Underdesigned and Opportunistic Computing in Presence of Hardware Variability

Puneet Gupta[1], *Member, IEEE,* Yuvraj Agarwal[2], *Member, IEEE,* Lara Dolecek[1], *Member, IEEE,* Nikil Dutt[6], *Fellow, IEEE,* Rajesh K. Gupta[2], *Fellow, IEEE,* Rakesh Kumar[4], *Member, IEEE,* Subhasish Mitra[5], *Member, IEEE,* Alexandru Nicolau[6], *Member, IEEE,* Tajana Simunic Rosing[2], *Member, IEEE,* Mani B. Srivastava[1], *Fellow, IEEE,* Steven Swanson[2], *Member, IEEE,* Dennis Sylvester[3], *Fellow, IEEE,*

***Abstract*—Microelectronic circuits exhibit increasing variations in performance, power consumption, and reliability parameters across the manufactured parts and across use of these parts over time in the field. These variations have led to increasing use of overdesign and guardbands in design and test to ensure yield and reliability with respect to a rigid set of datasheet specifications. This paper explores the possibility of constructing computing machines that purposely expose hardware variations to various layers of the system stack including software. This leads to the vision of underdesigned hardware that utilizes a software stack that opportunistically adapts to a sensed or modeled hardware. The envisioned Underdesigned and Opportunistic Computing (UnO) machines face a number of challenges related to the sensing infrastructure and software interfaces that can effectively utilize the sensory data. In this paper, we outline specific sensing mechanisms that we have developed and their potential use in building UnO machines.**

I. Introduction

The primary driver for innovations in computer systems has been the phenomenal scalability of the semiconductor manufacturing process that has allowed us to literally print circuits and build systems at exponentially growing capacities for the last three decades. After reaping the benefits of the Moore's law driven cost reductions, we are now beginning to experience dramatically deteriorating effects of material properties on (active and leakage) power, and die yields. So far, the problem has been confined to the hardware manufacturers who have responded with increasing guardbands applied to microelectronic chip designs. However the problem continues to worsen with critical dimensions shrinking to atomic scale. For instance, the oxide in 22 nm process is only five atomic layers thick, and gate length is only 42 atomic diameters across. This has translated into exponentially increasing costs of fabrication and equipment to achieve increasingly precise control over manufacturing quality and scale [1].

We are beginning to see early signs of trouble that may make benefits of semiconductor industry unattainable for all but the most popular (hence highest volumes) of consumer gadgets. Most problematic is the substantial fluctuation in critical device/circuit parameters of the manufactured parts across the die, die-to-die and over time due to increasing susceptibility to errors and circuit aging-related wear-out. Consequently, the microelectronic substrate on which modern computers are built is increasingly plagued by variability in performance (speed, power) and error characteristics, both across multiple instances of a system and in time over its usage life. Consider the variability in circuit delay and power as it has increased over time as shown in Figure 1. The most immediate impact of such variability is on chip yields: a growing number of parts are thrown away since they do not meet the timing and power related specifications. Left unaddressed this trend towards parts that scale in neither capability nor cost will cripple the computing and information technology industries.

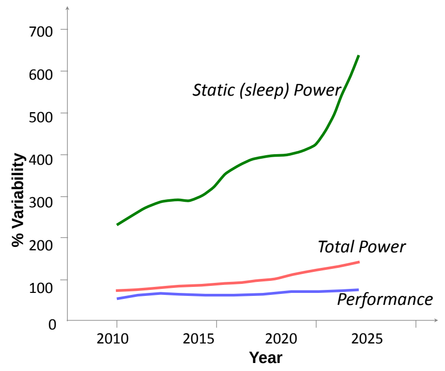

Fig. 1. Circuit variability as predicted by ITRS [3]

The solution may stem from the realization that the problem is not variability per se, rather how computer system designers have been treating the variability. While chip components no longer function like the precisely chiseled parts of the past, the basic approach to the design and operation of computers has remained unchanged. Software assumes hardware of fixed specifications that hardware designers try hard to meet. Yet hardware designers must accept conservative estimates of hardware specifications and cannot fully leverage software's inherent flexibility and resilience. Guardband for hardware design increases cost - maximizing performance incurs too much area and power overheads [2]. It leaves enormous performance and energy potential untapped as the software assumes lower performance than what a majority of instances of that platform may be capable of most of the time.

In this paper, we outline a novel, flexible hardware-

P. Gupta, L. Dolecek and M. Srivastava are with University of California, Los Angeles.
Y. Agarwal, R.K. Gupta and T. S. Rosing are with University of California, San Diego.
N. Dutt and A. Nicolau are with University of California, Irvine.
S. Mitra is with Stanford University.
R. Kumar is with University of Illinois, Urbana-Champaign.
D. Sylvester is with University of Michigan, Ann Arbor.

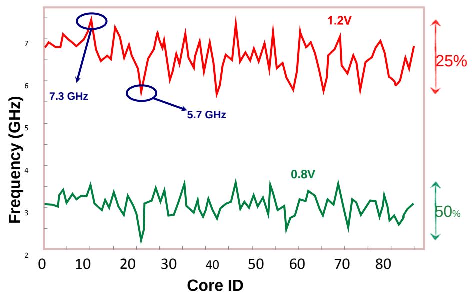

Fig. 2. Frequency variation in an 80-core processor within a single die in Intel's 65nm technology [5]

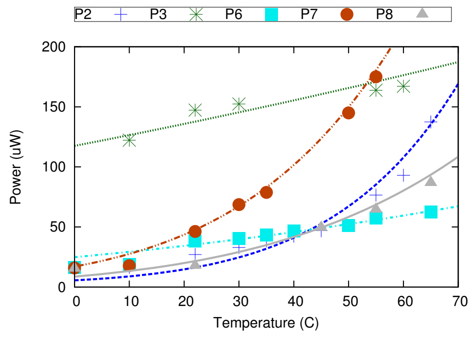

Fig. 3. Sleep power variability across temperature for ten instances of an off-the-shelf ARM Cortex M3 processor [7]. Only 5 out of 10 measured boards are shown for clarity.

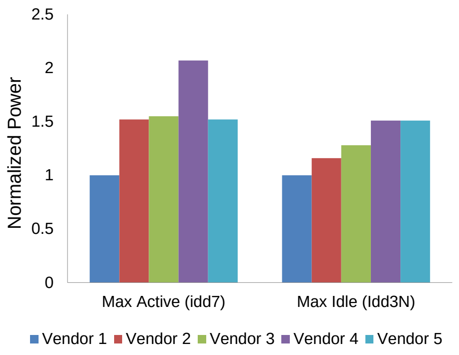

Fig. 4. Power variation across five 512 MB DDR2-533 DRAM parts [8].

software stack and interface that use adaptation in software to relax variation-induced guard-bands in hardware design. We envision a new paradigm for computer systems, one where nominally designed (and hence underdesigned) hardware parts work within a software stack that opportunistically adapts to variations.

This paper is organized as follows. The next section discusses sources of variability and exemplifies them with measurements from actual systems. Section 3 gives an introduction of underdesigned and opportunistic computing. Section 4 outlines various methods of variation monitoring with examples. Section 5 discusses UnO hardware while Section 6 discusses UnO software stack implications. We conclude in Section 7.

II. Measured variability in contemporary computing systems

Scaling of physical dimensions in semiconductor circuits faster than tolerances of equipments used to fabricate them has resulted in increased variability in circuit metrics such as power and performance. In addition, new device and interconnect architectures as well as new methods of fabricating them are changing sources and nature of variability. In this section, we discuss underlying physical sources of variation briefly; and illustrate the magnitude of this variability visible to the software layers by measured off-the-shelf hardware components.

There are four sources of hardware variation:

- *Manufacturing.* The International Technology Roadmap for Semiconductors (ITRS) highlights power/performance variability and reliability management in the next decade as a red brick (i.e., a problem with no known solutions) for design of computing hardware. As an example, Figure 2 shows within-die 3σ performance variation of more than 25% at 0.8V in a recent experimental Intel processor. Similarly, on the memory/storage front, recent measurements done by us show 27% and 57% operation energy variation and 50% bit error rate variation between nominally identical flash devices (i.e., same part number) [4]. This variability can also include variations coming from manufacturing the same design at multiple silicon foundries.
- *Environment.* ITRS projects Vdd variation to be 10% while the operating temperature can vary from - 30^oC to 175^oC (e.g., in the automotive context [6]) resulting in over an order of magnitude sleep power variation (e.g., see Figure 3) resulting in several tens of percent performance change and large power deviations.
- *Vendor.* Parts with almost identical specifications can have substantially different power, performance or reliability characteristics, as shown in Figure 4. This variability is a concern as single vendor sourcing is difficult for large-volume systems.
- *Aging.* Wires and transistors in integrated circuits suffer substantial wear-out leading to power and performance changes over time of usage (e.g., see Figure 5). Physical mechanisms leading to circuit aging include bias temperature instability (i.e., BTI where threshold voltage of transistors degrades over time), hot carrier injection (i.e., HCI where transistor threshold voltage degrades every time it switches), electromigration (i.e., EM where wire width shrinks as more currents passes through it).

In the remainder of this section, we give a few examples of variability measurement in modern off-the-shelf computing components.

A. Power Variability in Contemporary Embedded Processors

Energy management methods in embedded systems rely on knowledge of power consumption of the underlying computing platform in various modes of operation. In previous work

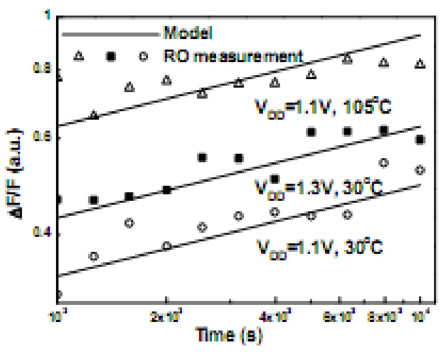

Fig. 5. Normalized frequency degradation in a 65nm ring oscillator due to NBTI [9].

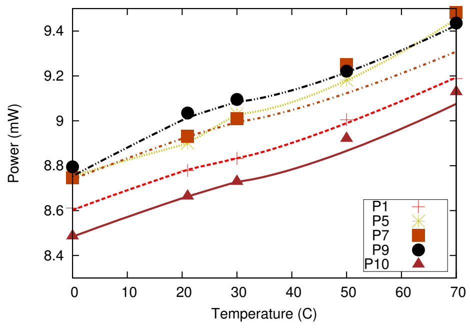

Fig. 6. Measured active power variability for ARM Cortex M3 processor. Only 5 out of 10 measured boards are shown for clarity.

[7], we measured and characterized instance and temperature-dependent power consumption for the Atmel SAM3U, a contemporary embedded processor based on an ARM Cortex M3 core, and discussed the implications of this variation to system lifetime and potential software adaptation. Figure 3 shows that at room temperature the measured sleep power spread was 9X which increased to 14X if temperature variation (20^oC to 60^oC) was included as well [1]. Such large variation in leakage power is to be expected given its near exponential dependence on varying manufacturing process parameters (e.g., gate length and threshold voltage) as well as temperature. To the best of our knowledge, this embedded processor was manufactured in 90nm or older technology. Given that the leakage as well as leakage variability are likely to increase with technology generations, we expect future embedded processors to fare much worse in terms of variability. Another interesting point to note is that six out of the ten boards that we measured were out of datasheet specification highlighting the difficulty in ensuring a pessimistic datasheet specification for sleep power.

On the same processor core, we measured active power variation as well. The power spread at room temperature is about 5% while it is 40% if temperature dependence is included as well. Actual switching power has a weak linear dependence on temperature and its process dependence is also relatively small. Most of the temperature-dependent variation observed is likely due to variation in short-circuit power at lower temperatures and leakage component of active power at higher temperatures.

B. Power Consumption Variability in Contemporary General Purpose Processors

On the other end of the power and performance spectrum are the higher-end modern processors from Intel and AMD that are meant for laptop, desktop and server class devices. Energy management is critically important in these platforms due to battery lifetime concerns (mobile devices) or rising energy costs (desktops and servers). In modern platforms, the processor is often a dominant energy consumer and with increasingly energy efficient designs has a very large dynamic range of operation. As a result characterizing processor power consumption accurately such that adaptive algorithms that can manage their power consumption is key.

However, it is likely that characterizing power consumption on just one instance of a processor may not be enough since part-to-part power variability may be significant and knowing its extent is therefore critical. Using a highly instrumented Calpella platform from Intel, we were able to isolate the power consumption of multiple subsystems within contemporary x86 processors (Intel Nehalem class). Using this Calpella platform, and an extensive suite of single-threaded and modern multi-threaded benchmarks such as the SPEC CPU2006 and PARSEC suites, we characterized the variability in power consumption across multiple cores on the same die, and also across cores on different dies. We used Linux `cpu-sets` to pin the benchmark and the OS threads to specific cores for these experiments. To minimize experimental error (which is eventually very small as measured by standard deviations across runs), we ran each benchmark configuration multiple times (n=5); we swapped the processors in and out and repeated the experiments to eliminate any effects due to the thermal couplings of the heatsink, as well as rebooted the Calpella platform multiple times to account for any effects due to the OS scheduling decisions.

Table I summarizes the variability in power consumption across cores on six instances of identical dual-core Core i5-540M processors for different configurations such as turning TurboBoost ON or OFF and also turning processor sleep states (C-states) ON or OFF. We do not show the variability across multiple cores on the *same* processor (within die variability) since it was quite low and instead focus on the variability across instances (across-die variability). We also measure benchmark completion times and use that to measure the variability in energy consumption. The data presented in the table is for our final benchmark set consists of nineteen benchmarks from the SPEC CPU 2006 benchmark set [10] chosen to cover a wider range of resource usage patterns.

As shown in Table I the variation in power consumption observed across dies is significant (maximum 22% for a particular benchmark and average 12.8% across all benchmarks) and depends on the processor configuration. Our data provides evidence that the power variation

[1]All temperature experiments were conducted in a controlled temperature chamber. Further details can be found in [7]

	TurboBoost = OFF				TurboBoost = ON			
	C-States = ON		C-States = OFF		C-States = ON		C-States = OFF	
	Power	Energy	Power	Energy	Power	Energy	Power	Energy
Maximum Variation (for any particular benchmark)	22%	16.7%	16.8%	16.8%	11.7%	10.6%	15.1%	14.7%
Average Variation (across all benchmarks)	14.8%	13.4%	14.7%	13.9%	8.6%	7.9%	13.2%	11.7%

TABLE I
VARIABILITY IN POWER AND ENERGY CONSUMPTION MEASURED ACROSS SIX IDENTICAL CORE I5-540M PARTS, WITH DIFFERENT PROCESSOR FEATURES ENABLED OR DISABLED. OUR DATA SHOWS THAT THE VARIABILITY ACROSS PARTS IS INDEED SIGNIFICANT AND IT DECREASES WITH TURBOBOOST AND PROCESSOR C-STATES ENABLED DUE TO REDUCTION IN LEAKAGE POWER. THE *maximum variation* VALUES PRESENTED IN DEPICTS VARIABILITY FOR A PARTICULAR BENCHMARK THAT SHOWS THE LARGEST VARIATION WHILE THE *average variation* SHOWS THE AVERAGE VARIATION IN POWER AND ENERGY ACROSS ALL THE BENCHMARKS FOR A PARTICULAR CONFIGURATION.

we observe is dominated by the differences in leakage power across processor dies. For instance, disabling the processor sleep states (C-states=OFF) increases the parts of the chip that remain powered on, thereby increasing the leakage power consumption and in turn increasing the power variability. As shown in Table I, the average power variation across all benchmarks indeed increases when C-states are disabled (C-States=OFF case); the varition increases from 8.6% (TurboBoost=ON, C-states=ON) to 13.2% (TurboBoost=ON,C-States=OFF).

C. *Variation in Memories*

The push toward "big data" applications has pushed the performance, reliability, and cost of memory technologies to the forefront of system design. Two technologies are playing particularly important roles: DRAM (because it is very fast) and NAND flash (because it is non-volatile, very dense/cheap, and much faster than disk. In both technologies, variability manifests itself as bit-errors, differences in performance, and differences in energy efficiency.

a) Flash Memory: To attain ever-greater flash memory densities, flash memory manufacturers are pushing the limits of the process technology and the principles of operation that underlie flash devices. The result is growing grow levels of variability in flash performance at multiple scales.

Careful measurements of flash behavior [4] allow us to quantify variability in flash's read, write, and erase performance, raw bit error rate, and energy consumption. The results show both systematic variation with flash blocks and random variation at multiple scales. Figure 7 show the systematic variation we observe in program latency for multi-level cell (MLC) flash devices. Power measurements of the same operations show that power consumption is roughly constant across pages, leading to a similar variation in energy per programmed bit. We have demonstrated that SSD firmware and the operating system can leverage this variation to improve performance of high-priority IO requests or reduce energy consumption during, for instance, battery powered operation.

Rising density has had as especially detrimental affect on flash's bit error rate and the number of program/erase cycles a flash block can sustain before it becomes unusable. We frequently observe variation of 10× or more in raw bit error rate across blocks on the same chip. Even more insidious is the decrease in data retention (i.e., how long data remains intact in an idle block) with program/erase cycling: Accelerated aging measurements show that by the end of its program/erase life

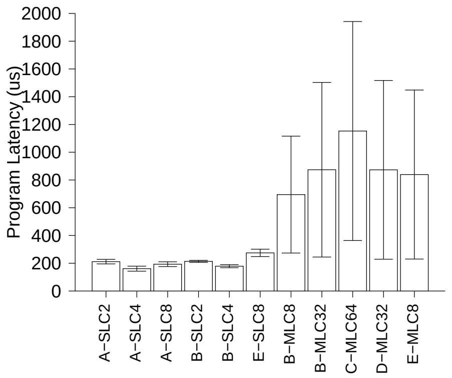

Fig. 7. Programming performance exhibits wide and predictable variation across pages within a single flash block. Since program power is constant, the same effect leads to energy efficiency variation as well. Chip identifier denote the manufacturer (the initial letter), cell type (MLC or SLC), and capacity. Error bars are one standard deviation.

time, data retention time drops by 90% (from 10 years to 1). We are using these and other measurements to predict when individual flash blocks will go bad, and allocate different types of data (e.g., long-term storage vs. temporary files) to different region of the flash memory within an SSD.

b) DRAM: [11] tested 22 double date rate (DDR3) dual inline memory modules (DIMMs), and found that power usage in DRAMs is dependent both on operation type (write, read, and idle) as well as data, with write operations consuming more than reads, and 1s in the data generally costing more power than 0s. Temperature had little effect (1-3%) across the -50 °C to 50 °C range. Variations were up to 12.29% and 16.40% for idle power within a single model and for different models from the same vendor, respectively. In the scope of all tested 1 gigabyte (GB) modules, deviations were up to 21.84% in write power. The measured variability results are summarized in Fig. 8. Our ongoing work addresses memory management methods to leverage such power variations (for instance variability-aware Linux virtual memory management [12]).

III. UnO Computing Machines

*U*nderdesigned *a*n*d* *O*pportunistic (UnO) computing machines provide a unified way of addressing variations due to manufacturing, operating conditions, and even reliability failure mechanisms in the field: difficult-to-predict spatiotemporal variations in the underlying hardware, instead

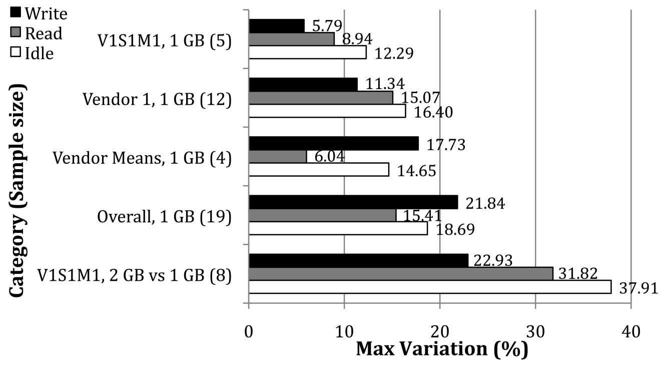

Fig. 8. Maximum variations in Write, Read, and Idle Power by DIMM Category, 30 °C. Instrumentation measurement error is less than 1%.

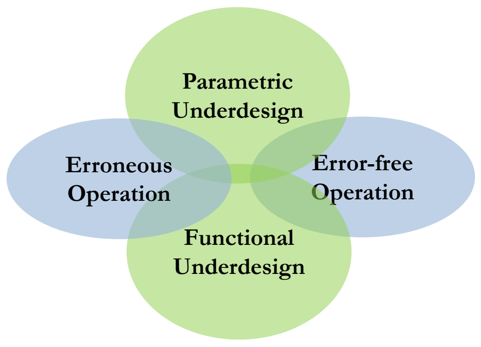

Fig. 9. A classification of UnO computing machines.

of being hidden behind conservative specifications, are fully exposed to and mitigated by multiple layers of software.

A taxonomy of UnO machines is shown in Figure 9. They can be classified along the following two axes.

- Type of Underdesign. Use parametrically under-provisioned circuits (e.g., voltage over-scaling as in [13], [14]) or be implemented with explicitly altered functional description (e.g., [15]–[17]);
- Type of Operation. Erroneous operation may rely upon applications level of tolerance to limited errors (as in [18]–[20] to ensure continued operation. By contrast, error-free UNO machines correct all errors (e.g., [13]) or operate hardware within correct-operation limits (e.g., [7], [21]). Furthermore, for erroneous systems, the exact error characteristics may or may not be known. In parametrically underdesigned erroneous systems, it may not be easy to infer the exact error behavior. For instance, voltage overscaling can cause timing errors thereby changing the functional behavior of the circuit but the exact input dependence of such errors may be tough to predict especially in the presence of process or environmental variations. On the other hand, such input to error mapping is known a priori for functionally underdesigned systems (though it may be too complex to keep track of).

A large class of applications is tolerant to certain kinds of errors due to algorithmic or cognitive noise tolerance. For example, certain multimedia and iterative applications are tolerant to certain numerical errors and other errors in data. An numerical or data error often simply affects the quality of output for such applications. The number of errors tolerable by such applications is bounded by the degree to which a degradation in output quality is acceptable. System correctness can be carefully relaxed for such applications for improved power and performance characteristics. Note that only certain classes of errors are acceptable. System still needs to avoid errors in control or errors that may cause exceptions or I/O errors. Our recent work [22] presents a compiler analysis framework for identifying vulnerable operations in error tolerant GPU applications. A large class of other applications, e.g., mission-critical applications, do not tolerate any error in the output. For such applications, the system needs to guarantee correct operation under all conditions. For such applications, hardware monitoring mechanisms could be used to to identify an aggressive, but safe operating point. Alternatively, hardware error resilience mechanisms can be used to guarantee that no hardware error is exposed to applications.

The IC design flow will use software adaptability and error resilience for relaxed implementation and manufacturing constraints. Instead of crash-and-recover from errors, the UnO machines make proactive measurements and predict parametric and functional deviations to ensure continued operations and availability. This will preempt impact on software applications, rather than just reacting to failures (as is the case in fault-tolerant computing) or under-delivering along power/performance/reliability axes. Figure 10 shows the UnO adaptation scheme. Underdesigned hardware may be acceptable for appropriate software applications (i.e., the ones that degrade gracefully in presence of errors or are made robust). In other cases, software adaptation may be aided by hardware signatures (i.e., measured hardware characteristics). There are several ways to implement such adaptation ranging from software-guided hardware power management to just-in-time recompilation strategies. Some examples are described later in section V. For hardware design flow, the objective is to develop efficient design and test methods, given that the goal is no longer to optimize yield for fixed specifications, but rather to ensure that designs exhibit well-behaved variability characteristics that a well-configured software stack can easily exploit.

IV. Sensing and Exposing Hardware Signatures

A hardware signature provides a way of capturing one or more hardware characteristics of interest and transmitting these characteristics to the software at a given point in time. Examples of hardware signatures includes performance, power, temperature, error rate, delay fluctuations and working memory size.

Hardware signatures may be collected at various levels of spatial and temporal granularities depending on hardware and software infrastructure available to collect them and their targeted use in the UnO context. UnO machines expose these signatures to opportunistic software via architectural assists to dynamically recognize and use interesting behaviors. Key challenges in realizing this vision are:

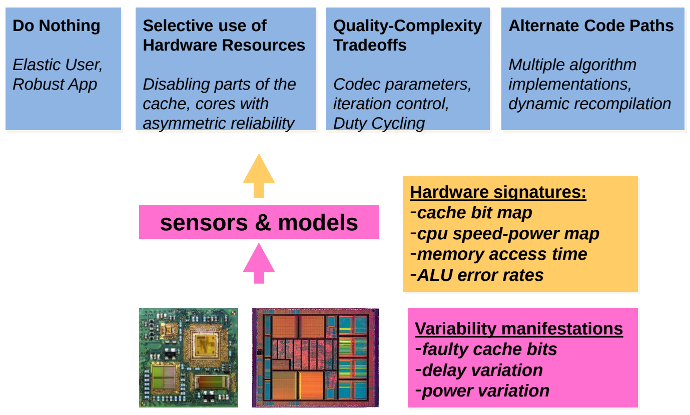
Do Nothing
Elastic User, Robust App
Selective use of Hardware Resources
Disabling parts of the cache, cores with asymmetric reliability
Quality-Complexity Tradeoffs
Codec parameters, iteration control, Duty Cycling
Alternate Code Paths
Multiple algorithm implementations, dynamic recompilation
sensors & models
Hardware signatures:
-cache bit map
-cpu speed-power map
-memory access time
-ALU error rates
Variability manifestations
-faulty cache bits
-delay variation
-power variation

Fig. 10. Examples of UnO adaptaion aided by embedded hardware monitoring.

1) Identification of hardware characteristics of interest and derivation of corresponding signatures that can be effectively sensed and utilized by the architecture and software layers.
2) Techniques (hardware, software, or combinations thereof) to enable effective sensing.
3) Design of the overall platform to support a signature-driven adaptive software stack.

A. Production Test of UnO Machines

Signatures can be collected at various points in time, for example during production test, boot-time, periodically during runtime, or adaptively based on environmental conditions (voltage, temperature), workload, and actions triggered by other sensing actions. While runtime sensing is central to UnO, it is supported by production-time sensing that goes beyond traditional production testing methods that generally focus on screening defective and/or binning manufactured parts according to their power / performance characteristics. Some of the challenges associated with production-time sensing are listed below:

1) Today, performance / power binning is generally limited to high-end processors. For complex SoCs with a large mix of heterogeneous components (including a significant proportion of non-processor components), it is difficult to create comprehensive tests to achieve high-confidence binning. In contrast, we expect UnO machines to support a large number of highly granular bins. This requires innovations in production testing techniques.
2) A wide range of parameters beyond traditional chip-level power and performance may need to be measured and calibrated for UnO machines . Examples include the noise margins and erroneous behaviors of memory cells and sequential elements, error rates and degradation behaviors of various design blocks and canary circuits under voltage / frequency over-scaling or voltage / temperature stress. behavior of various functional blocks and their error / degradation behaviors. With conventional production test methodologies, it may be highly difficult to accomplish these tasks cost-effectively.
3) Today's production testing methodologies typically don't factor in end-user application intent (some exceptions exist). Factoring in application intent can improve manufacturing yield since several chips give acceptable application behavior despite being defective (e.g., see [18]). Establishing this defect to application behavior mapping, especially in case of programmable, general purpose computing systems is highly difficult. Factoring in application behaviors during production testing may require complex fault diagnosis (which can result in expensive test time) and/or extensive use of functional tests. Functional test generation is generally considered to be "black art" and difficult to automate.

B. Runtime Sensing Methods

Techniques for collecting signatures during system runtime can be classified into the following four categories:

1) *Monitors*. By monitors, we refer to structures that are generally decoupled from the functional design (but can be aware of it). Such structures are inserted (in a sparse manner) to capture hardware characteristics of interest. Several examples exist in literature to monitor various circuit characteristics (e.g., achievable frequency [23], leakage power [24] or aging [25]). The challenge is to minimize the overheads introduced by the monitoring circuits; these overheads take the form of area, delay, and/or power, as well as design complexity while retaining accuracy.
2) *Error Detection Circuits and In-situ Sensors*. Circuit delay variation can be characterized using a variety of delay fault detection flip-flop designs (provided timing-critical paths are exercised) [26]–[30]. These approaches can be used either for concurrent error detection (i.e., detect a problem after errors appear in system data and systems) or for circuit failure prediction (i.e., provide early indications of impending circuit failures before errors appear in system data and states) [27]. Depending on the final objective, such error detection circuits may be employed in a variety of ways:

   Simply keep them activated all the time. Activate them periodically. Activate them based on operating conditions or signatures collected by (simpler) monitors.
3) *Online Self-Test and Diagnostics*. On-line self-test and diagnostics technques allow a system to test itself concurrently during normal operation without any downtime visible to the end user. Major technology trends such as the emergence of many-core SoC platforms with lots of cores and accelerators, the availability of inexpensive off-chip non-volatile storage, and the wide-spread use of test compression techniques for low-cost manufacturing test create several new

opportunities for on-line self-test and diagnostics that overcome the limitations of traditional pseudo-random Logic Built-In Self-Test (Logic BIST) techniques [31]. In addition to hardware support, effective on-line self-test diagnostics must be orchestrated using software layers such as virtualization [32] and the operating system [33].

4) *Software-Based Inference*. A compiler can use the functional specification to automatically insert ()in each component) software-implemented test operations to dynamically probe and determine which parts are working. Such test operations may be used, for instance, to determine working parts of the instruction set architecture (ISA) for a given processor core. Software-implemented test operations may be instrumented using a variety of techniques: special test sequences with known inputs and outputs, self-checking tests, and quick error detection tests [34]–[41].

One area of recent work in monitors that could be employed in UnO machines are aging sensors that estimate or measure the degree of aging (for different mechanisms, such as negative bias temperature instability, NBTI, or oxide degradation for example) that a particular chip has experienced. One design was presented in [25], where a unified NBTI and gate-oxide wear-out sensor was demonstrated in a 45nm CMOS technology. With a very small area, sprinkling hundreds or even thousands of such sensors throughout a design becomes feasible, providing a fairly reliable indication of system expected lifetime.

Alternatively, in situ sensors directly measure the hardware used in the chip itself. Such designs tend to incur relatively higher overhead in area, performance, or some other metric. One recent in situ design [42] re-uses header devices aimed at leakage reduction to evaluate the nature of leakage in a circuit block. Area overhead in this case can be kept in the 5% range, though such checks must be scheduled (this can be done without affecting user performance).

Since each method of runtime sensing inevitably makes a fundamental trade-off between cost, accuracy and applicability across various stages of system lifetime (see Table II), it is necessary to combine these approaches to meet the area/power/test time/accuracy constraints. Intelligent algorithms drawing from recent work in information theory, compressive sensing (e.g., see [43]), and similar fields can be used to adjust sampling rates based on prior observations, detect data outliers, and otherwise exploit the vast information being provided by the runtime sensing techniques. In addition, to build models of hardware variability that can be used by software layers (for example to decide which code versions to make available), randomness and dependencies must be captured, which has initially been approached using simplified probabilistic models [44]. As mentioned earlier, collective use of many sensors brings up questions of the appropriate monitoring granularity (i.e., what to test when). Temporal granularity is determined by time dependence of wear-out mechanisms, timescales of ambient/operating variability and initial design-time guardband. Statistical system adaptation techniques must be able to take advantage of fine granularity. This establishes a close tie between system design and software adaptability versus test procedures during manufacturing, and both must be co-optimized for the best result (e.g., examples shown in [45]). Frequently measuring high precision signatures (speed, power, error rate) at a large number of points in a circuit incurs large overheads. Preliminary work [21] shows that for discrete algorithm configurations or code versions, it is possible to derive optimal quantization strategies (e.g., what exact frequencies to test). Spatial granularity is dependent on the magnitude of within-die process variations, heterogeneity in design (e.g., use of multiple device types necessitating multiple measurements) and opportunities of leveraging heterogeneity/deviations in the ISA (e.g., LOAD consuming more power while ADD consuming less than expected).

Providing architectural and system support for test and monitoring is equally important. For instance, CASP [31] integrates microarchitecture support, as well as software-assisted techniques utilizing virtual machine monitors and operating systems, to efficiently orchestrate on-line self-test and diagnostics at the system level. CASP provides efficient on-line test access mechanisms to fetch high-quality test patterns from off-chip storage and apply them to various components in a SoC in four phases (see Figure 11).

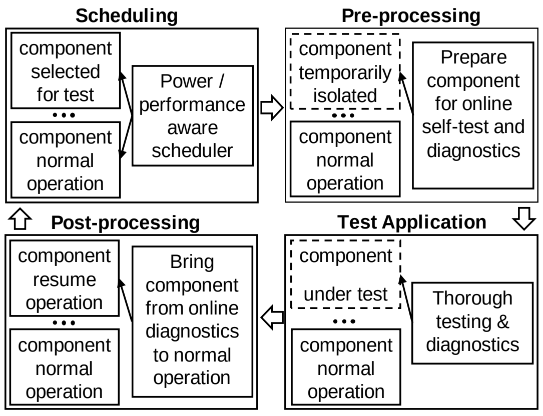
Scheduling
component selected for test
component normal operation
Power / performance aware scheduler
Pre-processing
component temporarily isolated
component normal operation
Prepare component for online self-test and diagnostics
Post-processing
component resume operation
component normal operation
Bring component from online diagnostics to normal operation
Test Application
component under test
component normal operation
Thorough testing & diagnostics

Fig. 11. The four phases of CASP operation which include software orchestration to minimize the performance impact of self-test.

A simple technique that stalls the component-under-test can result in significant system performance degradation or even visible system unresponsiveness [46]. An example software optimization is CASP-aware OS scheduling [33]. We modified the scheduler of a Linux operating system to consider the unavailability of cores undergoing on-line self-test and diagnostics. Experimental results on a dual quad-core Xeon system show that the performance impact of CASP on non-interactive applications in the PARSEC benchmark suite is negligible (e.g., $< 1\%$) when spare cores are available.

V. Variability-Aware Software

Variability has been typically addressed by process, device and circuit designers with software designers remaining isolated from it by a rigid hardware-software interface, which leads to decreased chip yields and increased costs [2]. Recently

	Monitors	Error indicators	On-line self-test and diagnostics	Software inferences
Hardware costs	Low	Generally high	Medium	Low
Accuracy	Low	High	High	Low
Coverage	Low	High	High	Low
Hardware changes	Required	Generally required	Generally required	Minimal
Diagnosability	Low	Medium to low	High	Low
On-line operation	Yes	Yes	Partial with system support	Yes with multi-programming
Applicability (failure mechanisms addressed)	Distributed effects Localized events not covered.	General as long as errors are created.	Transient errors cannot be detected.	General but diagnosis of problems is difficult

TABLE II
COMPARING DIFFERENT HARDWARE SIGNATURE GENERATION METHODS.

there have been some efforts to handle variability at higher layers of abstraction. For instance, software schemes have been used to address voltage [47] or temperature variability [48]. Hardware "signatures" are used to guide adaptation in quality-sensitive multimedia applications in [49]. In embedded sensing, [50], [51] propose sensor node deployment schemes based on the variability in power across nodes.

A software stack that changes its functions to exploit and adapt to runtime variations can operate the hardware platform close to its operational limits. The key to this achievement is to understand the information exchanges that need to take place, and the design choices that exist for the responses to variability, where they occur (i.e., which layer of software), and when they occur (design or run time).

The hardware variability may be visible to the software in several ways: changes in the availability of modules in the platform (e.g. a processor core not being functional); changes in module speed or energy performance (e.g., the maximum feasible frequency for a processor core becoming lower or its energy efficiency degraded); and changes in error rate of modules (e.g. an ALU computing wrong results for a higher fraction of inputs). The range of possible responses that the software can make is rich: alter the computational load (e.g., throttle the data rate to match what the platform can sustain); use a different set of hardware resources (e.g. use instructions that avoid a faulty module or minimize use of a power hungry module); change the algorithm (e.g., switch to an algorithm that is more resilient to computational errors); and, change hardware's operational setting (e.g., tune software-controllable control knobs such as voltage/frequency).

These changes can occur at different places in the software stack with corresponding trade-offs in agility, flexibility, and effectiveness: explicitly coded by the application developer making use of special language features and API frameworks; automatically synthesized during static or just-in-time compilation given information about application requirements and platform variability; and transparently managed by the operating system resource scheduler.

A. Selective use of Hardware Resources

One strategy for coping with variability is to selectively choose units to perform a task from a pool of available hardware. This can happen at different granularities where variability may manifest itself across different cores in a multicore processor, different memory banks, or different servers in a data center. We briefly describe a few examples below.

Large fraction of all integrated circuit failures are related to thermal issues [52], [53]. A number of strategies are used to try to minimize the effect of thermal variability on general purpose designs, starting from on-chip, to within a single server and finally at the large system level such as within datacenters. The cost of cooling can reach up to 60% of datacenter running costs [54]. The problem is further complicated by the fact that software layers are designed to be completely unaware of hardware process and thermal variability. [55] studied the effect of temperature variability on Mean Time To Failure (MTTF) of general purpose processors. It compared the default Linux scheduler with various thermal and power management techniques implemented at the OS and hardware levels of a 16 core high end general purpose processor. The difference in MTTF between default case and the best performing example (including different dynamic power management schemes) is as high as 73%. With relatively minor changes to the operating system layer, the MTTF of devices can more than double at lower energy cost by effectively leveraging hardware characteristics.

In [19], the authors present an Error Resilient System Architecture (ERSA), a robust system architecture which targets emerging applications such as recognition, mining, and synthesis (RMS) with inherent error resilience, and ensures high degree of resilience at low cost. While resilience of RMS applications to errors in low-order bits of data is well-known, execution of such applications on error-prone hardware significantly degrades output quality (due to high-order bit errors and crashes). ERSA avoids these pitfalls using a judicious combination of the following key ideas: (1) asymmetric reliability in many-core architectures: ERSA consists of small number of highly reliable components, together with a large number of components that are less reliable but account for most of the computation capacity. Within an application, by assigning the control-related code to reliable components and the computation intensive code to relaxed reliability components, it is possible to avoid highly conservative overall system design. (2) error- resilient algorithms at the core of probabilistic applications: Instead of traditional concurrent error checks typically used in fault-tolerant computing, ERSA relies on lightweight error checks such as timeouts, illegal memory access checks, and application-aware error compensation using the concepts of convergence filtering and convergence damping. (3) intelligent software optimizations: Error injection experiments on a multicore ERSA hardware prototype demonstrate that, even at very high error rates of 20 errors/flip-flop/10^8 cycles (equivalent to 25,000 errors/core/s), ERSA maintains 90% or better accuracy of output results, together with minimal impact on execution time, for probabilistic applications such as K-Means clustering, LDPC decoding, and Bayesian network inference. We also demonstrate the effectiveness of ERSA in tolerating high rates of static memory errors that are characteristic of emerging challenges related to SRAM Vccmin problems and erratic bit errors.

In addition to processors, variation is also found in memory subsystems. Experiments have shown that random access memory chips are also subject to variations in power consumption. [11] found up to 21.8% power variability across a series of 1GB DIMMs and up to 16.4% power variation across 1GB DIMMs belonging to the same vendor. Variation in memory power consumption, error, and latency characteristics can be handled by allowing the run time system to select different application memory layout, initialization, and allocation strategies for different application requirements. An architecture for hardware-assisted Variability-aware Memory Virtualization (VaMV) that allows programmers to exploit application-level semantic information [56] by annotating data structures and partitioning their address space into regions with different power, performance, and fault-tolerance guarantees (e.g., map look-up tables into low-power fault-tolerant space or pixel data in low-power non-fault-tolerant space) was explored in [57]. The VaMV memory supervisor allocates memory based on priority, application annotations, device signatures based on the memory subsystem characteristics (e.g., power consumption), and current memory usage. Experimental results on embedded benchmarks show that VaMV is capable of reducing dynamic power consumption by 63% on average while reducing total execution time by an average of 34% by exploiting: (i) SRAM voltage scaling, (ii) DRAM power variability, and (iii) Efficient dynamic policy-driven variability-aware memory allocation.

B. *Software Adaptations for Quality-Complexity Tradeoffs*

Application parameters can be dynamically adapted to explore energy, quality, and performance tradeoffs [58]. For example, Green [59] provides a software adaptation modality where the programmer provides "breakable" loops and a function to evaluate quality of service for a given number of iterations. The system uses a calibration phase to make approximation decisions based on the quality of service requirements specified by the programmer. At runtime, the system periodically monitors quality of service and adapts the approximation decisions as needed. Such adaptations can be used to maximally leverage underlying hardware platform in presence of variations.

In the context of multimedia applications, [21] demonstrated how by adapting application parameters to the post

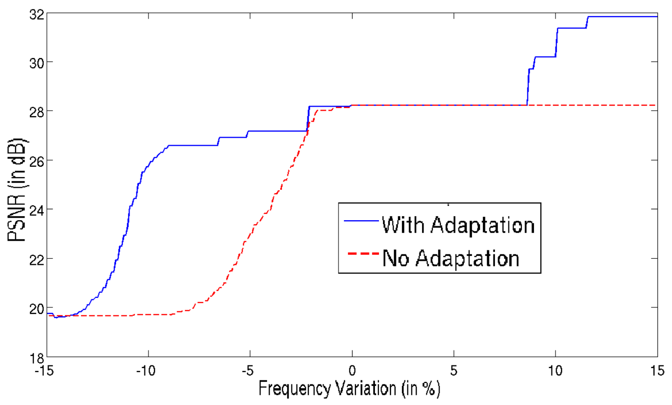

Fig. 12. Hardware guided adaptation improves PSNR (for samples of video sequences encoded using adaptive and non adaptive meth- ods, please see http://nanocad.ee.ucla.edu/Main/Codesign).

manufacturing hardware characteristics across different die, it is possible to compensate for application quality losses that might otherwise be significant in presence of process variations. This adaptation in turn results in improved manufacturing yield, relaxed requirement for hardware overdesign and better application quality. Figure 12 shows that quality-variation tradeoff can differ significantly for different hardware instances of a video encoder. In H.264 encoding, adapting parameters such as sub-pixel motion estimation, FFT transform window size, run length encoding mechanism, and size of motion estimation search window can lead to significant yield improvements (as much as 40% points at 0% over-design), a reduction in over-design (by as much as 10% points at 80% yield) as well as application quality improvements (about 2.6dB increase in average PSNR at 80% yield). Though this approach follows the error-free UnO machine model, exploting errors also as a quality tuning axis has been explored in [60].

In context of erroneous UnO machines, several alternatives exist ranging from detecting and then correcting faults within the application as they arise (e.g., [61], [62]) to designing applications to be inherently error-tolerant (e.g., application robustification [63]). In addition, many applications have inherent algorithmic and cognitive error tolerance. For such applications, significant performance and energy benefits can be obtained by selectively allowing errors.

C. *Alternate Code Paths*

One broad class of approaches for coping with hardware variability is for the software to switch to a different code path in anticipation of or in response to a variability event. Petabricks [64] and Eon [65] feature language extensions that allow programmers to provide alternate code paths. In Petabricks, the compiler automatically generates and profiles combinations of these paths for different quality/performance points. In Eon, the runtime system dynamically choses paths based on energy availability. Levels is an energy-aware programming abstraction for embedded sensors based on alternative tasks [66]. Programmers define task levels, which provide identical functionality with different quality of service and energy usage characteristics. The run-time system chooses the highest task levels that will meet the required battery lifetime.

While Petabricks, Eon, and Levels do not consider hardware variability, similar mechanisms for expressing application elasticity can be leveraged in a variability-aware software system. For example, the application can read the current hardware signature, or register interest in receiving notifications or exceptions when a specific type or magnitude of changes occur in that signature, as illustrated in scenarios 1 and 2 of Figure 13. Application response to variability events could be structured as transitions between different run-levels, with code-blocks being activated or deactivated as a result of transitions.

Algorithms and libraries with multiple implementations can be matched to underlying hardware configuration to deliver the best performance [67], [68]. Such libraries can be leveraged to choose the algorithm that best tolerates the predicted hardware variation and deliver a performance satisfying the quality-of-service requirement. With support from the OS, the switch to alternative algorithms may be done in an application-transparent fashion by relinking a different implementation of a standard library function.

D. Adjusting Hardware Operating Point

Variation-aware adjustment of hardware operating point, whether in context of adaptive circuits (e.g., [69]–[71]), adaptive microarchitectures (e.g., [28], [72]–[74]) or software-assisted hardware power management (e.g., [5], [75], [76]) has been explored extensively in literature. UnO software stack will be cognizant of actuation mechanisms available in hardware and utilize them in an application-aware manner.

E. An Example UnO Software Stack for Embedded Sensing

Throttling the workload is a common strategy to achieve system lifetime objectives in battery powered systems. A particularly common technique is duty cycling, where the system is by default in a sleep state and is woken up periodically to attend to pending tasks and events. A higher duty cycle rate typically translates into higher quality of service and a typical application-level goal is to maximize quality of data through higher duty cycles, while meeting a lifetime goal. Duty cycling is particularly sensitive to variations in sleep power at low duty cycling ratios. Variability implies that any fixed, network-wide choice of duty cycle ratio that is selected to ensure desired lifetime needs to be overly conservative and result in lower quality of sensing or lifetime.

In order to maximize the sensing quality in the presence of power variation, an opportunistic sensing software stack can help discover and adapt the application duty cycle ratio to power variations across parts and over time. The run-time system for the opportunistic stack has to keep track of changes in hardware characteristics and provide this information through interfaces accessible to either the system or the applications. Figure 13 shows several different ways such an opportunistic stack may be organized; the scenarios shown differ in how the sense-and-adapt functionality is split

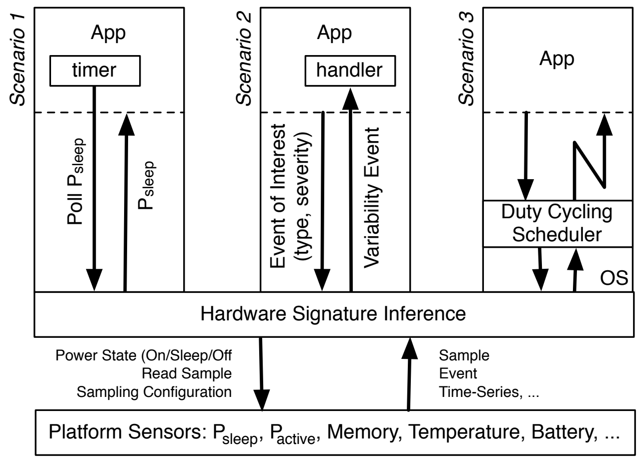

Fig. 13. Designing a software stack for variability-aware duty cycling

between applications and the operating system. Scenario 1 relies on the application polling the hardware for its current "signature". In the second scenario, the application handles variability events generated by the operating system. In the last scenario, handling of variability is largely offloaded to the operating system.

Using architecture similar to that of scenario 3 in Figure 13, [7] explored a *variability-aware duty cycle scheduler* where application modules specify a range of acceptable duty cycling ratios, and the scheduler selects the actual duty cycle based on run-time monitoring of operational parameters, and a power-temperature model that is learned off-line for the specific processor instance. Two programming abstractions are provided: tasks with variable iterations and tasks with variable period. For the first, the programmer provides a function that can be invoked repeatedly a bounded number of times within each fixed period. For the second, the application programmer provides a function that is invoked once within each variable but bounded period of time. The system adjusts the number of iterations or period of each task based on the allowable duty cycle. The scheduler determines an allowable duty cycle based on: (i) sleep and active power vs. temperature curves for each instance, (ii) temperature profile for the application, which can be pre-characterized or learned dynamically, (iii) lifetime requirement, and (iv) battery capacity.

In an evaluation with ten instances of Atmel SAM3U processors, [7] found that variability-aware duty cycling yields a 3–22x improvement in total active time over schedules based on worst-case estimations of power, with an average improvement of 6.4x across a wide variety of deployment scenarios based on collected temperature traces. Conversely, datasheet power specifications fail to meet required lifetimes by 7–15%, with an average 37 days short of a required lifetime of one year. Finally, a target localization application using variability-aware duty cycle yields a 50% improvement in quality of results over one based on worst-case estimations of power consumption.

With current technology characteristics, a variability-aware schedule is beneficial for smaller duty cycles ($< 5\%$). With the expected projection of sleep and active power variation with scaling of technology, there will be significant benefits of a variability-aware schedule even for higher duty cycles. Some classes of real-time, highly synchronized embedded sensing applications are not amenable to this type of adaptation. Furthermore, this scheme adds some complexity to the application, in the form of bounds to task activations, which may in turn lead to further complexities in data storage, inference and communication strategies. Nevertheless, the benefits of variability-aware duty cycling outweigh the added complexity for a large class of sensing applications. While this adaptation scheme indirectly leads to a form of energy-based load balancing across a network of sensors, further opportunities for network-wide adaptation exist in role selection for nodes, where a node could take different roles (e.g. data collector, router, aggregator) depending on its respective energy rank in the network.

F. Variability Simulation Challenges

Due to the statistical nature of variability, it is important to test variability-aware software techniques on large number of hardware platforms, which, unfortunately is impractical in most cases. It is, therefore, essential to develop simulation and emulation infrastructures for UnO computing systems which are scalable while retaining variability modeling accuracy. The work in this direction has been somewhat limited (e.g., see [77] in context of performance variability for a specific system or error emulation in [19], [63]) and a key challenge for UnO software methods is building such emulation platforms for large class of computing systems which model all manifestations (error, delay, power, aging, etc) of variability.

VI. Implications for Circuits and Architectures

Software on UnO machines can react to hardware signature of a resource by either changing its utilization profile or changing the hardwares operating point. In addition, the computing platform can be implemented just to report the hardware signature (with or without self-healing) or it may allow (software-driven) adaptation. The power/area cost will depend on the desired spatial granularity of healing/adaptation (i.e., instruction-level, component-level and platform-level). Since the mechanisms for such healing or adaptation (e.g., shutdown using power gating; power/performance change using voltage scaling, etc.) are fairly well understood, future research focus should be on optimizing application-dependent tradeoffs and reducing hardware implementation overheads.

The embedded sensing and H.264 encoder examples discussed in the next section fall into the category of parametrically underdesigned, error-free UnO systems. Certain classes of applications are inherently capable of absorbing some amount of errors in computations, which allows for quality to be traded off for power. Errors in hardware (which are circuit-observable but system-insignificant, such as errors in speculation units of a processor [78]) can be permitted to improve other design metrics such as power, area and delay. Error probability can be traded-off with design metrics by adjusting verification thresholds, or selective guardbanding, or selective redundancy, etc. Recent work ("stochastic computing" [79]) advocates that hardware should be allowed to produce errors even during nominal operation if such errors can be tolerated by software. Similarly, software should designed to make forward progress in spite of the errors produced by hardware. The resulting system will potentially be significantly more power efficient as hardware can now be under-designed exactly to meet the software reliability needs. Design methodologies that are aware of the error resilience mechanisms used during the design process can result in significantly lower power processor designs (e.g., [80]).

Unfortunately, current design methodologies are not always suitable for such underdesign. For example, conventional processors are optimized such that all the timing paths are critical or near-critical ("timing slack wall"). This means that any time an attempt is made to reduce power by trading off reliability (by reducing voltage, for example), a catastrophically large number of timing errors is seen [20]. The slack distribution can be manipulated (for example, to make it look gradual, instead of looking like a wall (Figure 14) to reduce the number of errors when power is reduced [81]. Furthermore, the frequently exercised components (or timing paths) that contribute the most to the error rate can be optimized at the expense of infrequently exercised components to perform the slack redistribution without an overhead. The resulting processor will produce only a small number of errors for large power savings. Also, the error rate will increase gradually with increased power savings [80]. Hardware can be explicitly designed for a target error rate. The errors are either detected and corrected by a hardware error resilience mechanism or allowed to propagate to an error tolerant application where the errors manifest themselves as reduced performance or output quality.

Corresponding opportunities exist at the microarchitectural level. Architectural design decisions have always been, and still are made in the context of correctness. Until now, error resilience has been viewed as a way to increase the efficiency of a design that has been architected for correctness but operates outside the bounds of the correctness guarantee. However, optimizing a design for correctness can result in significant inefficiency when the actual intent is to operate the design at a non-zero error rate and allow errors to be tolerated or corrected by the error resilience mechanism [82]. In other words, one would make different, sometimes counterintuitive, architectural design choices when optimizing a processor specifically for error resilience than when optimizing for correctness [83]. This is not surpising considering that many microarchitectural decisions such as increasing the size of caches and register files, etc., tend to modify the slack distribution of the processor to look more like a wall. Similarly, microarchitectural decisions such as increasing the size of instruction and load-store queues worsen the delay scalability of the processor. When the processors are optimized for non-zero error rates, such decisions need to be avoided (Figure 15).

Though, most existing work [84]–[86] introduces errors by intelligently over-scaling voltage supplies, there has been some work in the direction of introducing error into a system via manipulation of its logic-function, for adders [87], [88] as well as generic combinational logic [17]. Let

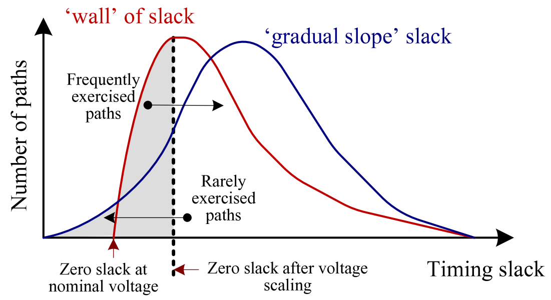
'wall' of slack
'gradual slope' slack
Number of paths
Frequently exercised paths
Rarely exercised paths
Zero slack at nominal voltage
Zero slack after voltage scaling
Timing slack

Fig. 14. The goal of a gradual slack, soft processor [20] design is to transform a slack distribution characterized by a critical "wall" into one with a more gradual failure characteristic. This allows performance / power tradeoffs over a range of error rates, whereas conventional designs are optimized for correct operation and recovery-driven designs, are optimized for a specific target error rate.

us consider an example functionally underdesigned integer multiplier circuit [15] with potentially erroneous operation. We use a modified 2x2 multiplier as a building block which computes 3×3 as 7 instead of 9. By representing the output using three bits (111) instead of the usual four (1001), we are able to significantly reduce the complexity of the circuit. These inaccurate multipliers achieve an average power saving of 31.78% - 45.4% over corresponding accurate multiplier designs, for an average percentage error of 1.39% - 3.32%. The design can be enhanced (albeit at power cost) to allow for correct operation of the multiplier using a correction unit, for non error-resilient applications which share the hardware resource. Of course the benefits are strongly design-dependent. For instance, in the multiplier case, the savings translate to 18% saving for a FIR filter and only 1.5% for a small embedded microprocessor. Functional underdesign will be a useful power reduction and/or performance improvement knob especially for robust and gracefully degrading applications.

Much research needs to be done to functionally or parametrically underdesign large general class of circuits automatically. Mechanisms to pass application intent to physical implementation flows (especially to logic synthesis in case of functional underdesign) need to be developed. Formally quantifying impact of hardware errors on applications is important (e.g., [89] computes the fundamental limits of accuracy of LDPC error control decoders due to noise in transmission as well as underdesigned hardware units). Error steering toward less important (depending on the application) parts of ISA during design or at runtime can be done with ISA extensions specifying latency and reliability constraints to the hardware. Annotations indicating access patterns, reliability requirements [90], etc. of code segments as well as data can be used effectively especially in case of heterogeneous circuits and systems (e.g., [19]).

An adaptive software stack cognizant of the exact hardware state enables a fluid specification for UnO computing platforms so that meeting a rigid specification becomes less important than selecting good power/performance/area/reliability operating points. Using an adaptable software stack – and knowing the bounds of this adaptability – we can relax the notion of harsh constraints for design of computing systems

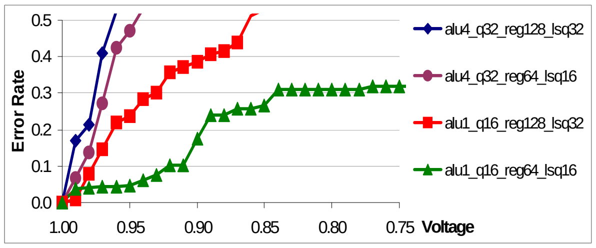

Fig. 15. Different microarchitectures exhibit different error rate behaviors, demonstrating the potential to influence the energy efficiency of a timing speculative architecture through microarchitectural optimizations. (Notations describe the number of ALUs, issue queue length, number of physical registers, and load store queue size for a microarchitecture.)

and alleviate the "last-MHz problem" that results in a very high cost to achieve the marginally high performance. Similarly, an application-aware fluid test specification can lead to large gains in parametric manufacturing yield. For instance in our recent simulation study [21], we show 30% improvement in hardware yield or equivalently 8% reduction in overdesign for simple H.264 video encoder by leveraging a software application that adapts to underlying hardware.

VII. Conclusions

At its core, handling variability of hardware specification at run-time amounts to detecting the variability using hardware or software based sensing mechanisms that we discussed earlier, followed by selecting a different execution strategy in the UnO machine's hardware or software or both. Though the focus in UnO systems is adaptation in the software layer, we realize that flexibility, agility, accuracy and overhead requirements may dictate choice of layer (in the hardware-software stack) where adaptation happens for different variation sources and manifestations. To realize the UnO vision, several challenges exist:

- What are the most effective ways to detect the nature and extent of various forms of hardware variations?
- What are the important software-visible manifestations of these hardware variations, and what are appropriate abstractions for representing the variations?
- What software mechanisms can be used to opportunistically exploit variability in hardware performance and reliability, and how should these mechanisms be distributed across application, compiler-generated code, and run-time operating system services?
- How can hardware designers and design tools leverage application characteristics and opportunistic software stack?
- What is the best way of verifying and testing hardware when there are no strict constraints at the hardware-software boundary and the hardware and application behaviors are no longer constant?

Our early results illustrating UnO hardware-software stack are very promising. For instance, our implementation of variability-aware dutycycling in TinyOS gave over 6X average improvement in active time for embedded sensing. With shrinking dimensions approaching physical limits of semiconductor processing, this variability and resulting benefits from UnO computing are likely to increase in future. Such a vision of a redefined – and likely flexible/adaptive – hardware/software interface presents us with an opportunity to substantially improve the energy efficiency, performance and cost of computing systems.

Acknowledgement

The focus of the National Science Foundation Computing Expedition on "Variability-Aware Software for Efficient Computing with Nanoscale Devices" http://www.variability.org is to address these challenges and develop prototypical UnO computing machines. Some of the work was supported by the NSF Variability Expedition award numbers CCF-1029030, CCF-1028831, CCF-1028888, CCF-1029025, CCF-1029783.

References

[1] S. Jones, "Exponential trends in the integrated circuit industry," 2008.
[2] K. Jeong, A. B. Kahng, and K. Samadi, "Impacts of guardband reduction on design process outcomes: A quantitative approach," *IEEE Transactions on Semiconductor Manufacturing*, vol. 22, no. 4, 2009.
[3] ITRS, "The international technology roadmap for semiconductors," http://public.itrs.net/.
[4] L. Grupp, A. M. Caulfield, J. Coburn, E. Yaakobi, S. Swanson, and P. Siegel, "Characterizing Flash Memory: Anomalies, Observations, and Applications," in *IEEE/ACM MICRO*, 2009.
[5] 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 core mapping and thread hopping for an 80-core processor," in *IEEE ISSCC*, 2010.
[6] R. W. Johnson *et al.*, "The Changing Automotive Environment: High Temperature Electronics," *IEEE Transactions on Electronics Packaging Manufacturing*, vol. 27, no. 3, 2004.
[7] L. Wanner, C. Apte, R. Balani, P. Gupta, and M. Srivastava, "Hardware variability-aware duty cycling for embedded sensors," *IEEE Transactions on VLSI*, 2012.
[8] H. Hanson, K. Rajamani, J. Rubio, S. Ghiasi, and F. Rawson, "Benchmarking for Power and Performance," in *SPEC Benchmarking Workshop*, 2007.
[9] R. Zheng, J. Velamala, V. Reddy, V. Balakrishnan, E. Mintarno, S. Mitra, S. Krishnan, and Y. Cao, "Circuit aging prediction for low-power operation," in *IEEE CICC*, 2009.
[10] J. McCullough, Y. Agarwal, J. Chandrashekhar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta, "Evaluating the effectiveness of model-based power characterization," in *USENIX Annual Technical Conference*, 2011.
[11] M. Gottscho, A. Kagalwalla, and P. Gupta, "Power variability in contemporary drams," *IEEE Embedded System Letters*, 2012 (to appear).
[12] L. Bathen, M. Gottscho, N. Dutt, P. Gupta, and A. Nicolau, "Vipzone: Os-level memory variability-aware physical address zoning for energy savings," in *IEEE/ACM CODES+ISSS*, 2012.
[13] T. Austin, D. Blaauw, T. Mudge, and K. Flautner, *Making Typical Silicon Matter with Razor*. IEEE Computer, Mar. 2004.
[14] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar, "Scalable effort hardware design: exploiting algorithmic resilience for energy efficiency," in *IEEE/ACM DAC*, 2010.
[15] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power in a multiplier architecture," *Journal of Low Power Electronics*, 2011.
[16] M. R. Choudhury and K. Mohanram, "Approximate logic circuits for low overhead, non-intrusive concurrent error detection," in *IEEE/ACM DATE*, 2008.
[17] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *IEEE/ACM DATE*, 2010.
[18] M. Breuer, S. Gupta, and T. Mak, "Defect and error tolerance in the presence of massive numbers of defects," *Design Test of Computers, IEEE*, vol. 21, no. 3, 2004.
[19] H. Cho, L. Leem, and S. Mitra, "Ersa: Error resilient system architecture for probabilistic applications," *IEEE Transactions on CAD of IC & S*, vol. 31, no. 4, 2012.
[20] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Designing processors from the ground up to allow voltage/reliability tradeoffs," in *International Symposium on High-Performance Computer Architecture*, 2010.
[21] A. Pant, P. Gupta, and M. v.-d. Schaar, "Appadapt: Opportunistic application adaptation to compensate hardware variation," *IEEE Transactions on VLSI*, 2011.
[22] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications," *IEEE Transactions on Multimedia*, 2012.
[23] T. B. Chan, P. Gupta, A. B. Kahng, and L. Lai, "Ddro: A novel performance monitoring methodology based on design-dependent ring oscillators," in *IEEE ISQED*, 2012.
[24] C. Kim, K. Roy, S. Hsu, R. Krishnamurthy, and S. Borkar, "An on-die cmos leakage current sensor for measuring process variation in sub-90nm generations," in *IEEE VLSI Circuits Symposium*, 2004.
[25] P. Singh, E. Karl, D. Sylvester, and D. Blaauw, "Dynamic nbti management using a 45nm multi-degradation sensor," in *IEEE CICC*, 2010.
[26] P. Franco and E. McCluskey, "On-line testing of digital circuits," in *IEEE VTS*, 1994.
[27] M. Agarwal, B. Paul, M. Zhang, and S. Mitra, "Circuit failure prediction and its application to transistor aging," in *IEEE VTS*, 2007.
[28] 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 *IEEE/ACM MICRO*, 2003.
[29] K. Bowman, J. Tschanz, N. S. Kim, J. Lee, C. Wilkerson, S.-L. Lu, T. Karnik, and V. De, "Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance," *IEEE Journal of Solid State Circuits*, vol. 44, 1, 2009.
[30] M. Nicolaidis, "Time redundancy based soft-error tolerance to rescue nanometer technologies," in *IEEE VTS*, 1999.
[31] Y. Li, S. Makar, and S. Mitra, "Casp: Concurrent autonomous chip self-test using stored test patterns," in *IEEE/ACM DATE*, 2008.
[32] Y. L. Inoue, H. and S. Mitra, "Vast: Virtualization-assisted concurrent autonomous self-test," in *IEEE ITC*, 2008.
[33] Y. Li, Y. Kim, E. Mintarno, D. Gardner, and S. Mitra, "Overcoming early-life failure and aging challenges for robust system design," *IEEE DTC*, vol. 26, no. 6, 2009.
[34] K. M. P. Parvathala and W. Lindsay, "Frits: A microprocessor functional bist method," in *IEEE ITC*, 2002.
[35] J. Shen and J. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," in *IEEE ITC*, 1998.
[36] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," in *ASPLOS*, Mar. 2008.
[37] ——, "Trace-based microarchitecture-level diagnosis of permanent hardware faults," in *DSN*, Jun. 2008.
[38] S. K. Sahoo, M.-L. Li, P. Ramachandran, S. V.Adve, V. S. Adve, and Y. Zhou, "Using likely program invariants to detect hardware errors," in *DSN*, Jun. 2008.
[39] D. Lin, T. Hong, F. Fallah, N. Hakim, and S. Mitra, "Quick detection of difficult bugs for effective post-silicon validation," in *IEEE/ACM DAC*, 2012.
[40] N. Oh, S. Mitra, and E. McCluskey, "Ed^{4}i: Error detection by diverse data and duplicated instructions," *IEEE Transactions on Computers*, vol. 51, no. 2, 2002.
[41] T. Hong, "Qed: Quick error detection tests for effective post-silicon validation," in *IEEE ITC*, 2010.
[42] P. Singh, Z. Foo, M. Wieckowski, S. Hanson, M. Fojtik, D. Blaauw, and D. Sylvester, "Early detection of oxide breakdown through in situ degradation sensing," in *IEEE ISSCC*, 2010.
[43] X. Li, R. R. Rutenbar, and R. D. Blanton, "Virtual probe: a statistically optimal framework for minimum-cost silicon characterization of nanoscale integrated circuits," in *IEEE/ACM ICCAD*, 2009.
[44] M. Qazi, M. Tikekar, L. Dolecek, D. Shah, and A. Chandrakasan, "Loop flattening and spherical sampling: Highly efficient model reduction techniques for sram yield analysis," in *IEEE/ACM DATE*, 2010.
[45] E. Mintarno, E. Skaf, J. R. Zheng, J. Velamala, Y. Cao, S. Boyd, R. Dutton, and S. Mitra, "Self-tuning for maximized lifetime energy-efficiency in the presence of circuit aging," *IEEE Transactions on CAD of IC & S*, vol. 30, no. 5, 2011.
[46] Y. Li, O. Mutlu, D. Gardner, and S. Mitra, "Concurrent autonomous self-test for uncore components in system-on-chips," in *IEEE VTS*, 2010.

[47] V. J. Reddi, M. S. Gupta, M. D. Smith, G.-y. Wei, D. Brooks, and S. Campanoni, “Software-assisted hardware reliability: abstracting circuit-level challenges to the software stack,” in *IEEE/ACM DAC*. ACM, 2009.

[48] J. Choi, C.-Y. Cher, H. Franke, H. Hamann, A. Weger, and P. Bose, “Thermal-aware task scheduling at the system software level,” in *IEEE/ACM ISLPED*, 2007.

[49] A. Pant, P. Gupta, and M. van der Schaar, “Software adaptation in quality sensitive applications to deal with hardware variability,” in *Great lakes Symposium on VLSI*. ACM, 2010.

[50] T. Matsuda, T. Takeuchi, H. Yoshino, M. Ichien, S. Mikami, H. Kawaguchi, C. Ohta, and M. Yoshimoto, “A power-variation model for sensor node and the impact against life time of wireless sensor networks,” in *ICCE*, oct. 2006.

[51] S. Garg and D. Marculescu, “On the impact of manufacturing process variations on the lifetime of sensor networks,” in *IEEE/ACM CODES+ISSS*. ACM, 2007.

[52] M. Pedram and S. Nazarian, “Thermal modeling, analysis, and management in vlsi circuits: Principles and methods,” *Proceedings of the IEEE*, vol. 94, no. 8, 2006.

[53] S. Sankar, M. Shaw, and K. Vaid, “Impact of temperature on hard disk drive reliability in large datacenters,” in *DSN*, 2011.

[54] E. Kursun and C.-Y. Cher, “Temperature variation characterization and thermal management of multicore architectures,” in *IEEE/ACM MICRO*, 2009.

[55] A. Coskun, R. Strong, D. Tullsen, and T. S. Rosing, “Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors,” in *SIGMETRICS*, 2009.

[56] S. Novack, J. Hummel, and A. Nicolau, “A simple mechanism for improving the accuracy and efficiency of instruction-level disambiguation,” in *International Workshop on Languages and Compilers for Parallel Computing*, 1995.

[57] L. Bathen, N. Dutt, A. Nicolau, and P. Gupta, “Vamv: Variability-aware memory virtualization.” in *IEEE/ACM DATE*, 2012.

[58] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, “Dynamic knobs for responsive power-aware computing,” *SIGARCH Comput. Archit. News*, vol. 39, Mar. 2011.

[59] W. Baek and T. M. Chilimbi, “Green: a framework for supporting energy-conscious programming using controlled approximation,” *SIGPLAN Not.*, vol. 45, jun 2010.

[60] S. Narayanan, J. Sartori, R. Kumar, and D. Jones, “Scalable stochastic processors,” in *IEEE/ACM DATE*, 2010.

[61] R. Hegde and N. R. Shanbhag, “Energy-efficient signal processing via algorithmic noise-tolerance,” in *IEEE/ACM ISLPED*, 1999.

[62] K.-H. Huang and J. Abraham, “Algorithm-based fault tolerance for matrix operations,” *IEEE Transactions on Computers*, vol. 33, no. 6, 1984.

[63] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi, “A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance,” in *IEEE DSN*, 2010.

[64] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, “Petabricks: a language and compiler for algorithmic choice,” *SIGPLAN Not.*, vol. 44, jun 2009.

[65] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger, “Eon: a language and runtime system for perpetual systems,” in *International Conference on Embedded Networked Sensor Systems*. ACM, 2007.

[66] A. Lachenmann, P. J. Marrón, D. Minder, and K. Rothermel, “Meeting lifetime goals with energy levels,” in *International Conference on Embedded Networked Sensor Systems*. ACM, 2007.

[67] M. Frigo and S. G. Johnson, “The design and implementation of FFTW3,” *Proceedings of the IEEE*, vol. 93, no. 2, 2005, special issue on “Program Generation, Optimization, and Platform Adaptation”.

[68] X. Li, M. Garzaran, and D. Padua, “Optimizing sorting with machine learning algorithms,” in *Parallel and Distributed Processing Symposium*, 2007.

[69] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, “Parameter variations and impact on circuits and microarchitecture,” in *IEEE/ACM DAC*, 2003.

[70] 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 Transactions on CAD of IC & S*, vol. 26, no. 11, nov. 2007.

[71] A. Agarwal, B. Paul, S. Mukhopadhyay, and K. Roy, “Process variation in embedded memories: failure analysis and variation aware architecture,” *IEEE Journal of Solid State Circuits*, vol. 40, no. 9, 2005.

[72] D. Sylvester, D. Blaauw, and E. Karl, “Elastic: An adaptive self-healing architecture for unpredictable silicon,” *Design Test of Computers, IEEE*, vol. 23, no. 6, 2006.

[73] K. Meng and R. Joseph, “Process variation aware cache leakage management,” in *IEEE/ACM ISLPED*, 2006.

[74] A. Tiwari, S. R. Sarangi, and J. Torrellas, “Recycle: pipeline adaptation to tolerate process variation,” in *ISCA*, 2007.

[75] S. Chandra, K. Lahiri, A. Raghunathan, and S. Dey, “Variation-tolerant dynamic power management at the system-level,” *IEEE Transactions on VLSI*, vol. 17, no. 9, 2009.

[76] R. Teodorescu and J. Torrellas, “Variation-aware application scheduling and power management for chip multiprocessors,” in *ISCA*, 2008.

[77] V. J. Kozhikkottu, R. Venkatesan, A. Raghunathan, and S. Dey, “Vespa: Variability emulation for system-on-chip performance analysis,” in *IEEE/ACM DATE*, 2011.

[78] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor,” in *IEEE/ACM MICRO*, Dec. 2003.

[79] N. Shanbhag, R. Abdallah, R. Kumar, and D. Jones, “Stochastic computation,” in *IEEE/ACM DAC*, 2010.

[80] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, “Recovery-driven design: A methodology for power minimization for error tolerant processor modules,” in *IEEE/ACM DAC*, 2010.

[81] ——, “Slack redistribution for graceful degradation under voltage overscaling,” in *In the 15th IEEE/SIGDA Asia and South Pacific Design and Automation conference*, 2010.

[82] N. Zea, J. Sartori, and R. Kumar, “Optimal power/performance pipelining for timing error resilient processors,” in *IEEE ICCD*, 2010.

[83] J. Sartori and R. Kumar, “Exploiting timing error resilience in architecture of energy-efficient processors,” in *ACM Transactions on Embedded Computing Systems*, 2011.

[84] K. V. Palem, “Energy aware computing through probabilistic switching: A study of limits,” *IEEE Transactions on Computers*, vol. 54, no. 9, 2005.

[85] M. S. Lau, K.-V. Ling, and Y.-C. Chu, “Energy-aware probabilistic multiplier: design and analysis,” in *CASES*. ACM, 2009.

[86] J. George, B. Marr, B. E. S. Akgul, and K. V. Palem, “Probabilistic arithmetic and energy efficient embedded signal processing,” in *CASES*. ACM, 2006.

[87] D. Shin and S. K. Gupta, “A re-design technique for datapath modules in error tolerant applications,” *Asian Test Symposium*, vol. 0, 2008.

[88] D. Kelly and B. Phillips, “Arithmetic data value speculation,” in *Asia-Pacific Computer Systems Architecture Conference*, 2005.

[89] S. M. S. T. Yazdi, H. Cho, Y. Sun, S. Mitra, and L. Dolecek, “Probabilistic analysis of gallager b faulty decoder,” in *IEEE International Conference on Communications (ICC) – Emerging Data Storage Technologies*, 2012.

[90] J. Henkel *et al.*, “Design and architectures for dependable embedded systems,” in *IEEE/ACM CODES+ISSS*, 2011.