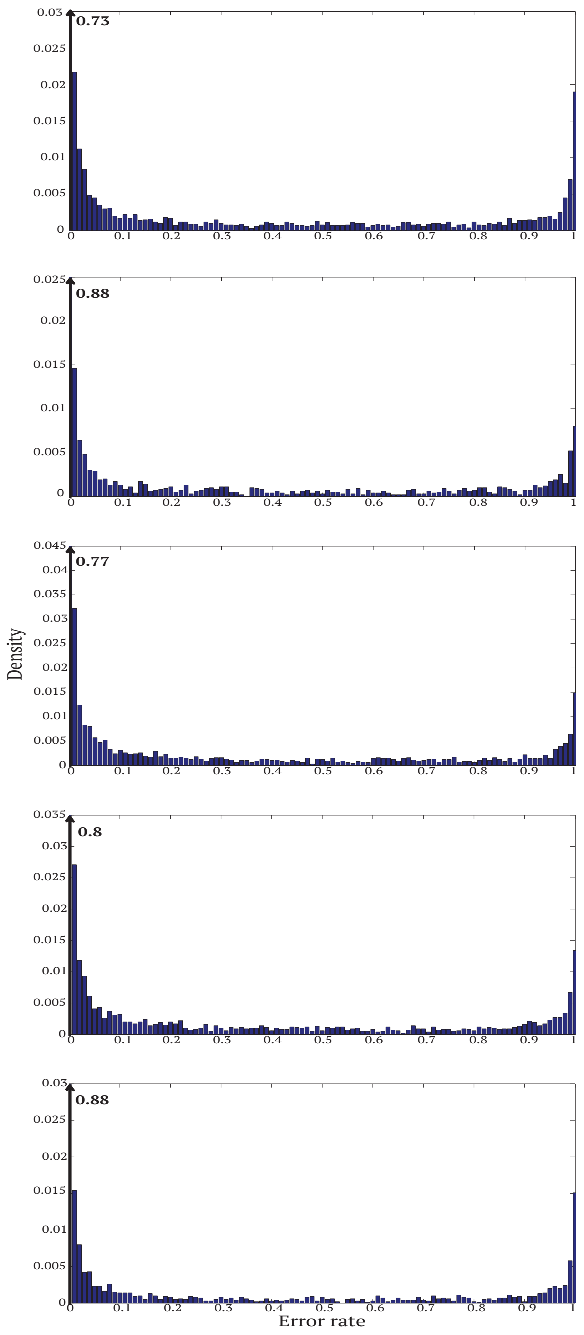

Figure 6: Error rate distribution among various static instructions. From top to bottom, figures correspond to `basicmath`, `bitcount`, `qsort`, `dijkstra`, and `stringsearch`.

Table 4: Distribution of errors among the 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*				*Inst. Decode*		*Operand Read*		*Address Generation*			*ALU*	*Bypass*			*Inst. Result*		
Benchmarks	**basicmath**	0.024	0.006	0.009	0.003	0.264	0.084	0.006	0.012	0.162	0.126	0.249	0.681	0.021	0.003	0.009	0.093	0.153	0.261
	bitcount	0.009	0.006	0.005	0.002	0.121	0.049	0.006	0.001	0.038	0.031	0.092	0.121	0.003	0.002	0.001	0.011	0.057	0.092
	qsort	0.114	0.042	0.006	0.012	0.792	0.714	0.060	0.030	0.384	0.306	0.294	0.473	0.018	0.006	0.036	0.588	0.372	0.366
	dijkstra	0.136	0.048	0.004	0.012	0.216	0.248	0.008	0.024	0.168	0.152	0.136	0.536	0.016	0.020	0.016	0.124	0.244	0.268
	stringsearch	0.012	0.003	0.001	0.001	0.068	0.074	0.008	0.002	0.091	0.064	0.052	0.151	0.002	0.007	0.001	0.064	0.019	0.079
	Avg	**0.059**	**0.021**	**0.005**	**0.006**	**0.292**	**0.234**	**0.018**	**0.014**	**0.169**	**0.136**	**0.165**	**0.228**	**0.012**	**0.008**	**0.013**	**0.176**	**0.169**	**0.213**

Fig. 6 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* [14]. 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 instructions with more relaxed timing requirements, use architecture and circuit level techniques such as time borrowing or increase timing gaurdbands by decreasing frequency or increasing voltage. 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 correction 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.

6.4 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 4 presents the distribution of errors among the hyperpaths and functional networks of LEON3. A hyperpath is considered faulty when it produces an error *and* it is activated by the instruction using it. 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 portion. For better visuality, Fig. 7 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.

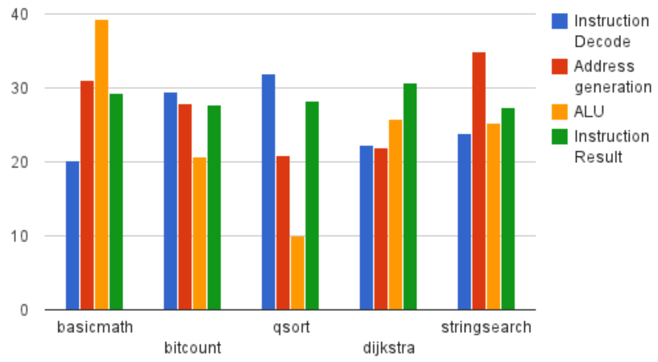

Figure 7: Percentage of errors in the most faulty processor networks

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

7. CONCLUSIONS

We have presented an analysis of the error behavior of erroneous systems (i.e. systems that allow timing errors). Performance of these systems is heavily dependent on the location of the errors and the rate at which they occur. We demonstrated the important role of software in determining this error behavior by developing a high level variation-aware timing model for an in-order RISC processor. The model, called Clustered Timing Model (CTM), was verified using an accurate timing analysis framework and used for fast timing simulation in the presence of process variation. Through these simulations, we introduced and characterized four aspects of how the error behavior is affected by the running software. Inter- and Intra-Program Variation were proposed as measures of the error rate variability in different programs and among instructions of a program. It was 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.

8. ACKNOWLEDGMENTS

This material is based upon the work supported by the National Science Foundation's 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.

9. REFERENCES

[1] C. Forzan, D. Pandini, "Statistical static timing analysis: A survey", Integration, the VLSI Journal, vol. 42, pp. 409-435, June 2009.

[2] D. Blaauw, K. Chopra, A. Srivastava, L. Scheffer, "Statistical Timing Analysis: From Basic Principles to State of the Art," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol.27, pp.589-607, April 2008.

[3] S. Dighe, S. Vangal, P. Aseron, S. Kumar, T. Jacob, K. Bowman, J. Howard, J. Tschanz, V. Erraguntla, N. Borkar, V. De, S. Borkar, "Within-die variation-aware dynamic-voltage-frequency scaling core mapping and thread hopping for an 80-core processor," Solid-State Circuits Conference Digest of Technical Papers (ISSCC), IEEE International , pp.174-175, Feb. 2010.

[4] P. Gupta, Y. Agarwal, L. Dolecek, N. Dutt, R.K. Gupta, R. Kumar, S. Mitra, A. Nicolau, T.S. Rosing, M.B. Srivastava, S. Swanson, D. Sylvester, "Underdesigned and Opportunistic Computing in Presence of Hardware Variability," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, pp.8-23, Jan. 2013

[5] M. Floyd, M. Allen-Ware, K. Rajamani, B. Brock, C. Lefurgy, A.J. Drake, L. Pesantez, T. Gloekler, J.A. Tierno, P. Bose, A. Buyuktosunoglu, "Introducing the Adaptive Energy Management Features of the Power7 Chip," Micro, IEEE , vol.31, no.2, pp.60-75, March-April 2011.

[6] D. Ernst, Nam Sung Kim; S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, T. Mudge, "Razor: a low-power pipeline based on circuit-level timing speculation," Microarchitecture, MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on , pp.7-18,

[7] L. Wanner, S. Elmalaki, Liangzhen Lai, P. Gupta, M. Srivastava, "VarEMU: An emulation testbed for variability-aware software," Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on, pp.1-10, Sept. 29 2013-Oct. 4 2013

[8] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," Workload Characterization, IEEE International Workshop on , pp.3-14, 2001.

[9] S. Rehman, M. Shafique, F. Kriebel, J. Henkel, "Reliable software for unreliable hardware: embedded code generation aiming at reliability," In Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '11). ACM, New York, NY, USA, 237-246.

[10] A. Rahimi, L. Benini, R.K. Gupta, "Analysis of instruction-level vulnerability to dynamic voltage and temperature variations," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012, pp.1102-1105, March 2012.

[11] A. Rahimi, L. Benini, R. Gupta, "Application-Adaptive Guardbanding to Mitigate Static and Dynamic Variability," Computers, IEEE Transactions on , pp. 1-10, 2013.

[12] P. Ndai, N. Rafique, M. Thottethodi, S. Ghosh, S. Bhunia, K. Roy, "Trifecta: A Nonspeculative Scheme to Exploit Common, Data-Dependent Subcritical Paths," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , vol.18, no.1, pp.53-65, Jan. 2010.

[13] S. Roy, K. Chakraborty, "Predicting timing violations through instruction-level path sensitization analysis," In Proceedings of the 49th Annual Design Automation Conference (DAC '12). ACM, New York, NY, USA, 1074-1081.

[14] J. Xin, R. Joseph, "Identifying and predicting timing-critical instructions to boost timing speculation," In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44 '11). ACM, New York, NY, USA, 128-139.

[15] G. Hoang, R.B. Findler, R. Joseph, "Exploring circuit timing-aware language and compilation," SIGPLAN Not. 47, 4 (March 2011), 345-356.

[16] J. Sartori, R. Kumar, "Architecting processors to allow voltage/reliability tradeoffs," In Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems (CASES '11). ACM, New York, NY, USA, 115-124.

[17] J. Samandari-Rad, M. Guthaus, R. Hughey, "VAR-TX: A variability-aware SRAM model for predicting the optimum architecture to achieve minimum access-time for yield enhancement in nano-scaled CMOS," Quality Electronic Design (ISQED), 13th International Symposium on , pp.506-515, March 2012.

[18] D. Sinha, Hai Zhou; N.V. Shenoy, "Advances in Computation of the Maximum of a Set of Gaussian Random Variables," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on , vol.26, no.8, pp.1522-1533, Aug. 200

[19] Aeroflox Gaisler, "TSIM ERC32/LEON Simulator," http://www.gaisler.com/index.php/products/simulators/t-sim

[20] S. Nadarajah, S. Kotz, "Exact Distribution of the Max/Min of Two Gaussian Random Variables," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , vol.16, no.2, pp.210-212, Feb. 2008

[21] TSMC 45nm standard cell library release note, TCBN45GSBWP, version 120A, Nov. 2009.