CONTRIBUTED PAPER

# Variability Mitigation in Nanometer CMOS Integrated Systems: A Survey of Techniques From Circuits to Software

*This article surveys challenges and opportunities in identifying variations and their effects, and investigates methods to combat these variations for improved microelectronics devices.*

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

**ABSTRACT** | Variation in performance and power across manufactured parts and their operating conditions is an accepted reality in modern microelectronic manufacturing processes with geometries in nanometer scales. This article surveys challenges and opportunities in identifying variations, their effects and methods to combat these variations for improved microelectronic devices. We focus on computing devices and their design at various levels to combat variability. First, we provide a review of key concepts with particular emphasis on timing errors caused by various variability sources. We consider methods to predict and prevent, detect and correct, and finally conditions under which such errors can be accepted; we also consider their implications on cost, performance and quality. We provide a comparative evaluation of methods for deployment across various layers of the system from circuits, architecture, to application software. These can be combined in various ways to achieve specific goals related to observability and controllability of the variability effects, providing means to achieve cross-layer or hybrid resilience. We then provide examples of real world resilient single-core and parallel architectures. We find that parallel architectures and parallelism in general provide the best means to combat and exploit variability to design resilient and efficient systems. Using programmable accelerator architectures such as clustered processing elements and GP-GPUs, we show how system designers can coordinate propagation of timing error information and its effects along with new techniques for memoization (i.e., spatial or temporal reuse of computation). This discussion naturally leads to use of these techniques into emerging area of "approximate computing," and how these can be used in building resilient and efficient computing systems. We conclude with an outlook for the emerging field.

**KEYWORDS** | Approximate computing; resilient systems; timing errors; variability

## I. SOURCES OF VARIABILITY

Variation in performance and power consumption is a common phenomenon in semiconductor manufacturing. What makes it particularly challenging, however, is its effect on manufacturing of devices as these scale down

Manuscript received February 9, 2015; revised July 10, 2015 and December 14, 2015; accepted January 4, 2016. Date of publication March 7, 2016; date of current version June 16, 2016. This work was supported by NSF Variability Expeditions (1029783), ERC-AdG MultiTherman (291125), and FP7 Virtical (288574).

**A. Rahimi** and **R. K. Gupta** are with the Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093 USA (e-mail: abbas@cs.ucsd.edu; gupta@cs.ucsd.edu).

**L. Benini** is with the Department of Information Technology and Electrical Engineering, Swiss Federal Institute of Technology in Zurich, 8092 Zurich Switzerland, and also with the Department of Electrical, Electronic and Information Engineering, University of Bologna, 40136 Bologna, Italy (e-mail: lbenini@iss.ee.ethz.ch).

Digital Object Identifier: 10.1109/JPROC.2016.2518864

Table 1 Design Impact of Performance and Power in the Presence of Variability Extracted From [3]

| Property | Ease of measuring | Variability | Effects of variability | Effect of missing specification |
|---|---|---|---|---|
| **Performance** | Medium | Medium: upto 60% | L, W, R, C, $V_{th}$, $\mu$ | Slower product, yield, timing error |
| **Leakage Power** | Easy | Large: upto 148% | L, $V_{th}$, $\mu$, $t_{ox}$ | Shorter battery life, yield, heat |
| **Dynamic Power** | Difficult | Workload dependent | C, $\alpha$ [8] | Shorter battery life, heat |

to near atomic scale feature dimensions. Any variation in dimensions, doping, etc. has a large effect on the resulting device and circuit behavior [1], [2]. To address this variation, designers resort to design guardbands. There is evidence that these guardbands are increasing rapidly, accounting for nearly 40% of the target performance, e.g., and eventually obliterating any gains due to device scaling [3]. As a consequence, reduction of design guardbands in design has become an important research challenge with recent results that seek to recover these guardbands through circuit-level changes [4].

Broadly speaking, there are three physical sources of variations: **1) Spatial variability**: Process variations cause static variations in channel length (L) and threshold voltage ($V_{\mathrm{th}}$) of devices due to random dopant fluctuations and subwavelength lithography. Static process variations manifest themselves as die-to-die (D2D) and within-die (WID) variations [1]. This includes systematic process and apparatus induced variations as well as random variations. D2D variations affect all devices on a die equally, whereas WID variations induce different characteristics for each device. **2) Temporal variability**: Aging and wearout mechanisms that cause slow temporal degradation in devices reliability. Device aging mechanisms are induced by negative bias temperature instability (NBTI), positive bias temperature instability, electromigration, time dependent dielectric breakdown, gate oxide integrity, thermal cycling, and hot carrier injection [5]. **3) Dynamic variability**: Environmental variations in ambient condition are caused by temperature fluctuations and supply voltage droops. Voltage droops result from abrupt changes in the switching activity, inducing large current transients in the power delivery system (dI/dt voltage drops), and contain high-frequency and low-frequency components which occur locally as well as globally across the die [6]. On the other hand, temperature variations occur at a relatively slow time scale with local hot spots on the die, depending on environmental, and workload conditions [7]. The origins of variability include time-independent DC component (process variations), slow-varying low-frequency components (aging and temperature), and fast-changing high-frequency components (voltage droops). The variations are expected to be worse with technology scaling [3].

Spatial parameter variations in the device geometries in conjunction with temporal degradation and undesirable fluctuations in the operating condition may prevent circuit from meeting the performance and power constraints. Table 1 illustrates design impact of performance and power in the presence of such variations [3]. The most immediate manifestations of variability are in path delay (therefore, performance) and power variations. Sequential elements are connected at the end of the paths to hold the circuit state. Path delay variations cause violation of timing specification resulting in circuit-level *timing errors* that could lead to an invalid state being stored in the sequential element. This could result in a malfunction of the digital system. Synchronous circuit designers commonly handle the timing errors by adding safety timing margins to the voltage and/or the clock frequency as guardband. This practice leads to overly conservative designs. Currently, the guardbands tend to accumulate as design closure is performed using a multicorner analysis, with an increasing number of corners [9]. As a result, the impact of guardbanding on the key design metrics (power, performance, and area) has been steadily increasing with technology scaling [3], leading to loss of operational efficiency and increased costs due to overdesign. Power variability is also challenging, for instance 13× variation in the sleep power across ten instances of ARM Cortex M3 core has been observed over a temperature range of 22–60 °C [10]. However, we narrow the scope of this paper to the path delay variation and its manifestation as timing errors. We identify the timing errors as the most threatening manifestation of variability and investigate various means to address it throughout this paper. We begin with a quantitative feel of the extent of variation currently seen in manufactured devices. Section II covers the delay variation in details.

Earlier variability-centric surveys [11], [12] have focused on the circuit and architecture levels, by contrast we focus on software, application and algorithmic methods. Further, we provide a holistic remedy for both data-level and task-level parallel architectures.

## II. DELAY VARIATION

Authors in [4] quantify the impact of individual process, voltage and temperature (PVT) variations on a standard cell inverter delay through SPICE simulations. Table 2 shows eight possible combinations of PVT corners in 65 nm technology. Between the worst-case and the best-case PVT corners, 1.8× delay variation has been observed; 1.46× comes from the process, 1.25× comes from the voltage, and 0.97× comes from the temperature due to temperature inversion effect [4].

For an Intel 80-core processor in 65 nm, Fig. 1 shows the WID core-to-core maximum frequency (Fmax)

Table 2 Inverter Delay for Different 65 nm PVT Corners [4]

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

variations for each of the 80 cores. The measurements have been done at a fixed operating temperature of 50 °C with three operating voltages: 1.2, 0.9, and 0.8 V. At the nominal voltage of 1.2 V, the fastest core displays the Fmax of 7.3 GHz while in the same die the slowest core can work with the Fmax of 5.7 GHz resulting in 28% WID clock frequency variation. Fig. 2 illustrates the delay distribution of the 80 cores for the same operating conditions [13]. The single die with 80 cores exhibits an increasing value of $\sigma/\mu$ for lower voltages: 5.93%, 6.37%, and 8.64% for 1.2, 0.9, and 0.8 V, respectively. Lowering the voltage from the nominal 1.2 V to 0.8 V, increases the critical paths variability ($\sigma/\mu$) by 45% [13].

Voltage overscaling (VOS) [14] and working at near-threshold (NT) voltage [15] have become popular approaches for building energy-efficient digital circuits. Operating at low voltages ($V_{DD} \leq 0.5$ V) unfortunately exacerbates the effects of delay variations [14], [16]–[19]. This indicates the importance of variability awareness at lower operating voltages where the delay uncertainty is further increased. The WID delay measurement for a 45 nm SIMD processor shows that reducing $V_{DD}$ from 1.0 to 0.53 V increases the delay variation by 6× [19]. Fig. 3 shows the normalized gate delay variation due to process variations as a function of $V_{DD}$ [16]. Working at near threshold voltage of 400 mV increases the performance variability by 5× compared to 1.3× at the nominal operating voltage. It is then clear that for logic working at NT voltages, the statistical WID variation in the voltage threshold ($V_{\text{th}}$) plays an important role in determining the path delay. $V_{\text{th}}$ variations result mainly from random fluctuations in the number of dopant atoms in the transistor channels [17]. Considering dynamic sources of variations, including temperature fluctuations, and voltage droops results in a total performance variability of 20× [16].

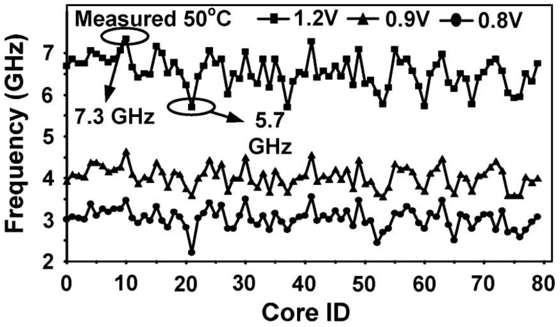


Fig. 1. *WID core-to-core maximum clock frequency variation for 80 cores on a single chip [13].*

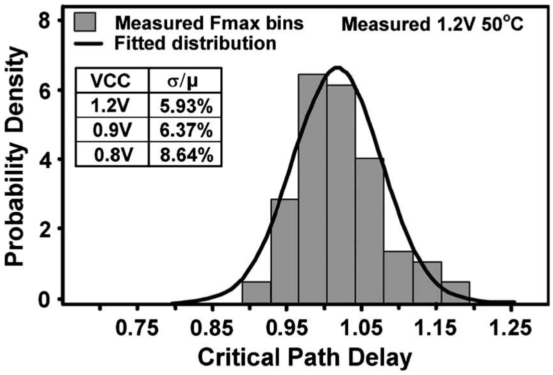


Fig. 2. *Critical path delay distribution and its coefficient of variation ($\sigma/\mu$) for 80 cores on a single chip [13].*

Given such a growing increase in performance variability, design methods are needed to make a design resilient to the timing errors especially so for circuits operating at low voltages where the effect of delay uncertainty is pronounced. The effects of the static process variations can sometimes be mitigated through binning or by postsilicon tuning during test time, while the dynamic variations manifest themselves on the field as a function of time and environment, and therefore cannot be compensated by *one-time* presilicon and postsilicon tuning techniques. Consequently, accurate design time

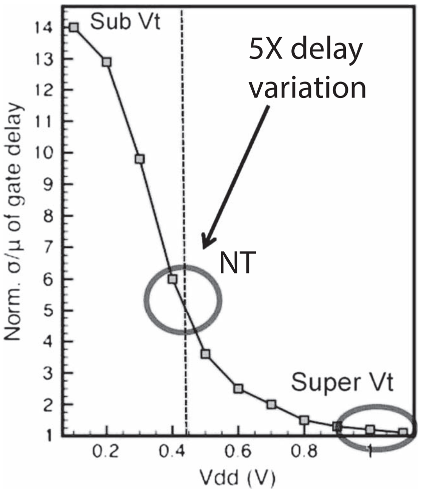


Fig. 3. *Impact of voltage scaling on gate delay variation due to process variation [16].*

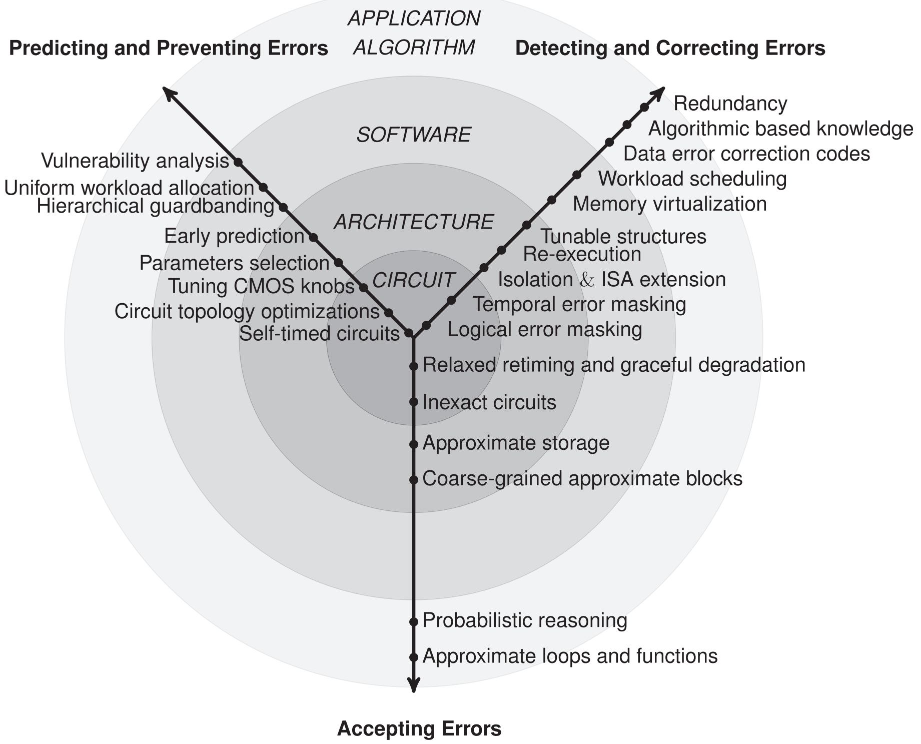


**Fig. 4.** *Taxonomy for timing error tolerance: Abstractions versus approaches.*

analysis coupled with efficient runtime techniques are required to overcome the variability challenges.

The rest of the survey paper is organized as follows. In the next three sections, we provide a taxonomy to classify various variability-tolerance approaches. Section III describes three classes of methods to handle timing errors at different abstraction levels. Section IV covers enhanced methods through combined approaches of error handling as well as cross-layer error information exchange. Granularity of observing and controlling timing errors is discussed in Section V. In Section VI, we illustrate single-core and parallel architectures where the approaches presented in the previous sections are integrated to enhance their resiliency. In focus of data-level and task-level parallel architectures in Section VII, we further describe how combined approaches can be applied in such architectures. In Section VIII, we conclude with an outlook for the emerging field of resilient and approximate computing.

## III. A TAXONOMY FOR TIMING ERROR TOLERANCE

This section classifies approaches to timing error handling into a conceptual Y-chart shown in Fig. 4. The Y-chart groups techniques to address variability into three classes based on *when* and *how* the timing errors should be manipulated. These three classes of the Y-chart are on radial axes. The first axis describes *design time* approaches to predict and avoid the timing errors. The second axis focuses on *runtime* approaches to detect and correct the timing errors, while the third axis neglects the timing errors if possible. Each class is divided into levels of abstraction, using concentric rings. Every abstraction level determines at *which* level of the computing stack the approaches can be applied: *circuit*, *architecture*, *software*, *application* and *algorithm*. At the top level outer ring, we consider approaches applicable to algorithm or application level; at the lower levels inner rings, we refine approaches into finer software, architecture, and circuit implementations. These three axes are covered in Section III-A–C.

### A. Predicting and Preventing Timing Errors

In this section, we describe a class of approaches that aim at reducing the excessive guardband or generally enabling better than worst-case design while avoiding the timing errors. This class typically relies upon modeling that derives rules for simultaneous guardband reduction and error prevention. Table 3 lists how these approaches are implemented at different design abstraction levels. Our description in Section III-A accordingly moves form circuit level in Section III-A1, to architecture level in Section III-A2, and finally to software level in Section III-A3.

Table 3 Predicting and Preventing Timing Errors: Abstractions Versus Approaches

| | Predicting and Preventing Timing Errors (No Timing Error) |
|---|---|
| **Software** | • Vulnerability analysis: instruction-level [20], sequence-level [21], [22], [23], procedure-level [24]<br>• Uniform workload allocation: idleness distribution [25], healthy kernels with dynamic recompilation [26]<br>• Hierarchically focused guardbanding [27] |
| **Architecture** | • Early prediction using micro indicators: PC-based prediction [28] and then instruction scheduling [29]; signature-based voltage droop prediction [30]<br>• Parameters selection: microarchitectural parameters (e.g., pipeline depth) [31], variable/adaptive latency register file and execution units [32], CRISTA [33], Trifecta [34], data-dependent operation speed-up [35], sub-system voltage/frequency and structure type [36], core allocation and thread hopping [13] |
| **Circuit** | • Tuning CMOS knobs: adaptive body biasing [37], [38], adaptive voltage and/or frequency scaling [39], [40], [41], [13], [42], [43], [44]<br>• Circuit topology optimizations: uncertainty-aware design methodology [45], slack redistribution [46], optimal V$_{DD}$ and V$_{th}$ selection [47], guardband reduction [4], clustering and power-gating [48]<br>• Self-timed circuits: delay insensitive circuits, quasi delay insensitive circuits (A8051 [49]), adaptive parameterized circuits [50], GALS clustered processor [51], GALS-based MPSoC[39], GALS-based processor clusters [52] |

1) *Circuit:* The prediction and prevention approaches at the circuit-level are threefold. Some of these approaches tune available CMOS knobs of the circuit. Other approaches either change the topology of the clocked circuits, or switch to clockless (self-timed) circuits to enhance immunity from the timing errors.

*Tuning CMOS Knobs:* These approaches tune electrical characteristics (e.g., power and delay) of a circuit block by leveraging CMOS knobs including, body bias, supply voltage, and clock frequency. These approaches are dynamic in nature, therefore enable adaptive circuit design that can be tuned after fabrication. Adaptive body biasing is one such runtime technique which carefully selects an appropriate body bias as an available parameter to tune the electrical circuit characteristics [37], [38]. Forward body bias reduces the voltage threshold ($V_{\text{th}}$), while reverse body bias increases the $V_{\text{th}}$. Increasing the $V_{\text{th}}$ improves performance (lower delay) at the expense of additional leakage power, while decreasing the $V_{\text{th}}$ reduces both performance and leakage power. Therefore, a slow circuit block can be forward biased, whereas a leaky circuit block can be reverse biased. Similarly, voltage and/or clock frequency can be tuned to compensate the variations [13], [39]–[44]. Table 3 lists various implementations of adaptive voltage/frequency scaling.

*Circuit Topology Optimizations:* These methods utilize the design time CAD optimizations to change the topology of a circuit for enhancing its resiliency against timing error. For a given circuit, there will be “a wall” of equally critical paths that are highly susceptible to timing errors especially so in voltage overscaling (VOS) regime [53]. To alleviate the effect of cluster of critical paths, some of the approaches focus on uncertainty-aware [45] circuit optimizations such as upsizing and downsizing gates (W/L ratio), use of multiple $V_{\text{th}}$ cells, and restructuring to reshape the path delay distribution. These methods strive to shift the timing slack of frequently-exercised and near-critical paths in a power/area-efficient manner [46]. Other approach clusters timing critical cells, and inserts sleep transistors in a row-based layout for power-gating those cells [48]. In [4], the authors describe a model of guardband reduction through standard cell synthesis, place and route optimization flow to quantify the impact on quality of results.

*Self-Timed Circuits:* In self-timed circuit, or asynchronous circuit, there is no need for a clock signal to determine a starting time for a computation. Delay insensitive circuit is among robust asynchronous circuits because it makes no assumptions on the delay of wires or gates. Quasi delay insensitive circuit is a subclass of the delay insensitive asynchronous circuit which makes minimal delay assumptions only on isochronic forks. Therefore, the quasi delay insensitive circuit blocks independently operate at their maximum speed for a given amount of variability. Authors in [49] design two versions of 8051 microcontroller: a synchronous logic (S8051) and a quasi delay insensitive asynchronous logic (A8051). Both cores are fabricated on the same die at 130 nm technology for performance measurements from nominal voltage to deep subthreshold. A8051 has $\sim 2\times$ larger area than the S8051, while both cores feature comparable energy and speed at nominal conditions. However, when PVT and workload are varied the S8051 requires $\sim 4\times$, $\sim 1.5\times$, and $\sim 2\times$ delay margins for process ($\Delta V_{\text{th}} = \pm 3\sigma$), voltage ($\Delta V = \pm 10\%$), and temperature ($\Delta T = 70\ ^{\circ}\text{C}$) variations, whereas the A8051 operates at actual speed [49].

Nevertheless, a delay insensitive circuit may spend energy to ensure its functional integrity that could be used instead on computation. To address this, a new methodology is proposed to compose several flavor of asynchronous circuit implementations (with different power/timing modes, possibly synchronous one) into a single adaptive parameterized circuit where the adaptability of such runtime reconfigurable solution outweighes the overheads [50]. To combat the variations, an out-of-order processor partitions its components to different clock domains using a globally asynchronous, locally synchronous (GALS) paradigm [51]. GALS is also used as a variation-tolerant communication network across multiple processers [39], and processor clusters [52].

*2) Architecture:* To avoid the timing errors, some architectural approaches utilize various indicators such as program counter (PC), cache misses, etc. to predict the timing errors and avoid them in advance. Alternatively, other approaches select appropriate parameters—among all available choices—for different architectural components that enable them to prevent the timing errors.

*Early Prediction Using Micro Indicators:* Some approaches leverage existing embedded indicators to tie an architectural event to a possible timing error. For instance, the PC is used as a predictor of an incoming timing error by monitoring an errant instruction [28]. This PC-based approach can pinpoint to a highly probable errant instruction and therefore trigger a preventive mechanism to avoid the timing error. For example, the instruction scheduling in an out-of-order processor can be enhanced for this purpose [29]. Other approaches extract signatures to link microarchitectural indicators to fast dynamic variations such as impending voltage droops [30]. To construct such a signature for the voltage droops, a voltage emergency predictor observes the pipeline flush and the cache miss [30]. Such a signature can accurately notice the likelihood of a voltage droop few cycles ahead of occurrence to avoid recurrence of the corresponding voltage droop.

*Parameters Selection:* Other architectural approaches try to select an appropriate *value* for a given microarchitectural parameter during the design time to enhance the resiliency. For instance, Liang and Brooks propose a joint architectural and statistical timing analysis method of selecting pipeline depth and size to mitigate the impact of clock frequency variations [31]. Similar techniques focus on multivariable latency blocks. A variable-latency technique alleviates the impact of the process variations on the register file and the execution units in a microprocessor [32]. Critical path isolation for timing adaptiveness (CRISTA) [33] isolates the long critical paths of the design and provides an extra clock cycle for those paths; therefore, CRISTA avoids possible delay failures in the critical paths by dynamically switching to two-cycle operation when they are activated. CRISTA assumes all standard operations are single cycle. Trifecta [34], a variable latency processor based on CRISTA, completes instructions that activate those long critical paths in two cycles. Data-dependent operation speedup is another architectural technique that dynamically allows more cycles depending on where the data are inside a combinatorial stage [35]. Another parameter selection method dynamically adapts the clock frequency, per-subsystem voltages, the issue queue size, and the functional unit structure for a processor [36]. Moving to an 80 cores chip, the voltage/frequency for every core is tuned to avoid the timing errors [13].

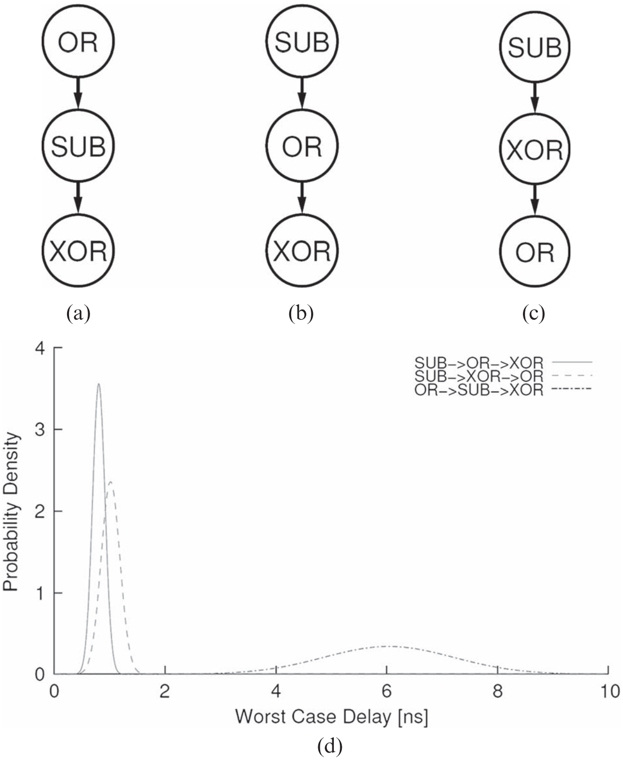


**Fig. 5.** ***Worst case ALU delay distribution for different instruction sequences [21]. (a) Original graph. (b) Reordered graph 1. (c) Reordered graph 2. (d) ALU delay distribution.***

*3) Software:* Software approaches presented in this section follow the architectural prediction approaches to extract a *warning* signature from the low-level program execution, rather than the architectural indicators. This signature increases capability of software to avoid the timing errors.

*Vulnerability Analysis:* Approaches presented here assess the vulnerability, or sensitivity, of unit(s) of a program to variations: instruction-, sequence-, and procedure-level vulnerability. Fig. 5 shows that the order of instructions can impact the worst case circuit delay distribution of an ALU [21]. This program-specific detail opens the door to code transformation for improving the timing resiliency. For instance, sometimes an OR instruction is exchanged with a SUB instruction that improves the worst case circuit delay distribution of the ALU significantly [21]. In the same vein, instruction-level vulnerability (ILV) [20] and sequence-level vulnerability (SLV) [23] are proposed to estimate the vulnerability of an instruction or a sequence of instructions to the timing errors. The ILV partitions the instruction set based on the delay distribution into three classes shown in Fig. 6; the higher delay, the higher likelihood of a timing failure, the higher

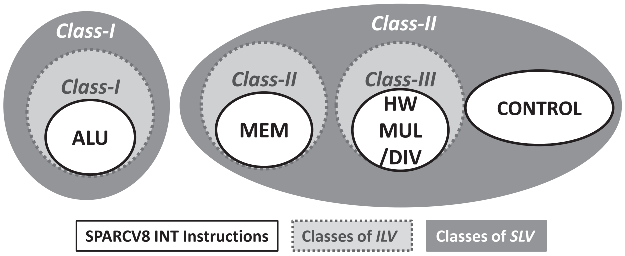


**Fig. 6.** ***ILV and SLV classifications for the integer SPARC V8 ISA [23].***

ILV. All the instructions within a class exhibit almost equal ILV. The ILV classification shows for the integer SPARC V8 instructions, the vulnerability of the ALU instructions is slightly lower than the memory instructions; the memory instructions also have a lower marginal vulnerability compared to the hardware multiplication and division (HW MUL/DIV) [20]. This is mainly because the path delay distribution of the exercised parts by a class of instruction is such that most of the paths have the same length, then we have an *all-or-nothing* effect, which implies that either all instructions within that class fail or all make it [20].

Considering a stream of instructions, the SLV classifies them into two classes: Class-I (only the ALU instructions), and Class-II (the rest of integer instructions, including the memory, the HW MUL/DIV, and control instructions). Based on SLV values, the vulnerability of the Class-I is lower than (or equal of) the Class-II [23]; this means that the Class-I requires a lower guardband compared to the Class-II. Fig. 6 summarizes the ILV and the SLV classifications. Compiler can benefit from these classifications to enhance the code resiliency. For instance, 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 of ALU 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 the ALU instructions, and as a result the percentage of sequences of Class-I is increased up to 41% and on average 31% for programs in EEMBC AutoBench [54] suite of benchmarks. Hence, an adaptive guardbanding can benefit from this compiler transformation technique to further reduce the guardband for the sequences of Class-I.

The aforementioned efforts strive to establish a link between the blocks of a processor that are activated during execution of an instruction or a sequence of instructions, to the observed timing errors. However, the *sensitivity* of the instructions to the timing errors is also impacted by the circuit block topology, structure, synthesis optimization strategies, technology, and workload. For instance, a hierarchical ALU displays a large difference between the instruction delays, whereas all instruction pairs (consecutive instructions) exhibit almost the same amount of delay in a high-performance ALU without hierarchies [22]. The latter is aligned with the all-or-nothing effect that is observed across the classes of SPARC V8 instructions: either all instructions within that class fail or all make it [20], [23]. Similarly, another observation on an ARM Cortex-M0 core shows that the number of timing errors increases dramatically after the first failing operating point [55]. This is mainly because the number of potential faulty flip-flops increases quickly with the scaled clock period. More than 50% of flip-flops are critical for a 10% timing slack which could wipe out the benefits of VOS or over clocking after the critical operating point [55].

Going further up on the software stack, a recent work exploits variations in the voltage droops among different procedure calls to form a runtime preventive procedure hopping [24]. During a characterization phase, the probability of voltage droops on different combinations of voltage/temperature (V/T) of a core is characterized at the level of procedures, where the problematic sequences of instructions [23], [56] could exist. This characterized metadata is then attached to each procedure at compile time, to be able to use it for runtime decisions about finding the best location to execute the procedure among available V/T-islands within a cluster of cores. Results show that the procedure hopping avoids the critical voltage droops during the execution of all procedures while incurring less than 1% latency penalty for migration of procedures [24].

*Uniform Workload Allocation:* The techniques apply adaptive workload allocation to address the nonuniform device aging. An idleness distribution technique applies idle cycles for fatigued cores to compensates the effects of aging and therefore avoiding the permanent failure [25]. This workload allocation technique mitigates aging-induced unbalanced cores lifetimes by means of core activity duty cycling on a multicore platform. Another work identifies fatigued resources using NBTI monitoring [57] in a VLIW architecture [26]. Then a compiler-directed scheme periodically shifts the instructions stress from a fatigued VLIW slot to a young one. It correlates the hardware stress time with instructions distribution, and equalizes the expected lifetime of each VLIW slot by regenerating *healthy* kernels that respond to the specific health state of the aged hardware [26].

*Hierarchically Focused Guardbanding:* A notion of hierarchically focused guardbanding to adaptively mitigate PVT variations and aging is proposed [27]. The method is guided by an online utilization of characterized models, and enables a focused adaptive guardbanding in view of monitors, observation granularity, and reaction times. The effectiveness has been shown at two levels of observation and adaptation: 1) applying adaptive guardbanding at granularity of kernel-level by employing

Table 4 Detecting and Correcting Timing Errors: Abstractions Versus Approaches

| | Detecting and Correcting Timing Errors |
|---|---|
| **Application Algorithm** | • Algorithmic based knowledge: algorithmic based fault tolerance [58], [59], encoder with iterative and error concealment [60], executive assertions and invariant checks [61], [62], partial recomputation and error localization [63]<br>• Error correction codes for data: tensor product codes for Flash [64] |
| **Software** | • Core: redundancy-based scheduling [65], [66], OS redundant multithreading [67], dynamic loop scheduling [68]<br>• Memory: virtualization [69], physical address zoning [70] |
| **Architecture** | • Tunable structures: per-stage voltage interpolation and variable latency FPUs [71], donor stages for time borrowing [72], reconfigurable L1 memory banks [18], fault-tolerant caches [73], [74]<br>• Restoring to a pre-error state or re-execution: counterflow pipelining [75], instruction replay [44], micro-rollback [76], checkpoint-rollback [77]<br>• Isolation and independent recovery: SIMD decoupling queues [78], [19]<br>• ISA extension: URISC [79] |
| **Circuit** | • Temporal error masking (clock adjustment and time borrowing): clock phase adjusting [80], global clock gating [75], two-phase transparent latch (Bubble Razor) [81], single-cycle local stalling [82], TIMBER [83], [84], TBFF with clock stretching [85], [86]<br>• Logical (spatial) error masking: non-intrusive redundancy [87], approximate redundant logic [88], SIMD lane weaving [19] |

coarse-grained monitors; and 2) the finer granularity of instruction-level monitoring that adapts guardband depending on the monitors configuration and the type of instructions executed within the kernels.

## B. Detecting and Correcting Timing Errors

As an alternative to earlier methods in Section III-A that seek to prevent timing errors from happening, in this section we examine techniques that work through the timing errors. Presented approaches here allow the timing errors to occur by operating at the edge of failure. Operating at the edge of failure further reduces the guardband. To combat the timing errors, these approaches require two main mechanisms: 1) an error detection mechanism: a mechanism to detect the incorrect state values caused by the timing errors; and 2) an error correction mechanism: a mechanism that is triggered upon an error detection to compensate the effects of errors during system operation. Not all timing errors need to be corrected. Only those errors that have an observable effect on resulting computation. In that sense, these approaches are a bit more flexible in their use and impact. Table 4 lists these approaches at different levels. Focusing on hardware approaches, Section III-B1 zooms in the circuit solutions, while Section III-B2 moves to the architectural techniques. The last two sections cover the software and application/algorithm approaches.

*1) Circuit:* These circuit-level techniques typically utilize circuit sensors for timing errors detection, and apply temporal or logical (spatial) error masking. We first describe the mechanism of circuit sensors followed by details of temporal and logical error masking. A common strategy of the circuit sensors is to detect variability-induced delays by sampling and comparing signals near the clock edge to detect the timing errors [75], [90]. The detection is implemented by the insertion of a delay to the clock or data line. To do so, two consecutive samples are captured and in case of a mismatch between the samples an ERROR signal is generated meaning that a timing error caused storing an invalid state. Let us focus on one of these circuit sensors: error-detection sequential (EDS) [89]. A resilient circuit can be constructed by replacing typical flip-flops [see Fig. 7(a)] with EDS circuits [see Fig. 7(c)] on the critical paths. The EDS circuit is a double-sampling with a time-borrowing latch design which consists of a flip-flop, a shadow latch, and an XOR gate. The main flip-flop and the shadow latch sample the input data on the rising and falling clock edges, respectively. The XOR logic gate compares the latch and flip-flop outputs to generate the ERROR signal as shown in Fig. 7(d). If the input data arrives late, with respect to the clock signal, due to any type of variations, it cannot meet the setup time of the flip-flop; as a result, the latch and flip-flop outputs differ, resulting in rising the ERROR signal. This generated ERROR signal can be propagated the rest of chip to invalidate the erroneously executed operation and trigger a proper recovery for correction. The error corrections are twofold: temporal and logical error masking.

*Temporal Error Masking:* To compensate the timing errors, temporal techniques tune timing references either by clock signal adjustment or by time borrowing. For example, the clock phase is controlled based on the measured timing slack [80]. As an architectural-independent technique, the global clock gating is used with Razor [75]. Upon an error detection, the global clock gating signal stalls the entire pipeline and then reloads the correct valid state into flip-flops. However, this technique does not scale well since the global signal has to be propagated across the entire chip in one cycle. Bubble Razor [81] utilizes a two-phase latch as opposed to the flip-flop.

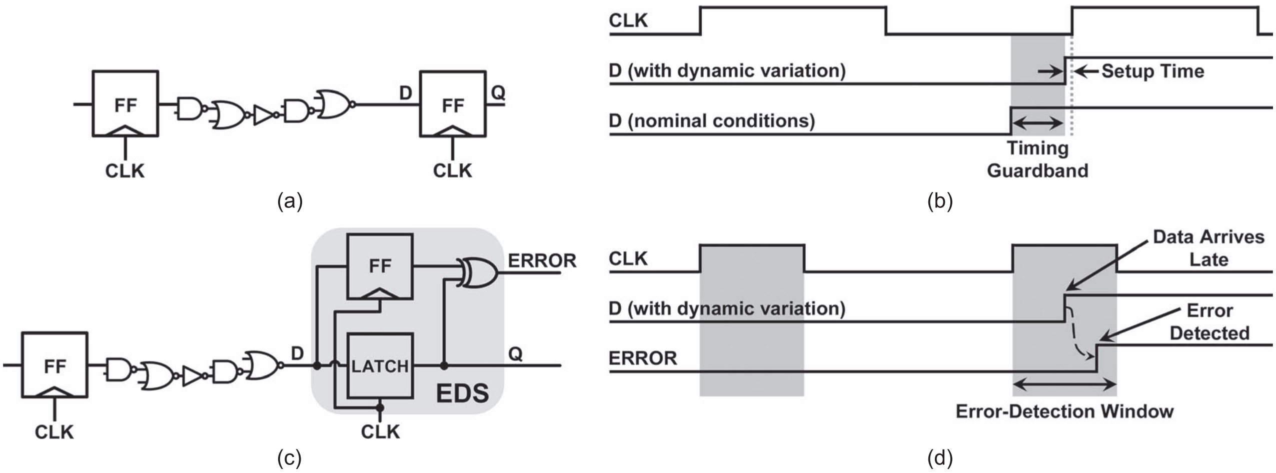


**Fig. 7.** ***(a) Conventional flip-flop at endpoint of critical path; (b) timing diagrams for variations and nominal conditions; (c) error-detection sequential (EDS) circuit is replaced with flip-flop at endpoint of critical path to enable resiliency; and (d) late arriving input data and error detection [89].***

Upon an error detection, Bubble Razor propagates *bubbles* to the neighboring latches as the clock gating control signals. The bubble propagation implements a local stalling scheme providing one extra cycle for correct data to arrive. Although Bubble Razor offers 1-cycle error correction, it can only be used for the two-phase latch based designs [81]. To address this concern, another local stalling technique is proposed in [82] which has the same penalty cost as Bubble Razor, but can be used in the flip-flop or pulsed-latch based designs.

Two sequential circuits are proposed that enable time borrowing: TIMBER flip-flop and TIMBER latch [83], [84]. TIMBER masks the timing errors by borrowing time from successive pipeline stages. References [85] and [86] present a more systematic time borrowing method that is composed of a special flip-flop (with a time-borrowing detection) and a clock shifter. The time-borrowing flip-flop uses clock shifter circuits to allow time borrowing on the critical paths, generates time-borrow signal for clock shifter to stretch the clock period. This pays back the borrowed time in the next clock cycle; therefore, no error recovery is needed.

*Logical Error Masking:* These spatial techniques replicate logic so in case of a timing error, the error-free output of the replicated version can be reused instead. A nonintrusive redundant circuit is implemented by synthesizing the Boolean function that represents the critical paths [87]. The synthesized circuit has at least 20% timing slack over the original logic circuit guaranteeing its immunity to timing errors. Another work introduces *in situ* fine-grained redundant approximation circuit that exhibits improved timing slack compared to the original circuit [88]. The approximate circuit can be constructed by simple structural analysis of the original circuit. It implements the spatial error masking by creating a logically-equivalent yet timing-improved circuit. A 10-lane SIMD core considers 2 spare lanes to mitigate the impact of process variation [19]. To utilize this coarse-grained spatial redundancy, a lane weaving technique is applied for each pipeline stage. The lane weaving bypasses a slow stage in a lane by routing the lane to one of neighbor lanes, either on the left or right [19].

*2) Architecture:* Architectural approaches to variability mitigation enhance resiliency, including tuning components through adaptive voltage/frequency/latency settings, backward error recovery, and ISA extension. These approaches are designed with availability of the circuit sensors (e.g., Razor [75] and EDS [89]) in mind.

*Tunable Structures:* These techniques adopt similar temporal and spatial techniques, as presented in Section III-B1, plus voltage adjustment but with a focus to a particular architectural structure. Reference [71] presents a mixed voltage interpolation and variable latency technique with emphasis on floating-point units (FPUs). The per-stage voltage interpolation chooses different voltage configurations among high $V_{DD}$ and low $V_{DD}$ that can tune the clock frequency of the pipeline—small difference between high $V_{DD}$ and low $V_{DD}$ eliminates the overhead of level shifters. At a coarser granularity, local adaptive $V_{DD}$ hopping [91], [92] is applied for individual core. The $V_{DD}$ hopping chooses one of the three discrete voltages based on online variability measurements reported by both Razor and ring oscillators. Reference [72] implements cycle time stealing which applies another form of time borrowing by transferring the timing slack of the faster stages to the slow ones by skewing clock arrival times.

Memory subsystem, spanning on-chip caches to SRAM memory banks, have been target of architecture level variability management. Common to these methods is level of redundancy and dynamic sizing of memory blocks. For instance, to mitigate the variations among L1 banks, a technique adds variable latency stages into the path from the cores to the memory banks [18]. This variation-tolerant architecture also supports a reconfigurable address-interleaving to bypass some of the slow memory blocks. Focusing on the error tolerance within a cache block, a technique replaces faulty cells due to process variation by dynamically downsizing the cache [74]. A recent work [73] characterizes the nature of bit faults during VOS for the SRAM-based cache on test chips manufactured in a 45 nm technology [93]. They observe that in the process variation-affected SRAMs, the bits that fail at some voltage level will also fail at all lower voltages [73]. Therefore, a simple and low-overhead fault tolerance cache is proposed that supports few $V_{DD}$ levels for the data array SRAM cells.

*Restoring to a Preerror State or Reexecution:* When a timing error is detected, these approaches trigger a series of architectural activities to recover from the timing errors. For instance, once a timing error is detected during an instruction execution, the Intel resilient core [44] prevents the errant instruction from corrupting the architectural state and an error control unit (ECU) initially flushes the pipeline to resolve any complex bypass register issues. To ensure a scalable error recovery, ECU replays the errant instruction multiple times at the same clock frequency (multiple-issue instruction replay). It has a recovery cost of $3 \times N$ cycles where $N$ is the number of pipeline stages. Similarly, in counterflow pipelining when an error is occurred within a stage, the stage sends a bubble toward end of pipeline stages and a flush toward head of pipeline stages (the fetch stage). Both bubble and flush signals are propagated cycle by cycle. The recovery penalty per error correction is $2 \times K$ cycles where $K$ is the order of the stage, which detects an error in the pipeline. Micro-rollback is another technique that locates private queue per each pipeline stage [76]. After an error-free execution of an instruction, the queues save a *snapshot* (i.e., operands and results). In case of an error, the snapshot can be reloaded into the pipeline and therefore the instructions can be restarted at their last known correct state. Checkpoint-restart has the similar trend but for a coarser granularity which often imposes high overhead. Special hardware structures are added to a core to support speculative execution for reducing the cost of checkpoint-restart [77].

*Isolation and Independent Recovery:* In the lock-step execution, any timing error within any of the lanes will cause a global stall to force recovery of the entire SIMD pipeline. Techniques are proposed to decouple the lanes through private queues that prevent the error events in any single lane from stalling all other lanes [19], [78]. This trick enables each lane to recover from the errors independently while causing slip between the lanes which requires additional architectural mechanisms to ensure the correct execution. Memoization techniques presented in Sections IV-B2 and VII-D address recovery issues for SIMD and GPGPU architectures.

*ISA Extension:* Ultra-reduced instruction set coprocessors (URISC) [79] extends a MIPS processor with a coprocessor that implements a new instruction called `SUBLEQ`. URISC executes the sequences of `SUBLEQ` that are semantically equivalent to any faulty instruction.

*3) Software:* Workload scheduling and memory allocation have been already deployed in the software to address various concerns in cores and memory subsystems. These two software modules can take into account the timing errors to ensure reliable operation. 1) Scheduling techniques guarantee reliable execution on the cores typically through replication and redundancy. 2) Variability-aware memory allocation adapts to the underlying variations in the memory modules and virtualizes the memory hierarchy for efficient address-space partitioning.

*Scheduling:* A loosely coupled triple modular redundancy (TMR) scheme for cluster-based many-core accelerators [52] is presented in [66]. The cluster controller blindly generates three replicas of the main thread for error detection and then a voting is applied to choose the correct result. Another dynamic TMR technique for reliable OpenMP tasking is presented in [65]. Programmer needs to annotate a reliable task through extended OpenMP task construct with a reliable clause, `#pragma omp task reliable`. To assure error tolerance on the cores, when a parent task creates a reliable child task into the runtime environment, it will dynamically replicate and submit three redundant children tasks for execution. Among the redundant executions a majority voting is applied for the error detection and correction [65]. An OS support for redundant multithreading is also proposed to detect and correct the errors during the execution of user-level applications using TMR [67]. Moving the focus from tasks to loops, a fault-tolerant loop scheduling scheme without checkpointing is presented [68]. A loop is transformed to ensure the correctness of the reexecution of loop iterations by buffering variables with anti-dependency.

*Memory Virtualization:* Variability-aware memory virtualization layer allows marking regions of memory through annotations [69]. Programmers apply annotations using high-level API to guide the OS. The memory virtualization layer partitions the memory space based on the power/performance characteristics, for instance, voltage scaled SRAMs, nominal $V_{DD}$ on-chip memories, low-power and high-power DRAMs [69]. This offers an opportunity to programmers for partitioning their application's address space into virtual address regions with different characteristics. For each annotated region, a mapping policy can be implemented to drive the dynamic variability-aware memory allocation. This method can opportunistically exploit for instance DRAM power variations through physical address zoning [70].

Table 5 Accepting Timing Errors: Abstractions Versus Approaches

| | Accepting Timing Errors |
|---|---|
| **Application Algorithm** | • Approximate loops and functions: Green [94], loop perforation [95]<br>• Probabilistic reasoning for program transformation [96] |
| **Software** | • Relaxed synchronization [97], selective discard of atomic operations [98] |
| **Architecture** | • Coarse-grained approximate execution: VOS meta-functions [99], ERSA [100], neural processing unit [101], relaxed neural blocks [102]<br>• Approximate storage: multi-level cell approximate storage [103] |
| **Circuit** | • Approximate [104] and inexact [105] circuits<br>• Graceful quality degradation (DCT [106] and DCT/IDCT [107]), relaxed retiming [53] |

*4) Application/Algorithm:* Approaches based on application and algorithm exploit algorithmic knowledge provided by a domain expert, or special coding to enhance application resiliency.

*Algorithmic-Based Knowledge:* These approaches rely on knowledge from the application domain experts. Algorithm-based fault tolerance is a system level approach that exploits some basic properties of computations to check the correctness of the computed output values [58], [59]. An important target of such optimizations are matrix operations. This scheme is used to detect and correct errors during matrix operations as the heart of many computation-intensive algorithms [58]. For multimedia applications, in-built error resiliency techniques can be used at the decoder to detect and correct for encoder induced errors [60]. Focusing only on error detection, assertions and invariant checks can be inserted based on algorithmic knowledge [61], [62]. The executable assertions aim to detect data errors by tracking a variable during a test procedure. Focusing only on algorithmic error correction, a partial recomputation-based approach is presented [63]. The partial recomputation-based approach identifies partitions of faulty and nonfaulty outputs through error localization. This lowers the high cost of recovery in traditional fault tolerance approaches, e.g., checkpoint-restart [77].

*Error Correction Codes:* To protect data in applications, the error pattern can be observed and characterized. Based upon that an appropriate error correction coding scheme can be applied to protect the memory subsystem. Data collected from a triple level cell Flash device demonstrates that the vast majority of cell-errors only had a single bit in error (errors affect 1 of the 3 bits of information) [64]. This observation leads to a new error correction code, based upon generalized tensor product codes, that enhances resiliency of Flash memory.

## C. Accepting Timing Errors

In this section, we describe a new class of approaches that have a relaxed behaviour toward the timing error handling. The aforementioned methods in Section III-A and B strive to achieve instruction executions exactly as specified by the application programs. In contrast, probabilistic or approximate programs can exhibit enhanced error resilience for applications 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. 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 is acceptable if the fidelity metrics satisfy a certain threshold. This provides an opportunity for ignoring the effect of timing errors as long as such errors do not lead to program failures, crashes, or hangs. Table 5 illustrates these techniques with spacial emphasis on application/algorithm, architecture, and circuit levels.

*1) Circuit:* Circuit-level approaches mostly focus on various design paradigms to synthesize circuits that produce approximate (inexact) results. Such approximate circuits have been used earlier to speedup performance and reduce the latency of computation [104]. The main application of such approximate circuit, or inexact circuit, is to reduce power, area, and complexity in exchange for a small loss of precision. For instance, an statistical approach can prune the logic gates of an adder based on the statistics of the data to be processed [105]. Different sets of design time techniques to generate the approximate circuit are reviewed in [105].

A synthesis methodology is also proposed to relax a circuit by ignoring the timing constraints on a subset of paths that are bottlenecks to retiming [53]. The resultant relaxed circuit shifts the path wall to a lower delay enabling additional VOS. This increases the range of VOS in which the timing error rate is acceptable [53]. A careful path relaxation could lead to graceful quality degradation (i.e., acceptable output) since not all intermediate computations are equally important. For instance, the paths that slightly contribute to PSNR improvement are relaxed in a DCT architecture [53]. Another technique is also proposed for the timing error acceptance in DCT/IDCT components [107]. This technique improves the quality-energy tradeoff in the VOS regime.

*2) Architecture:* When acting errors at the architectural level, the goal is to use relaxed specifications on components that can support approximation during execution and storage. Some approaches focus on a coarse-grained region of computation that is amenable for approximation [99]–[102]. The approximable region is then offloaded to a designated unit that uses less reliable cores, or VOS blocks, or neural processing elements. A methodology is proposed to enable VOS by generating approximate hardware blocks for meta-functions [99]. Meta-functions represent computational kernels commonly found in various application domains [99]. Application profiling has been used to train a neural network to mimic a region of application code [101]. The generated neural processing cores are coupled with the processor to enable VOS. In the same vein, a relaxed fault-tolerance is applied for implementation of a face-recognition neural network [102]. Error resilient system architecture (ERSA) is suited for applications consisting of a set of coarse-grained isolated tasks that can be expressed entirely with approximate computation [100]. ERSA isolates execution of an approximable data-intensive task from a control-intensive task. The former is executed on an array of relaxed reliability cores, while the latter is executed on a super reliable core.

Focusing on approximate storage, a recent writing mechanism enables applications to store data approximately [103]. It trades off accuracy for performance in multilevel cell accesses and leverages worn-out memory for approximate data instead of ignoring the cell.

*3) Software:* At the software level, accepting errors where possible naturally makes an execution inexact or approximate. The important question here how we distinguish such a computation from an exact one, and when/under what conditions is it acceptable? For instance, synchronization is a major bottleneck in scaling parallel programs. Relaxing synchronization points can be achieved through a program restructuring called relax [97], and check that systematically trades off quality results for performance. Similarly, atomic operations are relaxed in CUDA programs running in GPUs [98]. Atomic operations are typically used in sorting and reduction where threads must sequentialize writes to a variable. A compiler technique selectively bypasses atomic operations to generate a set of CUDA kernels with varying levels of approximation [98].

*4) Application/Algorithm:* These approaches enable the error tolerance at higher algorithmic level and program transformation. Reference [94] proposes a system called Green that allows programmers to approximate expensive functions and loops in a systematic manner. Green trades off quality of service for improvements in energy consumption, while providing statistical quality of service guarantees. Focusing on the loops, loop perforation [95] trades accuracy for performance by transforming loops to limit execution of a subset of their iterations. The technique distinguishes critical loops versus tunable loops. The loop perforation is only applied on the tunable loops whose perforation produces more efficient and still acceptable accuracy. Reference [96] proposes a new approach for program transformation that utilizes probabilistic reasoning as opposed to the use of standard discrete logical reasoning. The approach provides probabilistic guarantees for the results of the transformed program such that the deviation from the original program will rarely be large.

## IV. HYBRID APPROACHES

This section presents two new classes of approaches for the variability-tolerance: hybrid and cross-layer approaches. Hybrid approach fuses the aforementioned approaches of error handling, i.e., the three axis of Y-chart in Fig. 4. Combining these three approaches creates two sets of new hybrid approaches: 1) predicting and preventing with detecting and correcting timing errors described in Section IV-A; 2) detecting and correcting with accepting timing errors described in Section IV-B. Table 6 summarizes these hybrid approaches. On the other hand, cross-layer variability-tolerance can be realized by exchanging information across the layers of abstraction, i.e., the concentric rings in Fig. 4. Section IV-C covers these cross-layer approaches. The hybrid and cross-layer approaches enhance the scope of timing error handling and its efficiency.

### A. Predicting and Preventing With Detecting and Correcting Timing Errors

In this section, we describe a class of hybrid approaches that mix the prediction and prevention approach (see Section III-A) with the detection and correction approach (see Section III-B). The resulting hybrid approach relies upon both design time analysis to prevent most of probable timing errors and runtime measurement to correct any unforeseen timing error.

*1) Circuit:* Recovery-driven design is a design time approach that focuses on optimization of a core for a target timing error rate instead of the worst case design [132]. This design time optimizations can benefit from uncertainty-aware [45], and slack redistribution [46] techniques. The recovery-driven design also utilizes an online error recovery to correct the timing errors that are allowed to appear intentionally because of VOS. Reference [133] presents a forward timing error correction approach that

Table 6 Abstractions Versus Hybrid Approaches

| | Predicting and Preventing with Detecting and Correcting Timing Errors | Detecting and Correcting with Accepting Timing Errors |
|---|---|---|
| **Application Algorithm** | • Optimal application configurations [108] | • Approximate checks for sparse linear algebra [109] |
| **Software** | • Code transformations: instruction padding for improving timing speculation [56], dynamic code optimizer [110]<br>• Scheduling: voltage droop-aware thread scheduler [111], task dispatching [112], OpenMP schedulers for task-level vulnerability [113], [114] and work-unit vulnerability [115], GPU workload partitioning [116], latency tail-tolerant [117] | • OpenMP exact/approximate directives [118] |
| **Architecture** | • Microarchitectural event-guided [119]<br>• Collaborative architecture and compiler design [120]<br>• Recovery islands [121] | • ISA extensions: Truffle [122]<br>• Memoization: dynamic instruction reuse [123], region reuse [124], approximate vs. exact matching [125], [126], partial memristive memory-based computing using associative memory modules [127], [128]<br>• Approximate memories with asymmetric robustness: mixed-cell cache [129], energy vs. data integrity in SRAMs [130] and DRAMs [131] |
| **Circuit** | • VOS recovery-driven [132]<br>• Forward timing error correction [133]<br>• Redundant speculator with recovery [134] | • Accuracy-configurable blocks: adders [135], FPUs [118]<br>• Approximate error handling: approximate error correction [136], [137], approximate concurrent error detection [138] and masking [139] |

enhances the scope of TIMBER [83], [84] and time borrowing techniques [85], [86] without involving complex clock control. The approach synthesizes a timing error effect prediction logic based on Boolean differential equation to predict whether the timing errors occurred on a set of flip-flops would propagate to the next stage. In case of propagation, the errors are corrected systematically within the logic without time borrowing. An approximation method is proposed that focuses on circuit-level speculation for predicting results [134]. The approximation circuit implements a given logic function partially for reducing the logic delay of a stage. The approximated results are used to advance the pipeline. In case of a mismatch between the approximate result and exact result—computed by a duplicated logic function—a recovery mechanism is triggered to correct the effects of speculative executions.

2) *Architecture:* [119] provides an interleaved method that learns to predict the timing errors or corrects them if the prediction does not work. It has been achieved by partitioning the timing errors, that are induced by voltage droops, into predictable and unpredictable groups. This approach handles the predictable timing errors by avoiding voltage droops that are correlated with microarchitectural events, similar to preventive method discussed in [30]. The residual unpredictable errors are handled by a checkpoint-rollback mechanism. Further, it has been shown that microarchitecture and compiler collaborative design can lead to a cost-effective solution for dynamic voltage variations in commodity processors [120]. The work also provides a broad overview of possible directions to deal with voltage droops presented in [30], [110], [111], [119]. A design time methodology is presented in [121] to partition a given system-on-a-chip into recovery islands. Each island can therefore be recovered independently during operations.

3) *Software:* Software approaches can take into account the spatio–temporal effects of variability to reduce the overall cost of computation. These hybrid methods are divided into two main groups. The first group strives to improve resiliency of the program through a code transformer, while the second group improves scheduler to decide about a more reliable location for executing a given workload. Variability-aware workload scheduling monitors the variations and accordingly assigns workloads to proper computing cores for reducing the cost of error detection and correction. This can be extended to realize variability-aware memory allocation [69], [70] for adapting to the spatiotemporal variable memory modules. Fig. 8 shows integration of the variability-aware scheduler and memory allocator.

*Code Transformations:* These approaches probe an error prone region of the code, and then try to *reshape* it such that the code region exhibit lower errors. A recent work [56] makes the observation that some sequence of instructions can have a significant impact on the timing error rate. Consequently a code transformation is introduced which pads the instructions sequence with a NOP instruction. The NOP padding eliminates the critical path activation since the result is no longer forwarded directly from the execution stage. The ISA extension can further enhance the ability of a code transformer. For instance,

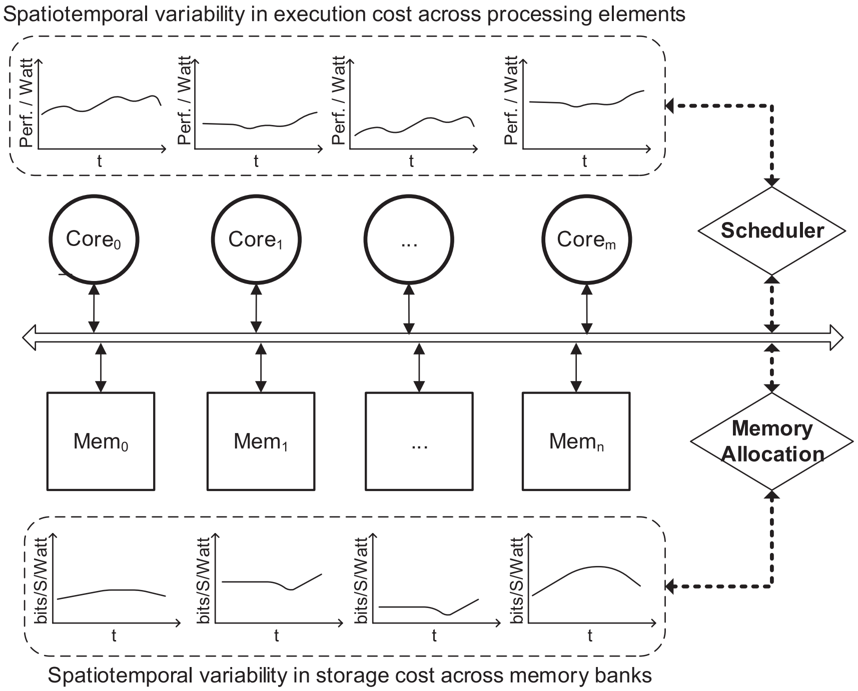

Spatiotemporal variability in execution cost across processing elements
Perf. / Watt
t
Core$_0$
Core$_1$
...
Core$_m$
Scheduler
Mem$_0$
Mem$_1$
...
Mem$_n$
Memory Allocation
bits/S/Watt
t
Spatiotemporal variability in storage cost across memory banks

**Fig. 8.** ***Variability-aware scheduler and memory allocator to combat spatiotemporal variations in processing elements and memory banks.***

[56] introduces BRINC instruction (broken increment) that performs addition in the lowest 4 bits of the operand, making it a more robust candidate for index computation in the unrolled loops. A code optimizer extracts problematic code sequences that cause the voltage droops, then the optimizer rearranges the instructions such that the voltage droops are suppressed [110].

*Scheduling:* Moving toward multicore architectures opens a choice for scheduling workload on cores such that the likelihood of timing errors will be reduced. In this trend, the voltage droop resiliency techniques are extended to multicore architectures. For example, a thread scheduler exploits the voltage droops phases to enhance scheduling decisions for smoothing out the voltage droops among the cores [111]. A variability-aware task dispatching technique enhances predictability and energy efficiency for multimedia streaming applications running on parallel multiprocessor arrays [112]. For processor clusters, a variation-aware task scheduling policy for OpenMP is presented in [113]. The OpenMP runtime environment computes metadata as the vulnerability of a task execution on a core, and uses this characterized information to reduce the timing error rate for the next task scheduling point. The variation-aware OpenMP is further extended to cover various constructs, including parallel sections and loops [115]. Using the notion of work-unit vulnerability, the timing errors are captured as descriptive metadata to characterize the impact of variability on different work-unit types running on various cores. As such, work-unit vulnerability provides a useful abstraction of hardware variability to efficiently allocate a given work-unit to a suitable core for execution [115].

Moving to GPUs, a recent work characterizes each application sensitivity to within-die frequency variations in the context of spatial multitasking [116]. The sensitivity information partitions the workload and enables variation-aware allocation of the resources to concurrently-executing applications on a GPU [116]. In large-scale Web services, the hardware variability combined with other shared resource issues can prevent the system to response to user actions quickly [117]. Therefore the role of software techniques to tolerate latency variability is crucial to realize systems so-called latency tail-tolerant [117].

*4) Application/Algorithm:* A new method is presented for adapting the application's algorithm at the software layer that can reduce the impact of the timing errors induced by process variation [108]. This enables an application to support multiple choices denoted by a software configuration set. Each configuration displays different performance and quality. Under process variation, an optimal software configuration can be chosen to maximize the application quality while meeting the performance constraints. This software configurations approach [108] is suitable for applications that are reconfigurable and adaptive, e.g., video encoding and decoding, multimedia stream mining, gaming, and embedded sensing.

## B. Detecting and Correcting With Accepting Timing Errors

In this section, we describe another class of hybrid approaches that mixes the detection and correction approach (see Section III-B) with the error acceptance approach (see Section III-C). The resulting hybrid approach detects timing errors and corrects them when it is necessary, or neglects them and ensures safety through a set of rules.

*1) Circuit:* The circuit techniques in this area offer either a configurable block that its precession can be selected during runtime, or provide a new way of handling timing errors with approximate operations.

*Accuracy-Configurable Blocks:* An accuracy-configurable integer adder offers two operating modes: exact and approximate [135]. During the exact operating mode the error detection and correction has to be applied, while in the approximate mode the errors can be ignored and left out uncorrected. The approximate mode has been implemented by power-gating the error correction module during execution of approximate operations [135]. Moving from the integer to single precision floating-point, a reconfigurable FPU dynamically switches between the accurate and approximate modes [118]. The approximate mode ignores the timing errors on the less significant $N$ bits of the fraction part where $N$ is reprogrammable memory-mapped register. The choice of operating mode is driven by the application requirement of the computational accuracy.

*Approximate Error Handling:* An approximate error correction method minimizes the error magnitude of large timing errors making it suitable for DSP accelerators [136], [137]. Upon an error detection, the approximate error correction stage corrects the error by generating an approximate response computed by interpolating both forward and backward samples. The backwards samples are collected by output buffers and the forward sample is approximately derived by pipeline lookahead.

Approximate concurrent error detection circuit is a nonintrusive technique which does not impose performance penalty for error detection [138]. Concurrent error masking based on approximate circuit [139] is able to mask errors dynamically and therefore there is no cost for recovery, for instance the rollback or the replay. A synthesis flow is then proposed to derive such approximate circuits that can target a specified input space for prediction and masking [139].

*2) Architecture:* Architectural approaches allow coexistence of exact and approximate instructions either by providing an ISA extension, or by applying different constraints for computational reuse. Memory blocks can also offer mixed cells with different levels of accuracy.

*ISA Extension:* Truffle [122] is a dual-voltage microarchitecture design that supports mapping of approximate EnerJ [140] programs through ISA extensions. It applies a high voltage for exact operations and a low voltage for approximate operations. Truffle duplicates all the functional units in the execution stage. Half of them are hardwired to a high voltage for executing the exact operations, while the other half operate at a low voltage for executing the approximate operations.

*Memoization:* Dynamic instruction reuse enables recalling of the outcome of an instruction in hardware tables, so a processor can reuse it temporally if the processor performs the same instruction with the same input values within a limited period of time before the entry is overwritten. To improve the hit rate of the table, recent reuse techniques [123], [124] seek to improve association of the entries of the table with *similar* inputs to the same output. These tolerant techniques rely upon the tolerance in the output precision of the multimedia algorithms to achieve high reuse rates, and work at the granularity of a FP instruction [123], or a region of dynamic FP instructions [124]. The dynamic tolerant region reuse is a method based on relaxing the conditions upon skipping regions of instructions by caching results of previous equal and also similar inputs [124]. Spatial memoization [125] and temporal memoization [126] techniques are proposed to use value locality inside data-parallel programs. These techniques recall (memorize) the context of error-free execution of an instruction, and then reuse the context to correct an errant instruction based on a matching constraint. Two matching constraints are considered: 1) exact matching constraint that enforces full bit-by-bit matching of the operands; and 2) approximate matching that relaxes the criteria of the exact matching during comparison of the operands by masking the less significant $N$ bits of the fraction parts. Another method increases spatiotemporal reuse of computational effort by utilizing nonvolatile associative memory modules (AMMs), particularly resistive memory-based computing [127], [128]. A profiling phase identifies frequent redundant computations, carefully prestore these key computations in the AMMs, and reuse them to avoid reexecutions.

*Approximate Memories With Asymmetric Robustness:* A mixed-cell cache architecture is proposed by allowing the use of both robust and nonrobust cells [129]. In the mixed-cell cache, nonrobust cache lines are more susceptible to failures at low voltage, while robust lines are resilient to such failures by using error-detection mechanisms (e.g., parity). A recent work applies aggressive voltage scaling (below the minimum $V_{DD}$) to the SRAM cells by exploiting features of the stored data in a given multimedia application [130]. The resulting errors are acceptable based on the produced image quality. Another example of energy versus data integrity tradeoffs is applied for embedded DRAMs [131]. It aims to reduce the very frequent refresh rate of DRAMs by revealing the statistical characteristics of the retention time, in addition to exploiting the error resilient nature applications.

*3) Software:* There have been approximate extensions to OpenMP through a set of extended custom directives[1] that allows a programmer to specify parts of a program that can be executed approximately or exactly [118]. A profiling technique is used to identify thresholds for tolerable error significance and error rate. This information guides OpenMP runtime scheduler to promotes execution units to accurate mode, or demotes them to approximate mode depending upon the annotated code region requirement.

*4) Application/Algorithm:* An algorithmic technique for error detection suitable for sparse problems is proposed in [109]. Sparse problems are often well structured, providing an opportunity for low-cost error detection through sampling. Only a representative random sample of the computations can be checked for error detection without missing major unseen errors. To do so, approximate random checking and approximate clustered checking are implemented for sparse linear algebra applications [109]. The former randomly samples the problem, and the latter performs sampling based on the problem's structure.

### C. Cross-Layer Variability-Tolerance

As it has been shown in Fig. 4 and Table 6, the variability can be handled at a specific layer. A system can implement the variability-tolerance technique using only a few layers of the system stack that simplifies the design of other layers at the expense of higher cost. On the contrary, cross-layer resilient systems distribute the responsibility for the error handling across the system stack, and can utilize available information at each level in the system stack to increase the computational efficiency [141]–[146]. Breaking rigid interfaces between the layers leads to more efficient computing systems where the hardware variability, instead of being hidden behind conservative specifications, is exposed to multiple layers of the system including circuit, architecture, software, and applications. For instance, underdesigned and opportunistic computing machines propose to explore the possibility of constructing computing machines that purposely expose hardware variations to various layers of the system stack including software [141]. This leads to flexible hardware-software stack and interface that use adaptation in software to relax the variation-induced guardbanding in the hardware design. Underdesigned and opportunistic machines take several ways to implement adaptation ranging from hardware power management to just-in-time recompilation strategies [141].

Bringing the soft errors into picture, dependable embedded systems consider involving several layers of the abstraction including hardware, system software and OS, and application [145]. The pyramid of dependable embedded hardware/software systems require to cooperate among layers to address adaptability of reliable systems. In a similar vein, to meet the application requirements a self-aware computing model exposes measurements and adaptation that traditionally are handled solely by hardware [147]. Such exposure allows hardware adaptations that can be specified by other parts of the system. A runtime decision system then coordinates with other, software based actions to use the available actions to meet application requirements with lower cost.

## V. OBSERVABILITY AND CONTROLLABILITY

An adaptive system can be characterized by an observe-decide-action loop. Observability is a measure for how well internal states of a system can be inferred by knowledge of its external outputs measures. Given the current state of the system a decision is made to meet the goal. Controllability denotes the ability to move a system around in its entire configuration space using only certain admissible manipulations. Both error observation and retaking control over them share a common spatial notion as they are bounded by a certain area of the chip. As a result, designing a robust system has to answer this question that *how fine* or *how coarse* we need to observe the errors and react. We classify observers and controllers into fine-grained and coarse-grained categories illustrated in Table 7.

### A. Observability

Observers, monitors, and sensors are designed to measure a quantitative metric. The desired sensing metric can be collected over a small or large area of the chip leading to fine-grained and coarse-grained measurements. Fine-grained observability often requires intrusive measurement within the internal structure of a desired block that leads to *in situ* monitoring. On contrary, coarse-grained observation is often done on the border of the desired block.

*1) Fine-Grained Observability:* Internal or *in situ* monitors such as Razor [75], [90], Razor II [162], Razor lite [163], and EDS [89] typically use double sampling with shadow latches through a delayed clock to detect process, voltage, and temperature variations. Intel resilient core [44] integrates EDS in the critical paths of each stage to detect the late transitions. Recently, a 45 nm decoupled 10-lane SIMD processor utilizes Razor for every lane in the specific context of data-level parallel architectures [19]. Bubble Razor [81] is based on two-phase transparent latch that does not require hold buffer insertion for protecting the short paths since the errors are only caused by the long paths taking more than a clock phase—unlike the flip-flop based design, in the two-phase transparent latch design, the short paths do not

[1] `#pragma omp approximate [clause]`
`#pragma omp accurate`.

Table 7 Granularity of Observability and Controllability

| | Observability (Sensing) | Controllability (Actuation) |
|---|---|---|
| **Coarse-grained** | • Replica circuits: Critical path monitors [148], tunable replica circuits [149], tunable replica bits [150],and design dependent monitors [151]<br>• Replica aging circuits: NBTI and oxide [152]<br>• On-chip sensors: ADC's and thermal sensors [153], droop detectors and thermal sensors [154], voltage droop sensors [155], repetitive sampling for measurement of high-frequency supply noise [156], phase-locked loop [157], ring oscillators [158], [159], and IDDQ testing [41] | • Adaptive per-core: voltage scaling [39], [41], [42], [160], frequency scaling [43], [44], [154], body biasing [154], and clock distribution with tunable-length delay [161]<br>• Cooperating feedback controllers: fast adaptive frequency scaling (8-10 cycles round-trip time), and slow adaptive voltage scaling [40] |
| **Fine-grained** | • In situ circuits: Razor [75], [90], Razor II [162], Razor lite [163], Bubble Razor [81], Pulse-triggered Razor [137], EDS [89], Subthreshold EDS [164], Canary [165], Transition-detector [43], TIMBER [83], [84], SlackProbe [166], and SRAM read/write stability [167]<br>• In situ aging circuits: NBTI and oxide [57] | • Bubble propagation and local staling [81], single-cycle local stalling [82], time borrowing and stretching clock period [85], [86], counterflow pipelining [75], instruction replay [44], and decoupling queues [19] |

cause any timing error. Canary circuit [165] uses the same mechanism of Razor but it delivers a delayed input signal to the main flip-flop rather than delaying the clock to checker part (the shadow latch). This does not require synthesis of a delayed clock tree. Instead of inserting circuit monitors at the end points, SlackProbe allows placing monitors at intermediate nets along the circuit paths [166]. Shifting focus from logic to memory, *in situ* current-sensing approach measures the read stability and write ability of the cells within an SRAM array without modifying the cell structure [167]. For slower variations, compact *in situ* aging sensors with digital outputs have been proposed to measure NBTI and gate oxide degradation [57].

*2) Coarse-Grained Observability:* Replica circuits or external sensors monitors are on the same die, but outside of the functional paths that enable nonintrusive operations. These coarse-grained observers are in twofold:

*Replica Circuits:* Compared to the *in situ* circuits, the replica circuits are less intrusive on system operation. Intel resilient core [44] places a tunable replica circuit (TRC) [168] per pipeline stage to monitor timing errors. TRC delay is tuned slower than the critical-path delays, therefore TRC might incur false positive. Similarly, critical path monitor (CPM) [148] measures the timing margin available to a circuit. IBM 8-core POWER7 employs five CPMs per each core to capture PVT variations and detect early wearout [169]. The replica circuits can be constructed from the topology of circuit under monitoring providing better tracking [151]. The replica scheme is also applied for the memory blocks. For instance, 8T SRAM arrays utilize a tunable replica bits (TRB) [150] therefore enables reduction of the minimum operating voltage. A replica aging sensor consists of two components, which are stressed and then separately measured for the gate oxide and NBTI degradation [152].

*On-Chip Sensors:* Foxton technology utilizes on-chip thermal sensors in conjunction with ADC to measure power and temperature of an Itanium family processor [153]. High-resolution, digital on-chip voltage droop sensors [155], [156] as well as thermal sensors [154] are widely used to measure distinct dynamic variations. These on-die variation sensors have been demonstrated in response to the slow-changing voltage, temperature, and aging variations. Less intrusive and low-overhead on-chip variability monitoring can be done by utilizing phase-locked loop [157] and ring oscillators [158], [159] available in the chip. For instance, a statistical data processing method can extract the operating voltage and temperature from several ring oscillators frequency measurements [159]. IDDQ testing is also used to determine the leaky processors to shift workload from them to efficient, less leaky ones [41].

*3) Hybrid Observability:* An adaptive MPSoC architecture combines both *in situ* and replica circuits to offer better observability [39]. This architecture mixes Razor II [162] as the *in situ* circuits with a distributed macro-block embedding several ring oscillators made of inverters, long wires, latches, XOR gates, large capacitive nets, etc. Razor II itself is not sufficient mainly because of two reasons: 1) it can detect only timing errors; it cannot anticipate the timing violations if the instrumented critical paths are not activated; and 2) razor-style is not able to measure the actual parameters variations, e.g., providing quantitative information about the internal voltage and temperature of the monitored circuit. Ring oscillators in conjunction with Razor II not only detect the timing errors, but also track the voltage and temperature fluctuations [39].

## B. Controllability

Likewise, retaking an error correction can be done at coarse-granularity or fine-granularity. Fine-grained error correction requires intrusive changes into the structure of the circuit or architecture to implant the ability of error handling, masking, or accepting. On the other hand,

Table 8 Resilient Single-Core Architectures

| Work | Architecture | Approach | Operation | PVTA | Sensing | Actuation | Validation |
|---|---|---|---|---|---|---|---|
| [154] | TCP/IP accelerator | predict-and-prevent | runtime | PVTA | $V_{DD}$ droop sensors [155] and thermal sensors | DFS and dynamic biasing | 90nm |
| [44] | LEON-3 (RISC) | detect-and-correct | runtime | PVT | in-situ [89] and replica [168] | DFS and instruction replay | 45nm |
| [90], [75] | Alpha (in-order) | detect-and-correct | runtime | PVT | fine-grained in-situ | DVS and clock-gating | 0.18 $\mu$m |
| [81] | ARM Cortex-M3 (RISC) | detect-and-correct | runtime | PVT | two-phase transparent latch timing | bubble propagation to gate off clock pulses | 45nm |
| [43] | ARM partial ISA (RISC) | detect-and-correct | runtime | PVT | transition-detectors | DFS | 65nm |
| [160] | 16-bit MSP430 MCU (RISC) | detect-and-correct | runtime | PVT | critical path replica sensing | DVFS | 65nm |
| [110] | Alpha (OoO) | predict-and-prevent | runtime | V | on-chip voltage sensor | dynamic binary optimizer | simulation |
| [71] | Floating point unit (FPU) | detect-and-correct | test time | P | post-fabrication delay measurements | voltage interpolation and variable latency | 0.13 $\mu$m |

coarse-grained error correction is less intrusive and often controls one of the CMOS knobs for adjusting the entire of desired block under control.

*1) Fine-Grained Controllability:* These techniques tweak the circuit at fine-grained that results recovery from errors within one cycle to few cycles. We review their cost of recovery cycle in ascending order. Once variation is detected in the current cycle, the timing error is compensated for the activated critical paths by dynamically stretching the clock period [85], [86]. Another resilient design is composed of a tunable-length delay, an on-die dynamic variation monitor, and a clock-gating mechanism [161]. The tunable-length delay avoids the critical path timing margin degradation during a fast voltage droop by extending the delay and changing the delay sensitivity to voltage in the clock distribution. Two local stalling techniques incur one cycle penalty for error correction that can be applied for both two-phase latch based design [81] or flip-flop based design [82]. Counterflow pipelining [75], and instruction replay [44] correct the errant operation by switching to multi cycles operation. Such techniques do not change the clock period but the latency of operation [44], [75]. In a coarser granularity, decoupling SIMD queues prevent error events in any single lane from stalling all other lanes to enable independent lane recovery [19]. However, the lanes are required to resynchronize when a micro-barrier (e.g., load, store instruction) is reached.

*2) Coarse-Grained Controllability:* CMOS knobs including voltage, frequency, and body bias have been leveraged at coarse-granularity of a core to mitigate variations. Among the available control knobs, adaptive frequency scaling is widely utilized in resilient implementations [43], [44], [154]. For instance, an on-die adaptive frequency controller changes clock frequency in response to error rate reported by transition detector circuits [43]. Bringing voltage into the picture, dynamic voltage and frequency scaling selects appropriate operating points for each core [39], [41], [42]. As the last knob, both NMOS and PMOS body bias generators have been utilized [154]. Adaptive biasing controller is based on a lookup table, which is indexed by the output of the sensors and is loaded with precharacterized data representing body bias values.

The 8-core POWER7 employs two cooperating feedback controllers to adaptively reduce the guardband [40]: the first one is an adaptive clock frequency controller that reacts quickly to voltage droops by coupling between the monitors output and a digital phase-locked loop. The round-trip time of this loop is 8–10 cycles from sensing through CPMs [148] back to actuating by the phase-locked loop. The phase-locked loop can adjust frequency for a wide range of 50%–125% of the nominal clock frequency. The second controller dynamically adjusts the processor voltage to achieve a desired performance level on a longer time scale. Another all-digital adaptive voltage scaling system features a dual control, too: fast frequency adaptation for protection against voltage droops and slow supply voltage adaptation for protection against process and temperature variations [160].

## VI. RESILIENT SYSTEMS: PUTTING IT TOGETHER

In the section, we illustrate how broad architectures can arrange the presented approaches to enhance their resiliency for various physical sources of variations: process, voltage, temperature, and aging (PVTA). Section VI-A considers approaches suitable for single-core architectures, while Section VI-B considers parallel architectures.

### A. Single-Core Architectures

Table 8 illustrates a list of resilient single-core architectures. The second column shows the type of architecture. The third columns focuses on the specific approach of variability tolerance used by the core; and its operation mode is listed in the next column. Given the ability of the core, the fifth column covers the type of PVTA

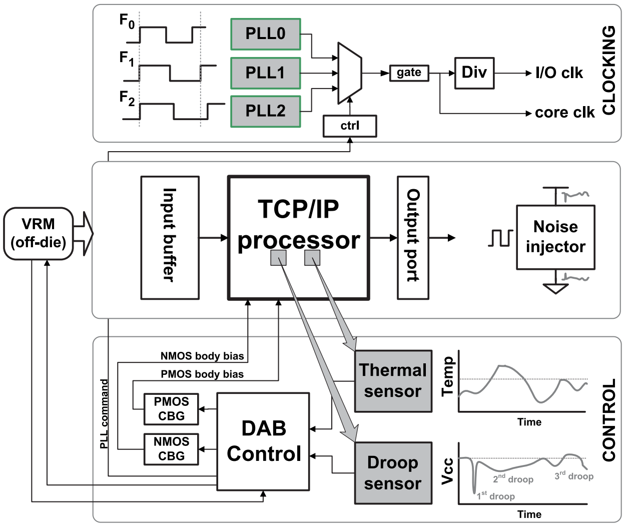


**Fig. 9.** ***Resilient TCP/IP core with voltage and thermal sensors, adaptive clocking and body bias generator [154].***

variations that can be handled. The next two columns describe the type of sensors and the actions used by the core. The eighth column comments on the validation process.

For instance, a TCP/IP accelerator core [154], shown in Fig. 9, approaches the variability using the prediction and prevention mechanism. The core can mitigate PVT variations and aging during the runtime operation by direct measurements of the variations using the voltage droop [155] and thermal sensors. This core utilizes fast single-cycle adaptive frequency (using three PLLs) and body biasing (NMOS and PMOS bias generators) techniques to deal with sudden changes in the temperature and supply voltage variations. A 45 nm LEON-3 RISC core relies on the detection and correction approach to mitigate the PVT variations during the runtime operation [44]. For detecting timing errors, the core utilizes both *in situ* EDS and replica TRC sensors. For correction, the core employs instruction replay and frequency scaling techniques. ARM-base processors [43], [81] unitize similar runtime techniques to detect and correct the PVT-induced timing errors. An out-of-order Alpha processor employs the on-chip voltage sensor to predict and prevent the voltage droops by code optimization [110]. As opposed to these runtime methods, a resilient FPU [71] relies only on the delay measurements during the test time after fabrication to mitigate the process variation.

## B. Parallel Architectures

Table 9 summarizes the details for resilient parallel architectures. The prediction and prevention as a common mechanism is used in most of the parallel architectures including, 8-core IBM POWER7 [40], 48-core Itanium [42], 16-core out-of-order SPARCv9 [41], and multimedia VLIW accelerators [25], [112]. To measure variations, IBM POWER7 uses CPMs [148], while 48-core IA processor

Table 9 Resilient Parallel Architectures

| Work | Architecture | Approach | Operation | PVTA | Sensing | Actuation | Validation |
|---|---|---|---|---|---|---|---|
| [40] | POWER7 (MIMD: 8 cores) | predict-and-prevent | runtime | PVT | 5 CPMs [148] per each of 8 cores | core-level fast DFS and slow DVS | 45nm |
| [42] | Many-core IA (48 cores) | predict-and-prevent | runtime | PT | on-die network of 48 thermal sensors | VA multiple voltage/ single frequency | 45nm |
| [41] | SPARCv9 (MIMD: 16 OoOs) | predict-and-prevent | runtime | P | IDDQ tests | per-core voltage/ frequency islands | simulation |
| [112] | Multimedia accelerator (ST231 + VLIW) | predict-and-prevent | runtime | PVT | core-level monitoring | task dispatching | simulation |
| [25] | MPSoC (SPMD: ST231 + VLIW) | predict-and-prevent | runtime | A | core-level monitoring | idleness distribution | simulation |
| [170] | Many-core (MIMD: 32×32 cores) | predict-and-prevent | offline | P | model: VARIUS [171] | task mapping | simulation |
| [39] | MPSoC (MIMD: 4 STxP70 cores) | detect-and-correct | runtime | PVT | 8 ring oscillator probes [159] and 64 RazorII [162] sensors per each core | local adaptive voltage and frequency scaling (Vdd-hopping [91] and digital frequency-locked loop [172]) | 32nm |
| [18] | ULP Processor Clusters (MIMD: 16 cores) | detect-and-correct | runtime | PVT | Razor [75] | Reconfigurable memory banks | simulation |
| [19] | Synctium-I (SIMD: 10 lanes) | detect-and-correct | runtime | PVT | Razor [75] | decoupling queues and lane weaving | 45nm |
| [173] | GPGPU | detect-and-correct | test time | P | $F_{max}$ measurements | disabling slowest core, or multiple clock domain (per-SM clocking) using GALS | simulation |
| [100] | LEON3 (MIMD: 1 SRC and 8 RRCs) | accepting error | off-line and runtime | PVT | light-weight execution monitors (watchdog timers, exceptions, memory protection units) | isolation of reliable task execution | FPGA |

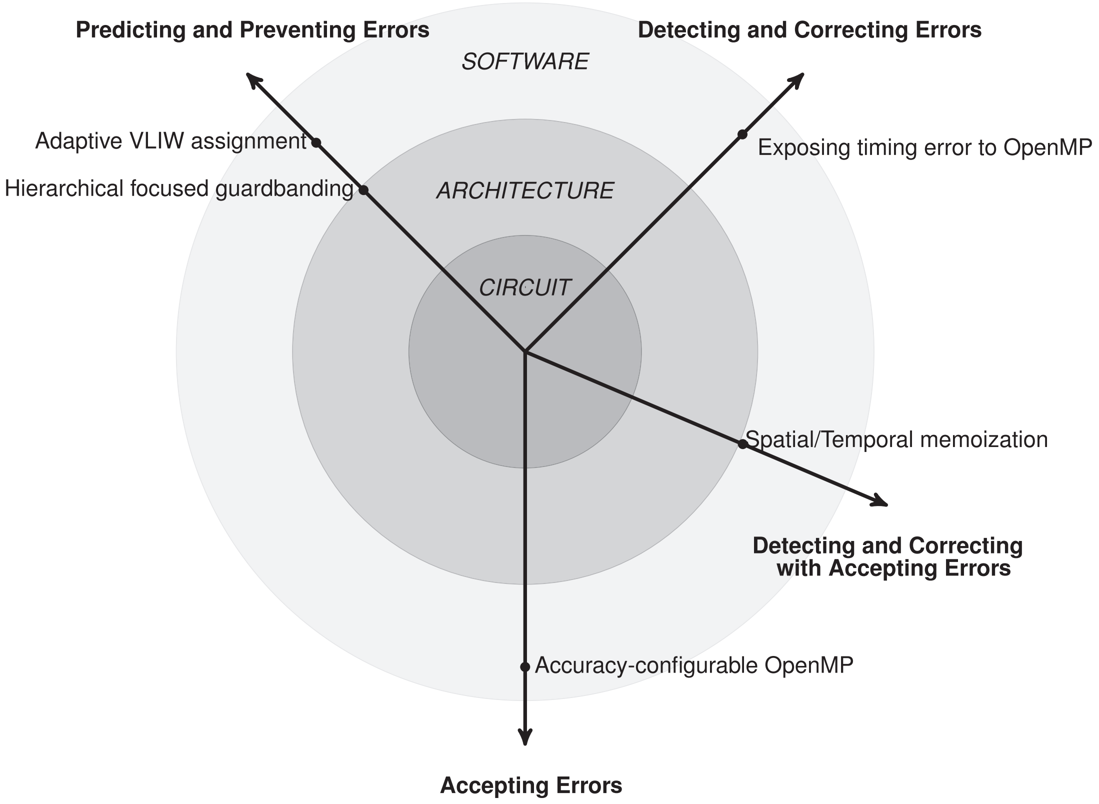

Predicting and Preventing Errors
Detecting and Correcting Errors
SOFTWARE
Adaptive VLIW assignment
Exposing timing error to OpenMP
Hierarchical focused guardbanding
ARCHITECTURE
CIRCUIT
Spatial/Temporal memoization
Detecting and Correcting with Accepting Errors
Accuracy-configurable OpenMP
Accepting Errors

**Fig. 10.** *Taxonomy of variation-tolerant parallel architectures.*

utilizes a network of 48 thermal sensors. Both multimedia accelerators focus on core-level monitoring: [112] computes an online policy on ST231 processor to mitigate PVT variation on VLIW engines, while [25] distributes idleness among VLIW engines to heal aging.

Detection and correction mechanism is the next popular approach used in the parallel architectures. Four STxP70 cores are interconnected together using a GALS network allowing them to benefit from local adaptive voltage and frequency scaling [39]. Every core is equipped with both ring oscillators [159]) and *in situ* Razor II [162] sensors. A 16-core resilient cluster [18] also uses Razor to bypass the slow memory banks which are shared within the cluster. Among data-level parallel architectures, a 10-lane SIMD decouples the lanes for independent error recovery during runtime operation [19], while a GPGPU disables the slowest core during test time to compensate the effects of process variations [173]. Finally, ERSA [100] can ignore the timing errors by executing the tolerable tasks on 8 relaxed reliability cores (RRCs).

## VII. VARIATION-TOLERANT PARALLEL ARCHITECTURES: A CASE STUDY

In this section, we provide a showcase of the earlier presented methods for their usage in the data-level and the task-level parallel architectures. The main focus of our attention for the data-level parallelism is on the SIMD and GPGPU architectures, and for the task-level parallelism is on the shared memory processor clusters. We show how we can choose, apply, and reuse a proper resilient approach given that limitations of such architectures. Fig. 10 illustrates another Y-chart of approaches suitable for parallel architectures. The main three axes include predicting and preventing errors (see Section VII-A), detecting and correcting errors (see Section VII-B), and accepting errors (Section VII-C). A new axis combines detecting and correcting errors with accepting errors to from the hybrid approaches described in Section VII-D.

### A. Predicting and Preventing Errors

We present two approaches suitable for GPGPUs that adaptively predict the delay variations and react accordingly to prevent the timing error.

*1) Adaptive VLIW Assignment:* For GPGPU architecture, we consider Evergreen family of AMD GPGPUs (a.k.a. Radeon HD 5000 series) that targets general-purpose data-intensive applications. The Radeon HD 5870 GPGPU consists of 20 compute units, a global front-end ultra-thread dispatcher, and a crossbar to connect the

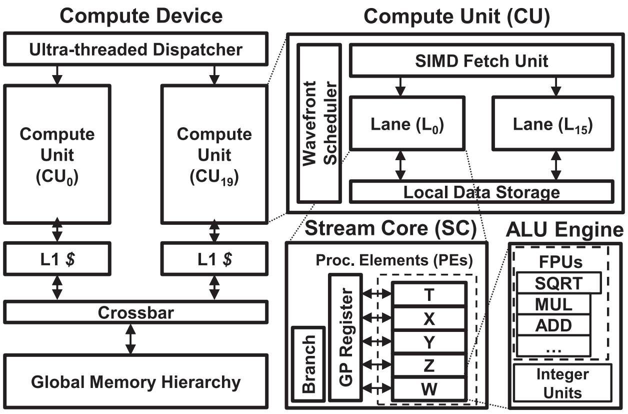


**Fig. 11.** ***Block diagram of the Radeon HD 5870 GPGPU.***

memory hierarchy. Each compute unit has 16 parallel lanes, or 16 Stream Cores (SCs). Within a compute unit, a shared instruction fetch unit provides the same machine instruction for all the lanes to execute in a SIMD fashion. Each lane or SC contains five Processing Elements (PEs) forming an ALU engine to execute Evergreen machine instructions in a vector-like fashion. Finally, the ALU engine has a pool of pipelined integer and FP units. The block diagram of the architecture is shown in Fig. 11.

We propose an online adaptive reallocation strategy to mitigate the NBTI-induced performance degradation in GPGPUs. As shown in Fig. 11, every stream core is a five-way VLIW processor capable of issuing up to five floating point scalar operations. Each VLIW slot is related to its corresponding PE. Four PEs (X, Y, Z, W) can perform up to four single-precision operations separately, while the remaining one (T) has a special function unit for transcendental operations. In an N-way VLIW processor, up to N data-independent instructions, available on N slots, can be assigned to the corresponding PEs and be executed simultaneously. On average, if M out of N slots are filled during a kernel execution, we call the achieved packing ratio is M/N. We observe that the instructions are not uniformly distributed among the PEs. For instance, the $PE_X$ executes roughly half of the ALU engine instructions (50.7%) during Reduction kernel execution, while only about one quarter of the ALU engine instructions (27.1%) are executed by $PE_X$ during Sobel kernel execution. Seven kernels selected from AMD APP SDK v2.5 [174] execute more than 40% of the ALU engine instructions only on $PE_X$. This nonuniform workload variation causes nonuniform aging among the PEs, and exhausts some PEs more than others and shortening their lifetime. Unfortunately, this nonuniformity happens within all the CUs since their workload is highly correlated together; therefore, no PE throughout the entire compute device is immune from this unbalanced utilization.

To address this issue, we propose an aging-aware compiler that uses a dynamic binary optimizer. The dynamic binary optimizer correlates the PE stress time-measured by online NBTI sensors—with instructions distribution, and equalizes the expected lifetime of the PEs. During the dynamic recompilation phase, the binary is optimized by customizing the kernel's code with respect to specific measured health state of GPGPU. This scheme uniformly distributes the stress of instructions throughout various VLIW resource slots, resulting in a healthy code generation that keeps the underlying GPGPU hardware healthy. The key idea of the aging-aware compilation is to assign the independent instructions uniformly to all slots: idling a *fatigued* PE and reassigning its instructions to a *young* PE through swapping the corresponding slots during the VLIW bundle code generation. This basically exposes the inherent idleness in the VLIW slots (average packing ratio of 0.3), and guides its distribution that does matter for aging. Thus, the job of the dynamic binary optimizer, for K-independent instructions, is to find K-young slots, representing K-young PEs, among all the available N slots, and then assign instructions to those slots. Therefore, the generated code is a *healthy* code that balances the workload distribution through various slots maximizing the lifetime of all PEs. The adaptation flow is illustrated in Fig. 12 and describes how these statistics can be obtained from the device, and how compiler can predict and thus control the nonuniform aging through four

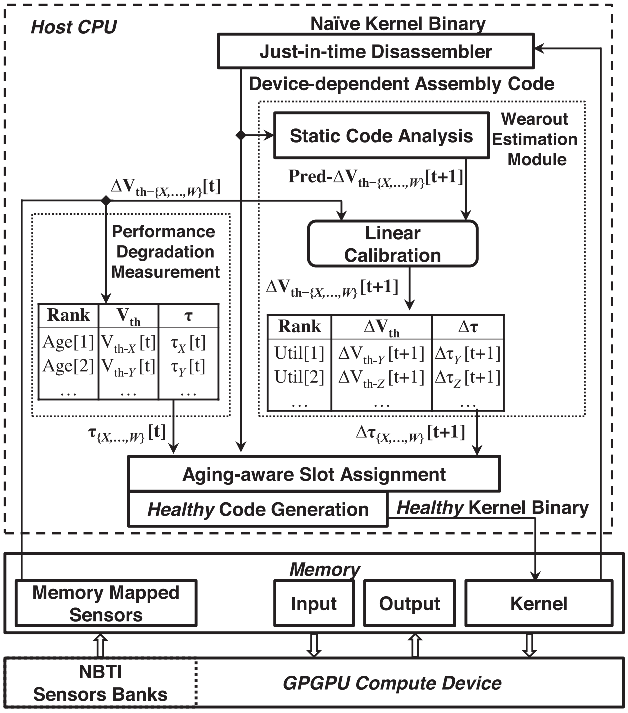

Host CPU
Naïve Kernel Binary
Just-in-time Disassembler
Device-dependent Assembly Code
Static Code Analysis
Wearout Estimation Module
$\Delta V_{th-\{X,\dots,W\}}[t]$
Pred-$\Delta V_{th-\{X,\dots,W\}}[t+1]$
Performance Degradation Measurement
Linear Calibration
$\Delta V_{th-\{X,\dots,W\}}[t+1]$
Rank | $V_{th}$ | $\tau$
Age[1] | $V_{th-X}[t]$ | $\tau_X[t]$
Age[2] | $V_{th-Y}[t]$ | $\tau_Y[t]$
... | ... | ...
Rank | $\Delta V_{th}$ | $\Delta\tau$
Util[1] | $\Delta V_{th-Y}[t+1]$ | $\Delta\tau_Y[t+1]$
Util[2] | $\Delta V_{th-Z}[t+1]$ | $\Delta\tau_Z[t+1]$
... | ... | ...
$\tau_{\{X,\dots,W\}}[t]$
$\Delta\tau_{\{X,\dots,W\}}[t+1]$
Aging-aware Slot Assignment
*Healthy* Code Generation
*Healthy* Kernel Binary
Memory
Memory Mapped Sensors
Input
Output
Kernel
NBTI Sensors Banks
GPGPU Compute Device

**Fig. 12.** ***Aging-aware kernel adaptation flow.***

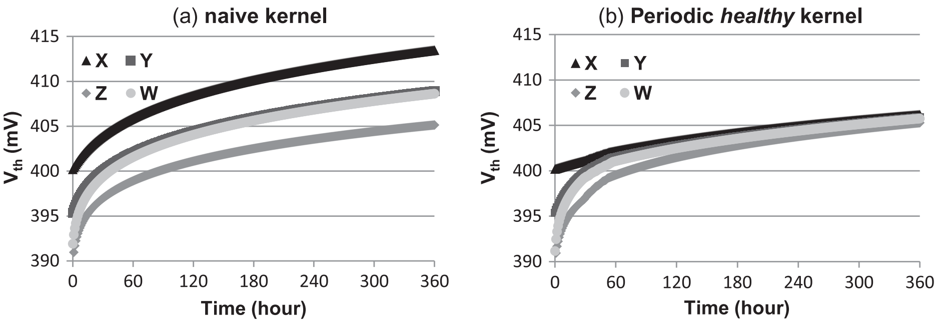


**Fig. 13.** ***Process variation and NBTI-induced for 360 hours: (a) the naive kernel; and (b) the periodic execution of healthy kernel.***

main steps: 1) reading the NBTI sensors; 2) kernel disassembling, static code analysis, and calibrating the workload predictions; 3) uniform slot assignment; and 4) healthy code generation.

We also evaluate the effectiveness of the proposed approach when executing the healthy kernel on a process variability-affected GPGPU with initial inter-PE of $\Delta V_{\text{th}} = 10$ mV. Fig. 13(a) shows the $V_{\text{th}}$ shift over time due to the naive kernel execution, and at the end of 360 hours there is an 8 mV $V_{\text{th}}$ variation among the PEs which limits the lifetime of $\text{PE}_X$ ($V_{\text{th}-X} = 413$ mV). On the other hand, Fig. 13(b) shows that adapting the kernel periodically leads to a uniform $V_{\text{th}}$ shift among all the PEs ($V_{\text{th}}$ variation is $\sim$0.6 mV), and the maximum $V_{\text{th}}$ shift is 406 mV at the end of 360 hours. [26] provides further details.

*2) Hierarchically Focused Guardbanding:* The notion of hierarchically focused guardbanding (HFG) is earlier introduced in Section III-A3. The HFG model takes into account the PVTA parameter variations, the clock frequency, and the physical details of placed-and-routed functional units (FUs) of GPGPUs through an ASIC analysis flow. The sensor instrumentation is required as the delay variation changes across the extreme corners of PVTA parameters. The question is what mix of monitors would be useful? Fig. 14 shows the minimum affordable clock period in the presence or absence of various sensors for three FUs. The sensors are sorted based on the time constant of the measured PATV parameters: from DC component to high-frequency components. For instance, the clock period of $\text{FP}_{\text{Add}}$ can be reduced from 1. 32 to 1.26 ns (a 0.06 ns guardband reduction) depends to the actual value of the WID process variation reported by a process monitor ($P_{\text{sensor}}$). The clock period can be further reduced to 1.08 ns if $\text{FP}_{\text{Add}}$ is equipped with the aging as well as the process sensor ($\text{PA}_{\text{sensors}}$). Adding the thermal sensor enables even 0.06 ns more reduction to 1.02 ns ($\text{PAT}_{\text{sensors}}$). Finally, considering the full set of sensors enables decreasing the clock period from 1.32 ns to 0.74 ns (a great guardband reduction of 0.58 ns) based on the measured values of variations reported by $\text{PATV}_{\text{sensors}}$. The more sensors we provide for a FU, the better conservative guardband reduction for that FU: the guardband can be reduced up to 8%, 24%, 28%, 44%, if we equip $\text{FP}_{\text{Add}}$ only with $P_{\text{sensor}}$, $\text{PA}_{\text{sensors}}$, $\text{PAT}_{\text{sensors}}$, and

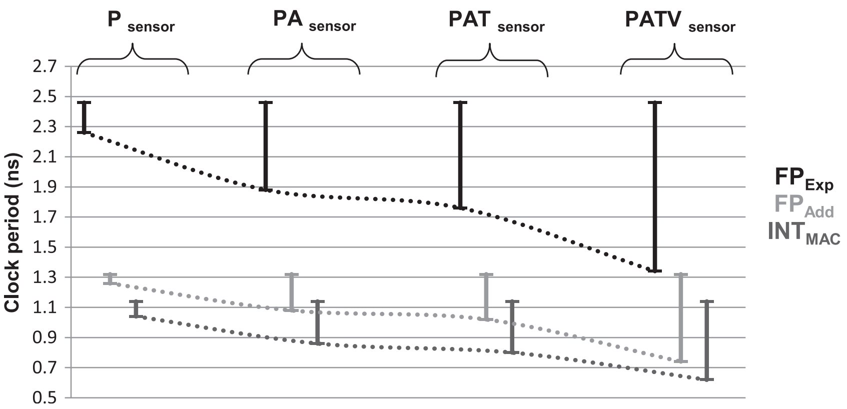


**Fig. 14.** ***Hierarchical sensors (sorted based on the time constant from DC to high-frequency components) for reducing guardband on the clock period.***

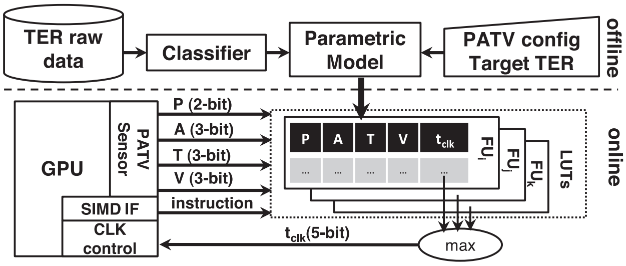


**Fig. 15.** ***Online utilization of characterized models through the HFG.***

PATV$_{sensors}$, respectively. As shown, this benefit is consistent across different FUs with a shift in the worst-case guardband. The FP FUs can exhibit even better guardband reduction (e.g., up to 47% for FP$_{exp}$ with PATV$_{sensors}$ case) due to the higher complexity of the circuit topology.

Employing any combination of PATV sensors provides the online measurement of the actual parameters variations, and thus a control system can adaptively apply an appropriate guardbanding utilizing the characterized models for the FUs. Among the available control knobs, the adaptive clock scaling is widely utilized in the resilient implementations [44], [154], [169]. Therefore, a control system can tune the clock frequency through an online model-based rule. To support the fast controller's computation, the parametric model (as the outcome of the ASIC analysis flow) can generate distinct lookup tables (LUTs) for every FUs. The LUTs are generated during the design time for specific configuration of the sensors, their resolution, and the desire target error rate for the FUs. Fig. 15 shows the online utilization of the LUTs with a full configuration of PATV$_{sensors}$.

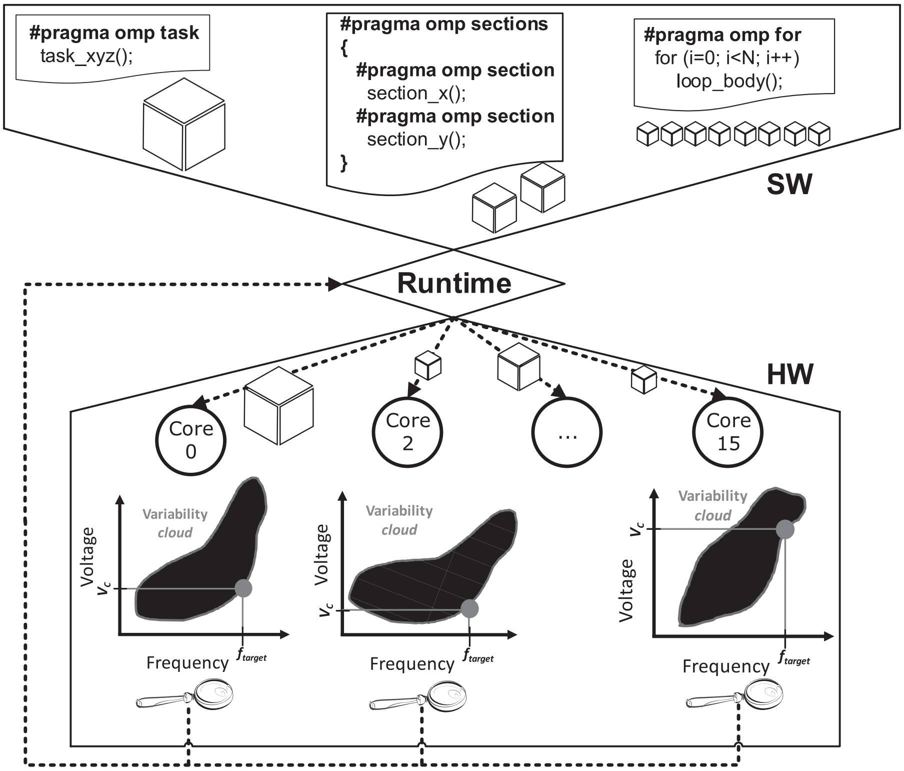
The next question to address is what type of monitoring observation granularity and what type of reacting time we need, e.g., cycle-by-cycle or tens of cycles or hundred of cycles? To analyze the effect of this choice of granularity, we apply the HFG to GPU architecture at two levels. **1**) Fine-grained granularity of instruction-by-instruction monitoring and adaptation that uses signals of PATV sensors come from the individual FUs that reside in the execution stage of GPU. The LUTs return the minimum clock period setting depending on the actual value of PATV sensors and the chain of FUs that will be activated by the fetched instruction. Thanks to the fast adaptive clocking circuit [154], the clock controller reduces the guardband that is compatible with PATV parameters and the demands of instructions. **2**) Coarse-grained granularity of kernel-level monitoring uses a representative PATV sensors for the entire execution stage of GPU pipeline. The clock adaptation is applied periodically before the kernel execution. The controller sets the clock period based on the current value of PATV sensors of the execution units and the chain of the FUs that potentially will be activated during kernel execution (in the static sense). Since the adaptation of clock during kernel execution is prohibited, the controller considers a 5% extra margin on the reported voltage and temperature values to recover the intrakernel dynamic variations. The presented guardbanding scheme is an adaptive resource management technique to proactively prevent the timing errors by applying a focused guardbanding. The HFG enhances the throughput of the GPU kernels by 70% employing the coarse-grained PVTA monitors and by applying the adaptive guardbands at kernel-level. The finer granularity of instruction-by-instruction monitoring and adaptation achieves 1.8×–2.1× throughput improvements depends to the PVTA monitors configuration and the type of instructions executed within the kernels. Reference [27] describes the HFG in detail.

## B. Detecting and Correcting Errors

We present mechanisms for a shared memory parallel architecture to detect and correct errors at various levels of the system. Generally speaking, the architecture consists of a set of computing units that enables various *options* for executing a given workload. One of these options, as our central focus, is the selection of an appropriate computing unit to execute the workload. This choice between alternative computing units enables parsimonious execution of the workload in the presence of timing errors. Having such a choice, enables abstracting the errors from lower levels to higher levels that can lead to efficient error handling and better management.

*1) Exposing Timing Error to OpenMP:* We consider OpenMP, as the de facto standard for parallel programming on the shared memory multicores systems. OpenMP is a combination of compiler directives and library routines that allows programmers to specify parallelism in their code without excessive details of parallel programming. Our shared memory architecture is a cluster of the processors that work together in MIMD fashion. Our goal is to provide runtime software support to increase the cost-effective countermeasures against the timing errors in hardware (see Fig. 16). We pursue this goal by exposing variability and its effect to the OpenMP programming model, thus enabling holistic variability management. We first describe how architectural support can expose the timing errors, then we describe online software characterization technique that enables variation-aware workload scheduling.

*Architectural support for exposing errors:* We now describe the architectural details of the variation-tolerant processing cluster, shown in Fig. 17. The architecture is inspired by STMicroelectronics Platform 2012 (P2012) [52], [176] as a programmable many-core accelerator for next-generation data-intensive embedded applications. In our implementation, we focus on a single cluster



```
#pragma omp task
task_xyz();
```

```
#pragma omp sections
{
  #pragma omp section
    section_x();
  #pragma omp section
    section_y();
}
```

```
#pragma omp for
for (i=0; i<N; i++)
  loop_body();
```

**Fig. 16.** *Timing error abstraction in OpenMP environment.*

consisting of 16 tightly-coupled 32-bit in-order RISC cores, a level-one (L1) tightly coupled data memory (TCDM) and a low-latency 16 × 32 logarithmic interconnection [175]. The TCDM is a software-managed scratchpad memory, configured as a shared, multiported, multibanked L1 memory that is directly connected to the logarithmic interconnection for fast accesses. The number of TCDM ports is equal to the number of banks (32) to enable concurrent access to different memory locations. Note that a range of addresses mapped on the TCDM space provides test-and-set read operations, which we use to implement basic synchronization primitives, e.g., locks.

The logarithmic interconnection is composed of mesh-of-trees networks to support *single cycle* communication between the cores and TCDM banks (see the left part of Fig. 17). When a read/write request is brought to the memory interface, the data is available on the

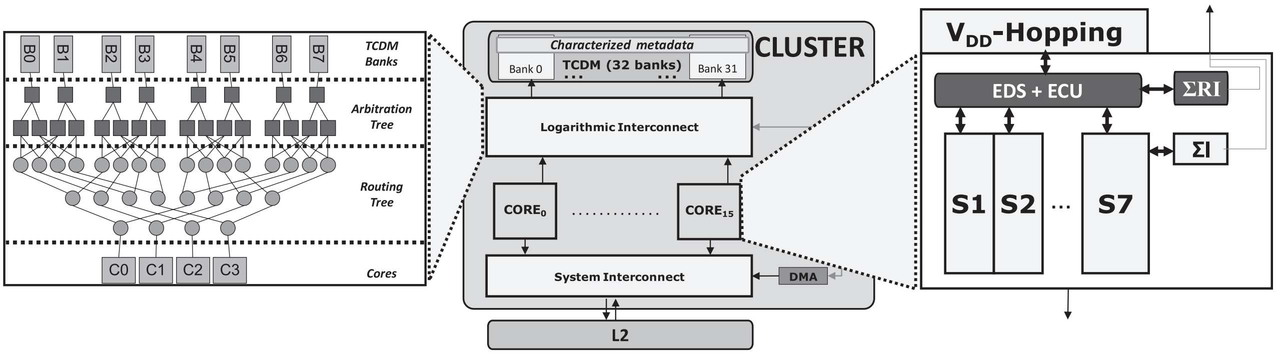


**Fig. 17.** *Variation-tolerant tightly-coupled processor cluster for OpenMP. The left part shows a 4 × 8 logarithmic interconnection [175]. The right part shows a resilient core that relies on error-detection sequential (EDS) [89] and error control unit (ECU) [44] to correct timing errors by the replica instructions; ΣI is the number of error-free executed instructions, and ΣRI is the number of replayed instructions.*

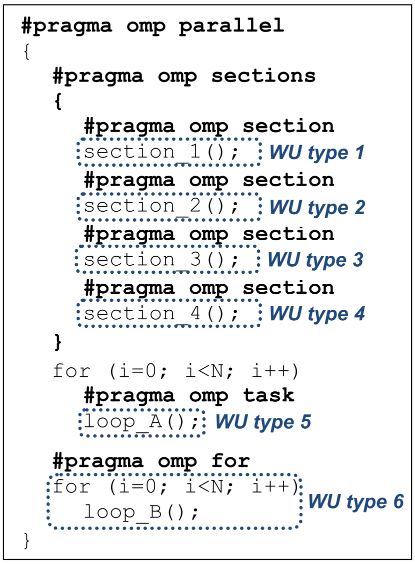
negative edge of the same clock cycle, leading to two clock cycles latency for a conflict-free TCDM access. The cores have direct access into the off-cluster L2 memory, also mapped in the global address space.

The cores within the cluster are equipped with two circuit-level resiliency techniques. First, each core relies on the EDS [89] circuit sensors to detect any timing error due to dynamic delay variation. To recover the errant instruction without changing the clock frequency, the core employs the multiple-issue instruction replay mechanism [44] in its error recovery unit (ECU). The ECU issues seven replica instructions (equal to the number of pipeline stages) followed by a valid instruction. Second, the cluster supports the $V_{DD}$ hopping technique [24], [39], [91], [92] that discretely tunes the voltage of slow cores—the cores that are affected by the static process variation. The $V_{DD}$ hopping improves the clock speed of the slow cores, thus enables all the components of the variability-affected cluster to work at same frequency. This technique avoids the intercore synchronization penalty that would significantly increase L1 TCDM latency. However, a core with higher vulnerability imposes extra cycles to correct the errant instructions. This varies the cost of recovery across the cores within a cluster.

*Online Work Unit Vulnerability Characterization:* OpenMP [177] consists of a set of compiler directives and library routines to specify parallel execution within a sequential code. Enclosing a code block within a `#pragma omp parallel` directive has the effect of launching multiple instances of that code over the available processors. Differentiating the actual work done by different processors in OpenMP is achieved by means of work-sharing constructs: `#pragma omp for`, `#pragma omp sections` and `#pragma omp task`. The `for` directive can only be associated to a loop nest, and distributes the loop iterations over available processors. Within a `sections` directive multiple section blocks can be specified, each containing a different parallel work-unit. `Sections` have limited expressiveness for describing task parallelism. For this reason, the latest OpenMP specifications have included the new `task` directive, which supports sophisticated forms of task parallelism. However, `task` implies significant overheads, which makes `sections` more convenient to outline few coarse grained tasks in a program.

As discussed earlier in Sections III-A3, B3, and IV-B3, the software-driven policies for the variability management need to characterize parallel work-units, WU, in terms of vulnerability to timing errors.[2] Each OpenMP work-sharing construct outlines an execution unit which runs a sequence of instructions. Enclosing portions of code within any of these constructs allows the programmer to statically identify several WU *types* in the program, as every directive syntactically delimits a unique stream of instructions. While at runtime the same stream may be dynamically instantiated several times (e.g., a work-sharing directive nested within a loop), from the point of view of our characterization it uniquely identifies a single WU type. As a direct consequence, there are as many types of WUs in a program as there are work-sharing directives in its code; for instance, as shown in Fig. 18, there are 6 WU types. Using the notion of work-unit vulnerability (WUV) [115], we capture the timing errors as high-level software knowledge for execution of the parallel work-sharing constructs, including `task`, `sections`, and `for` loop. WUV is a metric to estimate the execution time of each WU type per each core, under the hardware variability. This metric is quite useful for the purpose of simultaneous vulnerability measurement and load balancing. While the identification of WU types can be done statically at the compile time, WUV characterization has to be done online due to two main reasons. First, for a given fixed operating corner of a core, each WU type may exercise the core pipeline in a nonidentical manner. Consequently, each WU type may display different vulnerability depending upon composition of the various blocks of the core involved during execution of the WU (intra-corner WUV). Second, the characterization must reflect the variability-affected characteristic of every core (not known *a priori*) on every WU type. Since the amount of spatial and temporal variations changes from one core to another WUV is also a per-core metric (inter-corner WUV). WUV is defined as follows:

$$\mathrm{WUV}_{(i,j)} = \sum I + \sum RI \,|\, \forall\, \mathrm{core}_i, \forall\, \mathrm{WU\ type}_j \qquad (1)$$



```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    section_1();   WU type 1
    #pragma omp section
    section_2();   WU type 2
    #pragma omp section
    section_3();   WU type 3
    #pragma omp section
    section_4();   WU type 4
  }
  for (i=0; i<N; i++)
    #pragma omp task
    loop_A();   WU type 5

  #pragma omp for
  for (i=0; i<N; i++)
    loop_B();            WU type 6
}
```

**Fig. 18.** ***Outlined WU types in a OpenMP program:*** `task`, `sections`, `for` `loop`.

[2] Our platform does not have control over the errors happening while executing the library code. The functionality is preserved as each core is equipped with the replay mechanism.

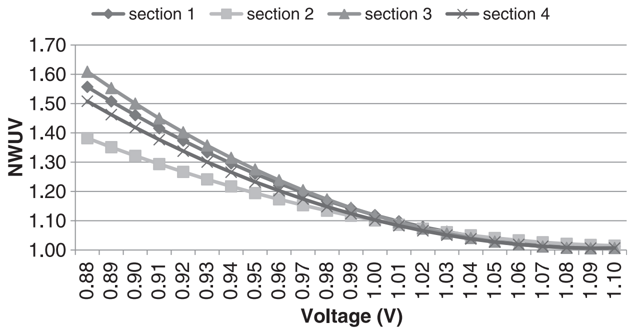


**Fig. 19.** ***Normalized WUV to voltage variations for 4 different sections types at 10 °C.***

where $\Sigma I$ is the number of error-free executed instructions; $\Sigma RI$ is the number of replayed instructions[3] during execution of WU type $j$ on core $i$, as reported by the ECU. Intuitively, for a given WU type if all the instructions run without any timing error, the corresponding WUV is equal to $\Sigma I$ as the total error-free dynamic instruction count. In the event of timing errors, WUV also accounts for the additional replica instructions. The lower the WUV, the lower number of recovery cycles, the lower the dynamic instruction count, and thus the higher throughput and energy efficiency.

To quantify WUV, the core collects $\Sigma I$ and $\Sigma RI$ statistics for (1) through the available counters in the ECU. This online characterization mechanism is distributed among all the cores in the cluster, thus enables full parallel WU execution monitoring and characterization. WUV is represented as a two-dimensional lookup table for different WU types and cores. This lookup table is physically distributed across all the banks of the L1 TCDM for fast parallel read/write operations (see the top part of Fig. 17). The OpenMP runtime accesses to the WUV metadata through a set of low-level APIs detailed in [115]. We now examine the WUV for four different `section` types when executing on fixed and variable operating corners. Fig. 19 shows normalized WUV value, as a metric which divides WUV value to its $\Sigma I$, for a fixed temperature of 10 °C with a supply voltage range of 0.88 V–1.10 V. The WUV for `sections` increases at the lower voltages. For example, the voltage variation of 0.22 V increases the intercorner WUV for `section 3` by 60% which means this WU might experience up to 60% higher timing errors depending upon the operating voltage of the core. Considering fix corners, among the four `sections` types, a maximum of 16% intracorner WUV is observed at (10 °C, 0.88 V) and a minimum of %1 intracorner WUV at (10 °C, 1.10 V). Hence based on WUV values, OpenMP runtime schedulers can optimize the system performance or energy efficiency by matching variability-affected core characteristics to WU types.

*Variation-Aware Work Unit Scheduling:* WUV consists of descriptive metadata to characterize the impact of variability on different work-unit types running on various cores. As such, WUV provides a useful abstraction of hardware variability to efficiently allocate a given work-unit to a suitable core for execution. The variation-aware OpenMP enables hardware/software collaboration with online variability monitors in hardware and runtime scheduling in software. We propose a set of scheduling algorithms in [115], that implement software-only countermeasure schemes, one for each work-sharing construct. Hence, the OpenMP runtime scheduler utilizes WUV metadata during scheduling to reduce the cost of error recovery caused by execution of a specific work-sharing constructs. Here, we focus on a reactive policy for variability-aware task scheduling (VATS) shown in Algorithm VII.1. This scheduler leverages the characterized WUV metadata to allocate tasks to cores so as to minimize both overall number of instruction replays and unbalanced loads. The main goal of this scheduler is to prevent allocation of tasks to unreliable cores.

**Algorithm VII.1** : VATS($\text{task}_j$)

```
for $i \leftarrow 1$ to $\text{N}_{\text{core}}$
    do $\begin{cases} \text{load}_i \leftarrow \text{loadQueue}_i + \text{WUV}(\text{core}_i, \text{task}_j) \\ \text{min} \leftarrow \text{findMinimum}(\text{load}_i) \end{cases}$
$\text{Queue}_{min} \leftarrow \text{insert}(task_j)$
return $(min)$
```

VATS scheduling policy strives to minimize the number of replayed instructions utilizing characterized WUV metadata. VATS also extends its awareness of the load on each queue associated with a core, thus avoids heavily unbalanced situations that could increase the total execution time. Each queue descriptor is enhanced with a status register that estimates the overall load (loadQueue), in terms of dynamic instructions count, of all tasks present into that queue. This is a better metric for workload-awareness than just the total task count, because different task types present in the queue may have various computational weight. To account for imbalance effects due to nonhomogeneous task durations and other system-level issues, VATS is further enhanced with a *most loaded queue-first* stealing algorithm.

Our experimental results indicate that the entire cost of online software characterization and countermeasures is paid off for the variability-affected 16-core cluster. The proposed OpenMP environment also saves both energy and total execution time for a wide range of parallelized applications. VATS reduces the execution time by 3%–36% and energy by 2%–46% for applications parallelized with `task` directives. The variation-aware scheduling for applications using `sections` directives also reaches to energy saving of 15%–50% and faster execution of 26%–49%. For further details about the error abstraction, you can refer to [113]–[115] and [178].

[3] Proportional to the number of errant instructions.

### C. Accepting Errors: Approximate Computing

We present techniques to enhance OpenMP and the shared-memory architecture presented in Section VII-B1 to support the approximate computing.

*1) Accuracy-Configurable OpenMP:* We propose a tightly-coupled processing cluster with shared, variation-tolerant, and accuracy-reconfigurable floating-point units (FPUs). This resilient shared-FPUs architecture supports online timing error detection, correction, and characterization. We introduce the notion of FP pipeline vulnerability (FPV), captured as metadata, to expose variability and its effects to a software scheduler for reducing the cost of error correction.

Our goal is to reduce the cost of a resilient OpenMP environment which is dominated by the error correction in the FPUs. Tolerance to error in execution is often a property of the application: some applications, or their parts, are tolerant to errors, while some other parts must be executed exactly as specified. We either explicitly ignore the timing errors—if possible—in a fully *controlled* manner to avoid undefined behavior of programs, or we try to reduce the frequency of timing errors by assigning computations to appropriate pipelines with lower vulnerability.

Using the notions of approximate and accurate computations, we describe a compiler and runtime environment to use approximate computations in a user- or algorithmically-controlled fashion. This is achieved via design-time profiling, synthesis, and optimization in conjunction with runtime characterization techniques. This approach eliminates the cost of error correction for specific annotated approximate regions of code if and only if the propagated error significance and error rate meet application-specific constraints on the quality of the output. For error-tolerant applications our OpenMP extensions specify parts of a program that can be executed approximately, thus providing a new degree of scheduling flexibility and error resilience. At design-time, code regions are profiled to identify acceptable error significance and error rate. This information drives synthesis of an application-driven hardware FPU. At runtime, as different sequences of OpenMP directives are dynamically encountered during program execution, the scheduler promotes FPUs to accurate mode, or demotes them to approximate mode depending upon the code region requirements. A ranking scheduler also utilizes the FPV metadata to identify the most suitable FPUs for the required computation accuracy for the minimum timing error rate.

*Accuracy-Configurable FPUs:* We extend the baseline cluster architecture with our resilient shared-FPUs. Similar to the DMA, our FPU design is also controlled via memory-mapped registers, accessible through a slave port on the peripheral interconnect. As shown in the rightmost part of Fig. 20, the FPU has three pipeline blocks which work in parallel. Each pipeline's inputs and outputs are retrieved from a minimal register file (one register file per pipeline to allow for parallel execution). For each pipeline there is a write-only *opmode* register that determines whether the current operation is accurate (i.e., exact) or approximate. Every pipeline block has two dynamically reconfigurable operating modes: 1) accurate; and 2) approximate. To ensure 100% timing correctness in the accurate mode, every pipeline uses the EDS sensors as well as the ECU to detect and correct any timing errors. During the accurate operation if a timing error is detected, the EDS circuits prevent pipeline from writing results to the register and thus avoid corrupting the architectural state. To recover the errant operation, the ECU adopts the multiple-issue operation replay mechanism [44].

In the approximate mode, the pipeline simply disables the EDS sensors on the less significant *N* bits of the fraction. The sign and the exponent bits are always protected by EDS. This allows the pipeline to ignore any timing error

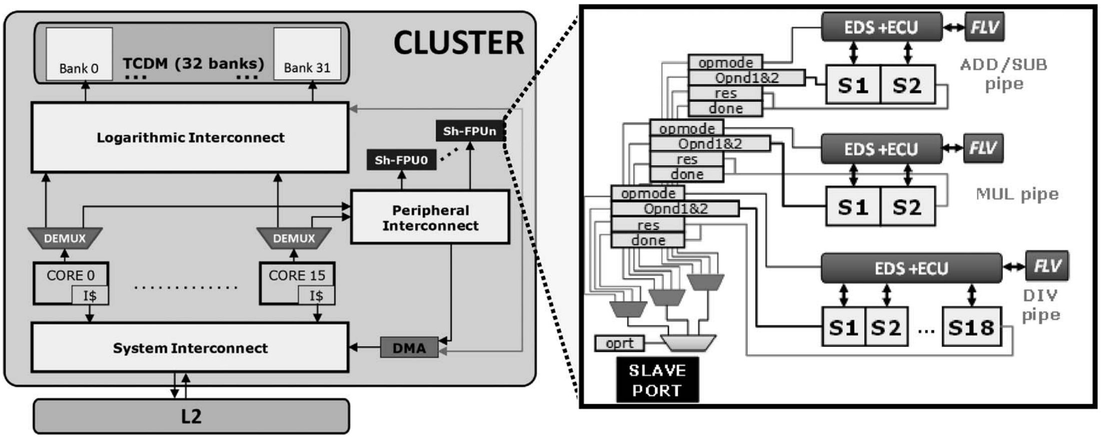


**Fig. 20.** *Variability-aware cluster architecture with accuracy-configurable FPUs.*

below the less significant *N* bits of the fraction and save on the recovery cost. We only disable the error detection circuits partially on *N* bits of the fraction. This enables the FP pipeline for executing the subsequent accurate or approximate software blocks without any problem in power retention. Further, this ensures that the error significance threshold is always met, but limits the use of the recovery mechanism to those cases where the error is present on the most significant bits. To characterize vulnerability of every FP pipeline to the timing error, we use FPV which is defined as the percentage of cycles in which a timing error occurs on the pipeline reported by the EDS sensors. To compute FPV, the ECU dynamically characterizes this per-pipeline metric over a programmable sampling period. The characterized FPV of each pipeline is visible to the software through the memory-mapped registers. Thus, the runtime scheduler leverages this characterized information for better utilization of FP pipelines, for example, it can assign fewer operations to a pipeline with higher FPV metadata. The runtime scheduler can also demote an error-prone pipeline to the approximate mode.

*OpenMP Compiler Extension for Approximation:* We provide two custom directives to OpenMP to identify approximate or accurate computations with an arbitrary granularity determined by the size of the structured block enclosed by the two custom directives

```
#pragma omp accurate
  structured-block

#pragma omp approximate [clause]
  structured-block.
```

The `approximate` directive allows the programmer to specify the tolerated error for the specific computation through an additional *clause*

```
error_significance_threshold (<value N>).
```

The error is specified as the least significant *N* bits of the fraction. By default, if the programmer does not specify an error significance threshold, it is assumed zero-tolerance (i.e., the `approximate` directive behaves as the `accurate`). By using this clause the `approximate` structured blocks have deterministic fully-predictive semantics: the maximum error significance for every FP instruction of the structured block is bound below the less significant *N* bits of the fraction. Moreover, any approximate instruction cannot modify any register other than its own. Let us consider the code snippet for Gaussian filter in Fig. 21. Here, the programmer has indicated the whole parallel block as the accurate computation, with the exception of the FP multiplication and accumulation of the input data. These two operations are annotated for the approximate computation with a tolerance threshold of less significant 20 bits of the fraction derived from a profiling stage. We use a profiling technique [118] to identify tolerable error significance and error rate thresholds in error-tolerant image processing applications. The compiler transforms the blocks to appropriate API calls implemented through the runtime library.

```
#pragma omp parallel
{
 #pragma omp accurate
 #pragma omp for
 for (i=K/2; i <(IMG_M-K/2); ++i) {
 // iterate over image
  for (j=K/2; j <(IMG_N-K/2); ++j) {
   float sum = 0;
   int ii, jj;
   for (ii =-K/2; ii<=K/2; ++ii) {
   // iterate over kernel
    for (jj = -K/2; jj <= K/2; ++jj) {
     float data = in[i+ii][j+jj];
     float coef = coeffs[ii+K/2][jj+K/2];
     float result;
     #pragma omp approximate   \
               error_significance_threshold(20)
       {
        result = data * coef;
        sum += result;
       }
    }
   }
   out[i][j]=sum/scale;
  }
 }
}
```

**Fig. 21.** ***Code snippet for Gaussian filter utilizing OpenMP approximation directives.***

*Runtime Support:* The runtime library is a software layer that lies between the variation-tolerant shared-FPU architecture and the compiler-transformed OpenMP program. The goal of our runtime scheduler is to inspect the status of the FPUs and allocate them to `approximate` or `accurate` software blocks to reduce the overall cost of timing error correction. This is accomplished in a twofold manner: 1) the runtime scheduler reduces the number of recovery cycles for accurate blocks by favoring utilization of FPUs with a lower FPV, thus lower the error rate and energy; and 2) the scheduler further reduces the cost of error correction by deliberately propagating the error toward the program, thus excluding the correction cost. The latter guarantees the quality of service for `approximate` blocks by demoting FPUs to the approximate mode for ignoring errors that match the tolerance expressed via the error significance threshold clause.

To allow for quick selection of best suited units for the accuracy target at hand, our scheduler ranks all the individual pipelines based on their FPV. The scheduler traverses the sorted list, starting from the head, until it finds an available pipeline. Once the target FP pipeline has been identified, it is configured to the desired operation mode on-the-fly, and a handler is returned to the program for offloading the consecutive FP instruction. Using this,

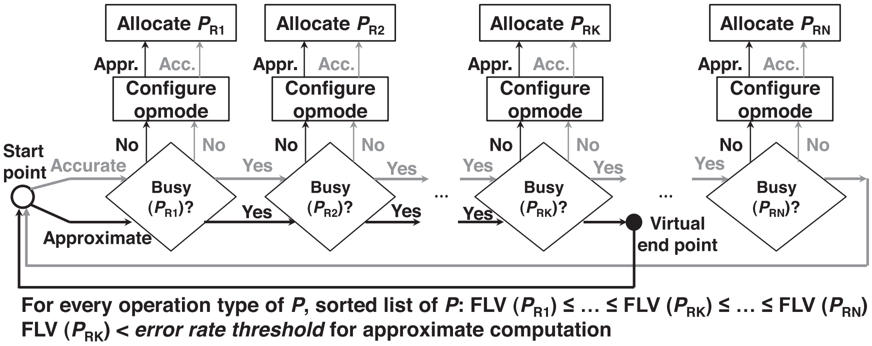


**Fig. 22.** ***Runtime scheduling based on FPV ranks.***

for every type of FP operations the ranking algorithm tries to highly utilize those pipelines with a lower FPV (and rarely allocate operations to the pipelines at the end of list), thus the aggregate recovery cycles for execution of FP operations will be reduced. Fig. 22 illustrates the ranking algorithm. For the approximate operations, in case of specifying an error rate threshold the scheduler limits its search to a certain element of the sorted list, e.g., until the $K$-th pipeline in Fig. 22. As soon as the scheduler finds a pipeline which has a higher FPV than the error rate threshold, it marks it as the virtual end point of the list for the approximate operations. Therefore, for the following approximate requests, the scheduler starts from the start point of the sorted list, and traverses down toward the virtual end point of the corresponding sorted list for finding a free pipeline. However, this virtualization technique limits the available parallelisms.

The presented collaborative OpenMP environment enables efficient execution of finely interleaved approximate and accurate operations enforced by various computational accuracy demands within and across applications. We demonstrate the effectiveness of our approach on a 16-core tightly-coupled cluster in the presence of timing errors. For the general-purpose error-intolerant applications, our approach reduces the recovery cycles that yield an average energy saving of 22% (and up to 28%), compared to the worst-case design. For the error-tolerant image processing applications with annotated approximate directives, 36% energy saving is achieved while maintaining acceptable quality degradation. In case of simultaneous execution of approximate and accurate applications, our approach avoids the overhead of frequent switching between the accurate and approximate modes which is imposed by interference of the accurate and approximate operations. More details about this work can be find in [118].

### D. Detecting and Correcting With Accepting Errors

In Section VII-B1, we have shown how a shared memory cluster of processors can schedule parallel work-units to address errors utilizing the fact that runtime system has the ability of choosing a favor core in close spatial proximity. On the contrary, such a choice of unit is not available in the data-level parallel architectures where the workload is uniform (SIMD) and all the computing units are fully utilized. Since such architecture has no choice for any alternative execution, it can utilize memoization or computational reuse that return a prestored result without triggering the recovery.

GPGPUs execute workload in SIMD fashion with high utilization. Parallel execution in such SIMD architectures provides an important ability to reuse computation (i.e., memoization) and reduce the cost of recovery from timing errors. We rely on the memoization to safely store the result of a portion of computing on a reliable medium, and then reuse the result rather than reexecution. To do so, we define two notions of memoization at the instruction level: concurrent instruction reuse (CIR), and temporal instruction reuse (TIR). Fig. 23 shows that for a SIMD architecture:

- CIR answers whether an instruction can be reused spatially across various parallel lanes;
- TIR answers whether an instruction can be reused temporally for a lane itself.

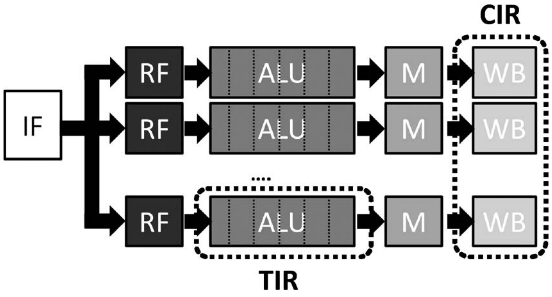


**Fig. 23.** ***Concurrent and temporal instruction reuse (CIR and TIR) for SIMD.***

CIR/TIR recalls the result of an error-free execution on an instance of data, then reuses this memoized context in case of meeting a matching constraint. Since different programs exhibit varying degrees of error tolerance, we consider two matching constraints that further extend the application of the memoization to approximate computing domain:

1) exact matching constraint that enforces full bit-by-bit matching of the single-precision instructions;
2) approximate matching that relaxes the criteria of the exact matching during the comparison by ignoring mismatches in the less significant $N$ bits of the fraction parts.

The latter constraint enables an *approximate error correction* technique suitable for applications in approximate computing to receive further benefits form the memoization technique. In a nutshell, the spatial and temporal memoization techniques leverage inherent value locality and similarity of applications by memoizing the result of an error-free execution on an instance of data; and by reusing this memoized result to exactly (or, approximately) correct any errant execution on other instances of the same (or, adjacent) data at a very low-cost.

These two techniques are fully compatible with the standard CMOS process. In [127], [128], we extend usage of such spatial and temporal reuse techniques in designing associative memory modules (AMMs) by leveraging the emerging CMOS-friendly memristor technology briefly described in Section IV-B2.

*1) Spatial Memoization (Concurrent Instruction Reuse):* To exploit the inherent spatial value locality across SIMD lanes, we propose a SIMD architecture consisting of a single strong lane and multiple weak lanes (SSMW). The SSMW is designed to maintain the lockstep integrity in the face of timing error. The key idea, for satisfying both resiliency and lockstep execution goals, is to always guarantee error-free execution of a strong lane (SS). Then, the rest of weak lanes (MW) can reuse the output of SS lane in the case of timing errors. In other words, SSMW provides an architectural support to leverage CIR for correcting the timing errors of MW lanes.

To measure the exposed spatial value locality over the parallel lanes, we have defined concurrent instruction reuse (CIR) as a metric for the entire kernel execution. CIR is defined as the number of simultaneous instructions executed on the lane1 ($L_1$) through $L_{15}$ of the CUs which satisfy the matching constraint, divided by the total number of instructions executed in all 16 lanes ($L_0 - L_{15}$). The matching constraint determines whether there is a value locality between the input operands of the instruction executing on $L_0$ and the input operands of another instruction executing on any of the neighbor lanes, i.e., $L_i$, where $i \in [1, 15]$. Thus, a tight (or, relaxed) matching locality constraint ensures that the instructions of $L_0$ and any of $L_i$ are working on the same (or, adjacent) instance of data, and consequently their outputs are equivalent (or, almost equivalent). This exchangeability allows the instructions of $L_0$ to correct any errant output of instructions executing on $L_i$. In the Radeon HD 5870 with 16-wide SIMD pipeline, the maximum theoretic CIR is 93.75% (15 out of 16).

Fig. 24 shows the CIR rate and the corresponding PSNR for various input pictures while using different matching constraints. As shown in Fig. 24(c), applying the exact matching constraint yields, on an average, a CIR rate of 27%. This means that 27% of the executed instructions on the whole SIMD can reuse the results of the executed instructions on the $L_0$ (SS lane) for the accurate error correction, without any quality degradation. Approximate matching relaxes the matching criteria through masking the less significant 12 bits of the fraction parts during comparison. Consequently, higher multiple data-parallel values fuse into a single value, resulting in a higher CIR rate for approximate error correction, e.g., up to 76% for Sobel. Applying the approximate matching, on average a CIR rate of 51% (32%) is achieved on the Sobel (Gaussian) filter with the acceptable PSNR of 29 dB (39 dB).

*2) Single Strong Multiple Weak (SSMW) Architecture:* We exploit the inherent value locality, therefore the SIMD is architected to maintain the lock-step integrity in the face of timing error: SSMW architecture, a resilient SIMD architecture. The key idea, for satisfying both resiliency and the lock-step execution goals, is to always guarantee error-free execution of a lane (SS). Then the rest of lanes (MW) can reuse its output in case of timing errors. In other terms, SSMW provides an architectural support to leverage CIR for correcting the timing errors of MW lanes. Note that to achieve this goal, SSMW *superposes* resilient circuit techniques on top of the baseline SIMD architecture without changing the flow of execution. SSMW employs two circuit resilient techniques. First, it guarantees the error-free execution of the SS lane in the presence of the worst-case PVT variations using voltage overdesign (VO). On the other hand, the MW lanes employ EDS to detect any timing error and propagate an error bit toward the tail of pipeline stages.

Second, SSMW also employs a CIR detector module for every PE of the MW lanes, as shown in Fig. 25. This module checks the matching constraint, and if it is satisfied, the module forwards the output result of the PE in the SS lane to the output of the corresponding PE in the weak lane. In case of simultaneous matching and timing error for any of the MW lanes, the errant weak lane can reuse the result of SS lane rather than triggering the recovery mechanism. The output result of the SS lane is broadcast via a voltage overdesign network across the MW lanes. The CIR detector module is a programmable

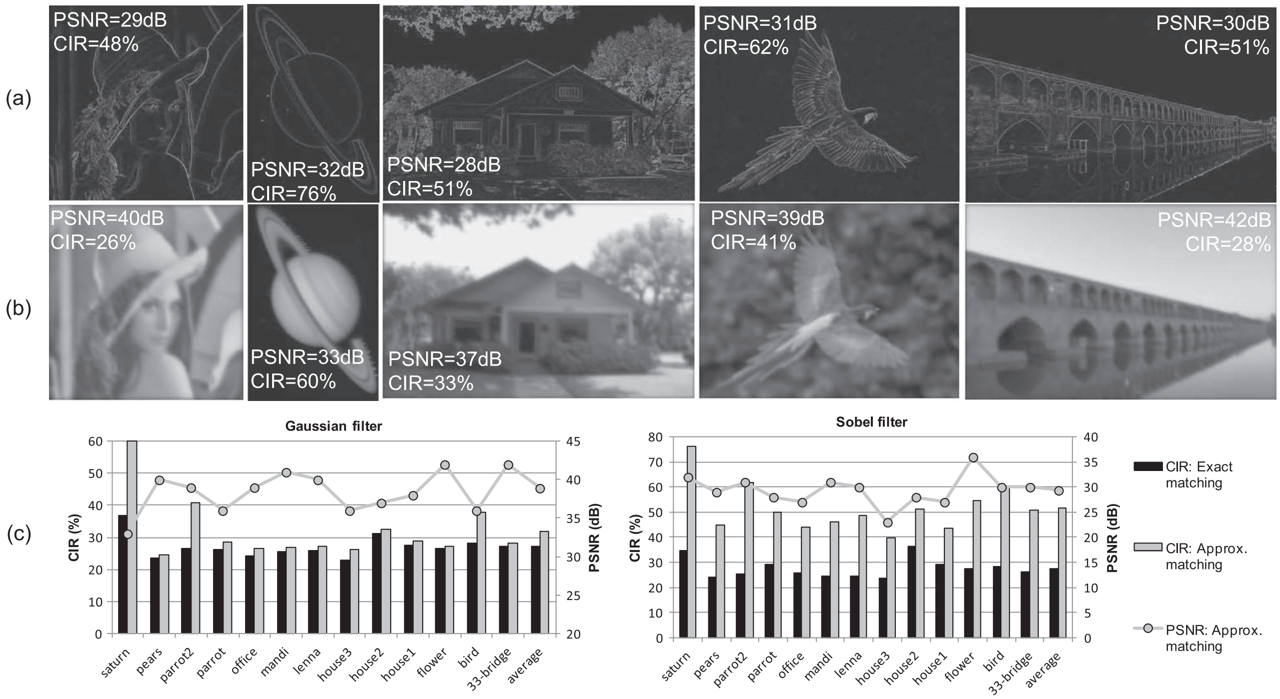


**Fig. 24.** *CIR of the FP with the corresponding PSNR for two kernels. (a) Sobel filter and (b) Gaussian filter using the approximate matching constraint—12 bits masked. (c) CIR and PSNR for Sobel and Gaussian filters with the exact and approximate constraints (the exact matching does not generate any noise because of no bitwise masking).*

combinational logic working on parallel with the first stage of the PE execution; since every PE executes one instruction per cycle, the module is thus shared across all FP functional units of the PE. To check the matching constraint, the module compares bit by bit the two operands of its own PE with the two operands of the PE on the SS lane. All the CIR detector modules share a masking vector to ignore the differences of the operands in the less significant *N* bits of the fraction part. The masking vector is a memory-mapped 32-bit register that is set by various application demands on the computation accuracy. If the two sets of the operations, with consideration of commutativity, meet the value locality constraint, the module sets a reuse-bit which will traverse alongside the corresponding instruction through the stages of the PE. At the last stage of the execution, the PE takes three actions based on the {reuse-bit, error-bit}. In case of no timing error, i.e. {1/0, 0}, the PE sends out its own computed result to the write stage. If a timing error occurred for the instruction during any of the stages, but it has a value locality with the instruction on the SS lane, i.e., {1, 1}, the PE sends out the computed result of the SS lane, and avoids the propagation of the error-bit to the next stage. Finally, in case of the error and lack of the value locality, i.e., {0, 1}, the PE triggers the recovery mechanism.

For five applications form AMD APP SDK v2.5 [174], on an average, the proposed SSMW eliminates the cost of recovery for 62% of the voltage-droop-affected instructions and reduces 12% of the total energy compared with recent resilient work.

*3) Temporal Memoization (Temporal Instruction Reuse):* TIR aims to exploit the value locality and similarity inside each processing element, i.e., FPU in our case. We observe the dispersion of the input operands at the finest granularity for individual FPUs. To expose the value locality for each FPU operations, we consider a private FIFO for every individual FPU. These FIFOs have a small depth and keep the distinct sets of the input operands in the order of instruction arrivals. The FIFO matches a set of incoming input operands and the current content of entries of FIFO using the matching constraint. The FIFO maintains a limited number of recent distinct sets. Therefore, if a set of incoming input operands does not satisfy either matching constraints, the FIFO will be updated by cleaning its last entry and inserting the new incoming operands accordingly.

To exploit the value locality, we tightly couple the FPU pipeline with our proposed temporal memoization module. This module has essentially a single-cycle LUT, and a set of flip-flops and buffers to propagate signals through the pipeline. The LUT is composed of two parts: 1) a FIFO with four entries; 2) a set of combinational comparators. In every entry, the FIFO maintains a set of input operands and the computed result provided by

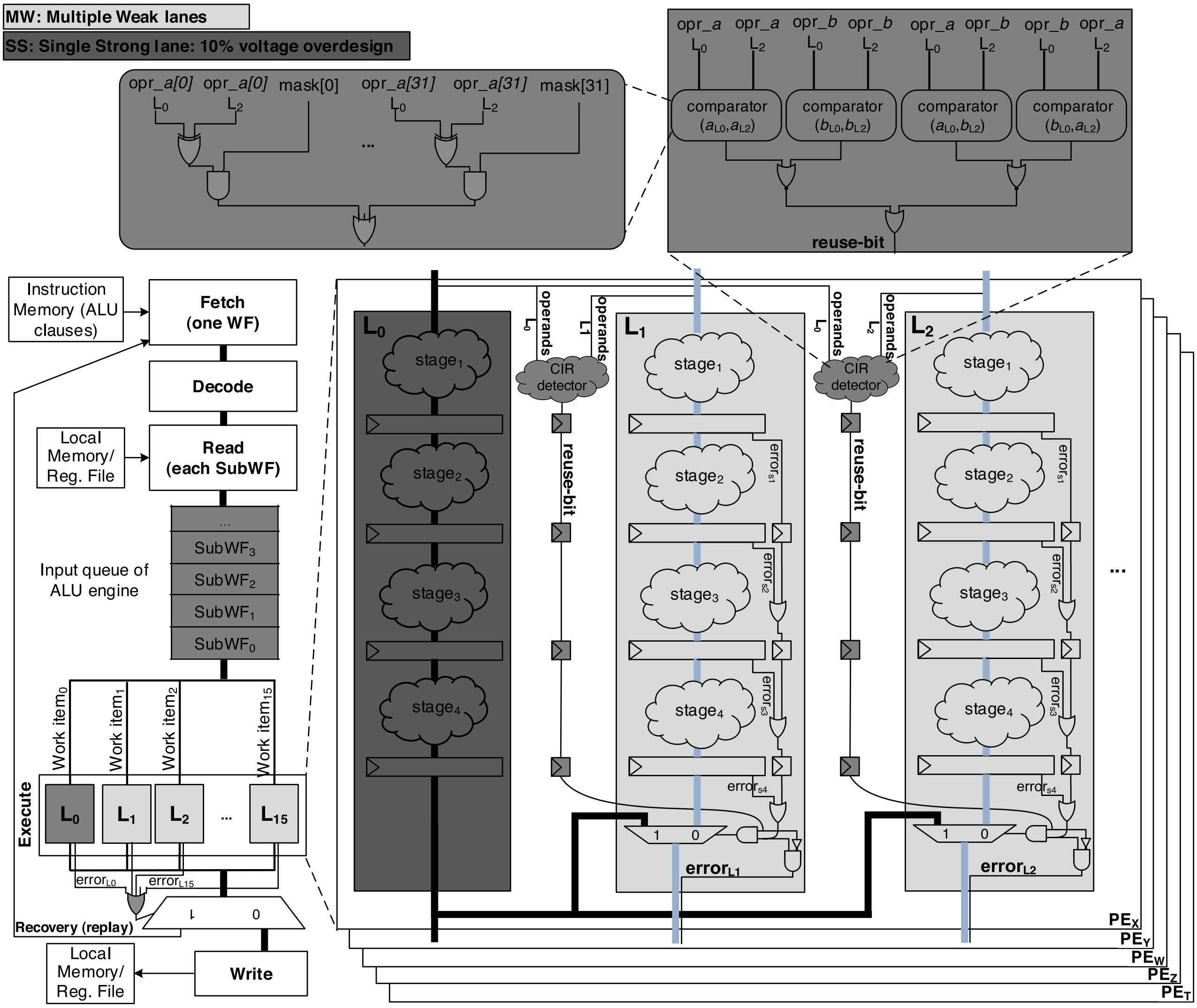


**Fig. 25.** ***Single strong lane and multiple weak lanes (SSMW) architecture.***

the output of the FPU in the last stage ($Q_S$). The parallel combinational comparators implement the two matching constraints, and are programmable through a 32-bit memory-mapped register as a masking vector. They concurrently make either a full or partial comparison of the input operands with the stored operands in each entry based on the masking vector. The LUT works in parallel with the first stage of the FPU. Therefore, for every set of input operands, the LUT searches the FIFO to find a match between the input operands and the operand values stored in the entries (i.e., whether the matching constraint is satisfied or not). A match directly results in reuse of results computed earlier. Consequently, this affords the temporal memoization module an opportunity to correct an errant instruction with zero cycle penalty.

To enable reuse, the LUT propagates a hit signal alongside with the previously-computed result ($Q_L$) toward the end of pipeline. The LUT raises the hit signal that squashes the remaining stages of the FPU to avoid the redundant computation by clock-gating; the clock-gating signal is forwarded to the rest of stages, cycle by cycle. The stored result is also propagated toward the end of pipeline for the reuse purpose. The hit signal selects the propagated output of the LUT ($Q_L$) as the output of the FPU; it also disables the propagation of timing error signal (if any) to the recovery unit, thus avoids the costly recovery. Therefore, each hit event reduces energy by locally retrieving the result from the LUT, rather than doing full reexecution by the FPU. In case of a LUT miss, the FIFO is updated to maintain the last recently computed values. It is implemented through a write enable signal ($W_{en}$) that ensures there is no timing error during execution of all stages of the FPU for computing $Q_S$. Finally, if simultaneous timing error and miss occurred, the error signal will be propagated to the

Table 10 Timing Error Handling With Temporal Instruction Reuse

| Hit | Error | Action | $Q_{Pipe}$ |
|---|---|---|---|
| 0 | 0 | Normal execution + LUT update | $Q_S$ |
| 0 | 1 | Triggering baseline recovery (ECU) | $Q_S$ |
| 1 | 0 | LUT output reuse + FPU clock-gating | $Q_L$ |
| 1 | 1 | LUT output reuse + FPU clock-gating + masking error | $Q_L$ |

recovery unit that triggers the baseline recovery. Table 10 summarizes these four states. For GPGPU applications, TER avoids costly recovery that improves the energy efficiency with an average savings of 8% (for 0% timing error rate) to 28% (for 4% timing error rate). The memoization techniques are explained in detail in [125], [126], and [179].

## VIII. CONCLUSION AND OUTLOOK

Microelectronic variability is a phenomenon at the intersection of microelectronic scaling, semiconductor manufacturing and how electronic systems are designed and deployed. Using timing variability we showed various levels of microelectronic circuit and system design where the effects of variability can be mitigated. These methods have a direct impact on the cost, performance and quality of the microelectronic systems. Methods to combat variability in practice have largely been confined to ever expanding design guardbands for the circuit designer. However, more effective methods can be devised that address variability across design abstraction levels. Such coordinated cross-layer methods are central to the emerging outlook on variability-tolerance as discussed below.

- **Application/Algorithm**. Emerging applications including graphics, multimedia, web search, data analytics, and cyber-physical system go beyond primarily numerical computations for scientific use to interacting with sensor and human interfaces. There exists a great potential to match the "impedance" on the accuracy of the computed result to application needs. In particular, the "acceptance criteria" for results of a computation is subject to quality tradeoffs, much the same way as quality of a signal received over a communications channel. The resulting accuracy requirements may not always need the hardware supported accuracy levels which are designed for worst case computational needs. This presents an opportunity to improve time and energy cost of computation by devising domain-specific resiliency techniques.

  Unfortunately, achieving this level of tradeoff is a much harder problem than knowing quality needs of a specific or specific class of applications. There needs to be engineered guarantees at all levels, certainly from hardware as well, that system and application developers can rely upon. Thus, the biggest technical challenge in this area is systematic methods of capturing/inferring acceptance precision and using this information to develop domain-specific resilience techniques. A careful study of acceptability differences between general purpose CPU and GPGPU architectures is needed to develop architecture-specific solutions. Recent work in this area can be classified into three broad groups as to how the accuracy versus cost tradeoff is made: 1) sampling data points rather than performing all specified computations, such as in BlinkDB; 2) changing task schedules based on computation quality needs; and 3) application-specific relaxation of precision.
- **Software**. Software presents a great unexploited potential for diagnosis and mitigation of variation effects. Software requires *runtime* monitoring and *re-calibration* mechanisms to determine the limits of efficiency. The key point is that at *design time* there is not enough knowledge and there is too much variability and sensitivity to have a viable design time approach. Distributed software techniques and paradigms will therefore become increasingly pervasive even at the chip level. The trend should be toward avoiding global variability bottlenecks, through arranging a mix of redundant execution (avoiding single-point of failure), globally-asynchronous communication and orchestration, and fine-grained rollback. Recent work in conceptualization of systems are physically asynchronous and logically synchronous (PALS) presents an interesting possibility of how distributed computation can be composed with some guarantees as to the quality and timing of results.
- **Architecture**. As mentioned earlier, variability mitigation is about cost and scale. Modular and scalable architectures such as those found in the programmable accelerators enable better observability and controllability of variations through explicit parallelism. Both hardware and software can enhance variability-tolerance by tuning two available axes: *configurations* and *choices*. Hardware and software can jointly "configure" available settings of an architecture and appropriate parameters explicitly coded in applications. They can also selectively "choose" a suitable hardware resource, or an alternative code path. For instance, one alternative can select an optimized approximate kernel rather than exact one results in significant resource reduction enabling integration larger number of parallel kernels on the fixed budget the underlying architecture.

- **Circuit**. Recent efforts have been done in designing robust clocked circuits. By coupling them with the large spectrum of asynchronous options, we can achieve parsimonious robustness. For a given subcircuit (either exact or approximate), a synthesis tool would have the choice of selecting a *communication* scheme among available different communication templates for realizing that subcircuit. In other words, the problem of determining the level of accuracy of a subcircuit will be transformed to "how much" energy we want to spend on ensuring the subcircuit functional "integrity" instead of spending the energy on the actual subcircuit computation.

Overall, variability mitigation presents a broad range of possibilities and techniques that can enable continued benefits from microelectronic scaling and manufacturing methods to the system designers. ■

## REFERENCES

[1] K. Bowman, S. Duvall, and J. Meindl, "Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution," in *Proc. IEEE Int. Solid-State Circuits Conf. Digest Tech. Papers (ISSCC 2001)*, pp. 278–279.

[2] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proc. Design Autom. Conf.*, Jun. 2003, pp. 338–342.

[3] *The ITRS Website*. [Online]. Available: http://www.itrs.net/ITRS%201999-2014%20Mtgs,%20Presentations%20&%20Links/2012ITRS/Home2012.htm.

[4] K. Jeong, A. Kahng, and K. Samadi, "Impact of guardband reduction on design outcomes: A quantitative approach," *IEEE Trans. Semiconductor Manufact.*, vol. 22, no. 4, pp. 552–565, Nov. 2009.

[5] X. Li, J. Qin, and J. Bernstein, "Compact modeling of mosfet wearout mechanisms for circuit-reliability simulation," *IEEE Trans. Device Mater. Rel.*, vol. 8, no. 1, pp. 98–121, Mar. 2008.

[6] K. Bowman, C. Tokunaga, J. Tschanz, A. Raychowdhury, M. Khellah, B. Geuskens, S.-L. Lu, P. Aseron, T. Karnik, and V. De, "Dynamic variation monitor for measuring the impact of voltage droops on microprocessor clock frequency," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, Sep. 2010, pp. 1–4.

[7] S. Murali, A. Mutapcic, D. Atienza, R. Gupta, S. Boyd, L. Benini, and G. De Micheli, "Temperature control of high-performance multi-core platforms using convex optimization," in *Proc. Conf. Des., Autom. Test Eur.*, 2008, pp. 110–115. [Online]. Available: http://doi.acm.org/10.1145/1403375.1403405, ser. DATE '08. New York, NY, USA: ACM.

[8] D. Kamel, C. Hocquet, O.-X. Standaert, D. Flandre, and D. Bol, "Glitch-induced within-die variations of dynamic energy in voltage-scaled nano-cmos circuits," in *Proc. ESSCIRC*, Sep. 2010, pp. 518–521.

[9] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge, "Opportunities and challenges for better than worst-case design," in *Proc. 2005 Asia South Pacific Design Autom. Conf.*, 2005, pp. 2–7. [Online]. Available: http://doi.acm.org/10.1145/1120725.1120878.

[10] L. Wanner, R. Balani, S. Zahedi, C. Apte, P. Gupta, and M. Srivastava, "Variability-aware duty cycle scheduling in long running embedded sensing systems," in *Proc. Design, Autom., Test Eur. Con. Exhib.*, Mar. 2011, pp. 1–6.

[11] S. Ghosh and K. Roy, "Parameter variation tolerance and error resiliency: New design paradigm for the nanoscale era," *Proc. IEEE*, vol. 98, no. 10, pp. 1718–1751, Oct. 2010.

[12] J. Crop, E. Krimer, N. Moezzi-Madani, R. Pawlowski, T. Ruggeri, P. Chiang, and M. Erez, "Error detection and recovery techniques for variation-aware cmos computing: A comprehensive review," *J. Low Power Electron. Appl.*, vol. 1, no. 3, pp. 334–356, 2011. [Online]. Available: http://www.mdpi.com/2079-9268/1/3/334.

[13] S. Dighe, S. Vangal, P. Aseron, S. Kumar, T. Jacob, K. Bowman, J. Howard, J. Tschanz, V. Erraguntla, N. Borkar, V. De, and S. Borkar, "Within-die variation-aware dynamic-voltage-frequency-scaling with optimal core allocation and thread hopping for the 80-core teraflops processor," *IEEE J. Solid-State Circuits*, vol. 46, no. 1, pp. 184–193, Jan. 2011.

[14] D. Jeon, M. Seok, Z. Zhang, D. Blaauw, and D. Sylvester, "Design methodology for voltage-overscaled ultra-low-power systems," *IEEE Trans. Circuits Syst. II: Exp. Briefs*, vol. 59, no. 12, pp. 952–956, Dec. 2012.

[15] B. Zhai, R. Dreslinski, D. Blaauw, T. Mudge, and D. Sylvester, "Energy efficient near-threshold chip multi-processing," in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Design (ISLPED)*, Aug. 2007, pp. 32–37.

[16] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. N. Mudge, "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits," *Proc. IEEE*, vol. 98, no. 2, pp. 253–266, Feb. 2010.

[17] R. Rithe, S. Chou, J. Gu, A. Wang, S. Datla, G. Gammie, D. Buss, and A. Chandrakasan, "The effect of random dopant fluctuations on logic timing at low voltage," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 5, pp. 911–924, May 2012.

[18] M. Kakoee, I. Loi, and L. Benini, "Variation-tolerant architecture for ultra low power shared-l1 processor clusters," *IEEE Trans. Circuits Syst. II: Exp. Briefs*, vol. 59, no. 12, pp. 927–931, Dec. 2012.

[19] 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," in *Proc. IEEE Int. Solid-State Circuits Conf. Digest Tech. Papers (ISSCC)*, Feb. 2012, pp. 492–494.

[20] A. Rahimi, L. Benini, and R. K. Gupta, "Analysis of instruction-level vulnerability to dynamic voltage and temperature variations," in *Proc. Design, Autom., Test Eur. Conf. Exhib.*, Mar. 2012, pp. 1102–1105.

[21] V. Kleeberger, S. Kiesel, U. Schlichtmann, and S. Chakraborty, "Program-aware circuit level timing analysis," in *Proc. 13th Int. Symp. Integr. Circuits (ISIC)*, Dec. 2011, pp. 102–105.

[22] V. B. Kleeberger, P. R. Maier, and U. Schlichtmann, "Workload- and instruction-aware timing analysis: The missing link between technology and system-level resilience," in *Proc. 51st Annu. Design Autom. Conf. Design Autom. Conf.*, 2014, pp. 49:1–49:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2596694.

[23] A. Rahimi, L. Benini, and R. K. Gupta, "Application-adaptive guardbanding to mitigate static and dynamic variability," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2160–2173, Sep. 2013.

[24] A. Rahimi, L. Benini, and R. K. Gupta, "Procedure hopping: A low overhead solution to mitigate variability in shared-l1 processor clusters," in *Proc. 2012 ACM/IEEE Int. Symp. Low Power Electron. Design*, 2012, pp. 415–420. [Online]. Available: http://doi.acm.org/10.1145/2333660.2333754.

[25] F. Paterna, L. Benini, A. Acquaviva, F. Papariello, G. Desoli, and M. Olivieri, "Adaptive idleness distribution for non-uniform aging tolerance in multiprocessor systems-on-chip," in *Proc. Design, Autom., Test Eur. Conf. Exhib.*, Apr. 2009, pp. 906–909.

[26] A. Rahimi, L. Benini, and R. K. Gupta, "Aging-aware compiler-directed vliw assignment for gpgpu architectures," in *Proc. 50th Annu. Design Autom. Conf.*, 2013, pp. 16:1–16:6. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488754.

[27] A. Rahimi, L. Benini, and R. K. Gupta, "Hierarchically focused guardbanding: An adaptive approach to mitigate pvt variations and aging," in *Proc. Design, Autom., Test Eur. Conf. Exhib.*, Mar. 2013, pp. 1695–1700.

[28] S. Roy and K. Chakraborty, "Predicting timing violations through instruction-level path sensitization analysis," in *Proc. 49th Annu. Design Autom. Conf.*, 2012, pp. 1074–1081. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228555.

[29] K. Chakraborty, B. Cozzens, S. Roy, and D. M. Ancajas, "Efficiently tolerating timing violations in pipelined microprocessors," in *Proc. 50th Annu. Design Autom. Conf.*, 2013, pp. 102:1–102:8. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488860.

[30] V. Reddi, M. Gupta, G. Holloway, G.-Y. Wei, M. Smith, and D. Brooks, "Voltage emergency prediction: Using signatures to reduce operating margins," in *Proc. IEEE 15th Int. Symp. High Performance Comput. Arch.*, Feb. 2009, pp. 18–29.

[31] X. Liang, and D. Brooks, "Microarchitecture parameter selection to optimize system performance under process variation," in *Proc. 2006 IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2006, pp. 429–436. [Online]. Available: http://doi.acm.org/10.1145/1233501.1233587.

[32] X. Liang and D. Brooks, "Mitigating the impact of process variations on processor register files and execution units," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarch. (MICRO-39)*, Dec. 2006, pp. 504–514.

[33] 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 Design Integr. Circuits Syst.*, vol. 26, no. 11, pp. 1947–1956, Nov. 2007.

[34] 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.

[35] F. Botman, D. Bol, J.-D. Legat, and K. Roy, "Data-dependent operation speed-up through automatically inserted signal transition detectors for ultralow voltage logic circuits," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 12, pp. 2561–2570, Dec. 2014.

[36] S. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas, "Eval: Utilizing processors with variation-induced timing errors," in *Proc. 41st IEEE/ACM Int. Symp. Microarch. (MICRO-41)*, Nov. 2008, pp. 423–434.

[37] J. Tschanz, J. Kao, S. Narendra, R. Nair, D. Antoniadis, A. Chandrakasan, and V. De, "Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage," in *Proc. IEEE Int. Solid-State Circuits Conf. Digest Tech. Papers (ISSCC 2002)*, Feb. 2002, vol. 1, pp. 422–478.

[38] S. Borkar, T. Karnik, and V. De, "Design and reliability challenges in nanometer technologies," in *Proc. 41st Design Autom. Conf.*, Jul. 2004, pp. 75–75.

[39] I. Miro-Panades, E. Beigne, Y. Thonnart, L. Alacoque, P. Vivet, S. Lesecq, D. Puschini, A. Molnos, F. Thabet, B. Tain, K. Ben Chehida, S. Engels, R. Wilson, and D. Fuin, "A fine-grain variation-aware dynamic vdd-hopping avfs architecture on a 32 nm gals mpsoc," *IEEE J. Solid-State Circuits*, vol. 49, no. 7, pp. 1475–1486, Jul. 2014.

[40] C. Lefurgy, A. Drake, M. Floyd, M. Allen-Ware, B. Brock, J. Tierno, J. Carter, and R. Berry, "Active guardband management in power7+ to save energy and maintain reliability," *IEEE Micro*, vol. 33, no. 4, pp. 35–45, Jul. 2013.

[41] S. Herbert, S. Garg, and D. Marculescu, "Exploiting process variability in voltage/frequency control," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 8, pp. 1392–1404, Aug. 2012.

[42] S. Dighe, S. Gupta, V. De, S. Vangal, N. Borkar, S. Borkar, and K. Roy, "A 45 nm 48-core ia processor with variation-aware scheduling and optimal core mapping," in *Proc. 2011 Symp. VLSI Circuits (VLSIC)*, Jun. 2011, pp. 250–251.

[43] D. Bull, S. Das, K. Shivshankar, G. Dasika, K. Flautner, and D. Blaauw, "A power-efficient 32 b arm isa processor using timing-error detection and correction for transient-error tolerance and adaptation to pvt variation," in *Proc. 2010 IEEE Int. Solid-State Circuits Conf. Digest Tech. Papers (ISSCC)*, Feb. 2010, pp. 284–285.

[44] K. Bowman, J. Tschanz, S. Lu, P. Aseron, M. Khellah, A. Raychowdhury, B. Geuskens, C. Tokunaga, C. Wilkerson, T. Karnik, and V. De, "A 45 nm resilient microprocessor core for dynamic variation tolerance," *IEEE J. Solid-State Circuits*, vol. 46, no. 1, pp. 194–208, Jan. 2011.

[45] X. Bai, C. Visweswariah, P. Strenski, and D. Hathaway, "Uncertainty-aware circuit optimization," in *Proc. 39th Design Autom. Conf.*, 2002, pp. 58–63.

[46] A. Kahng, S. Kang, R. Kumar, and J. Sartori, "Slack redistribution for graceful degradation under voltage overscaling," in *Proc. 2010 15th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2010, pp. 825–831.

[47] D. Bol, C. Hocquet, and F. Regazzoni, "A fast ulv logic synthesis flow in many-$v_t$ cmos processes for minimum energy under timing constraints," *IEEE Trans. Circuits Syst. II: Exp. Briefs*, vol. 59, no. 12, pp. 947–951, Dec. 2012.

[48] L. de Lima Silva, A. Calimera, A. Macii, E. Macii, and M. Poncino, "Power efficient variability compensation through clustered tunable power-gating," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 1, no. 3, pp. 242–253, Sep. 2011.

[49] K.-L. Chang, J. Chang, B.-H. Gwee, and K.-S. Chong, "Synchronous-logic and asynchronous-logic 8051 microcontroller cores for realizing the internet of things: A comparative study on dynamic voltage scaling and variation effects," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 3, no. 1, pp. 23–34, Mar. 2013.

[50] A. Mokhov, D. Sokolov, and A. Yakovlev, "Adapting asynchronous circuits to operating conditions by logic parametrisation," in *Proc. IEEE 18th Int. Symp. Asynchronous Circuits and Syst. (ASYNC)*, May 2012, pp. 17–24.

[51] D. Marculescu and E. Talpes, "Variability and energy awareness: a microarchitecture-level perspective," in *Proc. 42nd Design Autom. Conf.*, Jun. 2005, pp. 11–16.

[52] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a many-core computing accelerator for embedded socs: Performance evaluation of visual analytics applications," in *Proc. 49th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2012, pp. 1137–1142.

[53] S. Ramasubramanian, S. Venkataramani, A. Parandhaman, and A. Raghunathan, "Relax-and-retime: A methodology for energy-efficient recovery based design," in *Proc. 50th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, May 2013, pp. 1–6.

[54] *Eembc Benchmark Consortium*. [Online]. Available: http://www.eembc.org.

[55] L. Lai and P. Gupta, "A Case Study of Logic Delay Fault Behaviors on General-Purpose Embedded Processor Under Voltage Overscaling," Dept. Electr. Eng., Univ. California Los Angeles, Los Angeles, CA, USA, Tech. Rep. 90095, Aug. 2014.

[56] G. Hoang, R. B. Findler, and R. Joseph, "Exploring circuit timing-aware language and compilation," in *Proc. 16th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2011, pp. 345–356. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950405.

[57] P. Singh, E. Karl, D. Blaauw, and D. Sylvester, "Compact degradation sensors for monitoring nbti and oxide degradation," *IEEE Trans. Very Large Scale Integr. (VLSI) Sys.*, vol. 20, no. 9, pp. 1645–1655, Sep. 2012.

[58] K.-H. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. C-33, no. 6, pp. 518–528, Jun. 1984.

[59] A. Al-Yamani, N. Oh, and E. McCluskey, "Performance evaluation of checksum-based abft," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Syst.*, 2001, pp. 461–466.

[60] M. Makhzan, A. Khajeh, A. Eltawil, and F. Kurdahi, "A low power jpeg2000 encoder with iterative and fault tolerant error concealment," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 6, pp. 827–837, Jun. 2009.

[61] M. Hiller, "Executable assertions for detecting data errors in embedded control systems," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2000, pp. 24–33.

[62] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors-a survey," *IEEE Trans. Comput.*, vol. 37, no. 2, pp. 160–174, Feb. 1988.

[63] J. Sloan, R. Kumar, and G. Bronevetsky, "An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance," in *Proc. 43rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2013, pp. 1–12.

[64] R. Gabrys, E. Yaakobi, L. Grupp, S. Swanson, and L. Dolecek, "Tackling intracell variability in tlc flash through tensor product codes," in *Proc. IEEE Int. Symp. Inf. Theory Proc. (ISIT)*, Jul. 2012, pp. 1000–1004.

[65] O. Tahan and M. Shawky, "Using dynamic task level redundancy for openmp fault tolerance," in *Proc. 25th Int. Conf. Arch. Comput. Syst.*, 2012, pp. 25–36. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28293-5-3.

[66] C. Bolchini, A. Miele, and D. Sciuto, "An adaptive approach for online fault management in many-core architectures," in *Proc. Design, Autom., Test Eur. Conf. Exhib.*, Mar. 2012, pp. 1429–1432.

[67] B. Döbel, H. Härtig, and M. Engel, "Operating system support for redundant multithreading," in *Proc. 10th ACM Int. Conf. Embed. Software*, 2012, pp. 83–92. [Online]. Available: http://doi.acm.org/10.1145/2380356.2380375.

[68] Y. Wang, A. Nicolau, R. Cammarota, and A. Veidenbaum, "A fault tolerant self-scheduling scheme for parallel loops on shared memory systems," in *Proc. 19th Int. Conf. High Performance Comput. (HiPC)*, Dec. 2012, pp. 1–10.

[69] L. A. D. Bathen, N. D. Dutt, A. Nicolau, and P. Gupta, "Vamv: Variability-aware memory virtualization," in *Proc. Conf. Design, Autom. Test Eur.*, 2012, pp. 284–287. [Online]. Available: http://dl.acm.org/citation.cfm?id=2492708.2492779.

[70] L. A. D. Bathen, M. Gottscho, N. Dutt, A. Nicolau, and P. Gupta, "Vipzone: Os-level memory variability-driven physical address zoning for energy savings," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign Syst. Synthesis*, 2012, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/2380445.2380457.

[71] X. Liang, D. Brooks, and G.-Y. Wei, "A process-variation-tolerant floating-point unit with voltage interpolation and variable latency," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC 2008) Digest Tech. Papers*, Feb. 2008, pp. 404–623.

[72] A. Tiwari, S. R. Sarangi, and J. Torrellas, "Recycle: Pipeline adaptation to tolerate process variation," in *Proc. 34th Annu. Int. Symp. Comput. Arch.*, 2007, pp. 323–334. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250703.

[73] M. Gottscho, A. BanaiyanMofrad, N. Dutt, A. Nicolau, and P. Gupta, "Power/capacity scaling: Energy savings with simple fault-tolerant caches," in *Proc. 51st Annu. Design Autom. Conf.*, 2014, pp. 100:1–100:6.

[Online]. Available: http://doi.acm.org/10.1145/2593069.2593184.

[74] A. Agarwal, B. Paul, H. Mahmoodi, A. Datta, and K. Roy, "A process-tolerant cache architecture for improved yield in nanoscale technologies," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 13, no. 1, pp. 27–38, Jan. 2005.

[75] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarch. (MICRO-36)*, Dec. 2003, pp. 7–18.

[76] Y. Tamir and M. Tremblay, "High-performance fault-tolerant vlsi systems using micro rollback," *IEEE Trans. Comput.*, vol. 39, no. 4, pp. 548–554, Apr. 1990.

[77] M. Prvulovic, Z. Zhang, and J. Torrellas, "Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *Proc. 29th Annu. Int. Symp. Comput. Arch.*, 2002, pp. 111–122.

[78] E. Krimer, P. Chiang, and M. Erez, "Lane decoupling for improving the timing-error resiliency of wide-simd architectures," in *Proc. 39th Annu. Int. Symp. Comput. Arch.*, 2012, pp. 237–248. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337187.

[79] A. Rajendiran, S. Ananthanarayanan, H. Patel, M. Tripunitara, and S. Garg, "Reliable computing with ultra-reduced instruction set co-processors," in *Proc. 49th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2012, pp. 697–702.

[80] M. Kurimoto, H. Suzuki, R. Akiyama, T. Yamanaka, H. Ohkuma, H. Takata, and H. Shinohara, "Phase-adjustable error detection flip-flops with 2-stage hold driven optimization and slack based grouping scheme for dynamic voltage scaling," in *Proc. 45th ACM/IEEE Design Autom. Conf. (DAC 2008)*, Jun. 2008, pp. 884–889.

[81] M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D. Harris, D. Blaauw, and D. Sylvester, "Bubble razor: An architecture-independent approach to timing-error detection and correction," in *Proc. IEEE Int. Solid-State Circuits Conf. Digest Tech. Papers (ISSCC)*, Feb. 2012, pp. 488–490.

[82] I. Shin, J.-J. Kim, Y.-S. Lin, and Y. Shin, "A pipeline architecture with 1-cycle timing error correction for low voltage operations," in *Proc. IEEE Int. Symp. Low Power Electron. Design (ISLPED)*, Sep. 2013, pp. 199–204.

[83] M. Choudhury, V. Chandra, K. Mohanram, and R. Aitken, "Timber: Time borrowing and error relaying for online timing error resilience," in *Proc. Design, Autom., Test Eur. Conf. Exhib.*, Mar. 2010, pp. 1554–1559.

[84] M. Choudhury, V. Chandra, R. Aitken, and K. Mohanram, "Time-borrowing circuit designs and hardware prototyping for timing error resilience," *IEEE Trans. Comput.*, vol. 63, no. 2, pp. 497–509, Feb. 2014.

[85] K. Chae, S. Mukhopadhyay, C.-H. Lee, and J. Laskar, "A dynamic timing control technique utilizing time borrowing and clock stretching," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, Sep. 2010, pp. 1–4.

[86] K. Chae, C.-H. Lee, and S. Mukhopadhyay, "Timing error prevention using elastic clocking," in *Proc. IEEE Int. Conf. IC Design Technol. (ICICDT)*, May 2011, pp. 1–4.

[87] M. Choudhury and K. Mohanram, "Masking timing errors on speed-paths in logic circuits," in *Proc. Design, Autom., Test Eur. Conference Exhib. (DATE '09)*, Apr. 2009, pp. 87–92.

[88] F. Yuan and Q. Xu, "Intimefix: A low-cost and scalable technique for in-situ timing error masking in logic circuits," in *Proc. 50th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, May 2013, pp. 1–6.

[89] K. Bowman, J. Tschanz, N. S. Kim, J. Lee, C. Wilkerson, S. Lu, T. Karnik, and V. De, "Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance," *IEEE J. Solid-State Circuits*, vol. 44, no. 1, pp. 49–63, Jan. 2009.

[90] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, "Razor: Circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 24, no. 6, pp. 10–20, Nov. 2004.

[91] B. Calhoun and A. Chandrakasan, "Ultra-dynamic voltage scaling (udvs) using sub-threshold operation and local voltage dithering," *IEEE J. Solid-State Circuits*, vol. 41, no. 1, pp. 238–245, Jan. 2006.

[92] S. Miermont, P. Vivet, and M. Renaudin, "A power supply selector for energy- and area-efficient local dynamic voltage scaling," in *Proc. 17th Int. Workshop Integr. Circuit Syst. Design. Power Timing Modeling, Optimiz., Simulation*, 2007, pp. 556–565. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74442-9-54.

[93] L. Lai and P. Gupta, "Accurate and inexpensive performance monitoring for variability-aware systems," in *Proc. 19th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jun. 2014, pp. 467–473.

[94] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *Proc. 2010 ACM SIGPLAN Conf. Programming Lang. Design Implemen.*, 2010, pp. 198–209. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806620.

[95] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Foundations Software Eng.*, 2011, pp. 124–134. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025133.

[96] S. Misailovic, D. Roy, and M. Rinard, "Probabilistically accurate program transformations," in *Static Analysis*, Lecture Notes in Computer Science, E. Yahav, Ed., Berlin, Germany: Springer-Verlag, 2011, vol. 6887, pp. 316–333. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23702-7_24.

[97] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener, "Programming with relaxed synchronization," in *Proc. 2012 ACM Workshop Relaxing Synchronization Multicore Manycore Scalability*, 2012, pp. 41–50. [Online]. Available: http://doi.acm.org/10.1145/2414729.2414737.

[98] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarch.*, 2013, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540711.

[99] D. Mohapatra, V. Chippa, A. Raghunathan, and K. Roy, "Design of voltage-scalable meta-functions for approximate computing," in *Proc. Design, Autom., Test Eur. Conf. Exhib.*, Mar. 2011, pp. 1–6.

[100] H. Cho, L. Leem, and S. Mitra, "Ersa: Error resilient system architecture for probabilistic applications," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 31, no. 4, pp. 546–558, Apr. 2012.

[101] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarch. (MICRO)*, Dec. 2012, pp. 449–460.

[102] H. Mahdiani, S. Fakhraie, and C. Lucas, "Relaxed fault-tolerant hardware implementation of neural networks in the presence of multiple transient errors," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 23, no. 8, pp. 1215–1228, Aug. 2012.

[103] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarch.*, 2013, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540712.

[104] S.-L. Lu, "Speeding up processing with approximation circuits," *Computer*, vol. 37, no. 3, pp. 67–73, Mar. 2004.

[105] K. Palem and A. Lingamneni, "Ten years of building broken chips: The physics and engineering of inexact computing," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, pp. 87:1–87:23, May 2013. [Online]. Available: http://doi.acm.org/10.1145/2465787.2465789.

[106] N. Banerjee, G. Karakonstantis, and K. Roy, "Process variation tolerant low power dct architecture," in *Proc. Design, Autom., Test Eur. Conf. Exhib. (DATE '07)*, Apr. 2007, pp. 1–6.

[107] K. He, A. Gerstlauer, and M. Orshansky, "Circuit-level timing-error acceptance for design of energy-efficient dct/idct-based systems," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 23, no. 6, pp. 961–974, Jun. 2013.

[108] A. Pant, P. Gupta, and M. van der Schaar, "Appadapt: Opportunistic application adaptation in presence of hardware variation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 11, pp. 1986–1996, Nov. 2012.

[109] J. Sloan, R. Kumar, and G. Bronevetsky, "Algorithmic approaches to low overhead fault detection for sparse linear algebra," in *Proc. 42nd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2012, pp. 1–12.

[110] K. Hazelwood and D. Brooks, "Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization," in *Proc. 2004 Int. Symp. Low Power Electron. Design (ISLPED '04)*, Aug. 2014, pp. 326–331.

[111] V. Reddi, S. Kanev, W. Kim, S. Campanoni, M. Smith, G.-Y. Wei, and D. Brooks, "Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarch. (MICRO)*, Dec. 2010, pp. 77–88.

[112] F. Paterna, A. Acquaviva, A. Caprara, F. Papariello, G. Desoli, and L. Benini, "Variability-aware task allocation for energy-efficient quality of service provisioning in embedded streaming multimedia applications," *IEEE Trans. Comput.*, vol. 61, no. 7, pp. 939–953, Jul. 2012.

[113] A. Rahimi, A. Marongiu, P. Burgio, R. K. Gupta, and L. Benini, "Variation-tolerant openmp tasking on

tightly-coupled processor clusters," in *Proc. Design, Autom., Test Eur. Conf. Exhib.*, Mar. 2013, pp. 541–546.

[114] A. Rahimi, D. Cesarini, A. Marongiu, R. K. Gupta, and L. Benini, "Task scheduling strategies to mitigate hardware variability in embedded shared memory clusters," in *Proc. 52nd Annu. Design Autom. Conf.*, 2015, pp. 152:1–152:6. [Online]. Available: http://doi.acm.org/10.1145/2744769.2744915.

[115] A. Rahimi, D. Cesarini, A. Marongiu, R. K. Gupta, and L. Benini, "Improving resilience to timing errors by exposing variability effects to software in tightly-coupled processor clusters," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 4, no. 2, pp. 216–229, Jun. 2014.

[116] P. Aguilera, J. Lee, A. Farmahini-Farahani, K. Morrow, M. Schulte, and N. S. Kim, "Process variation-aware workload partitioning algorithms for gpus supporting spatial-multitasking," in *Proc. Conf. Design, Autom., Test Eur.*, 2014, pp. 176:1–176:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616606.2616823.

[117] J. Dean and L. A. Barroso, "The tail at scale," *ACM Commun.*, vol. 56, no. 2, pp. 74–80, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408794.

[118] A. Rahimi, A. Marongiu, R. K. Gupta, and L. Benini, "A variability-aware openmp environment for efficient execution of accuracy-configurable computation on shared-fpu processor clusters," in *Proc. Int. Conf. Hardware/Software Codesign Syst. Synthesis (CODES+ISSS)*, Sep. 2013, pp. 1–10.

[119] M. Gupta, V. Reddi, G. Holloway, G.-Y. Wei, and D. Brooks, "An event-guided approach to reducing voltage noise in processors," in *Proc. Design, Autom., Test Eur. Conf. Exhib. (DATE '09)*, Apr. 2009, pp. 160–165.

[120] V. Reddi and D. Brooks, "Resilient architectures via collaborative design: Maximizing commodity processor performance in the presence of variations," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 10, pp. 1429–1445, Oct. 2011.

[121] V. Kozhikkottu, S. Dey, and A. Raghunathan, "Recovery-based design for variation-tolerant socs," in *Proc. 49th Annu. Design Autom. Conf.*, 2012, pp. 826–833. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228510.

[122] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *Proc. 17th Int. Conf. Arch. Support Program. Lang. Operating Syst.*, 2012, pp. 301–312. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151008.

[123] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Trans. Comput.*, vol. 54, no. 7, pp. 922–927, Jul. 2005. [Online]. Available: http://dx.doi.org/10.1109/TC.2005.119.

[124] C. Alvarez, J. Corbal, and M. Valero, "Dynamic tolerance region computing for multimedia," *IEEE Trans. Comput.*, vol. 61, no. 5, pp. 650–665, May 2012. [Online]. Available: http://dx.doi.org/10.1109/TC.2011.79.

[125] A. Rahimi, L. Benini, and R. K. Gupta, "Spatial memoization: Concurrent instruction reuse to correct timing errors in simd architectures," *IEEE Trans. Circuits Syst. II: Exp. Briefs*, vol. 60, no. 12, pp. 847–851, Dec. 2013.

[126] A. Rahimi, L. Benini, and R. K. Gupta, "Temporal memoization for energy-efficient timing error recovery in gpgpus," in *Proc. Design, Autom., Test Eur. Conf. Exhib.*, Mar. 2014, pp. 1–6.

[127] A. Rahimi, A. Ghofrani, M. A. Lastras-Montano, K.-T. Cheng, L. Benini, and R. K. Gupta, "Energy-efficient gpgpu architectures via collaborative compilation and memristive memory-based computing," in *Proc. 51st Annu. Design Autom. Conf.*, 2014, pp. 195:1–195:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593132.

[128] A. Rahimi, A. Ghofrani, K.-T. Cheng, L. Benini, and R. K. Gupta, "Approximate associative memristive memory for energy-efficient gpus," in *Proc. 2015 Design Autom. Test Eur. Conf. Exhib.*, 2015, pp. 1497–1502. [Online]. Available: http://dl.acm.org/citation.cfm?id=2757012.2757158.

[129] S. M. Khan, A. R. Alameldeen, C. Wilkerson, J. Kulkarni, and D. A. Jimenez, "Improving multi-core performance using mixed-cell cache architecture," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Arch. (HPCA2013)*, Feb. 2013, pp. 119–130.

[130] F. Frustaci, M. Khayatzadeh, D. Blaauw, D. Sylvester, and M. Alioto, "13.8 a 32 kb sram for error-free and error-tolerant applications with dynamic energy-quality management in 28 nm cmos," in *Proc. IEEE Int. Solid-State Circuits Conf. Digest Tech. Papers (ISSCC)*, Feb. 2014, pp. 244–245.

[131] A. Teman, G. Karakonstantis, R. Giterman, P. Meinerzhagen, and A. Burg, "Energy versus data integrity trade-offs in embedded high-density logic compatible dynamic memories," in *Proc. 2015 Design, Autom., Test Eur. Conf. Exhib.*, 2015, pp. 489–494. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755753.2755864.

[132] A. Kahng, S. Kang, R. Kumar, and J. Sartori, "Recovery-driven design: A power minimization methodology for error-tolerant processor modules," in *Proc. 47th ACM/IEEE Design Autom. Conf. (DAC)*, Jun. 2010, pp. 825–830.

[133] J. Zhang, F. Yuan, R. Ye, and Q. Xu, "Forter: A forward error correction scheme for timing error resilience," in *Proc. Int. Conf. Comput.-Aided Design*, 2013, pp. 55–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=2561828.2561839.

[134] T. Liu and S.-L. Lu, "Performance improvement with circuit-level speculation," in *Proc. 33rd Annu. IEEE/ACM Int. Symp. Microarch. (MICRO-33)*, 2000, pp. 348–355.

[135] A. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *Proc. 49th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2012, pp. 820–825.

[136] P. Whatmough, S. Das, and D. Bull, "Hybrid circuit and algorithmic timing error correction for low-power robust dsp accelerators," in *Proc. IEEE Asian Solid-State Circuits Conf. (A-SSCC)*, Nov. 2013, pp. 29–32.

[137] P. Whatmough, S. Das, and D. Bull, "A low-power 1-ghz razor fir accelerator with time-borrow tracking pipeline and approximate error correction in 65-nm cmos," *IEEE J. Solid-State Circuits*, vol. 49, no. 1, pp. 84–94, Jan. 2014.

[138] M. Choudhury and K. Mohanram, "Approximate logic circuits for low overhead, non-intrusive concurrent error detection," in *Proc. Design, Autom., Test Eur. (DATE '08)*, Mar. 2008, pp. 903–908.

[139] M. Choudhury and K. Mohanram, "Low cost concurrent error masking using approximate logic circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 8, pp. 1163–1176, Aug. 2013.

[140] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *Proc. 32nd ACM SIGPLAN Conf. Programming Lang. Design Implementation*, 2011, pp. 164–174. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993518.

[141] P. Gupta, Y. Agarwal, L. Dolecek, N. Dutt, R. Gupta, R. Kumar, S. Mitra, A. Nicolau, T. Rosing, M. Srivastava, S. Swanson, and D. Sylvester, "Underdesigned and opportunistic computing in presence of hardware variability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 1, pp. 8–23, Jan. 2013.

[142] G. Karakonstantis, A. Chatterjee, and K. Roy, "Containing the nanometer 'pandora-box': Cross-layer design techniques for variation aware low power systems," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 1, no. 1, pp. 19–29, Mar. 2011.

[143] L. Leem, H. Cho, H.-H. Lee, Y. M. Kim, Y. Li, and S. Mitra, "Cross-layer error resilience for robust systems," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, pp. 177–180.

[144] N. Carter, H. Naeimi, and D. Gardner, "Design techniques for cross-layer resilience," in *Proc. Design, Autom., Test Eur. Conf. Exhib.*, Nov. 2010, pp. 1023–1028.

[145] J. Henkel, L. Bauer, J. Becker, O. Bringmann, U. Brinkschulte, S. Chakraborty, M. Engel, R. Ernst, H. Hartig, L. Hedrich, A. Herkersdorf, R. Kapitza, D. Lohmann, P. Marwedel, M. Platzner, W. Rosenstiel, U. Schlichtmann, O. Spinczyk, M. Tahoori, J. Teich, N. When, and H. Wunderlich, "Design and architectures for dependable embedded systems," in *Proc. 9th Int. Conf. Hardware/Software Codesign Syst. Synthesis (CODES+ISSS)*, Oct. 2011, pp. 69–78.

[146] R. K. Iyer, "Hierarchical application aware error detection and recovery," in *Proc. 41st Annu. Design Autom. Conf.*, 2004, pp. 79–79. [Online]. Available: http://doi.acm.org/10.1145/996566.996592.

[147] H. Hoffmann, J. Holt, G. Kurian, E. Lau, M. Maggio, J. E. Miller, S. M. Neuman, M. Sinangil, Y. Sinangil, A. Agarwal, A. P. Chandrakasan, and S. Devadas, "Self-aware computing in the angstrom processor," in *Proc. 49th Annu. Des. Autom. Conf.*, 2012, pp. 259–264. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228409.

[148] A. Drake, R. Senger, H. Deogun, G. Carpenter, S. Ghiasi, T. Nguyen, N. James, M. Floyd, and V. Pokala, "A distributed critical-path timing monitor for a 65 nm high-performance microprocessor," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC 2007) Digest Tech. Papers*, Feb. 2007, pp. 398–399.

[149] J. Tschanz, K. Bowman, S. Walstra, M. Agostinelli, T. Karnik, and V. De, "Tunable replica circuits and adaptive voltage-frequency techniques for dynamic voltage, temperature, and aging variation tolerance," in *Proc. Symp. VLSI Circuits*, Jun. 2009, pp. 112–113.

[150] A. Raychowdhury, B. Geuskens, K. Bowman, J. Tschanz, S. Lu, T. Karnik, M. Khellah, and V. De, "Tunable replica bits for dynamic variation tolerance in 8t sram arrays," *IEEE J. Solid-State Circuits*, vol. 46, no. 4, pp. 797–805, Apr. 2011.

[151] T.-B. Chan, A. Pant, L. Cheng, and P. Gupta, "Design dependent process monitoring for back-end manufacturing cost reduction," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2010, pp. 116–122.

[152] P. Singh, E. Karl, D. Sylvester, and D. Blaauw, "Dynamic nbti management using a 45 nm multi-degradation sensor," *IEEE Trans.Circuits Syst. I: Reg. Papers*, vol. 58, no. 9, pp. 2026–2037, Sep. 2011.

[153] R. McGowen, C. Poirier, C. Bostak, J. Ignowski, M. Millican, W. Parks, and S. Naffziger, "Power and temperature control on a 90-nm itanium family processor," *IEEE J. Solid-State Circuits*, vol. 41, no. 1, pp. 229–237, Jan. 2006.

[154] J. Tschanz, N. S. Kim, S. Dighe, J. Howard, G. Ruhl, S. Vangal, S. Narendra, Y. Hoskote, H. Wilson, C. Lam, M. Shuman, C. Tokunaga, D. Somasekhar, S. Tang, D. Finan, T. Karnik, N. Borkar, N. Kurd, and V. De, "Adaptive frequency and biasing techniques for tolerance to dynamic temperature-voltage variations and aging," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC 2007) Digest Tech. Papers*, Feb. 2007, pp. 292–604.

[155] A. Muhtaroglu, G. Taylor, T. Rahal-Arabi, and K. Callahan, "On-die droop detector for analog sensing of power supply noise," in *Proc. Symp. VLSI Circuits, Digest Techn. Papers*, Jun. 2003, pp. 193–196.

[156] S. Pant and D. Blaauw, "Circuit techniques for suppression and measurement of on-chip inductive supply noise," in *Proc. 34th Eur. Solid-State Circuits Conf. (ESSCIRC 2008)*, Sep. 2008, pp. 134–137.

[157] K. Kang, S. P. Park, K. Kim, and K. Roy, "On-chip variability sensor using phase-locked loop for detecting and correcting parametric timing failures," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 18, no. 2, pp. 270–280, Feb. 2010.

[158] M. Bhushan, A. Gattiker, M. Ketchen, and K. Das, "Ring oscillators for cmos process tuning and variability control," *IEEE Trans. Semiconductor Manufact.*, vol. 19, no. 1, pp. 10–18, Feb. 2006.

[159] L. Vincent, P. Maurine, S. Lesecq, and E. Beigne, "Embedding statistical tests for on-chip dynamic voltage and temperature monitoring," in *Proc. 49th ACM/EDAC/IEEE Design Autom. Conference (DAC)*, Jun. 2012, pp. 994–999.

[160] D. Bol, J. De Vos, C. Hocquet, F. Botman, F. Durvaux, S. Boyd, D. Flandre, and J. Legat, "Sleepwalker: A 25-mhz 0.4-v sub- mm$^2$ 7- $\mu$W/MHz microcontroller in 65-nm lp/gp cmos for low-carbon wireless sensor nodes," *IEEE J. Solid-State Circuits*, vol. 48, no. 1, pp. 20–32, Jan. 2013.

[161] K. Bowman, C. Tokunaga, T. Karnik, V. De, and J. Tschanz, "A 22 nm all-digital dynamically adaptive clock distribution for supply voltage droop tolerance," *IEEE J. Solid-State Circuits*, vol. 48, no. 4, pp. 907–916, Apr. 2013.

[162] D. Blaauw, S. Kalaiselvan, K. Lai, W.-H. Ma, S. Pant, C. Tokunaga, S. Das, and D. Bull, "Razor ii: *in situ* error detection and correction for pvt and ser tolerance," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC 2008) Digest Tech. Papers*, Feb. 2008, pp. 400–622.

[163] S. Kim, I. Kwon, D. Fick, M. Kim, Y.-P. Chen, and D. Sylvester, "Razor-lite: A side-channel error-detection register for timing-margin recovery in 45 nm soi cmos," in *Proc. IEEE Int. Solid-State Circuits Conf. Digest Tech. Papers (ISSCC)*, Feb. 2013, pp. 264–265.

[164] M. Turnquist, E. Laulainen, J. Makipaa, and L. Koskinen, "Measurement of a system-adaptive error-detection sequential circuit with subthreshold scl," in *Proc. NORCHIP*, Nov. 2011, pp. 1–4.

[165] T. Sato and Y. Kunitake, "A simple flip-flop circuit for typical-case designs for dfm," in *Proc. 8th Int. Symp. Quality Electron. Design*, 2007, pp. 539–544. [Online]. Available: http://dx.doi.org/10.1109/ISQED.2007.23.

[166] L. Lai, V. Chandra, R. Aitken, and P. Gupta, "Slackprobe: A low overhead *in situ* on-line timing slack monitoring methodology," in *Proc. Conference on Design, Autom., Test Eur.*, 2013, pp. 282–287. [Online]. Available: http://dl.acm.org/citation.cfm?id=2485288.2485358.

[167] H. Park and C.-H. Yang, "*In situ* sram static stability estimation in 65-nm cmos," *IEEE J. Solid-State Circuits*, vol. 48, no. 10, pp. 2541–2549, Oct. 2013.

[168] J. Tschanz, K. Bowman, S. Walstra, M. Agostinelli, T. Karnik, and V. De, "Tunable replica circuits and adaptive voltage-frequency techniques for dynamic voltage, temperature, and aging variation tolerance," in *Proc. Symp. VLSI Circuits*, pp. 112–113.

[169] C. Lefurgy, A. Drake, M. Floyd, M. Allen-Ware, B. Brock, J. Tierno, J. Carter, and R. Berry, "Active guardband management in power7+ to save energy and maintain reliability," *IEEE Micro*, vol. 33, no. 4, pp. 35–45, Jul. 2013.

[170] F. Chaix, G. Bizot, M. Nicolaidis, and N.-E. Zergainoh, "Variability-aware task mapping strategies for many-cores processor chips," in *Proc. IEEE 17th Int. On-Line Testing Symp. (IOLTS)*, Jul. 2011, pp. 55–60.

[171] S. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, "Varius: A model of process variation and resulting timing errors for microarchitects," *IEEE Trans. Semiconductor Manufact.*, vol. 21, no. 1, pp. 3–13, Feb. 2008.

[172] C. Albea, D. Puschini, P. Vivet, I. Miro-Panades, E. Beigné, and S. Lesecq, "Architecture and robust control of a digital frequency-locked loop for fine-grain dynamic voltage and frequency scaling in globally asynchronous locally synchronous structures," *J. Low Power Electron.*, vol. 7, no. 3, pp. 328–340, Aug. 2011.

[173] J. Lee, P. Ajgaonkar, and N. S. Kim, "Analyzing throughput of gpgpus exploiting within-die core-to-core frequency variation," in *Proc. 2011 IEEE Int. Symp. Perform. Anal. Syst. Software (ISPASS)*, Apr. 2011, pp. 237–246.

[174] *Amd app sdk v2.5*. [Online]. Available: http://www.amd.com/stream.

[175] A. Rahimi, I. Loi, M. Kakoee, and L. Benini, "A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters," in *Proc. Design, Autom., Test Eur. Conf. Exhib.*, Mar. 2011, pp. 1–6.

[176] 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. Design, Autom., Test Eur. Conf. Exhib.*, Mar. 2012, pp. 983–987.

[177] *The GNU Project, Gomp—An Openmp Implementation for GCC*. [Online]. Available: http://gcc.gnu.org/projects/gomp.

[178] A. Rahimi, L. Benini, and R. K. Gupta, "Analysis of Cross-Layer Vulnerability to Variations: An Adaptive Instruction-Level to Task-Level Approach," Dept. Comput. Sci. Eng., Univ. California San Diego, La Jolla, CA, USA, Tech. Rep. CS2014-1004, Feb. 2014, 92093.

[179] A. Rahimi, L. Benini, and R. K. Gupta, "Temporal Memoization for Energy-Efficient Timing Error Recovery in GPGPU Architectures," Dept. Comput. Sci. Eng., Univ. California San Diego, La Jolla, CA, USA, Tech. Rep. CS2014-1006, Jun. 2014, 92093.

## ABOUT THE AUTHORS

**Abbas Rahimi** (Student Member, IEEE) received the B.S. degree in computer engineering from the University of Tehran, Tehran, Iran, in March 2010, and the M.S. and a Ph.D. degrees in computer science and engineering from the University of California, San Diego, La Jolla, CA, USA, in September 2015.

He is currently a Postdoctoral Scholar in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley, CA, USA. He is a Member of the Berkeley Wireless Research Center and collaborating with the Berkeley Redwood Center for Theoretical Neuroscience. His research interests include brain-inspired computing, massively parallel memory-centric architectures, embedded systems and software with an emphasis on improving energy-efficiency and robustness in the presence of variability-induced errors and approximation opportunities. His doctoral dissertation has been selected to receive the 2015 Outstanding Dissertation Award in the area of new directions in embedded system design and embedded software from the European Design and Automation Association.

Dr. Rahimi received the Best Paper Candidate at 50th IEEE/ACM Design Automation Conference.

**Luca Benini** (Fellow, IEEE) received the Ph.D. from Standford University.

He is a Full Professor at the University of Bologna, Bologna, Italy, and he is the Chair of Digital Integrated Circuits and Systems at ETHZ. He has served as Chief Architect for the Platform2012/STHORM project in STmicroelectronics, Grenoble in the period 2009–2013. He has held visiting and consulting researcher positions at EPFL, IMEC, Hewlett-Packard Laboratories, and Stanford University. His research interests include energy-efficient system design and multicore 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 700 papers in peer-reviewed international journals and conferences, four books, and several book chapters.

Dr. Benini is a Member of the Academia Europaea.

**Rajesh K. Gupta** (Fellow, IEEE) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, Kalyanpur, India, in 1984, the M.S. degree in electrical engineering and computer science from the University of California, Berkeley, CA, USA, in 1986, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 1994.

He is a Professor of Computer Science and Engineering at the University of California, San Diego (UCSD), La Jolla, CA, USA, 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 coled 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.