UC San Diego
Technical Reports

Title
Analysis of Cross-layer Vulnerability to Variations: An Adaptive Instruction-level to Task-level Approach

Permalink
https://escholarship.org/uc/item/8m7737dh

Authors
Rahimi, Abbas
Benini, Luca
Gupta, Rajesh

Publication Date
2014-02-04

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Analysis of Cross-layer Vulnerability to Variations: An Adaptive Instruction-level to Task-level Approach

Abbas Rahimi
CSE, UC San Diego
La Jolla, CA 92093, USA
abbas@cs.ucsd.edu

Luca Benini
DEIS, University of Bologna
40136 Bologna, Italy
luca.benini@unibo.it

Rajesh K. Gupta
CSE, UC San Diego
La Jolla, CA 92093, USA
gupta@cs.ucsd.edu

***Abstract*—Variation in performance and power across manufactured parts and their operating conditions is an accepted reality in aggressive CMOS processes. This paper considers challenges and opportunities in identifying this variation and methods to combat or even use these variations for improved computing systems. We introduce the notion of instruction-level vulnerability (ILV) and concurrent instruction reuse (CIR) to expose variation and its effects to the software stack for use in architectural and runtime optimizations. Going further up on the hardware-software stack, we also introduce the notion of task-level vulnerability (TLV) as metadata to characterize dynamic variations. In fact, TLV is a vertical abstraction that reflects manifestation of circuit-level hardware variability in specific software context for parallel execution model.**

Keywords—Variability; timing errors; error recovery; SIMD; processor clusters; OpenMP

I. Introduction

Although scaling of physical dimensions in semiconductor circuits opens the way to billion-transistor dies, it also comes with the side effects of ever-increasing parameter variations [2]. Performance and power uncertainty caused by variability in the manufactured parts is a major design challenge in nanoscale CMOS technologies [1], [20]. Variations arise from different physical sources: 1) static inherent process parameter variations in channel length and threshold voltage variations due to random dopant fluctuations and sub-wavelength lithography; 2) dynamic environmental variations in ambient conditions such as temperature fluctuations and supply voltage droops; and 3) device aging mechanisms 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. Static process variations manifest themselves as die-to-die (D2D) and within-die (WID) variations. D2D variations affect all computing cores on a die equally, whereas WID variations induce different characteristics for each computing core (i.e., differ core-to-core characteristics). Other variations that impact computing cores are dynamic in nature and depend on the environment in which a core is used. Examples of these types of variations include dynamic voltage droop, on-die hot spots, and aging-induced performance degradation. The variations are expected to be worse in future technologies [3].

Such parameter variations in device geometries in conjunction with undesirable fluctuations in operating condition might prevent circuit from meeting timing and power constraints. The most immediate manifestations of variability are in path delay and power variations; for instance $13\times$ variation in the sleep power across ten instances of ARM Cortex M3 core and over a temperature range of 22–60°C has been observed [5]. Path delay variations cause violation of timing specification resulting in circuit-level timing errors that can result in a malfunction within the computing core. Hence, IC designers commonly use conservative guardbands for the operating frequency or voltage to ensure error-free operation for the worst-case variations. These guardbands have been steadily increasing, leading to a loss of operational efficiency and increased costs due to overdesign [4]. An alternative to overdesign is to make a design resilient to errors and variations. Various resilient techniques have been proposed to mitigate the variation-induced timing errors, including adaptive management of guardbanding through 'predict-and-prevent-error' mechanisms [47], [41], [38], [39], [8], [40], [9], [10], 'detect-then-correct-error' mechanisms [23], [24], [25], [26], [28], [33], and 'ignore-error' mechanisms [6], [7], [45], [11]. A brief review of the main concepts and their embodiments follows.

Predict-and-prevent mechanisms try to avoid timing errors while reducing guardbands. Mintarno et al. propose a framework with control policies to optimize dynamic control of self-tuning parameters during lifetime of a digital system that saves energy relative to traditional one-time worst-case guardbands [8]. A notion of hierarchically focused guardbanding is proposed to adaptively mitigate process, voltage, temperature variations and aging [38]. This is achieved by online utilization of a predictive model that enables a *focused* adaptive guardbanding in view of sensors, observation granularity, and reaction times. The instruction program counter of an out-of-order pipeline is used for an early prediction of an upcoming timing violation by searching in a large predictor table [41]. Rahimi et al. [39] propose a compiler technique that periodically regenerates healthy codes that reduces the aging-induced performance degradation of general-purpose graphic processing units (GPGPUs). For memories, various allocation technique reduces the effect of aging by distributing the idleness across the memory space, for instance sub-banks of a scratchpad memory [40], register file of an embedded core [9], and large register files [10] of the GPGPUs. The predictive techniques cannot eliminate the entire guardbanding to work efficiently at the edge of failure specially so with frequent timing errors in the voltage overscaling and near-threshold regimes.

Detect-then-correct mechanisms typically employ *in situ* or replica circuit sensors to detect the timing error in both logic and memory. These mechanisms focused on measures to mitigate variability through innovations in circuit-level designs.

For logic, *in situ* error-detection sequential (EDS) [23] and Razor [25] circuit sensors have been employed to detect timing errors, whereas an 8T SRAM memory array utilized tunable replica bits [24]. A common strategy is to detect variability-induced delays by sampling and comparing signals near the clock edge to detect timing errors. Alternatively, less intrusive on-chip monitors measure the timing margin available to a block, for instance IBM 8-core POWER7 employs five low-overhead critical path monitors (CPMs) per each core to capture PVT variations. In a similar vein, Intel resilient 45nm core places a tunable replica circuit (TRC) per pipeline stage to monitor worst-case delays. To ensure recovery, the timing errors are corrected by replaying the errant operation with a larger guardband through various adaptation techniques. For instance, Intel integer-scalar core [26] places EDS circuits in the critical paths of the pipeline stages. Once a timing error is detected during instruction execution, the core 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 scalable error recovery, the ECU supports two separate techniques: instruction replay at half frequency, and multiple-issue instruction replay at the same frequency. These techniques impose energy overhead and latency penalty of up to 28 extra recovery cycles per error for the resilient 7-stage integer pipeline [26]. Recently, OpenMP

Ignore-error methods ensure safety of error ignorance through a set of rules for *disciplined approximate* [45] programs. These methods do not strive to achieve instruction executions *exactly* as specified by the application programs. Disciplined approximate programs can exhibit enhanced error resilience at the application-level when multiple valid output values are permitted. Conceptually, such programs have a vector of 'elastic outputs', and if execution is not 100% numerically correct, the program can still appear to execute correctly from the users perspective. Programs with elastic outputs have application-dependent fidelity metrics such as peak signal to noise ratio (PSNR) associated with them to mathematically characterize the quality of the computational result [7]. The degradation of output quality for such applications (e.g., digital signal processing [11], multimedia and compression [43]) is acceptable if the fidelity metrics satisfy a certain threshold. An error resilient system architecture (ERSA) [6] presents a robust system that utilizes software optimizations and error-resilient algorithms of probabilistic applications based on their classification as recognition, mining and synthesis (RMS) applications [12]. Rahimi et al. [44] provide OpenMP extensions (as custom directives) for floating-point computations to specify parts of a program that can be executed approximately.

In this paper, we focus on 'detect-then-correct' mechanism as well as 'error-ignorance' method to show how a vertical abstraction of circuit-level variations into higher levels can enhances the the scope of these approaches, especially in parallel computation contexts. The rest of the paper is organized as follows. Section II surveys prior work in the circuit-level techniques and their limitations. Then, we describe how an instruction-level *memoization* technique can response to these deficiencies. It provides an important ability to reuse computation and error ignorance for reducing the cost of recovery from timing errors in a GPGPU context. In the next sections, we describe several efforts that have tried to characterize and use variability related information at higher levels. Section III and Section IV introduce the notions of instruction-level vulnerability (ILV) and task-level vulnerability (TLV) to expose hardware variations and its effects to the software stack. In fact, TLV is a vertical abstraction that reflects manifestation of circuit-level hardware variability in specific software context for efficient OpenMP parallel execution. Section V concludes this paper.

II. Concurrent Instruction Reuse for Timing Error Recovery

The aforementioned circuit-level 'detect-then-correct' mechanisms impose energy overhead and latency penalty for correcting the errant instruction. As energy becomes the dominant design metric, aggressive voltage scaling [22] and near-threshold operations [21] increase the rate of timing errors and correspondingly the costs (in energy, performance) of these recovery mechanisms. This cost is exacerbated in floating-point (FP) single-instruction multiple-data (SIMD) pipelined architectures where the pipeline dimensions are expanded both vertically (with wider parallel lanes) and horizontally (with deeper stages). The horizontally expanded deeper pipelines induce higher pipeline latency and higher cost of recovery through flushing and replaying the errant instruction. The FP pipelines consume higher energy-per-instruction than their integer counterparts and typically have high latency for instance over 100 cycles [27] to execute on a GPGPU. Effectively, these energy-hungry high-latency pipelines are prone to inefficiencies under timing errors. Similarly, in vertically expanded pipelines, there is a significant performance drop in a 10-lane SIMD architecture as single-stage-error probabilities increase [28]. In the lock-step execution, any error within any of the lanes will cause a global stall and force recovery of the entire SIMD pipeline.

Thus, in FP SIMD pipelines the error rate is multiplied by the wider width while the number of recovery cycles per error increases at least linearly with the pipeline length. This makes the cost of recovery per single error quadratically more expensive relative to scalar functional units. At the same time, parallel execution in the GPGPU architectures – described in Section II-A– provides an important ability to reuse computation and reduce the cost of recovery from timing errors. Accordingly, Rahimi et al. [13] exploit this opportunity to make three main contributions. **First**, we propose a novel *spatial memoization* technique to correct variation-induced timing errors on the SIMD architectures for efficient recovery. We observe that the entropy of data-level parallelism is low due to high spatial locality of values. The spatial memoization leverages this inherent value locality of applications by *memoizing* the result of an error-free execution on an instance of data. Then, it reuses this memoized result to exactly (or approximately) correct any errant execution on other instances of the same (or adjacent) data. Section II.B describes this technique in detail. **Second**, we propose a SIMD architecture consisting of a single strong lane and multiple weak lanes (SSMW) to support memoization at the level of instruction. The SS lane memoizes the output of an error-free FP instruction; therefore, if any MW lane faces an error, it reuses the output of SS lane instead of triggering recovery. Section II.C details the design of the SSMW architecture. **Third**, we demonstrate the effectiveness of our technique on

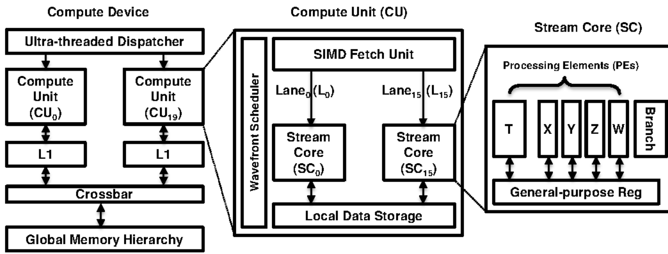

Fig. 1. Block diagram of the Radeon HD 5870 GPGPU.

the GPGPU architecture for error-tolerant image processing kernels and error-intolerant general-purpose kernels. A high rate of instruction reuse is observed, which avoids recovery on average for 62% of the errant instructions, thus significantly reduces the total cost of recovery.

A. Evergreen GPGPU Architecture

We focus on the 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 memory hierarchy. Each compute unit contains a set of 16 Stream Cores (SCs), i.e., 16 parallel lanes. Within a compute unit, a shared instruction fetch unit provides the same machine instruction for all SCs to execute in a SIMD fashion. Each SC contains five Processing Elements (PEs) – labeled *X, Y, Z, W,* and *T* – froming an ALU engine to execute Evergreen machine instructions in a vector-like fashion. Every SC is a five-way processor capable of issuing up to five FP scalar operations from a single very long instruction word (VLIW) consisting of five slots. Each slot is related to its corresponding PE. Four PEs (PE_X, PE_Y , PE_Z, PE_W) can perform up to four single-precision typical operations separately, while PE_T has a special function unit capable of handling transcendental operations. The block diagram of the architecture is shown in Fig. 1.

Instruction-level parallelism is exploited by packing data-independent instructions into a VLIW bundle. Within an SC, every VLIW slot supplies a parallel instruction (if available) to be assigned to the related PE for simultaneous execution. At the same time, the data-level parallelism is also exploited by the SIMD execution model that causes the same machine instruction to be executed concurrently by all 16 lanes in the lock-step fashion. This exposed data-level parallelism naturally facilitates the observation of availability of value locally across the lanes of a CU.

B. Spatial Memoization and CIR

Sodani and Sohi [34] introduced the concept of instruction reuse that comes from the observation that many instructions can be skipped if another instance has already been executed using the same input values. The instruction reuse memorizes the outcome of an instruction on hardware tables; therefore, a processor can reuse it temporally if the processor performs the same instruction with the same input values. Although this technique shows a high fraction of instruction reuse, particularly on the multimedia domain, the temporal memoization is fundamentally limited by: 1) the latency and energy overhead of the reuse tables; and 2) the low hit rate of the tables. To improve the hit rates, recent reuse techniques [36], [37] 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 multimedia algorithms to achieve high reuse rates, and work at the granularity of the floating-point instruction [37], or a region of floating-point instructions [36]. Their method is based on relaxing the conditions upon skipping an instruction or regions of instructions by caching results of previous equal and also similar inputs that relies in the tolerance in the output precision.

In response to these deficiencies, we propose a spatial memoization technique that does not require any table for saving and searching. This technique seeks whether a single instruction can be reused spatially, as opposed to temporally, across various parallel lanes of the SIMD architecture. Our analysis shows that the SIMD architecture explicitly exposes the value that is locally exhibited inside a parallelized program to all parallel lanes, thus facilitating the concurrent instruction reuse (CIR) in close proximity. We have examined error-tolerant image processing applications and error-intolerant general-purpose applications selected from AMD Accelerated Parallel Processing (APP) SDK v2.5 [42]; both application groups display significant value locality across the parallel lanes mainly because there is enough redundant contextual information (i.e., low entropy).

To measure the exposed spatial value locality over the parallel lanes, we have defined CIR as a metric for the entire kernel execution. CIR is defined as the number of simultaneous instructions executed on the $lane_1$ (L_1) through L_{15} of the CUs that satisfy a value locality constraint, which is divided by the total number of instructions executed in all 16 lanes (L_0–L_{15}). The value locality 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) value 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 theoretical CIR rate is 93.75% (15 out of 16).

In the following, we consider the single-precision FP instructions. For error-tolerant image processing applications, we have examined two filters: Sobel and Gaussian. These error-tolerant applications exhibit enhanced error resilience at the application-level when multiple valid output values are permitted, in effect, creating a relation from input values to (multiple) output values. Three value locality constraints are considered: α, β, γ. α is the tight constraint without masking, which enforces full bit-by-bit matching of the input operands of the instructions. β and γ relax the criteria of α during the comparison of the operands by masking the less significant 11, and 12 bits of the fraction parts, respectively. The tight value locality constraint α requires the full precision matching between the input operands of the pair of instructions, thus guaranteeing accurate error correction. On the other hand, β and γ need similar input operands, which yield approximate er-

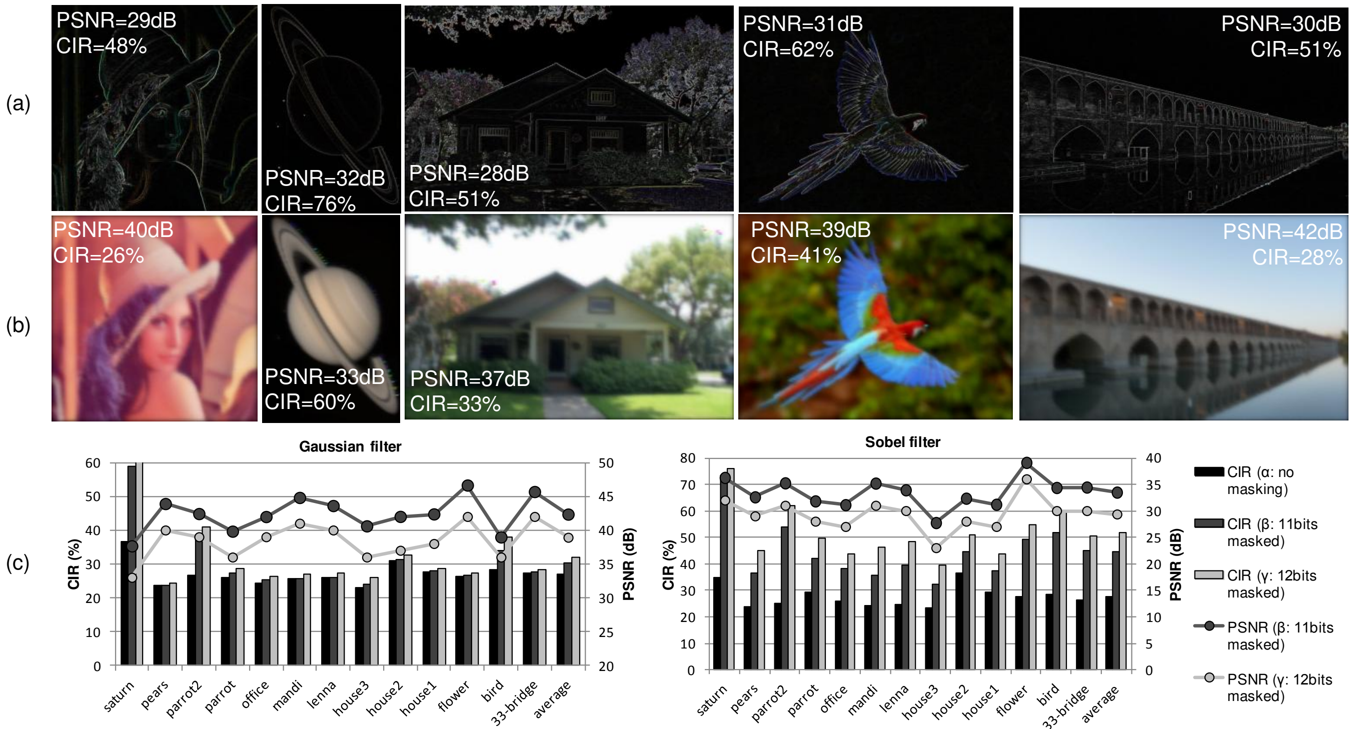

Fig. 2. CIR of the FP with the corresponding PSNR for two kernels. (a) Sobel filter applying the value locality constraint of γ. (b) Gaussian filter value locality constraint of γ. (c) Sobel and Gaussian filters with the three value locality constraints (α has no bitwise masking; thus, it does not generate any noise).

ror correction. With this approximation, the pair of instructions with two different input operands will have the same output and application can ignore error. As a result, the quality of the output is degraded but is acceptable in multimedia applications within the constraints of application-specific PSNR. For the filter kernels, Fig. 2 shows the CIR rate and the corresponding PSNR for various input pictures while using different value locality constraints. As shown in Fig. 2(c), applying the value locality constraint of α 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 L_0 for accurate error correction, without any quality degradation. By relaxing the value locality criteria from α toward γ, higher multiple data-parallel values fuse into a single value, resulting in a higher CIR rate for approximate error correction, for example up to 76% for Sobel. On average, by applying γ, a CIR rate of 51% (32%) is achieved on Sobel (Gaussian) with the acceptable PSNR of 29 dB (39 dB).

1) Concurrent Instruction Reuse for Error-Intolerant Kernels: To generalize the CIR concept, we have extended our analysis to the error-intolerant applications. In case of error-intolerant applications that do not have such inherent algorithmic tolerance, even a single bit error could result in unacceptable program execution. In this class, we have examined three applications: binomial option pricing, Haar wavelet transform, and eigenvalues of a symmetric matrix. To evaluate the scalability of CIR, the size of the input data of these applications are also enlarged. Option pricing is an important problem in financial engineering. Binomial option pricing is implemented for European-style options, and its input data are the number of samples to be calculated. Haar wavelet computes wavelet analysis on a 1-D input signal. The input data for Eigenvalues algorithm is a symmetric tridiagonal matrix.

These applications require 100% numerical correctness; thus, only the tight value locality constraint of α can be used. It enables the instructions of L_i to reuse the output of the instruction of L_0 while maintaining the full precision. The bars in Fig. 3 show the FP instruction count of these applications as a function of the input size, and the CIR of each instruction type is also shown. The FP instructions of binomial option pricing display high CIR rates: 60% for addition, 32% for multiplication, 26% for multiplication and addition, and 61% for the rest of FP instructions. By increasing the number of sampling input from 5000 to 9000, the number of executed FP instructions is almost doubled, whereas the rate of CIR is constant, confirming its scalability across various input sizes. For eigenvalues with an input matrix size of 100 $\times$ 100, a CIR rate of 91% for the total FP instructions is observed. Expanding the size of the input matrix by a factor of $\sim$6700$\times$ increases the FP instructions count by a factor of $\sim$4200$\times$ and further increases the CIR to 94%. The Haar wavelet transform also reveal a high CIR of 36% for the total FP instructions across various sizes of the input signal.

These high rates of CIR, across various application-specific requirements on the computational accuracy, confirms that the data-level parallelism exposed on the SIMD lanes is a promising observation point to exploit the inherent value locality inside the parallelized programs.

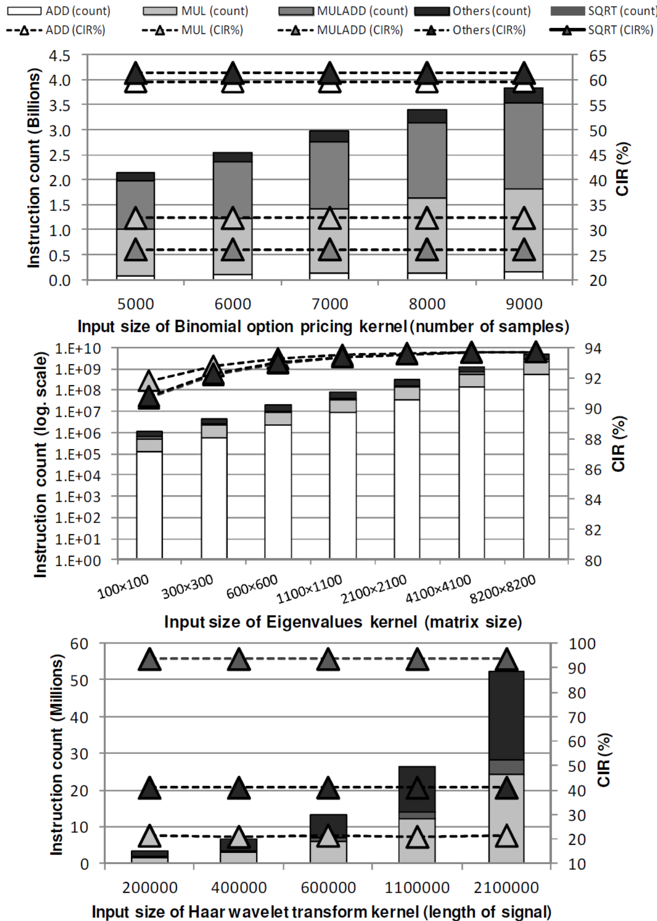

Fig. 3. FP instruction count and their CIR for three error-intolerant kernels (constraint of α thus no bitwise masking) with various input sizes.

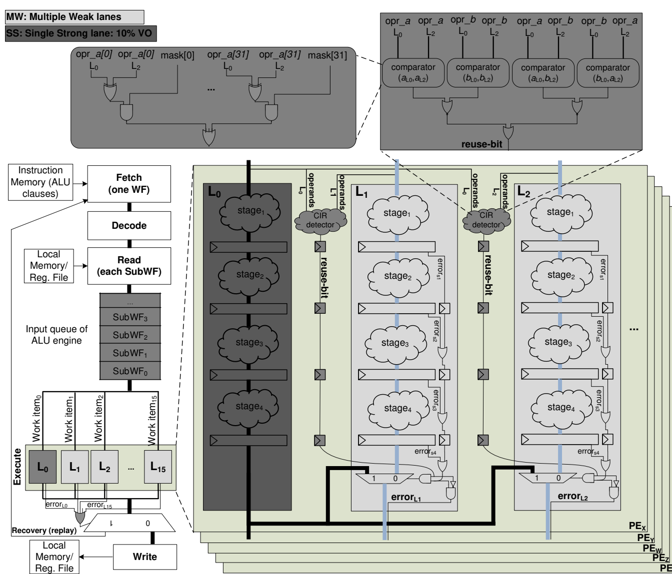

Fig. 4. FP SSMW execution unit.

C. Single-Strong-Lane-Multiple-Weak-Lane (SSMW) Architecture

As aforementioned, the cost of recovery per single timing error on a FP SIMD architecture is very expensive. Pawlowski et al. [33], [28] propose to decouple the SIMD lanes through private queues that prevent error events in any single lane from stalling all other lanes, thus enabling each lane to recover errors independently. The decoupling queues cause slip between lanes, which requires additional architectural mechanisms to ensure correct execution. Therefore, the lanes are required to resynchronize when a microbarrier (e.g., load, store) is reached, therefore incurring performance penalty [28]. This penalty limits their utility to low error rate circumstances.

In response to this deficiency, we exploit the inherent value locality; therefore, the SIMD is designed to maintain the lock-step integrity in the face of timing error, i.e., an SSMW architecture, which is a resilient SIMD architecture. The key idea, for satisfying both resiliency and lock-step execution goals, is to always guarantee error-free execution of a lane (SS). Then, the rest of the lanes (MW) can reuse its output 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. 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 major 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 circuits to detect any timing error and propagate an error bit toward the pipeline stages. Second, SSMW also employs a CIR detector module for every PE of the MW lanes, as shown in Fig. 4. This module checks the value locality 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. The output result of the SS lane is broadcast via a network across MW lanes.

The CIR detector module is a programmable 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 value locality constraint at the level of instruction, 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, considering 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 the 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 the case of an error

and lack of the value locality, i.e., {0, 1}, the PE triggers the recovery mechanism.

D. Effectiveness of SSMW Architecture

Our methodology is developed upon the AMD Evergreen GPGPUs but can be applied to other SIMD architectures as well. Multi2Sim [46], which is a cycle-accurate CPU–GPU simulation framework, is modified to collect the statistics for computing CIR. The Naive binaries of AMD APP SDK v2.5 [42] kernels are run on the simulator, and the input values for the kernels are generated by the default OpenCL host program. We analyzed the effectiveness of the SSMW architecture in the presence of timing errors on the TSMC 45-nm application-specific IC flow. The fetch and decode stages display low criticality [47]. To keep the focus on the processor architecture, we assume that the memory components are resilient, for example by utilizing the tunable replica bits [24].We have partially implemented the FP execution stage of the PE, consisting of three frequently exercised functional units: ADD, MUL, and SQRT. On Evergreen GPGPUs, every functional ALU has latency of four cycles and throughput of one instruction per cycle [49]. Therefore, the VHDL code of the three FP functional units are generated and optimized using FloPoCo [48] – an arithmetic FP core generator. To achieve balanced pipelines with latency of four cycles, the SQRT utilizes a fifth-degree polynomial approximation to decrease its delay.

The front-end flow with multiple V_{TH} cells has been performed using *Synopsys Design Compiler* with the topographical features, whereas *Synopsys IC Compiler* has been used for the back-end flow. The design has been optimized for timing, for the signoff frequency of 1 GHz at (SS/0.81V/125°C), and for power using high V_{TH} cells. Next, the voltage–temperature scaling feature of *Synopsys PrimeTime* is employed to analyze the delay variations under voltage droop. Finally, the variation-induced delay is back annotated to the post-layout simulation, which is coupled with Multi2Sim. To quantify the timing error, we consider two global voltage droop scenarios, i.e., 3% and 6%, across all 16 lanes during the entire execution of the kernels.

We consider five architectures for comparison: i) the lane decoupling queues architecture [28] without VO; ii-iii) the SIMD baseline architecture with 10% (or 6%) VO across all 16 lanes; iv-v) the SSMW architecture in which the SS lane, the CIR detector modules, and the broadcast network are guardbanded by 10% (or 6%) VO to guarantee error-free operations. Once SSMW cannot exploit CIR for an error event recovery, it relies on the single-cycle recovery mechanism presented in [33], [28].

Fig. 5 shows the energy efficiency of the FP execution stage during Gaussian filter execution for a wide range of error rates. At a low error rate, SSMW (10% VO) achieves up to 18% higher GFLOPS per watt compared with the baseline (10% VO). The energy efficiency gain of the decoupling queues disappears at an error rate of 12% and higher, whereas SSMW surpasses both architectures up to an error rate of 60%; SSMW achieves up to 16% higher GFLOPS per watt compared with the decoupling queues. Increasing the error rate beyond 60% removes the energy efficiency gain of SSMW.

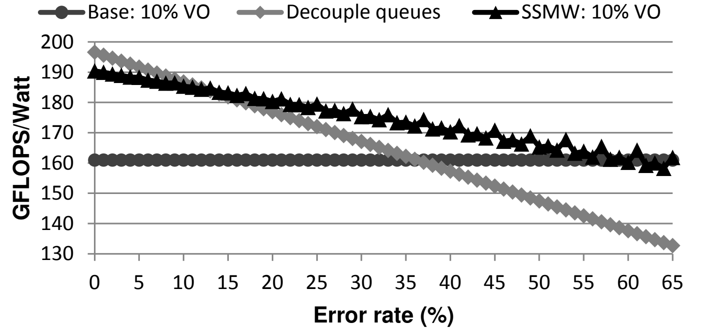

Fig. 5. Gaussian filter energy efficiency comparison for three architectures.

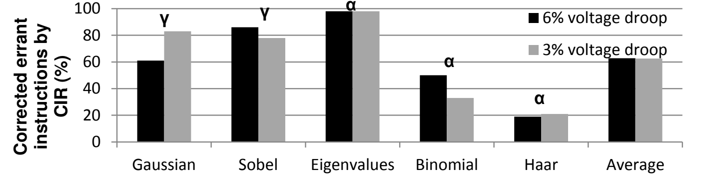

Fig. 6. Effectiveness of CIR for kernels during 3% and 6% voltage droops.

The CIR of Gaussian cannot afford to efficiently correct all errant instructions at this high error rate; thus, SSMW incurs the recovery cycles frequently.

Fig. 6 shows the effectiveness of SSMW, i.e., the percentage of the corrected errant instructions by CIR for all kernels when encountering 6% and 3% voltage droops during the execution. The applications set α for the accurate error correction and γ for the approximate error correction. On average, for all kernels, SSMW avoids the recovery for 62% of the errant instructions, confirming the effective utilization of the value locality.

Fig. 7 shows the total energy comparison of the kernels while experiencing 6% voltage droops.On average, SSMW(10% VO) reduces 8% of the total energy compared with its baseline counterpart. The CIR detector modules increase the delay of the baseline architectures up to 4.9% due to the SS-lane broadcast network and impose a maximum of 5.7% total power overhead. In comparison with decoupling queues, SSMW (10% VO) has on average 12% lower energy consumption. The SSMW (6% VO) has also 1% lower energy compared with the baseline with 6% VO, optimistically assuming that the baseline does not incur any timing error while operating at the edge of failure with 6% voltage droops.

In summary, the proposed SSMW architecture enables a spatial memoization technique that seeks to reduce error recovery costs by reuse of concurrent instructions while maintaining a lock-step execution of the SIMD architecture. The proposed memoization technique exploits the value locality in data-

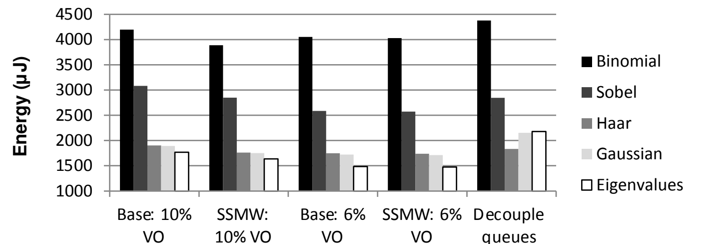

Fig. 7. Energy consumption of kernels during 6% voltage droops.

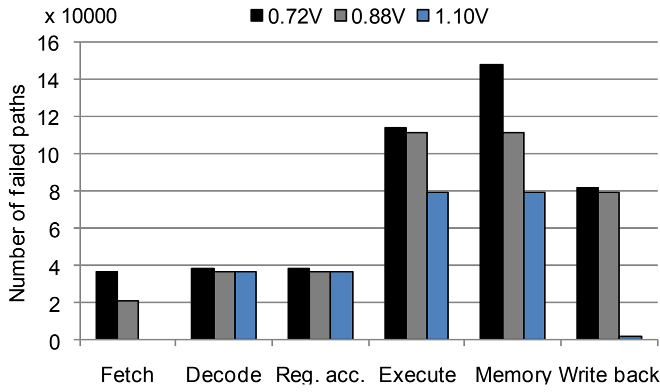

Fig. 8. Effect of voltage variation on the number of failed paths among the pipeline stages at a constant temperature of 125°C.

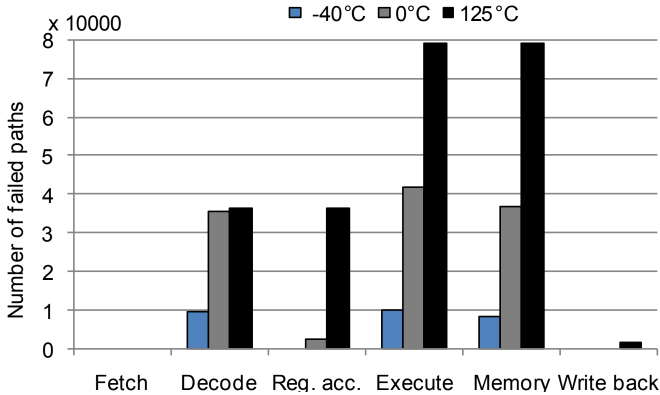

Fig. 9. Effect of temperature variation on the number of failed paths among the pipeline stages at a constant voltage of 1.10V.

parallel applications that is explicitly exposed to the parallel lanes. Error-tolerant and error-intolerant applications exhibit up to 76% and 94% CIR rate for the approximate and accurate error corrections, respectively. 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 work. Further, the spatial memoization can be utilized to spontaneously apply clock gating for MW lanes.

III. Instruction-level Vulnerability

In this section, we describe the notion of instruction-level vulnerability (ILV) [47] to expose variation and its effects to the software stack. To compute ILV, we quantify the effect of a full range of operating conditions on the performance of a 32-bit, RISC, integer-scalar LEON-3 [35] processor compliant with the SPARC V8 architecture. Specifically, we used a temperature range of -40°C–125°C, and a voltage range of 0.72V–1.1V. These operating condition (hence dynamic) variations cause the critical path delay to increase by a factor of 6.1× when the operating condition is varied from the one corner to the other.

We evaluate the critical paths of each pipeline stage for a given cycle time, while changing the operating conditions. Fig. 8 shows the number of failed paths with a negative slack for each parallel pipeline stages across three corners. The cycle time is set at 0.85ns (17FO4), and voltage varies from 0.72V to 0.88V, and then to 1.10V at a constant temperature of 125°C. As shown in Fig 8, most of the failed paths lie in the execute and memory stages in all three operating voltages. On the other hand, each of the fetch, decode, and register access stages contains less than 40K failed paths. Furthermore, there is a relatively small fluctuation in their number of critical paths across voltage variations for these stages. Quantitatively, the memory stage at operating voltage of 0.72V has 1.3×, 1.8×, 3.8× more critical paths in comparison to the execute, write back, and decode stages, respectively. Memory stage at operating voltage of 1.10V also faces 1.4×, 1.9× more critical paths when the voltage drops to 0.88V, 0.72V, respectively.

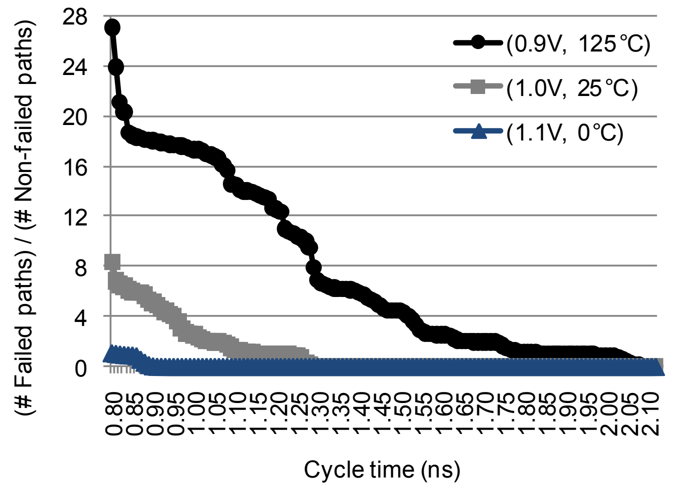

Fig. 10. The proportion of failed paths to non-failed paths versus clock scaling and across three base corners.

To analyze the effect of temperature fluctuation, variation in the number of the failed paths for each parallel pipeline stage is shown in Fig. 9. The cycle time is set at 0.85ns (17FO4), and temperature is varying from -40°C to 0°C, and then to 125°C at a constant voltage of 1.1V. As shown in Fig. 9, there is no failed path in the fetch stage when the temperature is varied, and only a small number of failed paths are found in the write back stage at the highest temperature. But the downside is that many paths fail within the execute and memory stages, like Fig. 8 Consequently, the execute and memory parts of the processor are not only very sensitive to voltage and temperature variations, but also exhibit a large number of critical paths in comparison to the rest of processor. Similarly, we would anticipate that the instructions that significantly exercise the execute and memory stages are likely to be more vulnerable to voltage and temperature variations.

Let us now examine the situation of all paths through the processor under different operating condition and frequency. The Y-axis of Fig. 10 shows the proportion of failed paths (paths with negative slack) to non-failed paths (paths with positive slack) for three base characterization corners: the best-case corner (1.10V, 0°C), the typical-case corner (1.0V, 25°C), and the worst-case corner (0.9V, 125°C). We observe that this proportion of failed paths suddenly drops below a certain threshold while the clock is finely scaled with a resolution of 0.01ns. For instance, the proportion falls below 0.5 with only 0.06ns clock scaling in the best-case corner; in the other words, the number of non-failed paths is twice as many as those which

fail. Alternatively, the number of non-failed paths is doubled when the cycle time is increased for 0.3ns in the worst-case corner. These provide an opportunity for an error-free running of some instructions that will not activate those failed paths.

From the previous analysis, we get the intuition that instructions will have different levels of vulnerability to variations depending on the way they exercise the non-uniform critical paths across the various pipeline stages. To capture this phenomenon, we define the concept of instruction-level vulnerability to dynamic variations. The classification of instructions is a valuable information to avoid the timing errors, and the processor can adapt itself accordingly by acquiring the knowledge about which class of instructions is running.

A. Instruction-level Classification

To quantify the ILV_i to voltage and temperature variations for each instruction_i, we compute the probability of failure of instruction_i using a set of Monte Carlo gate-level simulations with back-annotated delays. The ILV_i defines as the total number of violated cycles over the total simulated cycles for the instruction_i. If any of the stages have one or more violated flip-flop at a cycle_j, we consider that stage as a violated stage at cycle_j, since there is at least one activated path for instruction_i at cycle_j which is slow enough to miss the setup time of a flip-flop. Intuitively, if instruction_i runs without any violated path, ILV_i is *0*; on the other hand, ILV_i is *1* if instruction_i faces at least one violated path in any stage, in every cycle. We finely change the clock cycle to observe the paths failure for every exercised instruction, and then consequently evaluate its ILV. Our results indicate that the instructions exhibit a very wide range of delay under different operating conditions ranges from 0.76ns to 4.16ns.

More precisely, the ILV values evidence that the integer instructions are partitioned into three main classes: i) the logical and arithmetic instructions, ii) the memory instructions, and iii) the multiply and divide instructions. The first class shows an abrupt behavior when the clock cycle is slightly varied. Its ILV switches from *1* to *0* with a slight increase in the cycle time (0.02ns) for every corner, mainly because the path distribution of the exercised part by this class is such that most of the paths have the same length, then we have a all-or-nothing effect, which implies that either all instructions within this class fail or all make it. The second class, the memory operations, needs much more relaxed cycle time to be able to survive across conditions. For instance, only 0.02ns more guardbanding on the cycle time of the first class instruction can guarantee the error-free execution of the memory instructions while they are experiencing a full range temperature fluctuation. The third class is the multiply and divide instructions which need higher guardbanding in comparison to the first class instruction, ranges from 0.02ns at (1.1V, -40°C) to 0.30ns at (0.72V, 125°C). Since this class highly exercises the execution unit, it has a higher ILV in comparison with the rest of classes in the same clock cycle, for every corner. Moreover, 64%–82% (depends on the corner) of the failed paths in the execution stage lie in the hardware multiplier and divider. Earlier we have shown that the execution and memory units are not only vulnerable to the temperature and dynamic variations, but also cover most of the failed paths of the processor.

Based on these results, all instruction classes act similarly across the wide range of operating conditions: as the cycle time increases gradually, the ILV becomes *0*, firstly for the first class, then for the second class, and finally for the third class. Therefore, software stack can benefit from this characterized information toward reducing the cost of resiliency by acquiring the knowledge about which class of instructions is/will be running.

IV. Task-level Vulnerability

Going further up on the hardware-software stack, several efforts have tried to characterize and use variability related information for better management. We have earlier defined the notions of ILV that characterizes individual instructions as the most fine-grained abstraction of the processors functionality. Focusing on a stream of instructions, recent work [14], [15] determine sequences of instructions that have a significant impact on the timing error rate. Therefore, code transformations have been introduced for improving their timing speculation. Raising further the level of abstraction, a notion of task-level vulnerability (TLV) [16] is defined. In this section, we describe TLV, as an extension to the OpenMP v3.0 tasking programming model, that dynamically characterizes vulnerability of tasks. Here, the runtime system reduces the cost of recovery by matching different characteristics of each variability-affected core to various levels of vulnerability of tasks.

A. OpenMP Tasking and TLV

The OpenMP specification v3.0 introduces a task-centric model of execution. The new task directive is used to dynamically generate units of parallel work that can be executed by every thread in a parallel team. When an executing thread encounters the task construct, it prepares a task descriptor consisting of the code to be executed, plus a data environment inherited from the enclosing structured block. The tasking programming model is considered as a convenient abstraction for application development in shared memory multi-cores [29]. Thus we integrate TLV metadata as an extension to the OpenMP tasks. A task directive outlines an execution unit which runs a sequence of instructions. The OpenMP directives allow the programmer to statically identify several task types in the program. Every task directive syntactically delimits a unique stream of instructions. While at runtime the same stream may be dynamically instantiated several times (e.g., a task directive nested within a loop), from the point of view of our characterization it uniquely identifies a single task *type*. As a direct consequence, there are as many types of tasks in a program as there are task directives in its code.

ILV indicates that the classes of instructions have different levels of vulnerability to variations depending on the way they exercise the non-uniform critical paths across the various pipeline stages. We note that complex high-performance cores such as IBM POWER6 also confirm that vulnerability is not uniform across the instructions set [18]. We extend the notion of ILV to a more coarse-grained task-level metric, TLV. The vulnerability of a task type varies based on the class of instructions that it will execute. TLV is also a per-core metric since the amount of variation affecting different classes of instructions changes from one core to another. Therefore, each dynamic task (dynamic instance of a task type), can potentially

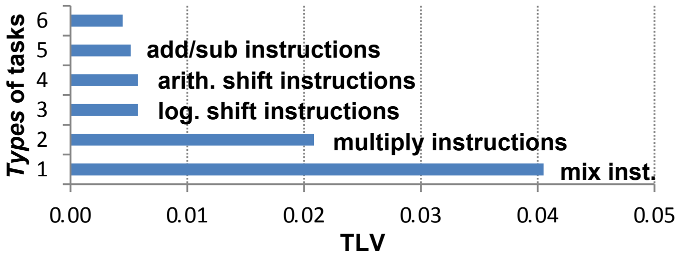

Fig. 11. Intra-corner TLV for six distinct task types.

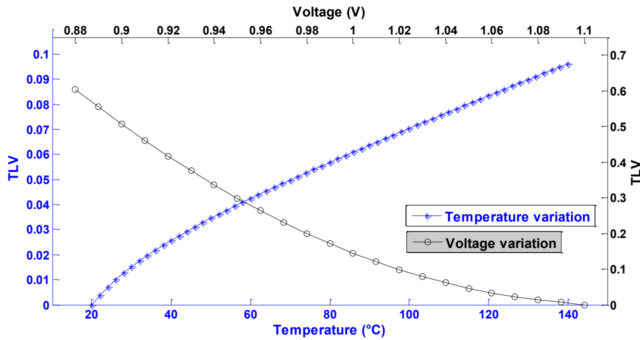

Fig. 12. TLV to dynamic voltage and temperature variations.

face a different density of the errant instructions imposed by both software context and hardware variations.

While the identification of task types can be done statically (i.e., at compile time), their characterization has to be done online due to two reasons. First, dynamic instances of the same task type may exercise the processor pipeline in a non-identical manner due to data-dependent control flow that executes different classes of instructions. Second, the characterization must reflect the variability-affected characteristic of every core (not known a priori) on every task type. Therefore, we define the notion of TLV as a metric to characterize vulnerability of each task type per each core, in the following:

$$TLV_{(i,j)} = \frac{\sum EI}{\sum I} \mid \forall core_i, \forall task_j \quad (1)$$

where ΣEI is the number of *errant instructions* during execution of $task_j$ on $core_i$, that are reported by the circuit sensors and need to be replayed for correct execution; ΣI is the total number of executed instructions. Intuitively, if all the instructions run without any timing error, TLV is *0*; on the other hand, TLV is *1* if every instruction causes at least one timing error. The lower TLV, the lower the number of errant instructions, the lower the cost of recovery, and thus the higher the instruction per cycle.

B. Intra-corner and Inter-corner TLV

We examine intra-corner TLV for a $core_i$ that runs a synthetic benchmark consisting of six distinct types of tasks. Each task is composed of a loop that executes a different class of instructions illustrated in Fig. 11. The $core_i$ works in the typical operating condition, i.e., the room temperature of 25°C and voltage supply of 1.1V. This operating corner is fixed, thus there will be no environmental variation during task execution. The TLV characterization for each task type

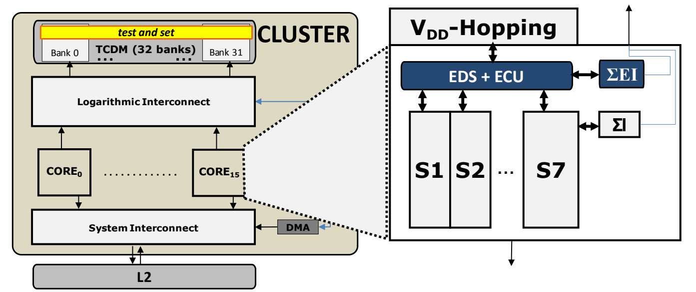

Fig. 13. Variation-tolerant tightly-coupled processor cluster.

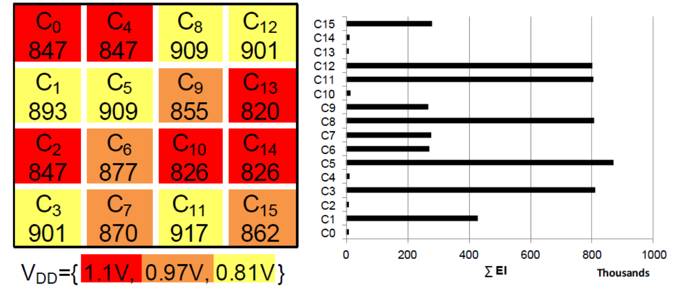

Fig. 14. VDD-Hopping in the variability-affected cluster (left); Number of errant instructions during the synthetic bench (right).

is shown in Fig. 11. As shown, TLV of each type of tasks is different even within the fixed operating condition in the $core_i$. For instance, TLV of task $type_1$ ($TLV_{(i,1)}$) is 9× higher than the $TLV_{(i,6)}$ indicating a considerable variation across the type of tasks. Furthermore, within same type of task, TLV can also be affected by the data-dependent control-flow that can cause execution of different classes of instructions. This TLV variation across task types indicates the need of online monitoring for every task types.

We also examine the TLV across different operating conditions. Specifically, we analyze the effects of a full range of dynamic variations, a temperature range of 20°C–140°C, and a voltage range of 0.88V–1.1V. As shown in Fig. 12, the average TLV of the six types of tasks is an increasing function of temperature. With a fixed voltage of 1.1V, by increasing the temperature the delay of critical paths is increased, thus more instructions will face the timing error which causes TLV to increase up to 0.096 at 140°C. In contrast, decreasing the voltage from the nominal point of 1.1V increases TLV. In lower voltages, the delay of critical paths highly increases, thus imposing a high rate of the errant instructions. For example, a dynamic voltage variation of 0.2V (ΔV=1.1V-0.9V) causes a TLV of 0.507, which implies that more than half of the total executed instructions within tasks failed due to the timing errors of the voltage variation. Fig. 11 illustrates TLV values across different types of task in the typical operating corner and Fig. 12 highlights that TLV of tasks is further magnified across various corners of operating conditions, thus TLV should be characterized for every different operating condition.

C. Variation-tolerant Tightly-coupled Processor Clusters

In this subsection, we describe the architectural details of a variation-tolerant processing cluster shown in Fig. 13 that supports TLV characterization. The cluster is inspired by

tightly-coupled clusters in STMicroelectronics P2012 [19] as the essential component of a many-core fabric. In our implementation, each cluster consists of sixteen 32-bit in-order RISC cores, an L1 software-managed tightly coupled data memory (TCDM) and a low-latency logarithmic interconnection [32]. The TCDM is configured as a shared, multi-ported, multi-banked scratchpad memory that is directly connected to the logarithmic interconnection. The number of TCDM ports is equal to the number of banks to enable concurrent access to different memory locations. Note that one bank of the TCDM provides test-and-set read operations, which we use to implement basic synchronization primitives (for example, locks). The logarithmic interconnection is composed of mesh-of-trees networks to support single cycle communication between processors and memories. When a read/write request is brought to the memory interface, the data is available on the negative edge of the same clock cycle, leading to two clock cycles latency for a conflict-free TCDM access.

The cluster is equipped with two core-level resiliency techniques. First, each core relies on the 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 [26] in its recovery unit; seven replica instructions followed by a valid instruction. Second, the cluster supports a per-core V_{DD}-hopping technique for tuning the voltage of each core individually to compensate the impact of static process variation. The core-level V_{DD}-hopping is employed in a variability-affected tightly-coupled cluster [17]. The V_{DD}-hopping improves the clock speed of the slow cores, thus enabling all components of the variability-affected cluster to work at same frequency (with memories at a 180° phase shift). This technique avoids the inter-core synchronization that would significantly increase L1 TCDM latency.

To observe the effect of static process variation on the frequency of individual cores within the cluster, [17] analyzed how critical paths of each core are affected due to WID and D2D process parameters variation. The maximum frequency distribution of every core is shown in Fig. 14 (left), in which each cores maximum frequency varies significantly due to the process variation. As a result, six cores (C_0, C_2, C_4, C_{10}, C_{13}, C_{14}) cannot meet the design time target frequency of 850 MHz. To compensate this core-to-core frequency variation, the V_{DD}-hopping technique measures the delay variation of each core and then applies the appropriate voltage accordingly (higher voltage for slow cores). The technique utilizes three discrete voltage modes (V_{DD}-high, V_{DD}-medium, V_{DD}-low), consequently, the cluster mitigates the core-to-core variations, and all cores can work with the design time target frequency. More details of V_{DD}-hopping and process variation analysis on the cluster is provided in [17].

In V_{DD}-hopping, cores in various voltage islands display different characteristics. Fig. 14 (right) shows that the number of errant instructions significantly varies across cores cooperating together within a single cluster for executing available tasks. For instance, C_0 faces 7.3K errant instructions, whereas C_1 has more than 428K errant instructions during the synthetic benchmark execution. As shown in Fig. 12, a core with lower voltage has higher TLV (higher ΣEI), and will impose higher extra cycles to correct those errant instructions. Thus a task scheduler that is aware of the individual core characteristics and tasks is better able to match them to reduce the overall penalty for correcting the errant instructions.

D. Decentralized TLV Characterization

To reduce the cost of recovery, TLV metadata guides the runtime scheduler. Since TLV depends on the type of task, we consider individual TLV characterization for every task type. As we already explained, TLV metadata is defined for a given core because different cores can display different variability characteristics. Therefore, each core needs to be characterized during online execution of a task. This results in TLV as a two-dimensional lookup table across tasks and cores. This lookup table is physically distributed across all the 32 banks of TCDM, thus it can be written/read with a two-cycle latency in case of conflict-free communication. Since TLV metadata is 32-bit, and every application will have a bounded number of N supported task types, the cluster needs to allocate a maximum of $N \times 4 \times C$ Bytes for the lookup table, where C is the number of cores in the cluster.

Algorithm 1 Pseudo-code to perform TLV characterization

```
while (HAVE_TASKS) do
    task_desc_t task = EXTRACT_TASK ()
    if (task) then
        float old_mdata = tlv_read_task_metadata (core_id)
                                  ▷ %Reset counter for this core %
        tlv_reset_task_metadata (core_id)
                                                 ▷ %Execute task %
        task.task_fn (task.task_data)
                                 ▷ %Task is executed. Fetch TLV ... %
        float mdata = tlv_read_task_metadata (core_id);
                                     ▷ %Update metadata in table %
        tlv_table_write (task.task_type_id, core_id,
(mdata+old_mdata)/2);
    end if
end while
```

The online characterization mechanism is distributed among all the cores in the cluster, thus enables fully parallel task-level monitoring and characterization. The cluster employs the circuit sensors and the error recovery unit of every core to perform characterization. To quantify TLV, the core collects the statistics of ΣEI and ΣI for Equation 1 through available counters. For instance, [26] does include a counter for the errant instructions ΣEI to change the frequency when the number of errors is above a certain threshold. Two function calls for profiling TLV of current task are inserted in the runtime library, right before and after actual execution (see Algorithm̃refalgo:characterization), and then the lookup table is updated with the new value. The former (*tlv_reset_task_metadata*) restarts the counters, and the latter two (*tlv_read_task_metadata* and *tlv_table_write*) transfers the characterized TLV metadata at the end of task execution to the lookup table for future inspection.

E. Variation-tolerant OpenMP Tasking Scheduler

The lookup table for the characterized TLV metadata acts as a software-accessible monitor that provides information to the runtime systems to guide task scheduling. We propose a reactive variation-tolerant scheduler that we call task-level

errant instruction management (TEIM). The OpenMP implementation that we consider [30] leverages a centralized task queue, where all the threads involved in parallel computation actively push and pop job descriptors. Typically, to avoid redundant computation, only a single thread from a parallel team executes the code within the task directive (pushing its task descriptor in the queue). The rest of the threads remain idle in wait for work to do. Whenever a thread is idle it tries to extract a task from the queue, thus tasks are scheduled to threads on a *first-come, first-served* basis [30].

Our TEIM technique enhances the above baseline scheduler with additional conditional checks. It utilizes TLV metadata to determine whether the querying thread is well suited to run the task on the head of the queue. The overall goal is a guided scheduling of tasks to cores, which reduces the number of errant instructions so that the replay logic is exercised less frequently. In other words, the scheduler tries to match the variability-affected characteristics of the cores with the level of vulnerability of tasks, thus reducing unnecessary recovery cycles. At each scheduling point, an idle core$_i$ runs the scheduler. Then, the scheduler checks two conditions to decide whether the core should execute a task$_j$ in the head of queue, or should skip it and lets other favoured cores execute it later. First, the scheduler reads the TLV metadata entry corresponding to the combination of task$_j$ and core$_i$. If TLV$_{(i,j)}$ is greater than a predefined target threshold (TLV_THR), there is no match between the characteristics of core$_i$ and task$_j$ (execution of task$_j$ on core$_i$ may cause at least TLV_THR$\times\Sigma I$ errant instructions, see Equation 1), so the scheduling attempt fails. Task$_j$ remains in the queue, ready to be reconsidered for scheduling at the next attempt (thus, the rest of cores can potentially execute it). Second, to avoid starvation, each core can skip tasks for a maximum number of ESCAPE_THR times. Beyond this threshold the core has to execute at least one task, independent of its TLV value. These thresholds can be tuned during a profiling phase as described in detailed in [16]. The TEIM algorithm is shown in the following.

Algorithm 2 TEIM algorithm in the variation-tolerant scheduler

▷ %Read metadata table%
TLV_metadata = tlv_table_read (task$_j$, core$_i$)
if (TLV_metadata $\geq$ TLV_THR AND escape_cnt [core$_i$] $\leq$ ESCAPE_THR) **then**
  escape_cnt [core$_i$] ++
  escape(task$_j$)
else
  schedule_to_core$_i$ (task$_j$)
  escape_cnt [core$_i$] = 0
end if

Thus far, we assumed that TLV characterization information is available for the scheduler to take decisions. When the program starts there is no such information for any task type. If no information is available in the lookup table for mapping of a particular task type on a particular core, a TLV of *0* will be returned, so the scheduler simply assigns the task to the requesting core, and enables online characterization. Once a task type is characterized, this information could be used for all the successive instances of the same type and thus the online characterization could be stopped. However, we rather keep the characterization active at every scheduling event and

TABLE I. ARCHITECTURAL PARAMETERS OF CLUSTER.

ARM v6 core	16	TCDM banks	16
I size	16KB per core	TCDM latency	2 cycles
I line	4 words	TCDM size	256KB
Latency hit	1 cycle	L3 latency	$\geqslant$ 60 cycles
Latency miss	$\geqslant$ 59 cycles	L3 size	256MB

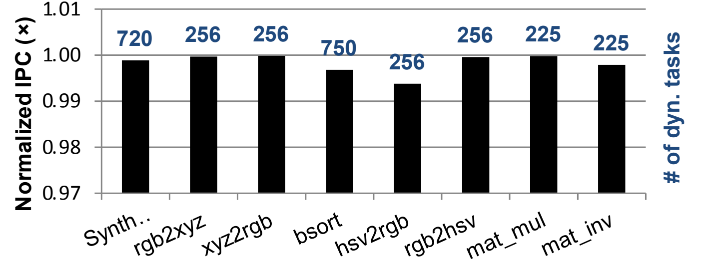

Fig. 15. Overhead of the variation-tolerant scheduler.

average the new characterized TLV value with the already TLV metadata available in the lookup table. This results in a better characterization for tasks that exhibit data-dependent control flow. Moreover, it also incorporates recent effects of dynamic variations on cores, including temperature fluctuation. Therefore, the scheduler uses the latest metadata generated from monitoring recent changes in both hardware and software. For each task scheduling point, the scheduler overhead for such decision-making is highly amortized over task execution.

F. Effectiveness of TEIM

We demonstrate our approach on a SystemC-based virtual platform [31] modeling the tightly-coupled cluster described in Section IV-C. Table I summarizes its parameters.

To emulate variations on the virtual platform, we have integrated variations models at the level of individual instructions using the ILV characterization methodology presented in Section III. Integration of ILV models for every core enables online assessment of presence or absence of errant instructions at the certain amount of dynamic voltage and temperature variations. We re-characterized ILV models of an in-order RISC LEON-3 core for 45-nm. This choice is because of availability of an advanced open-source RISC core that provides full back-end details for variation analysis. First, we synthesized the VHDL code of LEON-3 with the 45-nm TSMC technology library, general-purpose process. The front-end flow with normal V$_{TH}$ cells has been performed using *Synopsys DesignCompiler*, while *Synopsys IC Compiler* has been used for the back-end where the core is optimized for performance.

To observe the effects of a full range of dynamic voltage and temperature variations, we analyze the delay variability on the individual instructions, leveraging voltage-temperature scaling features of *Synopsys PrimeTime* for the composite current source approach of modeling cell behavior. Finally, delay variability is annotated to the gate-level simulations for creating ILV models. To utilize ILV models on the virtual platform, each core maps ARM v6 instructions to the corresponding ILV models in an instruction-by-instruction fashion during execution of tasks. Therefore, every core will face the errant instructions during tasks execution on the variability-affected cluster described in Section IV-C.

Our OpenMP implementation for the target cluster is based

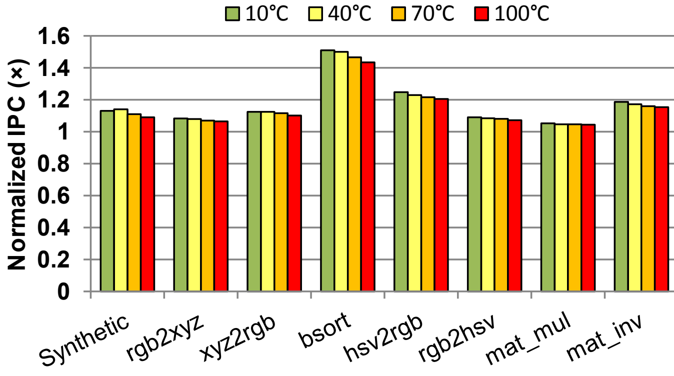

Fig. 16. Normalized IPC improvement of the variability-affected cluster using TEIM across a wide temperature range.

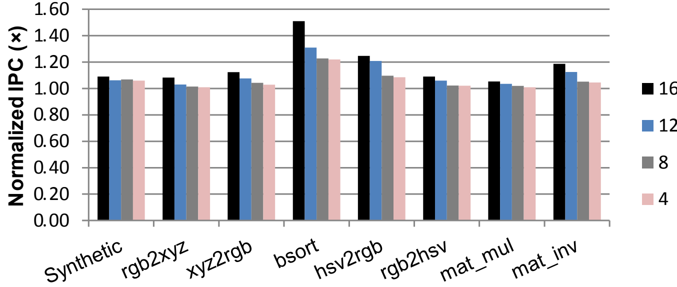

Fig. 17. Cluster IPC improvement using TEIM with various number of cores.

on [31]. To evaluate the effectiveness of the variation-tolerant technique, seven widely used computational kernel from the image processing domain are parallelized using OpenMP tasking. To quantify improvement of our technique, we have used normalized IPC of the cluster as a metric which divides the IPC of the cluster when using TEIM scheduler by the IPC of the cluster when using the baseline scheduler. First, we have quantified the overhead of TEIM technique on a variation-immune cluster (none of cores is affected by variations). Fig. 15 shows the normalized IPC of the variation-immune cluster for the benchmarks. On average, the normalized IPC of the cluster (the effective instructions) is slightly decreased by 0.998×. This tiny overhead is imposed by reading the TLV lookup table, and checking the conditions mentioned in Algorithm 2. During executions, the TLV lookup table only occupies 104–448 Bytes depending upon the number of task types. The number of dynamic tasks for each benchmark is illustrated on top of the bars in Fig. 15.

The variation-tolerant scheduler imposes negligible IPC degradation in the variation-immune cluster, while it outperforms the baseline scheduler in the variability-affected clusters and effectively amortizes the cost of TCPC. Fig. **??** shows the normalized IPC improvement of the variability-affected cluster (shown in Fig. 13). As shown, the normalized IPC is increased for all benchmarks, e.g., at 10°C, IPC of bsort is increased by a factor of 1.51× (1.17× on average for all benchmarks). TEIM technique decreases the number of cycles per cluster for each type of tasks, because cores incur fewer errant instructions and spend lower cycles for recovery. Thus, the effective IPC is increased (compared to the baseline scheduler, the cluster spends fewer cycles for the same amount of work). Moreover, this saving is consistent across a wide range of temperature variations with a slight decrease due to the slower critical paths. At temperature of 100°C (ΔT=90°C), TEIM achieves 1.15× IPC improvement, on average, thanks to the online TLV metadata characterization which reflects the latest variations, thus enables the scheduler to react accordingly. On average, each task is escaped 2.1 times because of no matching core. Overall, it shows that the tasks are postponed for a short latency in the queue, thus the performance penalty is avoided in the synchronization of tasks on a barrier.

Fig.17 shows the normalized IPC improvement of the cluster, when dedicating different number of cores for execution of tasks. On average, at 10°C, TEIM achieves 1.17×, 1.11×, 1.11×, and 1.07× IPC improvement when using only 16, 12, 8, and 4 cores, respectively. It shows effectiveness of TEIM in presence of various hardware resources, and variation scenario. TEIM achieves higher normalized IPC across higher number of cores (where there are higher variations and more voltage islands – see Fig. 13). TEIM is also effective with a 4-core scenario (C_0–C_3) in which the available two voltage islands are proactively utilized.

In summary, we propose a method for vertical abstraction of circuit-level variations into a high-level parallel software execution (OpenMP v3.0 tasking). Our method characterizes and mitigates variations at the level of tasks, identified by the programmer through annotations. The vulnerability of tasks is characterized by TLV metadata during introspective execution on individual cores. A variation-tolerant runtime scheduler (TEIM) is proposed to utilize characterized TLV metadata. TEIM matches different characteristics of each variability-affected core to various levels of vulnerability of tasks. Therefore, it enhances normalized IPC (compared to the baseline scheduler [30]) of a 16-core variability-affected cluster up to 1.51×. On average, it achieves 1.15×–1.17× normalized IPC improvement for a wide range of temperature fluctuation.

V. Conclusion

Manufacturing and environmental variability lead to timing errors in computing systems that are typically corrected by error detection and correction mechanisms at the circuit-level. The cost and speed of recovery can be improved by exposing variability in higher levels. This paper describes approaches that enhance the scope of *'detect-then-correct'* mechanism especially in the parallel execution context:

1) ILV, or instruction-level vulnerability, quantifies the effect of voltage and temperature variations on the performance of an in-order processor at the level of individual instructions. In fact, ILV data partitions instructions into various classes with different vulnerabilities.
2) Moving to data-level parallel SIMD architecture, a concurrent instruction reuse (CIR) technique is proposed to avoid the costly recovery in close proximity. It leverages the inherent value locality of applications by memoizing the result of an error-free instruction on an instance of data. Then, it reuses this memoized result to exactly (or approximately) correct any errant instruction on other instances of the same (or adjacent) data.
3) Finally, we present a variation-tolerant tasking technique for tightly-coupled shared memory processor clusters that relies upon modeling advance across

the hardware/software interface. This is implemented as an extension to the OpenMP v3.0 tasking programming model. Our method characterizes and mitigates variations at the level of tasks. The vulnerability of tasks is characterized by TLV metadata during introspective execution on individual cores. A variation-tolerant runtime scheduler is proposed to utilize characterized TLV metadata that matches different characteristics of each variability-affected core to various levels of vulnerability of tasks.

An ongoing work is focused on utilizing the spatial memoization to spontaneously apply clock gating for MW lanes. We will further explore variability-aware workload distribution and related programming models for multi-cluster architectures.

REFERENCES

[1] S. Ghosh, and K. Roy, *Parameter Variation Tolerance and Error Resiliency: New Design Paradigm for the Nanoscale Era,* Proc. of the IEEE , vol.98, no.10, pp.1718-1751, Oct. 2010.

[2] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, V. De, *Parameter variations and impact on circuits and microarchitecture,* Proc. IEEE/ACM DAC, 2003, pp. 338-342.

[3] ITRS [Online]. Available: http://public.itrs.net

[4] P. Gupta, et al., *Underdesigned and Opportunistic Computing in Presence of Hardware Variability* IEEE Trans. on CAD of Integrated Circuits and Systems, pp. 489-499, Jan. 2013.

[5] L. Wanner, R. Balani, S. Zahedi, C. Apte, P. Gupta, M. Srivastava, *Variability-aware duty cycle scheduling in long running embedded sensing systems,* Proc. IEEE/ACM DATE, 2011.

[6] Hyungmin Cho, L. Leem, S. Mitra, *ERSA: Error Resilient System Architecture for Probabilistic Applications,* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.31, no.4, pp.546-558, April 2012.

[7] J. Cong, and K. Gururaj, *Assuring Application-level Correctness against Soft Errors,* Proc. IEEE/ACM ICCAD, 2011, pp. 150-157.

[8] E. Mintarno, J. Skaf, R. Zheng, J.B. Velamala, Y. Cao, S. Boyd, R.W. Dutton, S. Mitra, *Self-Tuning for Maximized Lifetime Energy-Efficiency in the Presence of Circuit Aging,* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.30, no.5, pp.760-773, May 2011.

[9] F. Ahmed, M.M. Sabry, D. Atienza, L. Milor, *Wearout-aware compiler-directed register assignment for embedded systems,* Proc. IEEE ISQED, 2012, pp.33-40.

[10] M. Namaki-Shoushtari, A. Rahimi, N. Dutt, P. Gupta, R. K. Gupta, *Aging-aware GPGPU Register File Allocation,* Proc. of ACM/IEEE CODES+ISSS, 2013.

[11] R. Hegde, N. R. Shanbhag, *Energy-efficient signal processing via algorithmic noise-tolerance,* Proc. ACM/IEEE ISLPED, 1999, pp. 30-35.

[12] P. Dubey, *Recognition, Mining and Synthesis Moves Computers to the Era of Tera,* Technology at Intel Magazine, 2005.

[13] A. Rahimi, L. Benini, R.K. Gupta, *Spatial Memoization: Concurrent Instruction Reuse to Correct Timing Errors in SIMD Architectures,* IEEE Transactions on Circuits and Systems II, 2013.

[14] G. Hoang, R. B. Findler, R. Joseph, *Exploring circuit timing-aware language and compilation,* Proc. ACM ASPLOS, 2011, pp. 345-355.

[15] A. Rahimi, L. Benini, R.K. Gupta, *Application-Adaptive Guardbanding to Mitigate Static and Dynamic Variability,* IEEE Transactions on Computers, 2013.

[16] A. Rahimi, A. Marongiu, P. Burgio, R. K. Gupta, L. Benini, *Variation-tolerant OpenMP Tasking on Tightly-coupled Processor Clusters,* Proc. ACM/IEEE DATE, 2013.

[17] A. Rahimi, L. Benini, R.K. Gupta, *Procedure hopping: A low overhead solution to mitigate variability in shared-L1 processor clusters,* Proc. ACM/IEEE ISLPED, 2012, pp. 415-420.

[18] P. N. Sanda, et al., *Soft-error resilience of the IBM POWER6 processor,* IBM Journal of Research and Development, 52(3): 275-284, May 2008.

[19] L. Benini, E. Flamand, D. Fuin, D. Melpignano, *P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator,* Proc. ACM/IEEE DATE, 2012, pp. 983-987.

[20] D. Atienza, et al., *Reliability-aware design for nanometer-scale devices,* Proc. IEEE ASPDAC, 2008.

[21] M.R. Kakoee, et al., *Variation-Tolerant Architecture for Ultra Low Power Shared-L1 Processor Clusters,* IEEE Trans. on Circuits and Systems II, vol.59, no.12, pp.927-931, Dec. 2012.

[22] D. Jeon, et al., *Design Methodology for Voltage-Overscaled Ultra-Low-Power Systems,* IEEE Trans. on Circuits and Systems II, vol.59, no.12, pp.952-956, Dec. 2012.

[23] K.A. Bowman, et al., *Energy-Efficient and Metastability-Immune Resilient Circuits for Dynamic Variation Tolerance,* IEEE JSSC, 2009.

[24] A. Raychowdhury, et al., *Tunable Replica Bits for Dynamic Variation Tolerance in 8T SRAM Arrays,* IEEE JSSC, pp.797-805, April 2011.

[25] S. Das, et al., *A Self-tuning DVS Processor Using Delay-error Detection and Vorrection,* IEEE JSSC, vol. 41, no. 4, pp. 792-804, Apr. 2006.

[26] K.A. Bowman, et al., *A 45 nm Resilient Microprocessor Core for Dynamic Variation Tolerance,* IEEE JSSC, pp.194-208, Jan. 2011.

[27] M.-M. Papadopoulou, et al., *Micro-benchmarking the GT200 GPU* Technical report, Computer Group, ECE, University of Toronto, 2009.

[28] R. Pawlowski, et al., *A 530mV 10-lane SIMD Processor with Variation Resiliency in 45nm SOI,* Proc. IEEE ISSCC, 2012, pp. 492-494.

[29] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, G. Zhang, *The Design of OpenMP Tasks,* IEEE Transactions on Parallel and Distributed Systems, vol.20, no.3, pp.404-418, March 2009.

[30] FSF - The GNU Project. GOMP - An OpenMP implementation for GCC [online]. Available: http://gcc.gnu.org/projects/gomp

[31] D. Bortolotti et al., *Exploring instruction caching strategies for tightly-coupled shared-memory clusters,* Proc. Intern. Symposium on System on Chip (SoC), 2011, pp.34-41.

[32] A. Rahimi, I. Loi, M.R. Kakoei, L. Benini, *A Fully-Synthesizable Single-Cycle Interconnection Network for Shared-L1 Processor Clusters,* Proc. ACM/IEEE DATE, pp.1-6, 2011.

[33] E. Krimer, et al., *Lane Decoupling for Improving the Timing-error Resiliency of Wide-SIMD Architectures,* Proc. IEEE/ACM ISCA, 2012.

[34] A. Sodani and G.S. Sohi, *Dynamic Instruction Reuse,* Proc. IEEE/ACM ISCA, 1997, pp. 492-494.

[35] LEON3 [Online]. Available: http://www.gaisler.com/cms/

[36] C. A. Martinez, et al., *Dynamic Toler-ance Region Computing for Multimedia,* IEEE Transactions on Computers, pp.650-665, May 2012.

[37] C. Alvarez, et al., *Fuzzy Memoization for Floating-point Multimedia Applications,* IEEE Transactions on Computers , pp.922-927, July 2005.

[38] A. Rahimi, L. Benini, R. K. Gupta, *Hierarchically Focused Guardbanding: An Adaptive Approach to Mitigate PVT Variations and Aging,* Proc. ACM/IEEE DATE, 2013.

[39] A. Rahimi, L. Benini, R. K. Gupta, *Aging-Aware Compiler-Directed VLIW Assignment for GPU Architectures,* Proc. ACM/IEEE DAC, 2013.

[40] C. Ferri, et al., *NBTI-aware Data Allocation Strategies for Scratchpad Memory Based Embedded Systems,* Proc. LATW, 2011.

[41] K. Chakraborty, et al., *Efficiently Tolerating Timing Violations in Pipelined Microprocessors,* Proc. ACM/IEEE DAC, 2013.

[42] AMD APP SDK 2.5 [Online]. Available: www.amd.com/stream

[43] M. A. Breuer, *Multi-media Applications and Imprecise Computation,* Proc. IEEE DSD, 2005.

[44] A. Rahimi, A. Marongiu, R. K. Gupta, L. Benini, *A Variability-Aware OpenMP Environment for Efficient Execution of Accuracy-Configurable Computation on Shared-FPU Processor Clusters,* Proc. ACM/IEEE CODES+ISSS, 2013.

[45] H. Esmaeilzadeh, A. Sampson, L. Ceze, D. Burger, *Architecture Support for Disciplined Approximate Programming,* Proc. ACM ASPLOS, 2012, pp. 301-312.

[46] Multi2Sim [Online]. Available: http://www.multi2sim.org/

[47] A. Rahimi, L. Benini, R. K. Gupta, *Analysis of Instruction-level Vulnerability to Dynamic Voltage and Temperature Variations,* Proc. IEEE/ACM DATE, 2012.

[48] FloPoCo [Online]. Available: http://flopoco.gforge.inria.fr/

[49] AMD APP OpenCL Programming Guide, Chapter 6.6.1, pp. 157, 2012.