Application-Adaptive Guardbanding to Mitigate Static and Dynamic Variability

Abbas Rahimi, *Student Member*, *IEEE*, Luca Benini, *Fellow*, *IEEE*, and Rajesh K. Gupta, *Fellow*, *IEEE*

Abstract—Traditional application execution assumes an error-free execution hardware and environment. Such guarantees in execution are achieved by providing guardbands in the design of microelectronic processors. In reality, applications exhibit varying degrees of tolerance to error in computations. This paper proposes an adaptive guardbanding technique to combat CMOS variability for error-tolerant (probabilistic) applications as well as traditional error-intolerant applications. The proposed technique leverages a combination of accurate *design time* analysis and a minimally intrusive *runtime* technique to mitigate Process, Voltage, and Temperature (PVT) variations for a near-zero area overhead. We demonstrate our approach on a 32-bit in-order RISC processor with full post Placement and Routing (P&R) layout results in TSMC 45 nm technology. The adaptive guardbanding technique eliminates traditional guardbands on operating frequency using information from PVT variations and application-specific requirements on computational accuracy. For error-intolerant applications, we introduce the notion of *Sequence-Level Vulnerability* (*SLV*) that utilizes circuit-level vulnerability for constructing high-level software knowledge as metadata. In effect, the *SLV* metadata partitions sequences of integer SPARC instructions into two equivalence classes to enable the adaptive guardbanding technique to adapt the frequency simultaneously for dynamic voltage and temperature variations, as well as adapt to the different classes of the instruction sequences. The proposed technique achieves on an average $1.6\times$ speedup for error-intolerant applications compared to recent work [33]. For probabilistic applications, the adaptive technique guarantees the error-free operation of a set of paths of the processor that always require correct timing (*Vulnerable Paths*) while reducing the cost of guardbanding for the rest of the paths (*Invulnerable Paths*). This increases the throughput of probabilistic applications upto $1.9\times$ over the traditional worst-case design. The proposed technique has 0.022% area overhead, and imposes only 0.034% and 0.031% total power overhead for intolerant and probabilistic applications respectively.

Index Terms—Process, voltage, temperature (PVT) variations, timing error, variation-tolerant processor, adaptive guardbanding, resilient design, computation accuracy

1 Introduction

Performance and power uncertainty caused by variability in the manufactured parts is a major design challenge in nanoscale CMOS technologies [1]. Variations arise from different physical sources: (i) static inherent process parameter variations in channel length and threshold voltage variations due to random dopant fluctuations and subwavelength lithography; and (ii) dynamic environmental variations in ambient conditions such as temperature fluctuations and supply voltage droops. Such parameter variations in device geometries in conjunction with undesirable fluctuations in operating condition might prevent circuit from meeting timing constraints thus degrading parametric yield. These issues are expected to worsen with technology scaling [2]. Designers commonly use conservative guardbands for the operating frequency and voltage to ensure error-free operation for the worst-case variations over circuit lifetime that leads to loss of operational efficiency [3]. Therefore, accurate *design time* analysis coupled with efficient *runtime* techniques are required to overcome the variability challenges.

Indeed, several recent efforts have focused on measures to mitigate variability through innovations in circuit-level designs. Razor-style [4], [5] sequential circuit elements have been widely researched to detect Process, Voltage, and Temperature (PVT) variations coupled with adaptive recovery methods for quick online error detection and compensation. A recent 45 nm Intel in-order processor [6] replaces flip-flops connected to the endpoints of the critical paths of pipeline stages with Error-Detection Sequential (EDS) [7] circuits to detect late timing transitions. For error recovery, the processor supports two online techniques: (i) instruction replay at half frequency, and (ii) multiple-issue instruction replay at the same frequency. In a similar vein, Bubble Razor [8] leverages a two-phase latch timing technique in conjunction with a local replay mechanism on an ARM Cortex-M3 microprocessor. Alternatively, low-overhead and less intrusive on-chip Critical Path Monitors (CPM) [9] measure the timing margin available to circuits, and do not impose architectural modification. IBM 8-core POWER7 employs five CPMs per each core to capture PVT variations and detect early wearout conditions which impose only 0.12% area overhead [10]. To ensure recovery after error-detection, adaptive clock scaling is used in resilient silicon implementations [6], [10], [11]. For instance, fast single-cycle adaptive frequency technique to deal with sudden changes in temperature and supply voltage variations

- *A. Rahimi and R.K. Gupta are with the Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093-0404 USA. E-mail: {abbas, gupta}@cs.ucsd.edu*
- *L. Benini is with the Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna, 40136 Bologna, Italy. E-mail: luca.benini@unibo.it*

Manuscript received 22 July 2012; revised 06 Mar. 2013; accepted 13 Mar. 2013. Date of publication 30 May 2013; date of current version 07 Aug. 2014. Recommended for acceptance by S.W. Chung.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.72

has been demonstrated for a 3.4 GHz commercial TCP/IP processor [11].

Going further up on the hardware-software stack, several efforts have tried to characterize and use variability related information. We have earlier defined the notion of Instruction-Level Vulnerability (ILV) to dynamic voltage and temperature variations in order to expose and use variation in architectural/compiler optimizations [12]. Furthermore, Maniatakos *et al.* investigate correlation between (low-level) faults in the control logic of a microprocessor and their instruction-level impact on the execution of a workload in order to classify faults into instruction-level error types [13]. It has also been shown that a micro-architecture and compiler collaborative design can lead to a cost-effective solution for dynamic voltage variations in commodity processors [14]. Online procedure-level [15] and task-level [16] techniques have been proposed for variation-tolerant embedded multiprocessor SoCs. Finally, Cong *et al.* propose hybrid application-level correctness techniques to mitigate soft-error through control flow analysis for identifying critical program segments [17]. The static analysis phase identifies critical instructions and minimizes the number of instructions that are duplicated and checked at runtime, using a software-based fault detection and recovery technique [18].

These methods strive to achieve instruction executions *exactly* as specified by the application programs. In contrast, probabilistic programs can exhibit enhanced error resilience at the application-level when multiple valid output values are permitted. Conceptually, such programs have a vector of 'elastic outputs', and if execution is not 100% numerically correct, the program can still appear to execute correctly from the user's perspective [19]. Programs with elastic outputs have application-dependent fidelity metrics (such as peak signal to noise ratio) associated with them to mathematically characterize the quality of the computational result. The degradation of output quality for such applications (e.g., multimedia and compression [19]) is acceptable if the fidelity metrics satisfy a certain threshold. Martinez *et al.* propose dynamic tolerant region reuse [20], a method based on relaxing the conditions upon skipping regions of instructions by caching results of previous equal and also similar inputs that relies in the tolerance in the output precision of media algorithms. An Error Resilient System Architecture (ERSA) [21] presents a robust system that utilizes software optimizations and error-resilient algorithms of the probabilistic applications based on their classification as Recognition, Mining and Synthesis (RMS) applications [22]. Identification of error tolerant operations and the determination of the extent to which errors can be tolerated in individual operations remains an active area of research [23].

1.1 Contributions

We propose a near-zero area overhead adaptive guardbanding technique to meet application-specific requirements on computational accuracy. This work makes the following contributions:

I. We present a method to relate low-level hardware vulnerability information obtained using accurate and practical variation-aware analysis to high-level knowledge in software. Our analysis flow considers the dynamic voltage and temperature as well as static process variations, and validates results on a full post P&R layout of a 32-bit in-order RISC processor.
II. We propose an adaptive guardbanding technique to dynamically adjust the cycle time to PVT variations and application-level computation accuracy. For probabilistic applications represented by multimedia benchmarks from MiBench [52] and MediaBench [55], the technique achieves up to $1.9\times$ throughput improvement in comparison to the traditional worst-case design.
III. For error-intolerant applications, we introduce the notion of *Sequence-Level Vulnerability* (*SLV*) to dynamic voltage and temperature variations. Our experimental results and analysis show that *SLV* is not uniform across sequences obtained from a large set of general purpose benchmarks [52]–[57]. Effectively, the *SLV* partitions sequences of integer SPARC instructions into two classes: *ClassI*, which only consists of the arithmetic/logical instructions; and *ClassII*, a mixture of all types of instructions. We also show the effectiveness of compiler technique to achieve a favorable mix of sequences. Using *SLV* enables the processor to achieve $1.6\times$ average speed-up for intolerant applications, compared to [33], by adapting the cycle time for dynamic variations and different instruction sequences.

The minimally intrusive and parsimonious guardbanding in software greatly reduces the hardware cost with respect to the above-mentioned circuit techniques. Full layout results on TSMC 45 nm technology show that the proposed guardbanding imposes only 0.031% and 0.034% total power overhead for the probabilistic and the intolerant applications respectively. The total area overhead is 0.022%.

The rest of the paper is organized as follows. Section 2 surveys prior work in this specific topic area. Section 3 describes the effects of PVT variations. Sections 4 and 5 cover analysis of probabilistic and intolerant applications. The adaptive guard-banding technique is presented in Section 6. In Section 7, we explain our methodology for the characterization of *SLV* and present experimental results followed by conclusions in Section 8.

2 Related Work

There are three main sources of relevant prior work organized as circuit-level, architecture-level, and software techniques.

The most immediate manifestations of variability are variations in path delay and power. While path delay variations have been addressed extensively by the test community as delay fault testing problems, knowing the sources of path delay variations enables the circuit designers a better focus on measures to combat variability. For instance, a major source of variation is voltage droops and the fact that these errors matter only when they lead to changes in state. Combining these two observations have led the community to a rich literature in recent years for handling variability induced errors at the circuit-level. A common strategy is to detect error, and expand the window of recoverability using data-dependent path delays, time borrowing and/or tuning

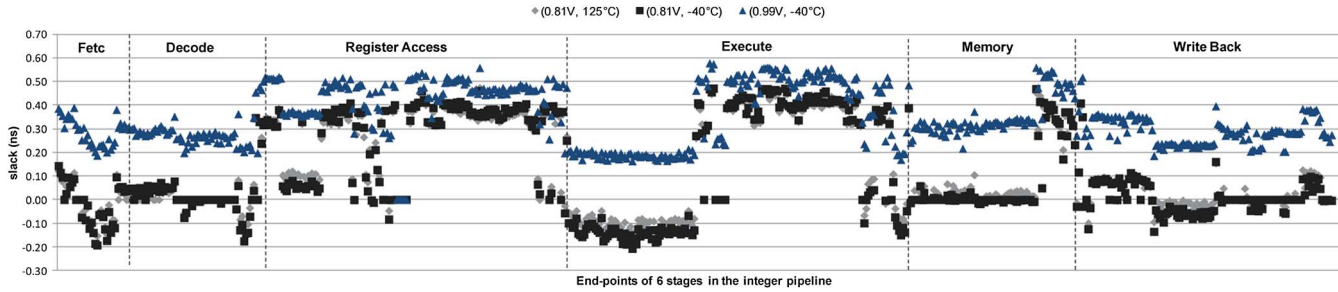


Fig. 1. Non-uniform slack variation of the integer pipeline stages caused by PVT—cycle time is set at 0.83 ns.

the supply voltage. Detection circuits typically use double sampling using shadow latches whereas tunable replica circuits enable non-intrusive operations.

Circuit-level techniques seek to identify variability-caused error conditions and correct for these errors through a variety of circuit-level modifications. Double sampling was originally applied in Razor [4], updated in Razor II [5] with modifications for transition detection, double sampling with time borrowing to EDS [7]. Recently, Intel has used EDS to build a resilient core [6] to quickly observe and detect the timing failure due to the fast dynamic variations as well as static variations on a processor die. In addition, Razor, Razor II, and Bubble Razor [8] have been used in ARM microprocessors. A more systematic time borrowing flip-flop was presented in [24] that uses clock shifter circuits to allow time borrowing on critical paths, generate time-borrow signal to clock shifter to stretch the clock period. Critical Path Isolation for Timing Adaptiveness (CRISTA) [25] isolates the long critical paths of the design and provides an extra clock cycle for those paths. Trifecta [26], a variable latency processor based on CRISTA, completes instructions that activate those long critical paths in two cycles. Another hardware-implemented technique tolerates the delay variability on critical paths by enabling a localized path-grained adaptation mechanism [27].

These techniques, while useful at the circuit-level, are fundamentally limited when it comes to their use in software. In addition, the overhead of the intrusive pipeline modifications and circuit sensors for the error detection in conjunction with the power-hungry error recovery techniques can be large, e.g., 1–3% [4], [5], 3.8% [6], 9.5% [27], 18% [26], 21% (same timing constraint) [8], and 30% [28] overheads in area, and 8% [24] power overhead. Further, Razor-style designs are shown to be not very effective in the face of aggressive voltage over-scaling [29].

Architectural and software techniques focus on methods that can be used by the computer architects, compiler writers to mitigate variability effects. Prominent works to combat soft-errors include the notion of architectural vulnerability factor [30], design of ERSA [21] as a resilient architecture, software-based error detection by duplicated instructions [18], and application-level correctness [17]. A common challenge to these approaches is that unlike the uniform soft-error models [17], [18], [21], [30], delay variations caused by PVT are deeply affected by the details of physical processor implementation, and do not uniformly alter architectural registers (see Fig. 1).

Liang and Brooks propose a joint architectural and statistical timing analysis method of selecting micro-architectural parameters, e.g., selection of pipeline depth and size, to mitigate the impact of frequency variations [31]. This is limited to combating only process variations. Collaborative architectures and compiler-based techniques have been proposed in [32] to rearrange instructions such that dynamic current fluctuations are suppressed. As a result, they enable mapping the dynamic voltage droops to the original source code. Similarly, a recent work makes the observation that the sequence of instructions in an application can have a significant impact on timing error rate and introduces code transformations for improving timing speculation [33]: e.g., (i) NOP padding which enforces extra cycle penalty; (ii) ISA extensions by adding a *brinc* instruction which requires intrusive architectural modification. Ultra-Reduced Instruction Set Co-processors (URISC) [28] extends a MIPS processor with a co-processor that implements a new instruction called *subleq*. URISC executes the sequences of *subleq* that are semantically equivalent to any faulty instruction. However, this technique has a considerable impact on the performance, e.g., up to $45\times$ performance penalty in case of a faulty multiplication instruction.

These architectural and compiler techniques either consider only process variations [31], or only dynamic variation [32]. Furthermore, [31], [33] use a superficially 'generic' variability model on high-level architectural simulators that do not consider the details of physical processor implementation and constraints (more details in Section 7). This limits use of software-assisted techniques that require highly-accurate *design time* analysis on the implemented cores with detailed and validated variability models given by the semiconductor fabrication process. We believe a combination of *design time* and *runtime* techniques is essential to meet the application-specific requirements on computational accuracy with minimal impact on architectural and circuit modifications; considering only one of them leads to unacceptable overhead.

3 PVT Variations

In this section, we analyze the delay variations caused by PVT variations on the paths of the 32-bit in-order LEON3 processor compliant with the SPARC V8 architecture [34]. This choice is keeping in view of the recent trends towards array processor architectures containing many simple RISC cores, e.g., GPUs [35], TILERA [36], and Platform 2012 [37]. More importantly, the availability of an advanced open-source RISC core with full back-end details is critical to accurate variation analysis. We note that other efforts for complex high-performance cores such as IBM POWER6 also confirm that vulnerability is not uniform across the instructions set [38]. While different

instruction sets will lead to different grouping of instructions depending upon the processor architecture and implementation, our methodology can be applied as long as there is a non-uniform vulnerability across the instructions.

Specifically, the effects of a full range of dynamic variations (an industrial temperature range of $-40°C$–$125°C$, and a voltage range of 0.72 V–0.99 V) as well as static process parameters variations (die-to-die and within-die) are analyzed on all paths throughout the entire integer pipeline of LEON3. Fig. 1 illustrates the delay variation in the six stages of the pipeline that results in positive/negative slacks for the flip-flops connected to the endpoints of the paths. The cycle time is set at 0.83 ns to meet the timing requirement of the typical-corner (0.9 V, $25°C$, TT). A higher voltage of 0.99 volts results in shorter delay (positive slack), while the lower temperature leads to a higher delay in the low-voltage region below 0.9 volts, since MOSFET drain current decreases when the temperature is decreased in nanometer CMOS technologies [39]. In addition to these dynamic operating conditions, the static process variations exacerbate the delay variation across various pipeline stages: Section 3.2 describes the details of modeling the process variations.

Given such variations across operating conditions and across different parts of the design, an adaptive guardbanding of the operating frequency is useful to ensure the error-free operation. Such a guardband can be much less conservative than a statically determined guardband. We divide pipeline paths into two groups: (a) *Vulnerable Paths* (VP): A set of paths that always require correct timing and any delay variability may result in catastrophic architectural failures and consequently visible errors in the outputs of a program; and (b) *Invulnerable Paths* (IP): A set of paths that do not require 100% timing correctness. The delay variation in IP does not cause catastrophic architectural failures since it affects only the vector of elastic outputs. The vector of elastic outputs does not require the complete numerical correctness. Thus, the delay variation in IP may degrade of the quality of fidelity metrics of the probabilistic applications.

Specifically for LEON3 pipeline shown in Fig. 1, a 20% voltage variation results in many negative slack values at the endpoints of the *fetch* and *decode* stages which causes the wrong instructions to be executed. Thus the paths that lie in these stages are considered as VP and must always meet the setup time of flip-flops in PVT variation. On the other hand, the scenario for IP is different. For example in the *execution* stage, some endpoints do not suffer from delay variation at all (those paths with a positive slack), and some endpoints have negative slack when voltage variation occurs. The execution stage has much more flexibility to deal with delay variation as long as it can produce an acceptable fidelity metric.

In Section 4.1, we present guardbanding technique that seeks to guardband VP for error-free operation, and at the same time effectively reduces the cost of guardbands on IP against fidelity metric of programs that are tolerant to imprecise and approximate computations. The tolerance levels can be specified based on algorithmic classifications such as RMS [22], and multimedia [19]. Section 5 also covers another adaptive guardbanding technique for intolerant applications in the general case.

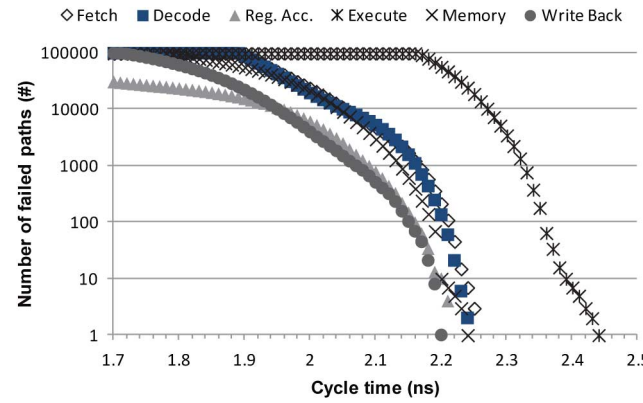


Fig. 2. Number of failed paths of LEON3 pipeline using STA.

3.1 Conventional Static Timing Analysis

Conventional Static Timing Analysis (STA) calculates the maximum delay variation using the worst-case corner, by simply combining the absolute worst-case combination of the process, voltage, and temperature parameters. The cycle time is finely varied to observe the behavior of the pipeline stages. The number of failed paths (i.e., paths with negative slack) for each stage using the STA in the worst-corner (0.72 V, $0°C$, Slow NMOS-Slow PMOS) is shown in Fig. 2. Increasing the cycle time form 1.8 ns to 2.25 ns reduces the number of failed path from hundreds of thousand paths to zero path for all stages except the *execution* stage which has a higher delay. The *execution* stage needs 10% more guardbanding, i.e., the clock cycle of 2.5 ns. Further, [12] shows that the *execution* and *memory* stages are highly vulnerable to dynamic variations. By setting the cycle time at 2.25 ns, we guarantee that no path will fail within the *fetch*, *decode*, *register access*, *memory*, and *write back* stages even in the worst-case process parameter variation. The paths in these stages are considered as VP because: (i) any failure in *fetch* or *decode* stages may cause the wrong instructions to be executed that cannot be masked even within the probabilistic application; and (ii) any failure in the *register/memory/write back* stages may cause an illegal access/operation on the memory/registers. It is therefore not surprising that both Intel resilient processor [6] and relaxed-reliability cores in ERSA [21] consider sufficient guardbanding in register stage, memory management unit, and L1 instruction cache. By sufficient guardbanding on VP through STA, the error-free operation of VP is guaranteed even if these paths display the worst-case process characteristics.

Unlike the above mentioned stages, with the cycle time of 2.25 ns, the *execution* stage has few failed paths in the worst-case process variation. If these paths are activated through the pipeline, there is no guarantee for 100% timing correctness of the *execution* stage. This lack of timing correctness causes inaccuracies in the result of execution of some instructions, which can be masked by the error resilience at the application-level of the probabilistic applications [19], [21], or proper software-based instruction duplication technique. Thus, these paths are considered as IP, since their violation might cause only application-level derating which strictly depends to the type of applications [38]. In Section 4, we examine the likelihood of these violations, and the type of applications that can accept or refuse this kind of inaccuracies.

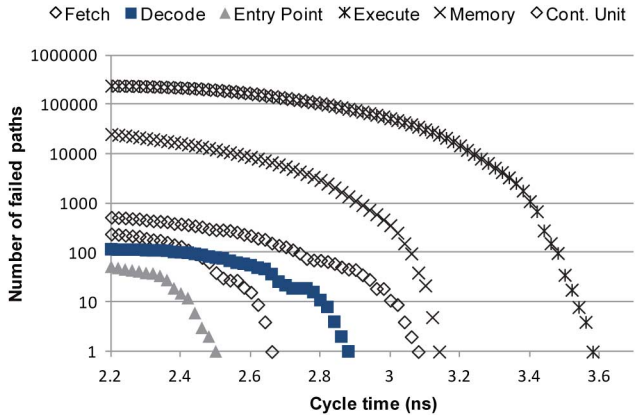


Fig. 3. Number of failed paths of THEIA pipeline using STA.

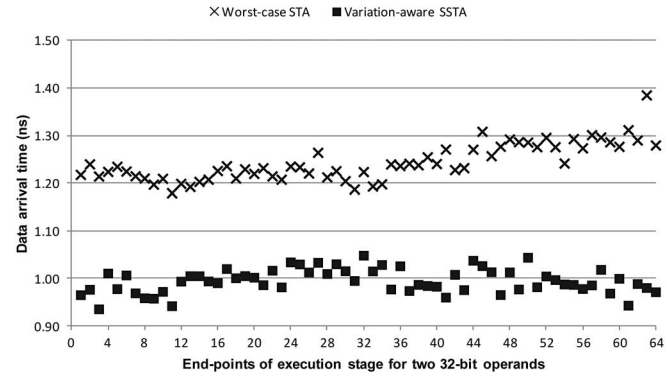


Fig. 4. Variation-aware SSTA versus the worst-case STA.

To observe the behavior of VP and IP on other architectures, we also consider a programmable Graphic Processing Unit (GPU), THEIA [40]. THEIA features a multi-core architecture, and uses a ray casting approach for rendering. Every core in THEIA runs a local copy of the shader code, and has a pipelined SIMD unit, capable of performing fixed-point arithmetic on 3D vectors. Each core includes *instruction entry point*, *fetch*, *decode*, *execute*, and *memory* stages in conjunction with a control unit. Similar to Fig. 2, the number of failed paths for each stage of a THEIA's core is shown in Fig. 3. As shown, VP display no failure with a clock cycle of 3.2 ns, while the *execution* stage faces high number of failed paths. In fact, the *execution* stage needs 14% more guardbanding compared to other stages. In comparison to LEON3, the *execution* stage of a THEIA's core imposes higher guardbanding, since it performs vector fixed-point operations which involve more complex units than the scalar integer operation in LEON3.

Indeed, several researches show that *execution* stage is critical not only for in-order or SIMD architectures, but also for various VLIW and out-of-order architectures [41]–[43]. For instance, despite the prior-art assumption that the register file defines the clock frequency of a clustered VLIW processor, the realistic physical layout experiments for an 8-issue-slot VLIW pipeline show that it is the *execution* stage and its bypass network that limits the clock speed [41]. Although a clock frequency speedup is achieved by partitioning a single cluster into two clusters (thus a shorter bypass network); in subsequent clustering there is a steady decrease of the bypass network delay, hence the delay of functional units is a deciding factor in clock frequency since it occupies up to 85% of the clock period in an 8-cluster VLIWs [41]. M. Ozawa *et al.* [42] also propose a cascade ALU architecture for out-of-order processors, in which the critical path lies in the ALU. Similarly, the ALU delay also determines the cycle time of a low-power out-of-order design [43].

3.2 Variation-Aware Statistical STA

Unlike the traditional STA, variation-aware Statistical Static Timing Analysis (SSTA) takes into account the actual distribution of the physical parameters instead [47]. As a result, the calculated slack distributions accurately reflect the true results obtained in silicon resulting in less pessimism in the analysis. The variation-aware SSTA is suitable for IP analysis where the processor does not need 100% timing correctness in case of the worst process variation. Our results illustrate the value of variation-aware SSTA. Fig. 4 distinguishes the data arrival time of the *execution* stage of LEON3 for two operands using the worst-case STA versus the variation-aware SSTA. The operating condition is set for (0.81 V, 125°C), and the process parameter for STA is set for the Slow NMOS-Slow PMOS (SS), while this parameter for variation-aware SSTA varies based on the process parameter variations supported by state-of-the-art commercial tools.

To perform an accurate *design time* SSTA, we use the variation-aware timing analysis engine of *Synopsys PrimeTime VX* [47], leveraging characterized parameters of 45 nm variation-aware TSMC libraries [48] derived from first-level process parameters by principal component analysis (PCA). PCA is a mathematical procedure that simplifies a data set by transforming a number of correlated parameters into a smaller number of uncorrelated parameters. After parasitic extraction from the physical design data, the die-to-die (D2D) and within-die (WID) process parameter variations are injected as normal distributions with zero means and standard deviations of $\sigma_{\mathrm{D2D}} = 5\%$ and $\sigma_{\mathrm{WID}} = 6.4\%$ [49]. Therefore, we change the variation components and analyze the delay variations with a given set of accurate variability models from commercial libraries [48], which are certainly more accurate than commonly used 'in-house model' extracted from predictive technology models [50]. As shown in Fig. 4, the data arrival time of the operands in the *execution* stage based on STA is upto 40% greater than the variation-aware SSTA due to pessimistic process parameters. For the fixed operating condition, STA results in 19% greater data arrival time on average compared to the variation-aware SSTA for the entire integer pipeline. These results set a baseline for the improvements from adaptive guardbanding techniques that raise the level of abstraction at which variability is addressed.

4 Probabilistic Applications

In moving from circuits to applications, we find a greater tolerance to failures simply because there is more contextual information available for recovery mechanisms to use. Given the increasing parallelism from hardware, the computer systems researchers have attempted to classify applications into core algorithmic categories such as RMS [22] that not only points to the structure of the computation but also a guidance on the degree of tolerance to individual data or even

computational errors. While a comprehensive framework for classifying applications according to degree of data and control tolerance to error and variation is still an area of active research [23], adaptive guardbanding proposed here does bring us a step closer to tie the mitigation of PVT guardbands to the type of applications.

4.1 Analysis of Adaptive Guardbanding for Probabilistic Applications

For probabilistic applications, the key idea is to guarantee the error-free operations of the paths that are vital for ensuring timing of the VP, while reducing the cost of guardbanding for the rest of the paths (IP). The adaptive guardbanding for the probabilistic applications dynamically decides on the cycle time based on the operating conditions, while guaranteeing the accuracy of the fidelity metric above a user-defined threshold (U_T) for the acceptable output. Timing error due to the delay variation in IP may alter the vector of elastic outputs (O_E). A fidelity metric of a probabilistic application P, $\mathrm{F}_P(I, O_E)$ is associated with its input I and the corresponding O_E. The execution of application P with input instance I in the presence of delay variation is acceptable iff $(\mathrm{A})^{\wedge}(\mathrm{B})^{\wedge}(\mathrm{C})$. The predicates (A)–(C) are defined as

$$\begin{aligned}&(\mathrm{A})\mathrm{F}_P(I, O_E) \geq \mathrm{U}_\mathrm{T},\\ &(\mathrm{B})\ \neg\exists\ path_j \in VP \mid \mathrm{Slack}_{\mathrm{STA}}(path_j) < 0,\\ &(\mathrm{C})\ \neg\exists\ path_k \in IP \mid \mathrm{Slack}_{\mathrm{SSTA}}(path_k) < 0.\end{aligned}$$

Specifically, the cycle time, for every operating condition is adjusted in such a way to satisfy that all paths in VP always meet the setup time of flip-flops even in the worst-case process parameter variation using STA (B); and that the paths in IP will not miss the setup time of any connected flip-flop, in a statistical sense, using the variation-aware SSTA (C). These two criteria guarantee the semantically correct execution of application P, e.g., an *addition* instruction is always executed as an *addition* instruction but it might generate inaccurate results, in case of large variations. To satisfy (A), the fidelity metric has to be greater than the U_T, thus guarantees the acceptable accuracy form the applications' point of view. For a given application P, the application writer is responsible to tune the acceptable threshold based on the end user's requirements [17], [21].

The adaptive guardbanding dynamically sets the cycle time to meet (A)–(C) requirements to mitigate the inter-corner variations for a given operating condition. The assigned cycle time guarantees the error-free operation of VP even in the worst-case process parameters variation, certified by STA. However, the guardband provided by the adapted cycle time cannot guarantee 100% timing correctness of IP within the *execution* stage in case of absolute worst-case combination of process parameters. This might cause inaccuracy in the result of the executed instruction. If the executed instruction produces O_E (thus affecting the fidelity metric), the predicate (A) guarantees that the program can produce an acceptable fidelity metric. On the other hand, if the executed instruction is a critical instruction, the proper application-level correctness techniques [17] is applied to identify the critical control flow instructions. The critical instructions are statically duplicated during compile time which guarantees the error-free execution in a fix operating condition.

TABLE 1
Effectiveness of Adaptive Guardbanding for the Probabilistic Applications under Dynamic Variations

Volt. (V)	Temp. (°C)	Cycle Time (ns)	The worst slack of *execution* (ns)			
			mean	std-dev	p99	p01
0.99	-40	0.80	0.247	0.028	0.325	0.196
0.81	-40	1.35	0.400	0.057	0.565	0.302
0.81	125	1.32	0.451	0.076	0.638	0.281
...	...	...	...	...	...	...

We use SSTA methodology (discussed in Section 3.2) to analyze the effect of within-die and die-to-die process parameters variations. It dynamically sets the cycle time depends to the operating conditions as shown in Table 1. For example, as soon as detecting the operating condition at (0.99 V, $-40°\mathrm{C}$), the adaptive guardbanding decreases the cycle time from 2.5 ns, calculated by the worst-case STA for (0.81 V, $0°\mathrm{C}$, SS), to 0.8 ns. This cycle time of 0.8 ns meets all timing requirements of VP, and at the same time provides positive slack for the *execution* stage in a statistical sense. As shown in the fourth column of Table 1, based on SSTA, the adaptive guardbanding strategy works well even with die-to-die and within-die process variation, while the paths are experiencing a full swing for voltage and temperature, and provides the positive slacks for the slowest path of the *execution* unit. Furthermore, the 1st percentile (p01) values are quite far from the zero slack, thus implying that the probability that actual slack of the path in the *execution* stage will be less than or equal to p01 value is 0.01.

The probability density functions of the slack value of top 1,000 critical paths within the *execution* stage are analyzed, at three operating conditions using the assigned cycle time in Table 1. All slack values are always positive when pipeline experience a full swing in voltage ($\Delta\mathrm{V} = 0.18\mathrm{V}$) and temperature ($\Delta°\mathrm{C} = 165°\mathrm{C}$). If an IP path in the *execution* stage is faced with the worst-case combination of process parameters, and does not meet the timing requirement, the effects of such variations may manifest itself as an error in a bit of the output vector. Depending upon the positional significance, a probabilistic application may tolerate errors in low-order bits [21], [19]; for the high-order bits of the *execution* stage, there is little likelihood of having errors even in a full swing of the operating conditions, as the smallest p01 slack values are quite positive: 0.22 ns/0.37 ns at (0.99 V, $-40°\mathrm{C}$)/(0.81 V, $125°\mathrm{C}$). The application writer can trade-off between the end user's accuracy requirements versus the cost of guardbanding using profiling and tuning mechanisms, thus satisfying predicate (A). The trade-off between the cycle time and the probability of having a failure in the *execution* paths is shown in Fig. 5. As shown, a higher cycle time results in lower probability of failure and thus a lower timing error rate. Therefore, the desired cycle time can be extracted to match with the tolerable error of the application. If the tolerable error of the application changes over different phases of the application, the policy of applying the adaptive guardbanding can be reprogrammed accordingly during the execution of the application.

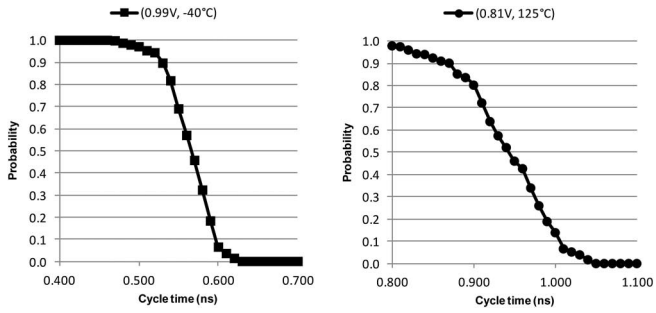


Fig. 5. Trade-off between the cycle time and the probability of having a failure in the *execution* stage.

5 Intolerant Applications

5.1 Sequence-Level Vulnerability (SLV)

Unlike the probabilistic applications, applications in general do not have such inherent algorithmic and cognitive tolerance thus even a single bit error in the execution unit could crash a program. We consider this class of applications as intolerant applications that require complete numerical correctness. Intolerant applications cover most of the general purpose applications, and even those probabilistic applications that there is no domain expert to define and analyze their fidelity metrics parameters. Therefore, the adaptive guardbanding for the intolerant applications has to guarantee 100% timing correctness for VP as well as IP. To alleviate such expensive constraint imposed by the intolerant programs, we have earlier defined the notion of Instruction-Level Vulnerability (ILV [12]) to dynamic voltage and temperature variations in order to expose and use variation in architectural/compiler optimizations. Equation (1) defines ILV as a function of current operating voltage and temperature (V, T), and the corresponding class of an instruction (inst_i) determined by partial function of ϕ. ILV is computed as the number of cycles with a failed path over the total Monte Carlo simulated cycles for the inst_i in [12]

$$ILV = \Im(\phi(\mathrm{inst}_i), V, T)$$
$$\phi(\mathrm{inst}_i) = \begin{cases} ClassI & \text{if } \mathrm{inst}_i \in \text{ALU instructions} \\ ClassII & \text{if } \mathrm{inst}_i \in \text{MEM instructions} \\ ClassIII & \text{if } \mathrm{inst}_i \in \text{HW MUL/DIV instructions.} \end{cases} \quad (1)$$

In fact, ILV data in [12] partitions integer SPARC V8 ISA (except control instructions) into three classes: *ClassI* consists of ALU instructions; *ClassII* covers all memory (MEM) instructions; and *ClassIII* has hardware multiply/divide (MUL/DIV) instructions. As shown in Equation (2), ILV indicates that the classes of instructions have different levels of vulnerability to dynamic variations depending on the way they exercise the non-uniform critical paths across the various pipeline stages. For instance, the hardware MUL/DIV instructions have a higher vulnerability in comparison to MEM instructions

$$\forall(V,T):\ ILV(ClassI,V,T) \le ILV(ClassII,V,T) \le ILV(ClassIII,V,T). \quad (2)$$

TABLE 2
Extracted High-Frequent Sequences of Instructions

Seq. #	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Inst. $_i$	ld	st	ld	ld	ld	st	ld	call	ld	call
Inst. $_{i+1}$	ld	st	bz	bz	st	ld	ld	st	ld	st
Inst. $_{i+2}$	ld	st	sub	and	ld	st	bz	st	sub	ld
Seq. #	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
Inst. $_i$	bz	ld	sub	and	sub	and	and	sub	sub	ALU
Inst. $_{i+1}$	st	and	bz	bz	add	add	sub	sub	and	ALU
Inst. $_{i+2}$	ld	st	bnz	bnz	bz	bz	bz	bz	bz	ALU

ILV does not cover the control instructions, because the characterization of a control instruction itself is meaningless unless it is considered within a *sequence* of instructions that affect the control instruction. Hence, we extend the notion of ILV; we introduce the notion of *Sequence-Level Vulnerability* (*SLV*) to expose dynamic variation in Equation (3). Different sequences of instructions exercise the critical paths of the pipeline differently resulting in various levels of vulnerability. The vulnerability of a sequence of instructions (seq_i) varies based on the class of instructions that it contains. *SLV* is also a function of current operating voltage and temperature to capture inter-corner dynamic variations. Therefore, *SLV* reflects the manifestation of variability-induced timing errors in the specific software context which is a sequence of instructions

$$SLV = \Im(\varphi(\mathrm{seq}_i), V, T). \quad (3)$$

5.2 SLV Characterization

To avoid an exponentially growing number of sequences for evaluations of *SLV*, the high-frequency sequences are extracted from various type of applications. We have profiled a large set of general purpose benchmarks containing 32 different applications, include MiBench [52], Parsec [53], Scimark2 [54], MediaBench [55], and CoreMark [56] benchmarks. The binaries of applications were dynamically instrumented. This allows us to extract the high-frequent sequences of the instrumented instructions as well as their operands distribution for the memory, and ALU instructions. This operands distribution helps to create the realistic values for the operands of the instructions. To distinguish sequences, a window of three instructions is considered since there are three stages before reaching the *execution* stage of LEON3. Then, for the sake of illustration, the top 20 high-frequent sequences are considered for the *SLV* analysis that are shown in Table 2.[1] After the sequence extraction, a sequence generator (see Fig. 8) applied Monte Carlo method for each of top 20 sequences, utilizing the operands distribution instrumented from the aforementioned benchmarks. Therefore, large samples of high-frequency sequences for SPARC ISA have been generated, including ALU, MEM, and control instructions.[2]

Then, to accurately evaluate *SLV* under different operating conditions, these sequences were fed to the post-layout simulations where the delay of the layout implementation of the processor is back-annotated. Therefore, *SLV* is calculated for

1. We later show our method is not limited to the top sequences and a sequence with a length of three instructions ($L = 3$).

2. The rest of ISA needs the floating-point and coprocessor units which are not available neither in our core nor in [6].

every individual sequence under a full range of operating conditions and cycle times to enable use of dynamic variations on sequences of instructions. To evaluate *SLV*, seq_i is run through the pipeline while varying the operands of the instructions using the following algorithm:

$$\begin{aligned}&\text{For } \mathrm{seq}_i \in \text{list of high-frequent sequences}\\ &\quad \text{For}(V,T) \in \{(0.72\mathrm{V}, -40^{\circ}\mathrm{C}), \ldots, (0.99\mathrm{V}, 125^{\circ}\mathrm{C})\}\\ &\quad\quad \text{For } Cycle\_Time \in \{1.0\mathrm{ns}, \ldots, 3.0\mathrm{ns}\}\\ &\quad\quad\quad \text{For } operands \in \text{list of operands}\\ &\quad\quad\quad\quad \text{Compute } SLV(\mathrm{seq}_i, V, T, Cycle\_Time)\end{aligned}$$

The *SLV* for each seq_i at the operating condition (V, T) with *Cycle_Time* is quantified in Equation (4), where N_i is the total number of clock cycles in Monte Carlo simulation of seq_i with random operands; and $\mathrm{Violation}_j$ indicates if there is a violated stage at clock cycle_j or not. In other terms, *SLV* is defined as the total number of violated cycles over the total simulated cycles for the seq_i. If any of the six stages have one or more violated flip-flop at clock cycle_j, we consider that stage as a violated stage at cycle_j since there is at least one activated critical path for seq_i at cycle_j that is slow enough to miss the setup time of a flip-flop. Intuitively, if seq_i runs without any violated path, *SLV* is zero; on the other hand, *SLV* is one if for every cycle seq_i faces at least one violated path in any stage

$$\begin{aligned}&SLV(\mathrm{seq}_i, V, T, Cycle\_Time) = \frac{1}{\mathrm{N}_i}\sum_{j=1}^{\mathrm{N}_i}\mathrm{Violation}_j,\\ &\mathrm{Violation}_j = \begin{cases} 1 & \text{If any stage violates at cycle}_j \\ 0 & \text{otherwise}\end{cases}.\end{aligned} \quad (4)$$

Fig. 6 shows the *SLV* values of the top sequences under a wide range of voltage and temperature variations while the cycle time is finely varied (steps of 10 ps). The *SLV* values are *0* during the long cycle times, as the cycle time decreases the *SLV* values increase towards *1* because the sequences experience higher timing violations. Let us first examine the behavior of the sequences under the full range of temperature variation (Fig. 6(b) and (c)). At the temperature of 125°C, all sequences have a *SLV* of *0* with clock cycle 1.35 ns. By decreasing the cycle time beyond 1.33 ns, $seq_1 - seq_{19}$ start to incur the timing violation as their *SLV* values increase, while seq_{20} is displaying a *SLV* of *0* until decreasing the cycle time to 1.28 ns. This trend also persists under $\Delta\mathrm{T} = 165^{\circ}\mathrm{C}$ temperature fluctuation with a shift in cycle time (Fig. 6(c)). As shown, these sequences are partitioned into two classes based on the *SLV* values. The $seq_1 - seq_{19}$ have higher within-corner *SLV* values, while the seq_{20} has lower within-corner *SLV* values.

Let us now examine the *SLV* values under dynamic voltage variations (Fig. 6(a) and (b)). A similar pattern of within-corner *SLV* variations is observed: the $seq_1 - seq_{19}$ show higher *SLV* values compared to the seq_{20} at equal cycle times. This classifies the $seq_1 - seq_{20}$ into two classes of sequences: *ClassI* and *ClassII*. As defined in Equation (5), *ClassI* is a sequence of instructions of length *L* in which every instruction has an ILV class of *ClassI*. In other words, when a sequence of instructions is composed of only ALU instructions, the sequence is classified as *ClassI*; otherwise it is classified as

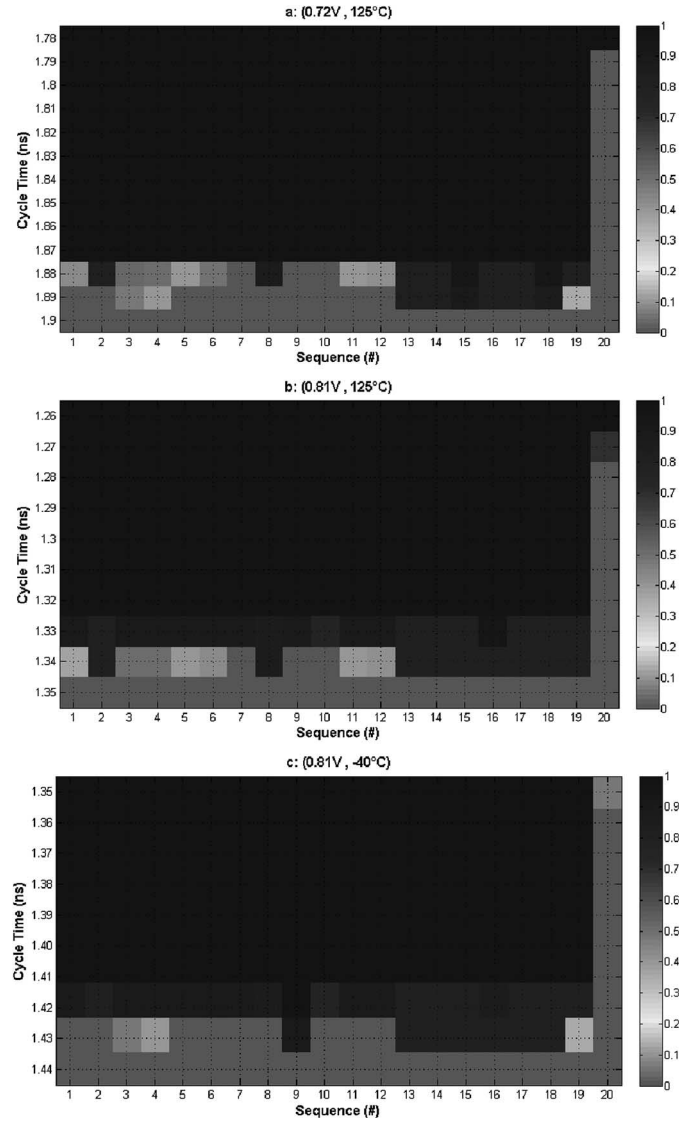


Fig. 6. Intra-corner SLV to dynamic variations ($\Delta\mathrm{T} = 165^{\circ}\mathrm{C}$ and $\Delta\mathrm{V} = 0.09\mathrm{V}$); a) (0.72 V, 125°C), b) (0.81 V, 125°C), c) (0.81 V, -40°C).

ClassII. Therefore, an instruction within the sequence of *ClassII* can be any instruction, including MEM, MUL/DIV, and various control instructions. For every operation condition (V, T), *ClassI* has a lower *SLV* (thus needs lower guardband) in comparison to *ClassII*

$$\begin{aligned}&\forall(V,T,\mathrm{seq}_i): SLV(ClassI,V,T) \le SLV(ClassII,V,T), \text{s.t.},\\ &\varphi(\mathrm{seq}_i) = \begin{cases} ClassI & \forall \mathrm{inst}_j \in \mathrm{seq}_i | \phi(\mathrm{inst}_j) = ClassI, 2 \le j \le L \\ ClassII & \text{otherwise}\end{cases}.\end{aligned} \quad (5)$$

Based on our analysis for the high-frequent sequences, as shown in Fig. 6, the seq_{20} is classified as *ClassI*, while the $seq_1 - seq_{19}$ are among *ClassII*. The seq_{20} has a lower *SLV* compared to all sequences in *ClassII*; since its instructions do not involve the critical paths of the memory and control (integer code conditions) components. Thus, we see that the *SLV* value of the two classes of the sequences at the same corner and with the same cycle time is not equal because their instructions do not uniformly exercise the various critical

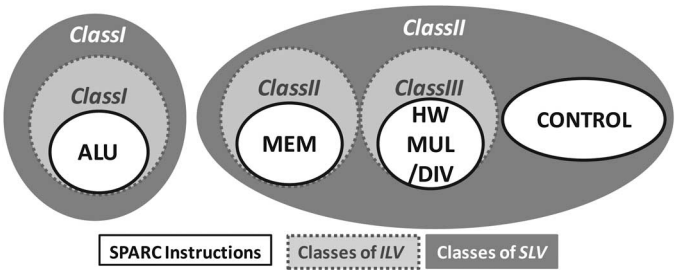


Fig. 7. ILV and SLV classification for integer SPARC V8 ISA.

paths of the pipeline. We know that the vulnerability of instructions is not uniform [12]. Sequences in *ClassII* need higher guardbands in comparison with *ClassI*, mainly because in addition of ALU's critical paths, the critical paths of memory are also activated for the load/store instructions as well as the critical paths of integer code conditions for the control instructions. As a result, in the same corner, sequences in *ClassI* run faster, thanks to their all ALU instructions which only exercise critical paths of the ALU component.[3] Fig. 7 summarizes ILV and *SLV* classification.

This intra-corner *SLV* enables the adaptive guardbanding to set the cycle time for each class of sequences accordingly, and thus eliminate the conservative guardbands across sequences up to 6%. Therefore, for intolerant applications, the adaptive guardbanding adjusts the cycle time depending upon the classes of the sequence, and the current operating conditions to make sure that the processor runs at the fastest speed compatible with both current hardware and software conditions. We classify any non-characterized sequence out of the analyzed high-frequent sequences as *ClassII*, thus it will have appropriate timing guardband in case of activation of the critical paths of non-ALU components. Relaxing the guardband can also be applied to any sequence of *ClassI* with a length of two ALU instructions ($ClassI_{L=2}$) or more (N) ALU instructions stream ($ClassI_{L=N}$). These chains of ALU instructions exercise the critical paths within only ALU component, therefore, for a given operating condition as shown in Equation (5), the *SLV* values of $ClassI_L$ for $L \in 2, 3, \ldots, \mathrm{N}$ are equal. This classifies ALU sequences into the same class of the sequences with consistency across a wide range of corners.

6 Adaptive Guardbanding

We propose a guardbanding technique that dynamically decides on the cycle time based on the *Application's Type*, the *Instruction Sequence*, and the operating conditions (V, T), to maximize performance. To ensure necessary observability, our approach employs on-chip low-overhead operating condition monitors using CPM [9]. POWER7 results show that five CPMs per each core are sufficient to finely capture PVT variation [10]. For controllability, a fast adaptive clocking circuit consisting of three Phase-Locked Loops (PLLs) is leveraged. Each PLL is running at independent frequencies, and a multiplexer quickly switches between them in a single cycle [11], [44]; therefore ultra-fast frequency changes are possible and PLL lock time is not an issue. This is well suited to mitigate the inter-corner dynamic variations where the timing guardbanding across corners are far apart. To mitigate the intra-corner guardband between the two classes of sequences, a finer clock speed adaptation is required which can be supported by an all-digital PLL. For instance, [44] proposes an all-digital PLL that provides multiple equally spaced clock phases with a small tuning step size of a few picoseconds; these phases are switched in a glitch-free reverse switching scheme. A phase switching frequency division architecture is also used to generate sub-integer division ratios and thus a larger variety of output frequencies [45]. These circuits techniques support very fast adaptation of the clock speed of the processor in immediate response to changes in the operating corners, various sequences of instructions, and the type of applications. The adaptive guardbanding adjusts the *Cycle_Time* as defined in Equation (6)

$$Cycle\_Time = \Im(Application's\ Type, Instruction\ Sequence, V, T). \quad (6)$$

Where *Application's Type* is probabilistic or intolerant; *Instruction Sequence* is the type of sequence which is either *ClassI* or *ClassII*; V and T are discretized current operating conditions reported by on-chip CPM sensors; $\Im$ is represented by a Programmable LookUp Table (PLUT) as shown in Table 3. The PLUT is a fully combinational module in the pipeline.[4] It is programmable through the memory-mapped I/O in arbitrary epochs of the post-silicon stages. The PLUT is connected to CPM (for monitoring the current operating condition (V, T)), the *fetch* stage (for monitoring the *Instruction Sequence*), and the single-cycle adaptive clocking module (for setting the $Cycle\_Time$). The *Application's Type* is also set at the start of running the application via memory-mapped I/O. The adaptive guardbanding monitors these four parameters every cycle, and then sends corresponding commands to the clock speed adjustment circuit to make sure that processor always runs at the fastest speed compatible with these conditions.

TABLE 3
PLUT for Adaptive Guardbanding

Application's Type	Instruction Sequence	Voltage (V)	Temperature (°C)	Cycle Time (ns)
Probabilistic	–	0.99	0	0.78
Probabilistic	–	0.81	125	1.32
Probabilistic	–	0.72	125	1.55
Intolerant	*ClassII*	0.81	-40	1.44
Intolerant	*ClassI*	0.81	-40	1.36
Intolerant	*ClassI*	0.72	125	1.80
…	…	…	…	…

As shown in Table 3, there is no intra-corner cycle time adaptation for the probabilistic application. The within-corner correct execution is guaranteed by static duplication of the critical instructions which is the application-aware version of the multiple-issue instruction replay [6]. Therefore, for the probabilistic application we do not require an online hardware recovery unit, and avoid the frequent changing of the cycle time within an operating corner.

3. ALU does not include the hardware multiply and divide units.

4. Note that PLU can be characterized and then optimized during *design time* stage depending upon the range of operating conditions and application's type.

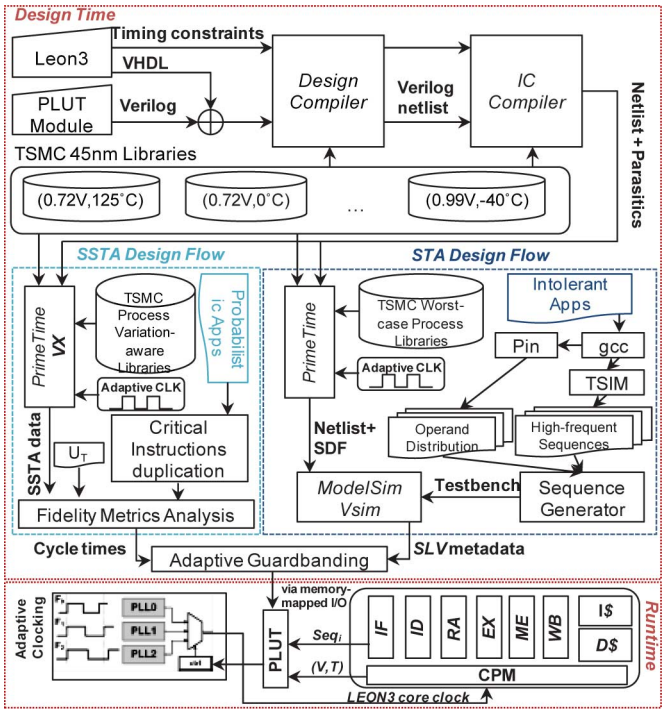


Fig. 8. Methodology for the adaptive guardbanding.

In our experiments, for characterization of the PLUT, we have used six sign-off operating corners available on an advanced real-life technology library [48]. PLUT conservatively matches a surrounding operating condition if the discretized reported operating condition does not appear in the PLUT. Note, this is conservative for few points in the PLUT, but will converge to ideal, while still being safe, if semiconductor fabrication process provides more characterized operating corners. Furthermore, for the intolerant applications, the adaptive guardbanding considers the worst-case process variation, and also considers a conservative guardband (as safe as *ClassII*) on the non-characterized sequence of instructions (sequences out of $seq_1 - seq_{20}$), thus guarantees 100% numerical correctness for the intolerant applications. As shown in Table 3, the PLUT assigns different cycle times to various types of applications at the same operating condition. Inherent resiliency of the probabilistic applications indicates that these can tolerate inaccuracies, while the intolerant applications do not accept such inaccuracies. Therefore, when running an intolerant application the sufficient guardbanding is guaranteed for IP as well.

7 Experimental Results

The experimental methodology for STA, and the variation-aware SSTA are described using Fig. 8 that shows both *design time* and *runtime* flows. During the *design time* analysis, the open-source synthesizable VHDL code of LEON3 [34] and Verilog description of the PLUT module have been synthesized with the TSMC 45 nm technology library, the general purpose process. The synthesized core enables the variation analysis of paths of the integer parallel pipeline unit, as well as the L1 instruction cache (I\$) and the L1 data cache (D\$), unlike the resilient core [6] that only considers the integer unit. The front-end flow with normal V_{TH} cells has been performed using *Synopsys Design Compiler* with the topographical features enabled, while *Synopsys IC Compiler* has been used for the back-end. The design is optimized for performance with the tight timing constraints, e.g., the clock period of 1.2 ns. For SSTA, the sign-off stage has been made with variation-aware timing analysis of *Synopsys PrimeTime VX*, leveraging characterized parameters of TSMC 45 nm variation-aware libraries discussed in Section 3.2. The dynamic variations are also analyzed utilizing the six accessible TSMC characterized sign-off corners [48]. Finally, for the post-layout simulations *Mentor Graphics ModelSim* is employed.

At the runtime, in every cycle, the PLUT module sends the desired cycle time to the adaptive clocking circuit utilizing the characterized *SLV* of the current sequence and the operating condition monitored by CPM. For detecting the current sequence, the PLUT looks at a window of three instructions (available on IF, ID, RA stages), thus it detects the class of the current instructions sequence before they reach the *execution* stage (the stage that needs more guardbanding as shown in Fig. 2). The previous stages (IF, ID, RA) are in a safe guardband, thus they will not have any failure if a sequence of *ClassI*/*ClassII* is running while the cycle time is set for a *ClassII*/*ClassI*. If the pipeline architecture does not have enough stages before the *execution*, the prefetch buffer [51] can be monitored instead. By detecting changes in the class of sequences, the single-cycle adaptive clocking circuit sets the core frequency accordingly. If an adaptive clocking circuit has long-latency clock switching, the PLUT can look ahead of a prefetch buffer coupled with phase prediction techniques to be able to decide about the desired core frequency in advance. Note that the core consists of the integer pipeline, L1 I\$, and L1 D\$ that are clocked by a single clock domain. Communication with L2 caches and uncore part can be done via globally asynchronous, locally synchronous interconnection supporting synchronization across multiple clock domains [37].

7.1 Effectiveness of Adaptive Guardbanding

In this section, we investigate the effectiveness of our adaptive guardbanding technique when executing real word applications.[5]

7.1.1 Probabilistic Applications

As probabilistic applications, we have selected multimedia benchmarks from MiBench and MediaBench suites: H264 is a video decoder while Libmad is a MP3 decoder; Susan is an image recognition program; DCT, Huffman coding and Ycc2rgb are important kernels in the JPEG decoder; GSM implements a decoder for the GSM communications standard, and LDPC is a linear error correcting code. The appropriate fidelity metric analysis and application-level correctness technique based on [17] are performed to identify the critical control flow instructions of these applications. Then, the critical instructions are statically duplicated during compile time. Finally, the adaptive guardbanding determines the cycle time based on the given error probability 0.01% which can guarantee the acceptable fidelity metrics [17].

In the traditional worst-case design, the maximum throughput of applications is limited by 400 MIPS (million

5. For those applications that have encoder and decoder parts, we consider their back-to-back executions.

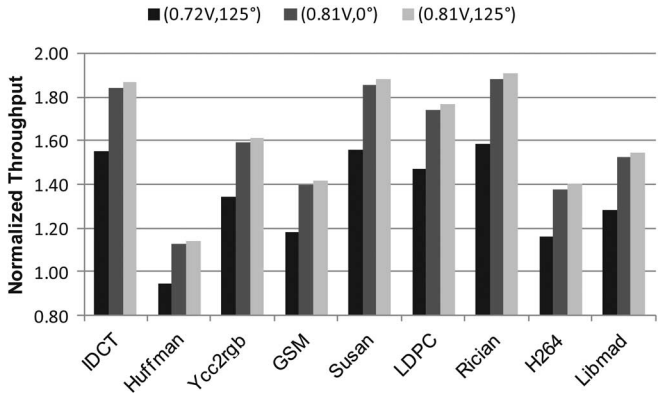


Fig. 9. Normalized throughput improvement by utilizing SSTA compared to the worst-case design for probabilistic applications.

instructions per second), analyzed by the worst-case STA in Section 3.1. Fig. 9 shows the normalized throughput of the applications in various operating conditions, covering $\Delta V = 0.09$ V dynamic voltage variation and $\Delta T = 125°C$ temperature variation. In comparison with the worst-case design, the adaptive guardbanding changes the throughput of these applications from $0.95\times$ to $1.9\times$ depends to the current operating condition. Throughput of Rician is increased up to $1.9\times$ at (0.81 V, 125°C). On the other hand, throughput of Huffman coding at the operating condition of (0.72 V, 125°C) is degraded by $0.95\times$ because 69% of its instructions are the critical control flow instructions which are duplicated, and cancel out the benefit of faster execution of the total instructions. On average, the throughput of these applications is enhanced by $1.52\times$. This shows that utilizing SSTA and adapting to the operating conditions highly surpasses the traditional worst-case STA, and also hides the overhead of the critical instructions duplication.

7.1.2 Intolerant Applications

For the intolerant applications, we have selected applications from six categories of MiBench, each suite targeting a specific area of the embedded market, including automotive, consumer devices, office automation, networking, security, and telecommunications. In addition, we have also considered EEMBC AutoBench [57] suite of benchmarks, suitable for embedded processor in automotive, industrial, and general-purpose applications. Without loss of generality, every probabilistic application can be considered as an intolerant application and benefits from *SLV* utilization if there is no domain expert to define and analyze its fidelity metric. Fig. 10 shows the percentage of sequences of *ClassI* with various lengths of ALU instructions, $L \in 2, 3, \ldots, 7$, during execution of the intolerant applications. For instance, $ClassI\_L = 2$ shows the percentage of sequences that have exactly two consecutive ALU instructions, $ClassI\_L = 3$ represents sequences with just three consecutive ALU instructions, and so on. The compiler[6] optimizes the applications codes with *-O3* optimization option; and then the applications are profiled during execution using TSIM [58], a cycle-accurate instruction-level simulator. Fig. 10(a) shows on average 26% of the

6. GNU Compiler Collection, version 3.4.4, with floating-point, mul/div emulation.

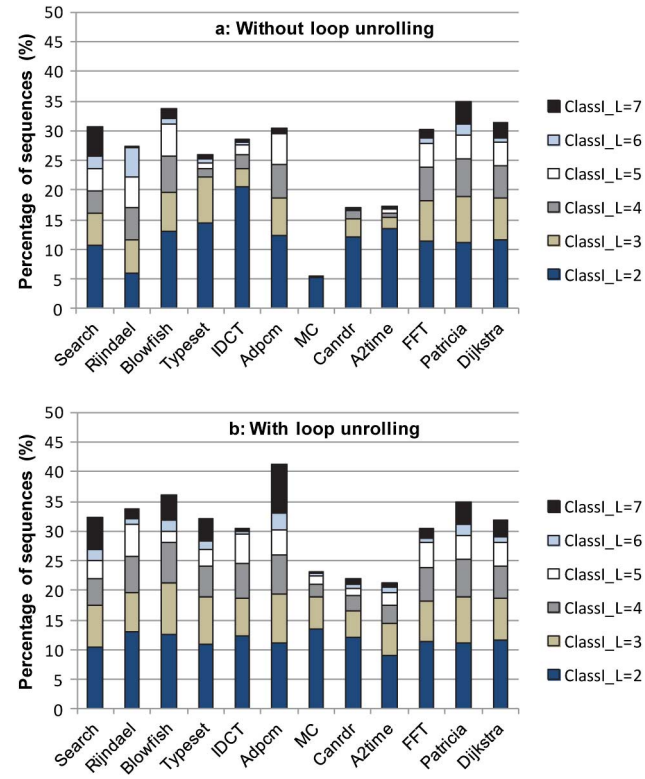


Fig. 10. Percentage of sequences of *ClassI* during program execution: a) without loop unrolling technique; b) using loop unrolling technique.

total executed sequences belong to *ClassI*, while the remaining sequences belong to *ClassII*. Patricia has the maximum number of sequences of *ClassI*, 35%. The adaptive guardbanding technique with the sequence detector of three instructions benefits from the sequences of *ClassI* with a length of 3 or more instructions.

Fig. 10(b) shows the percentage of sequences of *ClassI* when the compiler utilizes loop unrolling technique. Loop unrolling is a loop transformation technique that attempts to increase speed of a program by reducing instructions that control the loop. It increases the number arithmetic instructions with regard to the memory and control flow instructions, at the expense of register pressure and program size. Therefore, applying the loop unrolling produces a longer chain of ALU instructions, and as a result the percentage of sequences of *ClassI* is increased up to 41% and on average 31%. Hence, the adaptive guardbanding benefits from this compiler transformation technique to further reduce the guardband for sequences of *ClassI*. Considering the sequence detection with a length of three instructions, the adaptive guardbanding reduces the cycle time for 20% of the executed sequences on average (up to 30% for Adpcm). Note that the adaptive guardbanding technique also reduces the guardband for the other sequences of *ClassI* with a longer length of three instructions, since each sequence of *ClassI* with L instructions is composed of two consecutive sequences with a length of L-1 instructions, considering the overlap between the two sequences.

Table 4 lists the maximum and the average throughput improvement of the adaptive guardbanding technique utilizing the loop unrolling during compilation phase of the intolerant applications. The throughput improvement is

TABLE 4
Throughput Improvement of the Intolerant Applications Utilizing the Adaptive Guardbanding with Loop Unrolling.

Throughput improvement (×)	only *SLV* (intra-corner)		*SLV* + inter-corner	
	max	average	max	average
(0.72V,125°)	1.04	1.03	1.36	1.35
(0.81V,0°)	1.06	1.05	1.80	1.78
(0.81V,125°)	1.05	1.05	1.88	1.87

evaluated across various operating conditions. The second and the third columns of Table 4 show the maximum and the average throughput improvement of the applications utilizing *SLV* only within a fixed operating corner. Thus, the applications benefit from the higher rate of execution of the sequences of *ClassI* accomplished by the loop unrolling method. The last two columns show the maximum and the average normalized throughput (the worst-case design is the baseline) improvements utilizing *SLV* and inter-corner adaptation. In comparison with the worst-case design, the adaptive guardbanding enhances the throughput of these applications by a factor of $1.35\times$ to $1.88\times$ depending upon the current operating condition. This shows that utilizing the operating corner monitors and the online *SLV* coupled with offline compiler techniques can result in a significant throughput improvement for general-purpose applications where there is strict requirement on computational accuracy.

We compare our *SLV* technique (without the loop unrolling) with the code transformation technique proposed in [33] which pads the instructions sequence with a NOP instruction. The NOP padding eliminates the critical path activation for the forwarding paths of a processor for a Read-After-Write (RAW) register dependency. In other words, the result is no longer forwarded directly from the *execution* stage, it instead is forwarded a cycle later from the pipeline register in the *memory* stage. For comparison, we have identified the code sequences with a RAW register dependence and padded them with NOP instruction. Those NOP padded sequence are clocked as fast as the *ClassI*. The authors in [33] assume that they can clock that sequence $2.15\times$ faster than the typical frequency of a processor, while Intel shows that in the resilient processor the clock can increase up to $0.16\times$ in a fixed operating corner [6]; our results in Section 5.2 also indicates that intra-corner clock guardbanding for various sequences is bounded by $0.06\times$. Fig. 11 shows the normalized (baseline is [33]) throughput of our adaptive guardbanding utilizing *SLV* by adapting the cycle for dynamic operating conditions and different classes of the sequences. On average, our technique achieves $1.65\times$ higher throughput because [33] imposes one extra cycle for executing the NOP instruction, and does not adapt to the operating conditions. Fig. 12 shows the energy overhead of the NOP padding across various operating corners. It imposes 74 nJ to 564 nJ energy overhead, depending upon the number of NOP instructions and the current operating condition.

Multi-instruction code substitution, as another code transformation techniques in [33], is not applicable for an embedded RISC machine where there are almost no alternatives for representing an equivalent set of instructions, unless paying the expenses of intrusive pipeline modification, ISA extension, and leveraging co-processors. Nevertheless, there is a considerable performance and energy penalty for replacing a multi-instruction sequence with an equivalent set of instructions [28].

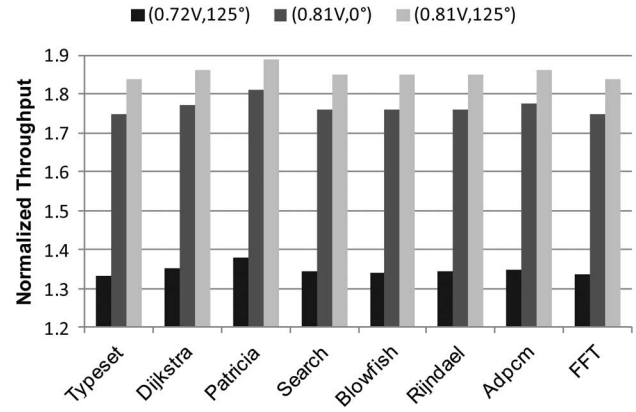


Fig. 11. Normalized throughput improvement utilizing *SLV* compared to [33] for the intolerant applications.

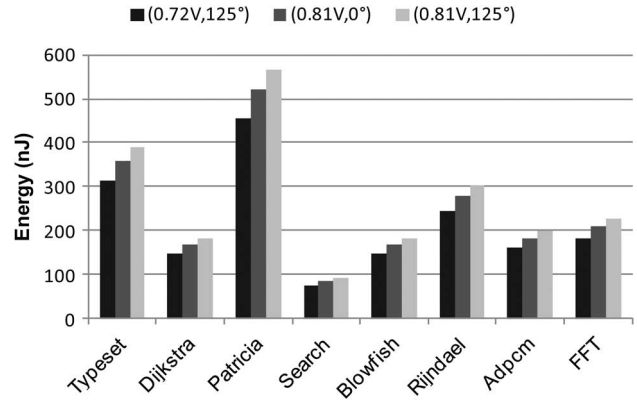


Fig. 12. Energy overhead of NOP padding [33] across corners.

The common strategy in circuit techniques [6], [7] is to allow the timing errors to happen. Then, an extra cost is paid to compensate errors through the error recovery technique: the multiple-issue instruction replay imposes up to 28 extra recovery cycles per error [7]. This cost of recovery has shown to be high, thus leading to massive performance degradation if processor blindly relies on the error recovery in face of frequent timing errors, especially so in aggressive voltage over-scaling and near-threshold computation [46]. However, our proposed approach guarantees the correct execution at lower cost: (i) It proactively prevent timing errors on VP by applying the adaptive guardbanding across the operating corners and the sequence of instructions. For the error intolerant applications, even if some residual timing error probability remains mainly because of using Monte Carlo method described in Section 5.2, our approach relies on the processor with error recovery capability that guarantees the correct execution with 100% numerical correctness. In this way, our online adaptive guardbanding implies that recovery actions will have to be undertaken in an extremely small number of cases, hence the recovery penalty is minimal. (ii) Our technique allows timing errors to happen on IP while meeting the application-specific requirements on computational accuracy for the error-tolerant applications, hence no penalty of recovery.

TABLE 5
Area and Power Overheads of Adaptive Guardbanding

	LEON3	Intolerant	Probabilistic
Total power (W)	2.00E-01	6.79E-05	6.20E-05
Leakage power (W)	1.04E-02	1.24E-06	1.20E-06
Total area (cell)	744018	164	164

7.2 Overhead of Adaptive Guardbanding

Table 5 lists the overhead of hardware implementation of the adaptive guardbanding technique. The area overhead in comparison to LEON3 core (including *I*\$ and *D*\$) is near-zero (0.022%). Five CPMs, as PVT sensors, occupy 0.12% area [10]. The adaptive guardbanding also imposes only 0.034%/0.031% average total power overhead for the intolerant/probabilistic applications, in the worst-case operating condition; the power leakage overhead is 0.012%. This coarse grained monitoring and adaptation approach is less intrusive and expensive and nicely complements the fine-grained approaches such as Razor and EDS.

8 Conclusion

A variation-aware cross-layer approach is presented that spans circuits, architectural pipeline to the applications. We have proposed a *design time* analysis in conjunction with the minimally intrusive *runtime* adaptive guardbanding technique to combat PVT variations while guaranteeing various applications demands on computation accuracy. We introduce the notion of *Sequence-Level Vulnerability* (*SLV*) to capture variability characteristics that can be used by the compiler, runtime system or even by the application programmer. The adaptive guardbanding technique enables an in-order RISC processor to run at the fastest speed compatible with the operating conditions, various sequences of instructions, and the type of applications. This increases the throughput of probabilistic applications upto $1.9\times$ over the traditional worst-case design. Utilizing *SLV* achieves on an average $1.6\times$ speedup for the intolerant applications, compared to [33], by adapting the cycle for dynamic variations and different instruction sequences. The concrete full layout results in TSMC 45 nm technology confirm that our technique incurs only 0.022%, 0.012%, and 0.034% overheads for the total area, leakage power, and total power respectively.

Our ongoing work is focused on the creation of *instruction groups* that can be run at higher frequency/lower power in parallel execution context which could schedule instruction from multiple streams trying to obtain a favorable sequence mix in each execution hardware unit.

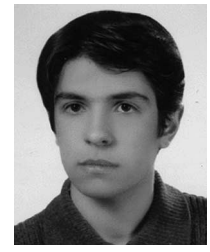
Acknowledgment

This material is based upon work supported by the NSF Variability Expeditions under award n. CCF-1029783, and FP7 ERC-AdG MultiTherman GA n. 291125.

References

[1] S. Ghosh and K. Roy, "Parameter variation tolerance and error resiliency: New design paradigm for the nanoscale era," in *Proc. IEEE*, vol. 98, no. 10, pp.1718–1751, Oct. 2010.
[2] ITRS [Online]. Available: http://public.itrs.net
[3] K. Jeong, A. B. Kahng, and K. Samadi, "Impact of guardband reduction on design outcomes: A quantitative approach," *IEEE Trans. Semicond. Manuf.*, vol. 22, no. 4, pp. 552–565, Nov. 2009.
[4] D. Ernst, *et al.*, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proc. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2003, pp. 7–18.
[5] S. Das, *et al.*, "RazorII: In situ error detection and correction for PVT and SER tolerance," *IEEE J. Solid-State Circuits*, vol. 44, no. 1, pp. 32–48, Jan. 2009.
[6] K. A. Bowman, *et al.*, "A 45 nm resilient microprocessor core for dynamic variation tolerance," *IEEE J. Solid-State Circuits*, vol. 46, no. 1, pp. 194–208, Jan. 2011.
[7] K. A. Bowman, *et al.*, "Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance," *IEEE J. Solid-State Circuits*, vol. 44, no. 1, pp. 49–63, 2009.
[8] M. Fojtik, *et al.*, "Bubble razor: An architecture independent approach to timing error detection and correction," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers (ISSCC)*, 2012, pp. 488–490.
[9] A. Drake, *et al.*, "A distributed critical-path timing monitor for a 65 nm high-performance microprocessor," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers (ISSCC)*, 2007, pp. 398–399.
[10] C. R. Lefurgy, A. J. Drake, M. S. Floyd, M. Allen-Ware, B. Brock, J. A. Tierno, and J. B. Carter, "Active management of timing guardband to save energy in POWER7," in *Proc. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2011, pp. 1–11.
[11] J. Tschanz, *et al.*, "Adaptive frequency and biasing techniques for tolerance to dynamic temperature-voltage variations and aging," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers (ISSCC)*, 2007, pp. 292–604.
[12] A. Rahimi, L. Benini, and R. K. Gupta "Analysis of instruction-level vulnerability to dynamic voltage and temperature variations," in *Proc. IEEE/ACM Des. Autom. Test Eur. Conf. Exhib. (DATE)*, 2012, pp. 1102–1105.
[13] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas, and Y. Makris, "Instruction-level impact analysis of low-level faults in a modern microprocessor controller," *IEEE Trans. Comput.*, vol. 60, no. 9, pp. 1260–1273, Sept. 2011.
[14] V. J. Reddi and D. Brooks, "Resilient architectures via collaborative design: Maximizing commodity processor performance in the presence of variations," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 30, no. 10, pp. 1429–1445, Oct. 2011.
[15] A. Rahimi, L. Benini, and R. K. Gupta, "Procedure hopping: a low overhead solution to mitigate variability in shared-L1 processor clusters," in *Proc. ACM/IEEE Int. Symp. Low-Power Electron. Des. (ISLPED)*, 2012, pp. 415–420.
[16] A. Rahimi, A. Marongiu, P. Burgio, R. K. Gupta, and L. Benini, "Variation-tolerant OpenMP tasking on tightly-coupled processor clusters," in *Proc. IEEE/ACM Des. Autom. Test Eur. Conf. Exhib. (DATE)*, 2013, pp. 541–546.
[17] J. Cong and K. Gururaj, "Assuring Application-level correctness against soft errors," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*, 2011, pp. 150–157.
[18] N. Oh, P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 63–75, Mar. 2002.
[19] M. A. Breuer, "Multi-media Applications and Imprecise Computation," *Proc. IEEE Euromicro Conf. Digital Syst. Des. (DSD)*, 2005, pp. 2–7.
[20] C. A. Martinez, J. C. Corbal San Adrian, and M. V. Cortes, "Dynamic tolerance region computing for multimedia," *IEEE Trans. Comput.*, vol. 61, no. 5, pp. 650–665, May 2012.
[21] L. Leem, H. Cho; J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *Proc. IEEE/ACM Des. Autom. Test Eur. Conf. Exhib. (DATE)*, 2010, pp. 1560–1565.
[22] P. Dubey, "Recognition, mining and synthesis moves computers to the era of tera," *Technol. Intel. Mag.*, 2005.
[23] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner, "Verifying GPU kernels by test amplification," in *Proc. ACM Programm. Language Des. Implementation (PLDI)*, Jun. 2012, pp. 383–394.
[24] K. Chae, S. Madhyay, C. Lee, and J. Laskar, "A dynamic timing control technique utilizing time borrowing and clock stretching," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, 2010, pp. 1–4.
[25] S. Ghosh, S. Bhunia, and K. Roy, "CRISTA: A new paradigm for low-power, variation-tolerant, and adaptive circuit synthesis using critical path isolation," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, pp. 1947–1956, Nov. 2007.

[26] P. Ndai, N. Rafique, M. Thottethodi, S. Ghosh, S. Bhunia, and K. Roy, "Trifecta: A nonspeculative scheme to exploit common, data-dependent subcritical paths," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 18, no. 1, pp. 53–65, Jan. 2010.
[27] G. Yan, Y. Han, and X. Li, "ReviveNet: A self-adaptive architecture for improving lifetime reliability via localized timing adaptation," *IEEE Trans. Comput.*, vol. 60, no. 9, pp. 1219–1232, Sept. 2011.
[28] A. Rajendiran, S. Ananthanarayanan, H. D. Patel, M. V. Tripunitara, and S. Garg, "Reliable computing with ultra-reduced instruction set co-processors," in *Proc. IEEE/ACM Des. Autom. Conf. (DAC)*, 2012, pp. 697–702.
[29] J. Sartori and R. Kumar, "Alleviating the voltage-scaling limitations of razor-based designs," in *Proc. IEEE Workshop Logic Synth. (IWLS)*, 2009.
[30] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proc. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2003, pp. 29–40.
[31] X. Liang and D. Brooks, "Microarchitecture Parameter Selection To Optimize System Performance under Process Variation," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*, 2006, pp. 429–436.
[32] K. Hazelwood and D. Brooks, "Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization," in *Proc. ACM/IEEE Int. Symp. Low-Power Electron. Des. (ISLPED)*, 2004, pp. 326–331.
[33] G. Hoang, R. B. Findler and R. Joseph, "Exploring circuit timing-aware language and compilation," in *Proc. ACM Int. Conf. Architectural Support Programm. Languages Operating Syst. (ASPLOS)*, 2011, pp. 345–355.
[34] LEON3 [Online]. Available: http://www.gaisler.com/cms/
[35] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, Whitepaper, V1.1, 2009.
[36] S. Bell, *et al.*, "TILE64—Processor: A 64-Core SoC with mesh interconnect," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers (ISSCC)*, 2008, pp. 88–598.
[37] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Proc. IEEE/ACM Des. Autom. Test Eur. Conf. Exhib. (DATE)*, 2012, pp. 983–987.
[38] P. N. Sanda, P. Kudva, R. Mata, V. Pokala, R. Haraden, and M. Schallhorn, "Soft-error resilience of the IBM POWER6 processor," *IBM J. Res. Develop.*, vol. 52, no. 3, pp. 275–284, May 2008.
[39] R. Kumar and V. Kursun, "Reversed temperature-dependent propagation delay characteristics in nanometer CMOS circuits," *IEEE Trans. Circuits Syst.*, vol. 53, no. 10, pp. 1078–1082, Oct. 2006.
[40] THEIA [Online]. Available: http://opencores.org/project, theia_gpu
[41] A. Terechko, M. Garg, and H. Corporaal, "Evaluation of speed and area of clustered VLIW processors," in *Proc. IEEE Int. Conf. VLSI Des.*, 2005, pp. 557–563.
[42] M. Ozawa, M. Imai, Y. Ueno, H. Nakamura, and T. Nanya, "Performance evaluation of Cascade ALU architecture for asynchronous super-scalar processors," in *Proc. IEEE Int. Symp. Asynchronous Circuits Syst. (ASYNC)*, 2001, pp. 162–172.
[43] E. Gunadi and M. Lipasti, "CRIB: Consolidated rename, issue, and bypass," in *Proc. ACM/IEEE Int. Symp. Comput. Archit. (ISCA)*, 2011, pp. 23–32.
[44] S. Hoppner, H. Eisenreich, S. Henker, D. Woalter, G. Ellguth, and R. Schuffny, "A Compact Clock Generator for Heterogeneous GALS MPSoCs in 65-nm CMOS Technology," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2012.
[45] B. A. Floyd, "Sub-Integer Frequency Synthesis Using Phase-Rotating Frequency Dividers," *IEEE Trans. Circuits Syst.*, vol. 55, no. 7, pp. 1823–1833, Aug. 2008.
[46] R. Pawlowski, E. Krimer, J. Crop, J. Postman, N. Moezzi-Madani, M. Erez, and P. Chiang, "A 530 mV 10-lane SIMD processor with variation resiliency in 45 nm SOI, " *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers (ISSCC)*, 2012, pp. 492–494.
[47] *PrimeTime® VX User Guide*, Jun. 2011.
[48] *TSMC 45 nm Standard Cell Library Release Note*, TCBN45GSBWP, version 120A, Nov. 2009.
[49] S. Herbert and D. Marculescu, "Characterizing chip-multiprocessor variability-tolerance," in *Proc. IEEE/ACM Des. Autom. Conf. (DAC)*, 2008, pp. 313–318.
[50] *Predictive Technology Model (PTM)* [Online]. Available: http://ptm.asu.edu/
[51] ARM Cortex-M3 Technical Reference Manual, rev. r1p1, 2006.
[52] *MiBench* [Online]. Available: http://www.eecs.umich.edu/mibench/
[53] *PARSEC Benchmark Suite* [Online]. Available: http://parsec.cs.princeton.edu/
[54] *SciMark 2.0 Benchmark* [Online]. Available: http://math.nist.gov/scimark2/
[55] *MediaBench* [Online]. Available: http://euler.slu.edu/~fritts/mediabench/
[56] *CoreMark Benchmark* [Online]. Available: http://www.coremark.org/home.php
[57] *EEMBC Benchmark Consortium* [Online]. Available: http://www.eembc.org
[58] *TSIM ISS* [Online]. Available: http://www.gaisler.com/index.php/products/simulators/tsim



Abbas Rahimi received the BS degree in computer engineering from the School of Electrical and Computer Engineering at the University of Tehran, Tehran, Iran, in March 2010. He is currently pursuing the PhD degree in the Department of Computer Science and Engineering, the University of California, San Diego, La Jolla, CA, USA. Since June 2010, he has also been with the Microelectronic Group at the University of Bologna, Bologna, Italy. His research interests include the resilient system design, design for robustness, and high-performance on-chip interconnections. He received the Best Paper Candidate at 50th IEEE/ACM Design Automation Conference.

Luca Benini received the PhD degree in electrical engineering from Stanford University, California, in 1997. He is a Full Professor at the Department of Electrical, Electronic and Information Engineering (DEI) of the University of Bologna. He also holds a visiting faculty position at the Ecole Polytechnique Federale de Lausanne (EPFL) and he is currently serving as Chief Architect for the Platform 2012 project in STmicroelectronics, Grenoble. His research interests include energy-efficient system design and Multi-Core SoC design. He is also active in the area of energy-efficient smart sensors and sensor networks for biomedical and ambient intelligence applications. He has published more than 600 papers in peer-reviewed international journals and conferences, four books and several book chapters. He is a member of the Academia Europaea.

Rajesh K. Gupta received the BTech degree in electrical engineering from the Indian Institute of Technology, Kanpur, Kalyanpur, India, in 1984, the MS degree in electrical engineering and computer science from the University of California, Berkeley, in 1986, and the PhD degree in electrical engineering from Stanford University, California, in 1994. He is a Professor of computer science and engineering at the University of California, San Diego (UCSD), La Jolla, and holds the Qualcomm endowed chair. He directs the smart buildings/smart grids task force at UCSD in his role as Associate Director for the California Institute for Telecommunications and Information Technology (CalIT2). His recent contributions include SystemC modeling and SPARK parallelizing high-level synthesis, both of which are publicly available and have been incorporated into industrial practice. Earlier, he led or co-led DARPA-sponsored efforts under the Data Intensive Systems (DIS) and Power Aware Computing and Communications (PACC) programs that demonstrated architectural adaptation and compiler optimizations in building high-performance and energy-efficient system architectures. He currently leads the National Science Foundation Expedition on Variability.