Exploiting Errors for Efficiency: A Survey from Circuits to Applications

PHILLIP STANLEY-MARBELL, University of Cambridge
ARMIN ALAGHI, University of Washington
MICHAEL CARBIN, Massachusetts Institute of Technology
EVA DARULOVA, Max Planck Institute for Software Systems
LARA DOLECEK, University of California at Los Angeles
ANDREAS GERSTLAUER, The University of Texas at Austin
GHAYOOR GILLANI, University of Twente
DJORDJE JEVDJIC, National University of Singapore
THIERRY MOREAU, University of Washington
MATTIA CACCIOTTI, École Polytechnique Fédérale de Lausanne
ALEXANDROS DAGLIS, Georgia Institute of Technology
NATALIE ENRIGHT JERGER, University of Toronto
BABAK FALSAFI, École Polytechnique Fédérale de Lausanne
SASA MISAILOVIC, University of Illinois at Urbana-Champaign
ADRIAN SAMPSON, Cornell University
DAMIEN ZUFFEREY, Max Planck Institute for Software Systems

When a computational task tolerates a relaxation of its specification or when an algorithm tolerates the effects of noise in its execution, hardware, system software, and programming language compilers or their runtime systems can trade deviations from correct behavior for lower resource usage. We present, for the first time, a synthesis of research results on computing systems that only make as many errors as their end-to-end applications can tolerate. The results span the disciplines of computer aided design of circuits, digital system design, computer architecture, programming languages, operating systems, and information theory. Rather than over-provisioning the resources controlled by each of these layers of abstraction to avoid errors, it can be more efficient to exploit the masking of errors occurring at one layer and thereby prevent those errors from propagating to a higher layer.

We demonstrate the potential benefits of end-to-end approaches using two illustrative examples. We introduce a formalization of terminology that allows us to present a coherent view across the techniques traditionally used by different research communities in their individual layer of focus. Using this formalization,

Authors' addresses: Phillip Stanley-Marbell, University of Cambridge, (phillip.stanley-marbell@eng.cam.ac.uk); Armin Alaghi, University of Washington; Michael Carbin, Massachusetts Institute of Technology; Eva Darulova, Max Planck Institute for Software Systems; Lara Dolecek, University of California at Los Angeles; Andreas Gerstlauer, The University of Texas at Austin; Ghayoor Gillani, University of Twente; Djordje Jevdjic, National University of Singapore; Thierry Moreau, University of Washington; Mattia Cacciotti, École Polytechnique Fédérale de Lausanne; Alexandros Daglis, Georgia Institute of Technology; Natalie Enright Jerger, University of Toronto; Babak Falsafi, École Polytechnique Fédérale de Lausanne; Sasa Misailovic, University of Illinois at Urbana-Champaign; Adrian Sampson, Cornell University; Damien Zufferey, Max Planck Institute for Software Systems.

0360-0300/2020/5-ART51 $15.00
https://doi.org/https://doi.org/10.1145/3394898

we survey tradeoffs for individual layers of computing systems at the circuit, architecture, operating system, and programming language levels as well as fundamental information-theoretic limits to tradeoffs between resource usage and correctness.

Additional Key Words and Phrases: Approximate computing, error efficiency, cross-layer optimization.

ACM Reference Format:
Phillip Stanley-Marbell, Armin Alaghi, Michael Carbin, Eva Darulova, Lara Dolecek, Andreas Gerstlauer, Ghayoor Gillani, Djordje Jevdjic, Thierry Moreau, Mattia Cacciotti, Alexandros Daglis, Natalie Enright Jerger, Babak Falsafi, Sasa Misailovic, Adrian Sampson, and Damien Zufferey. 2020. Exploiting Errors for Efficiency: A Survey from Circuits to Applications. *ACM Comput. Surv.* 53, 3, Article 51 (May 2020), 40 pages. https://doi.org/https://doi.org/10.1145/3394898

1 INTRODUCTION

Computing systems solve specific computational problems by transforming an algorithm's inputs to its outputs. This, as well as counteracting the effects of noise in the underlying hardware substrate [16, 105, 193], requires resources such as time, energy, or hardware real-estate. Because of the increasing pervasiveness of computing systems and the diminishing returns from performance improvements of process technology scaling [7, 23, 157], resource efficiency is becoming an increasingly important challenge.

Computing systems are reaching the fundamental limits of the energy required for fully-reliable computation [16, 133]. At the same time, many important applications have nondeterministic specifications or are robust to noise in their execution. We dedicate the next section of the review (Section 2) to providing an overview of application domains with quality versus resource usage tradeoffs and we provide two detailed examples in Section 3. They thus do not necessarily require fully-reliable computing systems and their resource consumption can be reduced. For instance, many applications processing physical-world signals often have multiple acceptable outputs for a large part of their input domain. Because all measurements of analog signals have some amount of measurement uncertainty or noise and digital signal representations necessarily introduce quantization noise, it is not always necessary to perform exact computation on data resulting from uncertain measurements of real-world physical signals.

These observations about the fundamental limits of computation and the possibility of trading correctness for resource usage have always been implicit in computing systems design dating back to the ENIAC [233], but have seen renewed interest in the last decade. This interest has focused on techniques to trade precision, accuracy, and reliability for reduced resource usage in hardware. These recent efforts harness nondeterminism and take advantage of application tolerance to coarser discretization in time or value (i.e., precision or sampling rate), to obtain significant resource savings for an acceptable reduction in accuracy and reliability. These techniques have been referred to in the research literature as *approximate computing* and include:

- Programming languages to specify computational problem and algorithm nondeterminism.
- Compilation techniques to transform specifications which expose nondeterminism or flexibility, into concrete deterministic implementations.
- Hardware architectures that can exploit nondeterminism exposed at the software layer, or which expose hardware correctness versus resource usage tradeoffs to the layers above.
- New devices and circuits to implement architectures that exploit or expose nondeterminism and correctness versus resource usage tradeoffs.

In the same way that computing systems that only use as much energy as is necessary are referred to as being *energy-efficient*, we can refer to the computing systems investigated in this survey as being *error-efficient*: they only make as many errors as their end-to-end applications can tolerate [215].

1.1 Context of this survey

This survey explores research results in hardware and software systems in which the system's designers or end-to-end applications can trade lower resource usage for increased occurrence of deviations from correctness. These deviations from correctness may occur within an individual layer of the system (e.g., at the circuit layer), or they may occur in the context of an end-to-end computing system application (e.g., a microcontroller-based pedometer application driven by sensor measurements from an accelerometer). Existing related surveys [10, 77, 145, 149, 191, 235] present valuable analyses of techniques at subsets of the layers of computing systems. These existing surveys provide complementary coverage of the relevant literature but neither introduce any broadly-applicable mathematical formalization of the research problem, nor do they survey the information-theoretic foundations of the tradeoffs between resource usage and correctness like we do in this article (Section 4 and Section 10, respectively).

This survey is the result of research ideas and discussions started at a multi-disciplinary workshop, involving the authors, held in April 2017. We wrote the survey shortly thereafter and submitted it for review in June 2018. As a result, our coverage of the research literature is skewed towards articles published prior to 2018. We have made every attempt to update the survey with relevant recent results at the time of acceptance for publication.

Any survey of an active research field can necessarily never be exhaustive. We do not explicitly cover the long history of fundamental research results on changing the structure of algorithms to trade, e.g., average case performance for deterministic execution (approximation and randomized algorithms). Instead, we limit ourselves to results that apply to compile-time program transformations of applications which implement a fixed algorithm. Throughout the survey, our objective is to highlight representative examples that provide reusable insights, rather than compile an exhaustive list of all related publications and related domains.

1.2 Contributions and outline

This survey presents:

- **A cross-disciplinary overview** of research results on correctness versus resource usage tradeoffs, spanning the hardware abstractions and disciplines of: transistors, circuits, microarchitecture and architecture, programming languages, operating systems, and applications.
- **An overview of existing uses of quality-versus- resource-usage tradeoffs** across application domains and **examples of two end-to-end applications** (Section 2 and Section 3).
- **Mathematical formalization and terminology** for describing resource usage versus correctness tradeoffs of computing systems that interact with the physical world. The formalization is consistent with existing widely-used terminology and at the same time provides a coherent way to discuss tradeoffs across domains of expertise (Section 4).
- **Detailed discussions of the state of the art** across the layers of system implementation stack, from circuits, to microarchitecture and architecture, to the programming language and operating system layers of abstraction (Section 5 – Section 8).
- **A taxonomy** tying together the ideas introduced in the survey (Section 9).
- **A discussion of limits of computation in the presence of noise**. (Section 10).
- **A set of open challenges** across the layers of abstraction (Section 11).

2 EXISTING QUALITY VERSUS RESOURCE USAGE TRADEOFFS

The idea of trading quality for resources and efficiency is inherent to all computing domains. Several research communities have developed techniques to exploit tolerance of applications to noise, errors, and approximations to improve the reliability or efficiency of software and hardware systems. In the same way that there have always been attempts to make hardware and software more tolerant to faults independent of specific research on fault-tolerant computing, there has also always been a pervasive use of techniques for approximation (e.g., Taylor series expansions) independent of recent interest in approximate computing. The following highlights some of these efforts across application domains.

2.1 Scientific computing

Scientific computing can be defined as "the collection of tools, techniques, and theories required to solve on a computer mathematical models of problems in science and engineering" [71]. Most of these models are real-valued, and exact analytical solutions rarely exist or are costly to compute [44, 134]. As a result, numerical approximations and their associated quality-efficiency tradeoffs have always been important in scientific computing [55].

These numerical approximations are introduced at different levels of abstraction. Because the real-world is too complex to be represented exactly, practical considerations require resorting to models, incurring modeling errors [134]. Even with a model in hand, analytical solutions may not exist and numerical solutions are needed to approximate the exact answers [26, 47], introducing further deviations from the expected result. And finally, most models are real-valued and thus have to be approximated by finite-precision arithmetic, adding roundoff errors [85].

Roundoff errors can be bounded to some extent automatically using techniques such as interval arithmetic [104]. Dealing with most of the errors introduced by modeling, numerical approximation, and finite-precision arithmetic, is rarely automated by software tools. The state of the art in dealing with modeling and numerical errors often requires manual intervention of the programmer or domain expert and is typically on a per-application basis. Because of the resulting complexity of the error analysis, the resulting error bounds are often only asymptotic.

2.2 Embedded, digital signal processing, and multimedia systems

Many computing systems that interact with the physical world or which process data gathered from it, have high computational demands under tightly-constrained resources. These systems, which include many embedded and multi-media systems, must often process noisy inputs and must trade fidelity of their outputs for lower resource usage. Because they are designed to process data from noisy inputs, such as from sensors that convert from an analog signal into a digital representation, these applications are often designed to be resilient to errors or noise in their inputs [213].

Several pioneering research efforts investigated trading precision and accuracy for signal processing performance [7] and exploiting the tolerance of signal processing algorithms to noise [83, 192]. When the outputs of such systems are destined for human consumption (e.g., audio and video), common use cases can often tolerate some amount of noise in their I/O interfaces [207–209, 212, 214].

2.3 Computer vision, augmented reality, and virtual reality

Many applications in computer vision, augmented reality, and virtual reality are compute-intensive. As a result, many of their algorithms (e.g., stereo matching algorithms) have always been implemented with quality versus efficiency tradeoffs in mind [72, 187, 222]. The implementations of these algorithms have used techniques including fixed-point implementations of expensive floating-point numerics [135] and algorithmic approximations such as removing time-consuming backtracking steps [24] when implementing these algorithms on FPGA accelerators.

2.4 Communications and storage systems

The techniques we survey often involve computation on noisy inputs or data processing in the presence of noise in much the same way research in communication systems and information theory considers communication over a noisy channel. As one recent example of work that could be viewed as either traditional information theory and communication systems research or approximate computing, Huang *et al.* [90] present a simple yet effective coding scheme that uses a combination of a lossy source/channel coding to protect against hardware errors for iterative statistical inference.

2.5 Big data and database management systems

Approximate query processing in the context of databases and big data research leverages sampling-based techniques to trade correctness of results for faster query processing. Early work in this direction investigated sampling from databases [153, 154]. More recently, BlinkDB [2], an approximate query engine, allows users to trade accuracy for response time. BlinkDB uses static optimizations to stratify data in a way that permits dynamic sampling techniques at runtime to present results annotated with meaningful error bars. Other recent efforts include Quickr [101] and ApproxHadoop [69].

2.6 Machine learning

Machine learning techniques learn functions (or programs) from data and this data is in practice either limited or noisy. Larger datasets typically lead to more accurate trained machine learning models, but in practice training datasets must be limited due to constraints on training time. As a result, many machine learning methods must inherently grapple with the tradeoffs between efficiency and correctness of the systems.

There are several techniques that allow machine learning systems to trade accuracy for efficiency. These techniques include *random dropout* [203], which randomly removes connections within a neural network to prevent overfitting during training and to improve overall training accuracy. Techniques such as weight de-duplication and pruning [37, 79], low-intensity convolution operators [89, 92], network distillation [174], and algorithmic approximations based on matrix decomposition [50, 110] take advantage of redundancy to minimize the parameter footprint of a given neural network. Weight quantization is yet another technique to reduce computation and data movement costs in hardware [45, 97].

2.7 Approximation and Randomized Algorithms

Approximation has been studied extensively in algorithmics and theoretical computer science. *Approximation algorithms* seek efficient solutions to computationally hard problems (e.g., knapsnack or SAT solving), such that the distance of the approximate solution is guaranteed to be within a fixed bound from the exact solution [228, 234]. These algorithms often use heuristic search methods (e.g., greedy search), or relax hard optimization problems to instances of linear programs. Often, they are *fully polynomial-time approximation schemes*—polynomial-time algorithms that for each fixed error bound ε have their running time bounded by a polynomial in the size of $\frac{1}{\varepsilon}$ and the size of the problem.

Randomized algorithms use a sequence of random bits to achieve good *average case performance* over the input space. These algorithms often ensure that the computed solution is within a small distance from the exact solution with *high probability* [146]. Especially interesting is the class of randomized algorithms that perform *property testing*, i.e., fast randomized algorithms for decision-making [70]. Examples include testing for similarity of strings or almost-sortedness of a sequence's elements. These algorithms typically query only a small fraction of the data set (often logarithmic in size), and make the correct decisions with high probability.

These approximations are purely algorithmic in nature. They often come with strong theoretical guarantees, but do not take advantage of approximation opportunities in the system stack. Each of these topics have been extensively covered by existing monographs, e.g., [70, 146, 228, 234], to which we refer the interested readers.

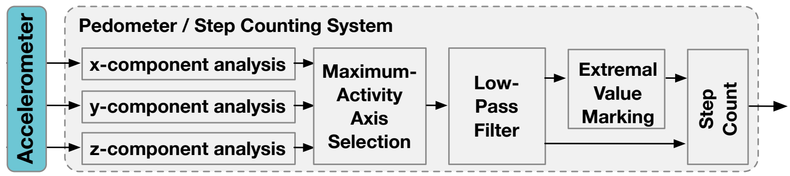


Fig. 1. The block diagram of one canonical pedometer application implementation.

3 ILLUSTRATIVE END-TO-END EXAMPLES

Many applications from the domains of signal processing and machine learning have traditionally had to grapple with tradeoffs between precision, accuracy, application output fidelity, performance, and energy efficiency (see, e.g., Section 2.2 and Section 2.6). Many of the techniques applied in these domains have been reimagined in recent years, with a greater willingness of system designers to explicitly trade reduced quality for improved efficiency.

We discuss two applications from the signal processing and machine learning domains: a pedometer and digit recognition. Using these examples, we suggest ways in which resource usage versus correctness tradeoffs can be applied across the layers of the hardware stack, from sensors, over I/O, and to computation. We use these applications to demonstrate how end-to-end resource usage could potentially be improved even more when tradeoffs are exploited at more than one layer of the system stack.

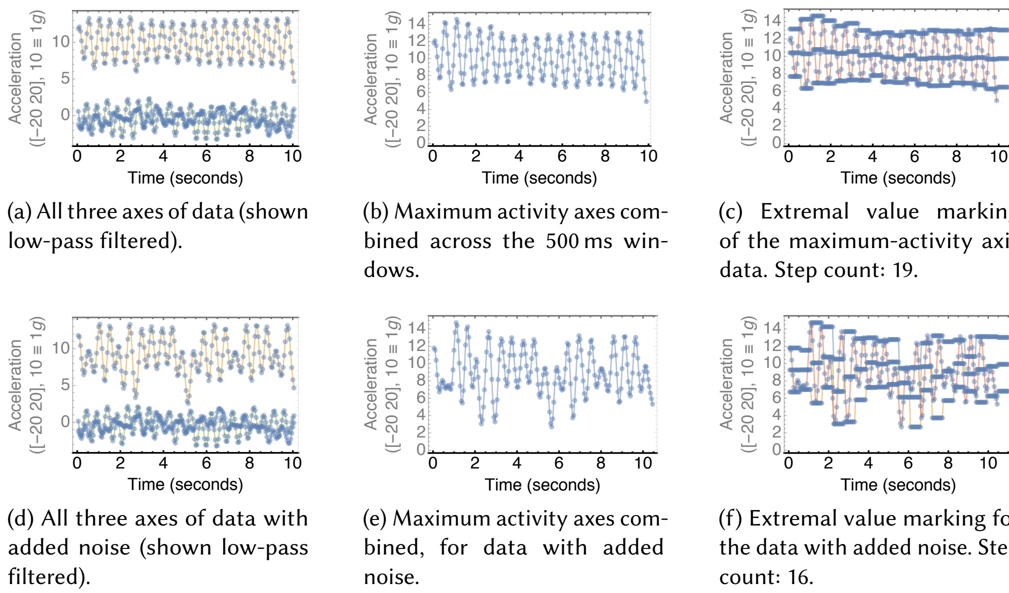
3.1 Example: a pedometer application

Applications which process data measured from the physical world must often contend with noisy inputs. Signals such as temperature, motion, etc., which are analyzed by such sensor-driven systems, are usually the result of multiple interacting phenomena which measurement equipment or sensors can rarely isolate. At the same time, the results of these sensor signal processing applications may not have a rigid reference for correctness. This combination of input noise and output flexibility leads to many sensor signal processing applications having tradeoffs between correctness and resource usage.

One concrete example of such an application is a pedometer (step counter). Modern pedometers typically use data from 3-axis accelerometers to determine the number of steps taken during a given period of time. Even when a pedometer's wearer is nominally motionless, these accelerometers will detect some distribution of (noisy) measured acceleration values. At the same time, small errors in the step count reported by a pedometer are often inconsequential and therefore acceptable.

Figure 1 shows a block diagram for an implementation of one popular approach [242]. Our implementation takes as input 3-axis accelerometer data and returns a step count for time windows of 500 ms. The pedometer algorithm first selects the accelerometer axis with the maximum peak-to-peak variation (the *maximum activity axis selection* block in Figure 1). The algorithm uses the selections to create a new composite sequence of accelerometer samples. Next, the pedometer algorithm performs low-pass filtering, and then, for each 500 ms window, computes the maximum and minimum acceleration values and the midpoint of this range (the *extremal value marking* block in Figure 1). Finally, the algorithm counts how many times the low-pass filtered signal crosses the per-window midpoints in one direction (e.g., from above the midpoint to below it), and it reports this count as the number of steps.

Figures 2a–2c show the progression of a sequence of accelerometer samples through the stages of the pedometer algorithm, which outputs a step count of 19 at the end. Figures 2d–2f show a modified version of the data where we have replaced 5% of the samples with zeros to simulate intermittent failures at a sensor. Even though the data in the final stage of the algorithm (Figures



(a) All three axes of data (shown low-pass filtered).

(b) Maximum activity axes combined across the 500 ms windows.

(c) Extremal value marking of the maximum-activity axis data. Step count: 19.

(d) All three axes of data with added noise (shown low-pass filtered).

(e) Maximum activity axes combined, for data with added noise.

(f) Extremal value marking for the data with added noise. Step count: 16.

Fig. 2. Intermediate stages of data from a pedometer application.

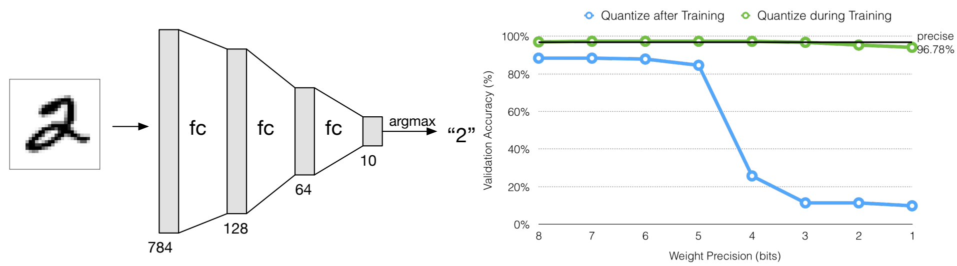
2c and 2f) looks qualitatively different, the final output of the algorithm is relatively close the noise-free output.

Applying individual tradeoffs. The hardware and system stack for a typical pedometer comprises sensors (e.g., accelerometers), I/O links (e.g., SPI or I2C) between those sensors and a processor, a runtime or embedded operating system, the implementation of the pedometer algorithm, and a display. A system's designer may exploit the resource versus correctness tradeoffs at each of these layers or components independently, using the techniques surveyed in Sections 5-8 of this article. For example, a system designer could apply Lax [213] to sensors, VDBS encoding [208, 209, 214] to the I2C or SPI communication between sensors and a microcontroller, and could ensure that the potentially inexact data does not affect the overall safety of the application using EnerJ [180] or FlexJava [158].

Potential for end-to-end optimization. This survey argues for exploring the end-to-end combination of techniques for trading correctness for efficiency, across the levels of abstraction of computing systems. Rather than treating each layer of the hardware and system software stack as an independent opportunity, this article argues that greater resource-correctness tradeoffs are possible when the entire system stack is considered end-to-end. For example, the insensitivity of the pedometer algorithm to input noise highlighted in Figure 2 might be determined by program analyses. These analyses could in turn be used to inform instruction selection for generated code as well as determining sensor operating settings (e.g., sampling rate, operating voltage, on-sensor averaging) and sensor I/O settings (e.g., choices for the I/O encoding for the sensor samples as they are transferred from a sensor).

3.2 Example: digit recognition

Digit recognition is the computational task of determining the correct cardinal number corresponding to an image of a single handwritten digit. Digit recognition is to computer vision scientists what fruit flies are to biologists: a quick and easy way to evaluate the feasibility of a novel idea. One can envision a simple system that analyses handwritten digits directly from a camera source. An image



(a) The MLP topology

(b) The effects of quantization on classification accuracy

Fig. 3. The multi-layer perceptron (MLP) trained on the MNIST dataset: topology and effects of quantization.

is captured by a CCD or CMOS sensor. Each pixel of the image is converted from an analog reading to a quantized value in the digital domain. That raw digital image is now sent for processing to a processor or dedicated ASIC to produce a cardinal number that preferably matches the ground truth handwritten digit. Such system could be used to label postal codes in an automated mail sorting facility. The inherent analog nature of the processing system's camera sensor, combined with the imperfect shapes of the human-written numerals, make digit recognition a fundamentally noisy and error-prone process.

One popular technique used to classify digits is the use of neural networks [116]. Neural networks are trained with a correctly labeled training data set, which is used to iteratively improve classification accuracy via the back propagation algorithm. Neural networks—and more broadly machine learning—are well suited for object recognition problems due to the fact that it would be very difficult to write a program describing how to classify an image based on highly diverse ground-truth examples. Instead, neural networks learn from training examples as a human would do, and derive internally a set of features that can help discriminate between the different classes of objects.

Neural networks are naturally imperfect discriminators: the quality of the classification relies upon how diverse the training set is, and how resilient the classifier is to noise and input modifications (e.g. rotations, lateral shifts, etc.). In addition, if the classifier is trained for too long over a dataset that is too limited, it can end up overfitting the training set. Errors that occur as bit-flips internally can be learned around during training in order to gracefully recover from hardware errors [53]. Aggressive quantization techniques such as network binarization [46, 168] can work well in certain domain problems. Randomized pruning techniques can also provision against overfitting, and help improve the trained classifier's robustness [117]. Consequently, neural networks constitute an ideal target for approximation techniques across the stack.

Figure 3(a) shows a simple network architecture for performing handwritten digit recognition. The network consists of three fully-connected layers (labeled “fc” in the Figure). The input layer takes in a 28×28 image, which is flattened into a one-dimensional array of 784 pixel values. Each fully connected layer performs a weighted sum of all input values to produce a single output value that is applied a bias, followed by a non-linear activation function. The latter determines if the output neuron should be considered active or not. The layers before the final output layer are known as hidden layers: they identify hidden features that help discriminate between different classes of digits. The final layer has 10 outputs, each corresponding to a different class of digits. The neuron with the highest value determines what digit the neural network has classified the input image as.

	Default 2's Complement Encoding					Error-Aware Ternary Encoding				
Bits	Value	b0 stuck at 0	b0 stuck at 1	b1 stuck at 0	b1 stuck at 1	Value	b0 stuck at 0	b0 stuck at 1	b1 stuck at 0	b1 stuck at 1
00	0	0 (Δ=0)	1 (Δ=1)	0 (Δ=0)	-2 (Δ=2)	1	1 (Δ=0)	0 (Δ=1)	1 (Δ=0)	0 (Δ=1)
01	1	0 (Δ=1)	1 (Δ=0)	1 (Δ=0)	-1 (Δ=2)	0	1 (Δ=1)	0 (Δ=0)	0 (Δ=0)	-1 (Δ=1)
10	-2	-2 (Δ=0)	-1 (Δ=1)	0 (Δ=2)	-2 (Δ=0)	0	0 (Δ=0)	-1 (Δ=1)	1 (Δ=1)	0 (Δ=0)
11	-1	-2 (Δ=1)	-1 (Δ=0)	1 (Δ=2)	-1 (Δ=0)	-1	0 (Δ=1)	-1 (Δ=0)	0 (Δ=1)	-1 (Δ=0)

Fig. 4. Application-aware codes can be critical in enabling robustness to errors in neural networks. In the case of ternary neural networks, we can craft codes that ensure a deviation of at most 1 from the original value when a single bit flip error occurs.

Applying individual tradeoffs. Figure 3 (b) shows the results for accuracy of the neural network with quantization of weights starting with a 32-bit floating point baseline all the way down to a 1-bit weight. The network is trained on the MNIST data set with quantization of weights either during or after training. The results show that as long as re-training is applied, this neural network is extremely tolerant even to aggressive levels of quantization. Quantization additionally enables weight prunability and compressibility: weights represented with fewer bits lead to fewer distinct values, and more weights end up in the zero-valued bin. This creates opportunities for sparse matrix compression [78], which can be directly implemented in hardware.

Potential for end-to-end optimization. Once the network has been quantized and compressed, we can further leverage resource versus correctness tradeoffs by storing the weights in approximate voltage-overscaled SRAM cells [169], which occasionally produce read errors. Recent work [108] motivate the use of approximate SRAMs by highlighting that they consume a significant percentage of overall power in accelerator. The authors show that both re-training and fault detection mechanisms can mitigate the destructive effects of voltage-overscaled SRAM read upsets on classification tasks.

In addition, in the case where weights are quantized to -1, 0 and 1, choosing the right encoding shown in Figure 4 can ensure that the effects of a single bit-flip due to an erroneous read remains limited. With this encoding, single bit-flip errors would cause in the worst case a deviation of 1 from the original weight value, as opposed to a value polarity flip from -1 to 1 or vice-versa. The latter is allowed under the default 2's complement encoding, and could potentially lead to catastrophic classification degradation.

To conclude, a synergistic combination of (1) training-corrected quantization and pruning, (2) hardware optimization of SRAMs to minimize power via voltage overscaling, and (3) selecting a weight encoding that minimizes catastrophic bit-flips altogether enables a more efficient digit classification system that can embrace the noisy nature of the image capturing system and the variable nature of handwritten digits.

4 TERMINOLOGY

The terminology used to describe resource usage versus correctness tradeoffs has historically differed across research communities (e.g., the computer-aided design and design/test communities versus the programming language and system software communities). The differences in terminology are sometimes inevitable: a "fault" in hardware is usually a stuck-at logic- or device-level fault while a "fault" in an operating system is usually the failure of a larger macro-scale component. In this article, we attempt to provide a uniform scaffolding for terminology. In doing so, we acknowledge that this terminology will by necessity need to be reinterpreted when applied to the different layers of abstraction in a computing system and we do precisely that at the beginning of each of the following sections.

4.1 Computation in the physical world

We consider computing systems that make observations of the physical world (e.g., using sensors or other data input sources) and compute a discrete set of actions that the system (or a human) then applies back to the physical world. Such end-to-end systems are therefore *analog in, analog out*. In this process, the computing system *measures* the physical world, *computes* on a sample of its measurement, and then computes a set of *actions or actuations* to be applied to the world. Figure 5 shows the steps of computation in the physical world.

We denote the domain of quantized values by $\mathbb{Q}$. In practice, quantized values are often bounded integers or finite-precision floating-point numbers. When working with a relation $r \subseteq A \times B$, the domain and range of a relation r are defined as $\mathrm{Domain}(r) := \{x \mid \exists y.(x, y) \in r\}$ and $\mathrm{Range}(r) := \{y \mid \exists x.(x, y) \in r\}$. The composition of two relations f and g, denoted $f \circ g$, is allowed if $\mathrm{Range}(f) \subseteq \mathrm{Domain}(g)$ and it is defined as $f \circ g = \{(x, z) \mid \exists y.(x, y) \in f \wedge (y, z) \in g\}$. A *left-total* relation is a relation that covers all members of its input domain. In other words, an output exists for every possible input.

Physical world: We assume that all of our systems are situated in the physical world and we model inputs from this world with real numbers, $\mathbb{R}$. This assumption is consistent with most applications that trade errors for efficiency (see Section 2 and Section 3), such as sensing applications (as in Section 3.1), cyber-physical systems, computer graphics, computer vision, machine learning (as in Section 3.2), and scientific computing.

Measurement and analog processing step: Each computation situated in the physical world begins with a *measurement* in which the computing system makes an observation of the physical world. In metrology, this quantity is referred to as the *measurand*. We denote the result of a measurement by a probability distribution. We restrict our focus to distributions that we can represent with a *probability density function (PDF)*, $f : \mathbb{R} \to \mathbb{R}$.

Measurements may include within their internal processes computations that transform the measured distributions to yield new distributions. These internal processes may be nondeterministic. We include this facility to account for systems that may perform computation directly in the unsampled and unquantized analog domain and Section 5.3 of the survey gives examples of such systems. A measurement is therefore a function of type $f : \mathbb{R} \to (\mathbb{R} \to \mathbb{R})$, mapping a real value (the measurand) to a function in the form of the probability distribution (the measurement). The result of the measurement step of a computation is therefore still in the domain of continuous-time real-valued quantities.

Sampling and quantization step: Between the measurement step and a subsequent discrete (digital) computation step, we assume that there is a sampling and quantization step that generates discrete-time samples with discrete values from the real-valued distribution resulting from the measurement step. A *sampler* is therefore a relation $f : (\mathbb{R} \to \mathbb{R}) \times \mathbb{Q}^m$ that samples and quantizes a discrete value from a probability distribution. ($\mathbb{Q}^m$ denotes the set of allowable quantized values.) The process of quantization adds an implicit noise, known as the *quantization noise* to the real-valued input.

Digital computation step: In the discrete world, we consider the computations that take as input a discrete sample from the measured world and performs a potentially nondeterministic computation to produce a discrete output. Therefore, a discrete computation f is a left-total relation $f \subseteq \mathbb{Q}^m \times \mathbb{Q}^o$ where $\mathbb{Q}^m$ is the input and $\mathbb{Q}^o$ is the output.

Actuation step: The digital outputs can be used back in the physical world as inputs to real-valued actuation which modifies the state of the physical world. An actuation computation is therefore a nondeterministic function that we model as a left-total relation $f \subseteq \mathbb{Q}^o \times \mathbb{R}$.

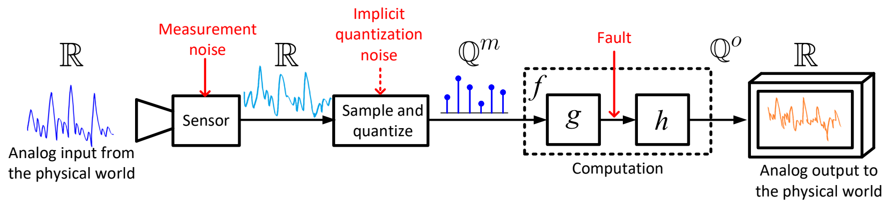


Fig. 5. Steps of computation in the physical world.

4.2 Computation and correctness

Following the terminology defined in Section 4.1, we can express any computation that processes data from the physical world as a composition of the steps of measurement, sampling and quantization, digital computation, and actuation. Each of these steps defines a *computation*:

Computation: A computation f is a nondeterministic function that we denote as a left-total relation $f \subseteq \mathbb{I} \times \mathbb{Q}$ where we instantiate the domain $\mathbb{I}$ and $\mathbb{O}$ to fit the computation's corresponding step from Section 4.1. For example, as we will see later in Section 5.1, at the circuit level, the input domain $\mathbb{I}$ and the output domain $\mathbb{O}$ are voltage levels.

We model computations as left-total relations to account for nondeterminism where for the same input, the computation may produce different outputs on different executions. The relations are left-total in that there exists at least one output for every value in the input domain. This modeling assumptions also dictates that computations terminate. If a computation is deterministic, then we model it as a function $f : \mathbb{I} \to \mathbb{O}$.

Specification: For any computation $f \subseteq \mathbb{I} \times \mathbb{O}$, a system's developers and users can provide its *specification* as a relation $f^* \subseteq \mathbb{I} \times \mathbb{O}$ that defines the set of *acceptable* mappings between the function's inputs and outputs. A specification need not be executable itself and multiple implementations can satisfy the same specification.

Correctness: A computation and its corresponding definition as a relation is *correct* if it *implements* its specification. A computation f implements a specification f^* iff $\forall i, o.\ (i, o) \in f \Rightarrow (i, o) \in f^*$. This definition means that every output of f for a given input, must be valid according to the specification.

Faults: To define faults, we first decompose a computation f into two computations $g \subseteq \mathbb{I} \times \mathbb{M}$ and $h \subseteq \mathbb{M} \times \mathbb{O}$, where $\mathbb{M}$ is a domain of values for the output of g and $h \circ g \equiv f$. Given this decomposition, a *fault* is an anomaly in the execution of g on an input i such that g produces an anomalous, unexpected value m in that $(i, m) \notin g$.

Errors: An error occurs when a computation encounters a fault and the computation's resulting output does not satisfy its specification. Given a computation f and its decomposition into g and h as above, the semantics of an error is that if the execution of g on i produces a faulty value m (as above), then that fault is an error if the result of the continued execution via h does not satisfy f's specification — namely, that $(i, h(m)) \notin f^*$.

Masking: A fault does not always result in an error; a fault can instead be *masked*. If a computation encounters a fault and the computation's resulting output satisfies its specification, then the fault has been masked by the computation's natural behavior. Given a computation f and its decomposition into g and h as before, the semantics of a masked error is that if the execution of g on i produces a faulty value m (as above), then that fault is masked if the result of the continued execution via h satisfies f's specification — namely, that $(i, h(m)) \in f^*$.

Precision and accuracy: We define *precision* as the degree of discretization of the state space determined by $\mathbb{Q}^m$ (from the sampling and quantization step, Section 4.1) and we define *accuracy* as a distance between the functions f and f^* defined above.

4.3 Standard viewpoints

Let h be the identity function and $f = g$. Then the aforementioned definitions give a semantics to faults that affect the output of a single, monolithic function f. Take f^* to be f, then the function's specification is given by its exact behavior. This form of specification is the standard assumption for computing systems wherein they must preserve the exact semantics (up to nondeterminism) of the computation. Most existing approaches to trading errors for efficiency fit this viewpoint: they typically start from an existing program as their specification and approximate it to allow for more efficient implementations.

4.4 Quantifying errors

Approaches to quantifying errors include absolute errors, relative errors, and error distributions. In most contexts, the evaluator of a system is interested in the error of not only a single input, but a whole domain of inputs. Depending on the application domain, upper or lower bounds on the worst-case error, or average errors may be of interest. When a computation runs repeatedly, the *error frequency* or *error rate* captures how often a computation returns an incorrect result.

5 TRANSISTOR-, GATE-, AND CIRCUIT-LEVEL TECHNIQUES

Transistors provide the hardware foundations for all computer systems. As a result, their physical properties determine the efficiency at which a particular computation can be performed. When collections of transistors are used to form gates and analog circuits, and when collections of gates are used to implement digital logic circuits, the organization of the transistors, gates, and circuits can be designed to trade efficiency for correctness.

5.1 Notation

Following the notation introduced in Section 4, input $\mathbb{I}$ can be defined as a voltage level that is switched to $\mathbb{O}$ as a computation f is executed. Therefore, $f : \mathbb{I} \rightarrow \mathbb{O}$ is a switching of voltage at a transistor or a group of transistors forming a circuit element, for instance, a byte in a memory or an adder in an arithmetic/logic unit (ALU). Such computation can be regarded as $f^* : \mathbb{I} \rightarrow \mathbb{O}$ where the relation between f^* and f is the difference in the electrical operating points of the individual transistors. This difference saves computational costs like power consumption and latency while introducing timing errors and incorrect voltage levels.

5.2 Analog input / analog output systems: a comparison reference for quantization

When using finite-precision arithmetic, computation always involves errors that are caused by quantization. Quantization is a fundamental mechanism for trading energy for accuracy and recent work has highlighted examples of its effectiveness [148].

The effect of quantization errors can be observed by treating the inputs and outputs of a computing system as real-valued analog signals and comparing these signals to an ideal (error-free) computing system that accepts analog inputs and produces analog outputs. When such ideal outputs are not available, designers often use the output of the highest precision available (e.g., double-precision floating point) as the reference from which to determine the error of a reduced-precision block. Such analyses are common in the design process of digital signal processing algorithms such as filters [159] where the choice of number representation and quantization level enables a tradeoff between the performance and signal-to-noise ratio properties of a system.

5.3 Analog computing: data processing with currents and voltages

Analog computing systems [103, 130] eliminate the need for discretization and the resulting restriction on precision that is inherent in digital circuits. While, in theory, analog circuits provide unbounded precision, in practice their precision is limited by factors such as noise, non-linearities, the degree of control of properties of circuit elements such as resistors and capacitors, and the degree of control of implicit parameters such as temperature. At higher precision, analog blocks tend to be less energy-efficient than digital blocks of equivalent precision [186]. Because they usually do not use minimum-size transistors, analog circuits may also be larger in area than their digital circuit equivalents. Designing analog computation units is also a challenging task. Nevertheless, analog circuits can be an attractive solution for applications that tolerate low-precision computation [186].

5.4 Probabilistic computing: exploiting device-level noise for efficiency

A line of research pioneered by Palem *et al.* [35, 156] ("probabilistic computing") proposes harnessing intrinsic thermal noise of CMOS circuits to improve the performance of probabilistic algorithms that exploit a source of entropy for their execution. Chakrapani *et al.* [31] show an improvement in the energy-performance product for algorithms such as Bayesian inference, probabilistic cellular automata, and random neural networks using this approach and they establish a tradeoff between the energy consumption and the probability of correctness of a circuit's behavior. These techniques have also shown energy savings for digital signal processing algorithms that do not employ probabilistic algorithms but which can tolerate some amount of noise [67, 107].

5.5 Stochastic computing: unary representation and computing on probabilities

Stochastic computing (SC) uses a data representation of bit streams that denote real-valued probabilities [6]. In theory, the probabilities can have unbounded precision, but in practice, the length of the bit-streams determines precision [4]. SC was first introduced in the 1960s [65] and its main benefit is that it allows arithmetic operations to be implemented via simple logic gates: a single AND gate performs SC multiplication. This made SC attractive in the era of expensive transistors. But as transistors became cheaper, SC's benefit faded away, and its main drawbacks, i.e., limited speed and precision, became dominant [4]. For this reason, SC was only used in certain applications, such as neural networks [51, 111] and control systems [223].

SC has seen renewed interest over the last decade [4], mainly because of its energy efficiency. SC's probabilistic nature copes with new inherently random technologies such as memristors [112]. Furthermore, the unary encoding of numbers on SC makes the computation robust against errors [160], and allows variable precision computation [5]. With the low precision requirement of modern machine learning applications, SC is becoming an attractive alternative to conventional binary-encoded computation [119].

Despite what the name suggests, most existing SC circuits are in fact deterministic. Unless a true random number source (e.g., memristors) is used, SC circuits will produce the same output if inputs are unchanged. Although pseudo-random number generators with long periods can imitate a true random number source, in reality they are still deterministic. Furthermore, several studies have shown that adding determinism to SC is in fact useful [75, 151]. If carefully chosen, deterministic number sources can increase the accuracy of SC circuits without adding any extra overhead.

5.6 Voltage overscaling: improved efficiency from reduced noise margins

The term *voltage overscaling* is often used to refer to reducing supply voltages more than is typically deemed safe for a given clock frequency. Voltage overscaling exploits the quadratic relationship between supply voltages and dynamic power dissipation. Let V_{dd} be the supply voltage of a CMOS circuit (e.g., an inverter), let f be its clock frequency (reciprocal of its delay) and let C be the effective capacitance of the load of the circuit. Then, the dynamic power dissipation P is [162]

$$P \propto C V_{\mathrm{dd}}^2 f. \tag{1}$$

The delay of a gate in a circuit, and hence the clock frequency f, is however not independent of supply voltage V_{dd}. Let V_t be the device threshold voltage and let α be a process-dependent parameter (the velocity saturation exponent [176]). Then, as supply voltage V_{dd} decreases, the delay of charging its load capacitance for a gate increases and the maximum clock frequency achievable at a given voltage follows the relation

$$f \propto \frac{(V_{\mathrm{dd}} - V_t)^{\alpha}}{V_{\mathrm{dd}}}. \quad (2)$$

As a result, overscaled voltages cause circuit delays, which in turn lead to timing errors in circuits at a fixed clock speed. Several approaches have explored the idea of carefully and systematically accepting such errors in exchange for the large (quadratic) power savings that voltage overscaling can potentially provide [84, 100, 102, 114, 196]. In unmodified circuits, this often leads to catastrophic errors at close-to-nominal voltages, as many digital circuits are optimized to minimize timing slack. However, for several application domains, such as image and video processing, inherent dependence of errors on known input characteristics can be exploited to redesign circuits such that they allow for significant overscaling with small and graceful degradation of output quality [13, 81, 147, 225]. However, voltage overscaling has potential issues with timing closure and meta-stability. Furthermore, timing errors in the critical paths of a circuit due to voltage overscaling tend to affect the most significant bits of a computation first and hence can lead to large errors. Dedicated logic modifications targeting lower significant bits as described next can instead provide better accuracy with additional switching activity savings for the same timing and hence voltage reduction [137].

5.7 Pruned circuits for efficiency at the expense of precision and accuracy

Pruning circuits refers to deleting or simplifying parts of a circuit based on the probability of their usage or importance to output quality. Recent research has shown how circuit pruning improves latency, energy, and area without the overheads associated with voltage scaling [122, 136, 189, 197].

Pruning can be applied to digital circuit building blocks such as adders and multipliers, enabling quality-cost tradeoff opportunities through different logic simplification and pruning techniques. Approximate adders attempt to simplify carry chains [232, 243] or to use approximate 1-bit full adders [76, 131, 137] at lower significant digits of a sum computation. Accuracy-configurable adders have also been proposed for adaptive-accuracy systems that require a functional unit like an adder or multiplier to vary the degree of tradeoff between correctness and resource usage based on the quality demand of computation [99]. Unlike approximate adders, approximate multipliers have a higher design space exploration requirement, as they are composed of 2×2 partial products that are summed up by deploying an adder tree to compute the final result [113]. Correctness versus resource usage tradeoffs can be deployed in multipliers (partial products) or adders, or both, for a chosen number of least-significant bits [95, 170].

Approximate adders and multipliers provide the combinational building blocks for approximate datapath and processor designs. At the sequential logic level, the challenge is in determining the amount of approximation to apply to each addition or multiplication operation in a larger computation in order to minimize output quality loss while maximizing energy savings. For example, in a larger computation that consists of multiple accumulations, using an adder with a zero-centered error distribution [137] will result in positive and negative errors canceling each other and thus averaging in the final output of a larger accumulation. Similarly, approximate multipliers with unbiased error distribution are promising for accumulation-based algorithms like multiply-accumulate (MAC) [80]. By contrast, in other computations, an approximate combinational block that always over- or under-estimates may be beneficial.

Determining the best tradeoff for each functional unit in a larger sequential design has been investigated for fixed register transfer level (RTL) designs [152, 166, 231]. Pure RTL optimizations, on the other hand, do not exploit changes in approximated component characteristics for a complete RTL re-design. In the context of custom hardware/accelerator designs, selection of optimal approximated operator implementations can instead be folded into existing C-to-RTL high-level synthesis (HLS) tools [118, 120]. For programmable processors, accuracy configuration of the datapath can be exposed through the instruction-set architecture (ISA) [229]. A compiler then has to determine the precision of each operation in a given application (see Section 6 and Section 7).

5.8 Approximate memory: reducing noise margins for efficiency in storage

Memory costs are often higher than that of computations in many data-intensive applications [15]. Approximate memories have been investigated in the research literature, to trade quality for energy, latency, and lifetime benefits [181, 198]. Reducing the refresh rate of DRAM provides an opportunity to improve energy efficiency while causing a tolerable loss of quality [98, 165]. For static random access memory (SRAM), by contrast, the tradeoff between correctness and resource usage is typically achieved by voltage overscaling, where the main concern is in dealing with the failures in the standard 6-Transistor (6T) cells of an SRAM array under reduced static noise margins (SNMs) [66]. As a result, hybrid implementations combining 6T with 8T SRAM cells [32] or with standard cell memory (SCMEM) [19] have been employed to achieve aggressive voltage scaling in order to get better quality versus cost tradeoffs. In case of emerging non-volatile random-access memory (NVRAM) technologies, such as spintronic memories (e.g., STT-MRAM), reducing the read current magnitude can reduce energy of read operations at the expense of reduced noise margins and hence accuracy of the content being read [167]. Similarly, significantly increasing the read current magnitude reduces the read pulse duration, decreasing the read latency while potentially disturbing the written content with noise.

5.9 Summary

The circuit-level techniques surveyed in this section must ultimately be deployed in the context of concrete applications. For example, one case study found that for applications such as Fast Fourier Transforms (FFTs), motion compensation filters, and k-means clustering, applying traditional fixed-point optimizations to limit the size of operands was more effective than applying circuit-level approximations such as approximate adders and multiplier circuits [14]. This is because approximating some bit values still requires information about those bits to be stored and used in downstream computations. The additional overhead of this bookkeeping in many cases is not worth the quality benefits. Carefully selecting the most suitable approximation strategies and comparing their cost versus quality tradeoffs can therefore lead to a better solution for certain applications.

6 ARCHITECTURE AND MICROARCHITECTURE-LEVEL TECHNIQUES

Architectural and microarchitectural techniques that trade correctness for resource usage have focused primarily on correctness at the software or application level and have focused on reducing resource usage in memory, in the processor, and in on- or off-chip I/O.

6.1 Notation

Architectural techniques create abstractions that allow operating systems, programming languages, and applications to specify their precision and accuracy requirements through specialized instructions and instruction extensions. Following the notation introduced in Section 4, the computation function $f : \mathbb{Q}^m \rightarrow \mathbb{Q}^o$ is defined over the quantized sets $\mathbb{Q}^m$ and $\mathbb{Q}^o$ embodied by software-visible machine state such as registers, memory, and storage. The computation function f is implemented using either general-purpose cores or specialized hardware accelerators. Microarchitectural techniques facilitate the efficient implementation of the computation function f at the level of hardware functional units, such as memory controllers and processor pipelines, or by the efficient hardware representation of the sets $\mathbb{Q}^m$ and $\mathbb{Q}^o$.

6.2 Trading resource usage for correctness in processor cores

Early work trading resource usage for correctness such as Razor and related techniques [57, 202], relied on voltage overscaling as the primary underlying circuit-level mechanism to increase energy efficiency. As a result, these techniques provided no direct means to improve performance, but provided higher energy efficiency at the expense of nondeterministic faults. To mask such faults and hide them from applications, voltage overscaling approaches typically rely on error recovery mechanisms. The key insight is that sophisticated error recovery mechanisms can be much more resource-efficient in ensuring correctness compared to voltage over-provisioning. Carefully balancing the error recovery overhead against the benefits of voltage overscaling can provide higher energy efficiency without sacrificing output quality or program safety [57, 202].

Truffle [58] was the first architecture to willingly introduce uncorrected nondeterministic errors in processor design for the sake of energy efficiency. Truffle uses voltage overscaling selectively to implement approximate operations and approximate storage. The Truffle architecture provides ISA extensions to allow the compiler to specify approximate code and data and its microarchitecture provides the implementation of approximate operations and storage through dual-voltage operation. For error-free operations, a high voltage is used, while a low voltage can be used for approximate operations. Voltage selection is determined by the instructions, with the control-flow and address generation logic always operating at a high voltage to ensure safety.

In addition to improving energy efficiency, architectures that enable tradeoffs between resource usage and correctness may result in higher performance compared to an error-free baseline [59, 150, 204, 238]. Examples of approaches include offloading parts of a processor's workload to computing units that can perform the desired functionality much faster at the cost of deviation from correct behavior. Because of their performance advantage, such computing units are often called accelerators. Accelerators that trade resource usage for correctness include, most notably, neural accelerators [59, 150, 204, 238], which implement a hardware neural network trained to mimic the output of a desired region of code.

Temam *et al.* empirically show that the conceptual error tolerance of neural networks translates into the defect tolerance of hardware neural networks [220], paving the way for their introduction in heterogeneous processors as intrinsically error-tolerant and energy-efficient accelerators. St. Amant *et al.* demonstrate a complete system and toolchain, from circuits to a compiler, that features an area- and energy-efficient analog implementation of a neural accelerator that can be configured to approximate general purpose code [204]. The solution of St. Amant *et al.* comes with a compiler workflow that configures the neural network's topology and weights. A similar solution was demonstrated with digital neural processing units, tightly coupled to the processor pipeline [59], delivering low-power approximate results for small regions of general-purpose code. Neural accelerators have also been developed for GPUs [238], as well as FPGAs [150].

6.3 Approximate memory elements

Memory architectures that trade resource usage for correctness permit the value that is read from a given memory address to differ from the most recent value that was written. The traditional view of memory elements assumes that every memory access pair consisting of a write followed by a subsequent read operation, applied to a input $\mathbb{I}$, results in the same read result for a given write value. In contrast, approximate memory elements may perform non-identity transformations of the input $\mathbb{I}$. At the architecture level, the benefits of doing so are exposed and leveraged through reduced read/write latency, reduced read/write access energy, fewer accesses to memory, increased read/write bandwidth, increased capacity [74, 94, 181], improved endurance [181], and reduced leakage power dissipation [125]. These techniques have been applied to memory components ranging from CPU registers [58], caches [58, 183–185], main memory [125], to flash storage [74, 94, 181].

Underlying memory technologies employ circuit-level mechanisms to trade accuracy for reduction in latency or access energy (or both) (see Section 5.8). Exposing such mechanisms as tunable knobs to the architecture allows adaptive selection of different approximation/efficiency operation points to better fit varying application requirements. For volatile memory technologies, such as SRAM and DRAM, voltage scaling and refresh rate scaling can be used to reduce static or dynamic energy at the expense of faults, observed as bit flips [58]. In the case of DRAM, differentiating between rows that contain data for which errors can be tolerated versus rows that applications require to remain correct motivates selective refresh rate reduction approaches [125]. In non-volatile memories, mechanisms that reduce read or write noise margins for energy gains can be leveraged at the architectural level through dedicated instructions for imprecise loads and stores [167]. For multi-level solid-state memories that perform write operations iteratively until the written value is in the desired range, reducing the number of write iterations significantly reduces the latency and energy of such approximate writes [181], increasing write bandwidth as a side effect, at the expense of reduced data retention. Furthermore, mapping data that applications can tolerate to be incorrect onto blocks that have exhausted their hardware error correction resources can significantly extend endurance.

Other architectural methods for trading resource usage for correctness in memories include predicting memory values instead of performing an actual read operation. For example, on the occurrence of a cache miss, *load value approximation* (LVA) [185, 221] provides predicted data values to a processor which may differ from the correct values in main memory. Doing so hides cache miss latency and thereby reduces the average memory access time at the expense of having data values in the cache that differ from what they would be had they been faithfully loaded from main memory. The correct values in main memory may subsequently be read from memory to train the predictor and improve its accuracy, or the main memory access may be skipped entirely to save energy. Conventional value prediction considers any execution relying on predicted values speculative and provides expensive microarchitectural machinery to roll back execution in the case of a mismatch between the predicted and actual values. LVA, by contrast, allows imperfect predictions, trading correctness of values in the cache for reduced micro-architectural complexity and reduced memory latency.

The correctness of values obtained from memories can also be traded for an increase in effective storage capacity. One way to achieve this is to avoid storing similar data multiple times. For example, storing similar data in the same cache line can save on cache space in situations when substituting a data item for a similar one still yields acceptable application quality [183, 184]. Another way to trade errors for capacity is through deliberate reduction in storage resources dedicated to error-correction [74, 94]. By providing weaker error-correction schemes for data whose accuracy does not have a critical impact on the output quality, significant storage savings have been demonstrated in the case of encoded images and videos [74, 94].

6.4 Approximate communication

As in the case of approximate memory elements, approximate communication systems may perform non-identity transformation f^* of input $\mathbb{I}$ to efficiently transfer the input through a communication channel or network. The idealized computation function f corresponds to an identity transformation over an infinitely large input. Examples of inputs include signals on intra- and inter-chip wires, such as memory buses and on-chip networks. The architectural techniques trading resource usage for correctness in such systems usually rely on more efficient but less reliable links, network buffers, and other network elements, or employ lossy in-network compression to minimize data movement, while overlapping the compression and communication.

The conventional approach to trade resource usage for correctness in communication over a channel is to employ lossy compression at the source and decompression at the destination, with the goal of reducing the amount of data transferred through the channel, as well as to reduce latency. Such approaches have been widely used for decades in long-distance communication, such as media streaming applications. However, when the communicating parties are two processors on a board, two cores on a chip, or a core and a cache, the communication latency is in the order of nanoseconds and any compression/decompression latency added to the critical path of program execution may be prohibitive.

At the circuit level, transmitting bits over a wire on-chip or over a printed circuit board trace costs energy. For single-ended I/O interfaces, where the signaling of information is with voltage levels, the energy cost is typically due to the need to charge the wire capacitance when driving a logic '1', and to discharge that capacitance when driving a logic '0'. Building on this observation, and on the body of work on low-power bus encodings [38, 205], value-deviation-bounded serial encoding (VDBS encoding) [212, 214] trades correctness for improved communication energy efficiency by lossy filtering of values to be transmitted on an I/O link. VDBS encoding reduces the number of '0' to '1' and '1' to '0' signal transitions and hence reduces the energy cost of I/O. Because VDBS encoding requires no decoder, it can be implemented with low overhead, requiring less than a thousand gates for a typical implementation [208]. Extensions of VDBS encoding have extended the basic concept to exploit temporal information in information streams [109, 155] and to employ probabilistic encoding techniques [209].

A recent study leverages data similarity between cache blocks to perform lossy compression in networks-on-chip (NoCs) [22]. The key idea is in simple data-type aware approximation using approximate matching between data to be sent and data items that have been recently sent to perform a quick lossy compression. Performing approximation at the network layer allows a significant data movement reduction without losing the precise copy of the data and without extending the critical path, as the communication and compression are overlapped.

An orthogonal approach to trading resource usage for correctness in communication by compression, is to reduce the safety margins of communication links to trade off their reliability for bandwidth, latency, or both. For on-chip networks, achieving reliable transmission in low-latency high-bandwidth interconnects requires features like forward error correction (FEC), but FEC can increase communication latency, by up to three fold in one study [64]. An approach to counteract such high overheads is to allow higher bit error rates at the link layer by removing forward error correction or employing a weaker but more efficient error correction mechanisms, with a variable amount of redundancy based on application needs [64]. A low-diameter network is one approach to keep the end-to-end bit error rate under control, minimizing the number of hop counts, and thus prevent excessive accumulation of errors [64].

Allowing errors in communication can be particularly challenging in parallel programs, which rely on communication for synchronization. In such contexts, failure to deliver correct messages on time can affect control flow and lead to catastrophic results [241]. Yetim *et al.* propose a mechanism to mitigate inter-processor communication errors in parallel programs by converting potentially catastrophic control flow or communication errors into likely tolerable data errors [241]. Their main insight is that data errors have much less impact on the application output compared to errors in control flow. Their approach is to monitor inter-processor communication in terms of message count, and to ensure that the number of communicated items is correct, either by dropping excess packets or by generating additional packets with synthetic values. Ensuring the correct number of exchanged messages improves the integrity of control flow in the presence of communication errors and consequently improves the output quality of approximate parallel programs.

6.5 Summary

Microarchitectural techniques that trade correctness for resources build on circuit level techniques (Section 5) to exploit information at the level of hardware structures such as caches, register files, off-chip memories, and so on. Architectural techniques expose microarchitectural techniques to software through constructs such as instruction extensions, new instruction types, or new hardware interfaces to accelerators. Exposing information about hardware techniques to software allows software to take advantage of the implemented techniques, while exposing information from software to hardware allows hardware to, for example, more aggressively leverage tradeoffs between correctness and resource usage. In the same way that circuit-level techniques form a foundation for the approaches discussed in this section, circuit-level, microarchitectural, and architectural techniques similarly form a foundation for operating system and runtime system techniques.

7 PROGRAMMING LANGUAGE TECHNIQUES

Many programming-language- and compiler-level techniques that trade correctness for efficiency provide abstractions for dealing with errors introduced at lower levels of the system stack, or introduce higher-level approximations directly and these errors combine into whole-application errors. Previous research presented a variety of automated transformations, together with the reasoning mechanisms that verify or estimate the accuracy of the whole-application results.

This section focuses on the techniques for *reasoning* about approximations and *managing* their impact on program execution. We divide the techniques into two broad categories: (1) *static compile-time techniques* ensure that tradeoffs are safe to apply for all inputs, without running a program and (2) *dynamic tuning techniques* typically execute the program on a set of representative inputs to estimate the benefit and safety of the transformations (off-line), or monitor the program execution at runtime to tune the level of approximation (on-line).

7.1 Notation

Following the notation introduced in Section 4, programming language techniques usually operate at a level of abstraction where the computation's implementation $f : \mathbb{I} \times \mathbb{O}$ is defined over a sets represented by, e.g., integers or floating-point numbers. These integer and floating-point number representations serve as an abstraction for the actual bit-level representations of program state in hardware. The idealized specification f^* that f implements may thus, for instance, assume unbounded integers or real numbers for its output $\mathbb{O}$ and may represent an entire computational problem or specific algorithm. Examples of errors introduced by the discrepancy between f and f^* are floating-point roundoff errors, errors due to skipping entire portions of a computation or due to missing synchronization. To quantify the end-to-end error, a developer typically specifies a distance $d(f, f^*)$. Examples distances include absolute error, relative error, worst-case error, or error probability. Selecting the appropriate distance and its acceptable threshold are typically application-dependent.

7.2 Static compile-time techniques

Static techniques aim to make resource versus correctness tradeoffs safe to apply without having to run a program. Safety of approximate programs ensures that (1) the errors caused by the approximation never make the program crash, diverge, or violate other important program properties, and (2) the result of the approximation is acceptable, i.e., that the magnitude and frequency of errors is within certain bounds. Errors introduced at the lower levels of the stack do not affect every operation of a high-level program equally. Ideally, errors in lower layers of the stack should be restricted to locations such that when they propagate to higher layers of the stack, they only affect those parts of the program where errors can be tolerated.

Conventional programming languages, however, do not provide a transparent way to mark what can be potentially approximate. Several approaches propose annotations that allow the developer to make the effects of lower-level errors explicit: EnerJ [180] ("approx" and "precise" data types), Asymmetric RHL [20, 29] ("relax" and "relate" variable annotations), and FlexJava [158] ("loosen" and "tighten" variable annotations). These techniques use type inference, theorem proving, and taint analysis, respectively, to mark all data/instructions that may be affected by errors. They can verify important safety properties, e.g., EnerJ ensures that approximate data cannot impact the values of precise data.

Rely [30] extends the scope of program analysis to probabilistic errors: it automatically derives the probability of a result being exact, given the probabilities of individual operations being exact. Decaf [21] presents an alternative approach for deriving probabilities using type inference. More recently, Parallely [62] extends the reliability analysis to parallel message-passing programs by reducing them to equivalent sequential programs. For a more precise analysis that considers the case of diverging control flow, Lohar *et al.* [127] present a sound static analysis that can take advantage of probability distributions provided by the developer.

The probability of a computed value being incorrect does not capture the numeric magnitude of the error. Numeric error estimation has been addressed in the form of static analysis for bounding errors due to input disturbances [34], optimizing finite-precision arithmetic [39, 48] and approximating elementary function calls [93] while guaranteeing sound error bounds. Numeric error magnitude can also be estimated by differential program verifiers to check relative safety, accuracy, or termination with respect to some reference implementation by reduction to a *satisfiability modulo theories* (SMT) problem [82].

The above approaches either quantify the probability or the error magnitude, but not both. Furthermore, they do not optimize directly for performance or energy usage. Zhu *et al.* [244] propose a framework which explores a randomized combination of resource-correctness tradeoffs provided by a user. It presents a tradeoff space exploration algorithm based on linear programming, which provides near-optimal guarantees. Chisel [139] combines a reliability analysis with error bounds computation. It automatically finds approximations satisfying a specification and minimizes energy by reduction to an integer linear problem. Lohar *et al.* [128] provide a dataflow-based probabilistic error analysis which computes the probabilities of different error magnitudes.

Static techniques are desirable as they can provide strong correctness guarantees. However, for a static optimization technique, a faithful resource cost model is needed. Until now, these models have been mostly high-level, coarse, and additionally not consistent across different techniques or evaluations, making combinations and comparisons of different techniques challenging. These models necessarily have to abstract over the underlying hardware in order to be scalable and widely applicable, but they also need to reflect reality as much as possible. Here, a tighter collaboration between the software and hardware is needed (see Challenge 2 in Section 11).

7.3 Dynamic tuning techniques

Static guarantees are in practice achievable only for small programs. For many applications such strong guarantees may not be necessary. Dynamic or testing-based validation techniques trade correctness for practical scalability and have been widely used to identify resource versus correctness tradeoffs and to validate the quality of these tradeoffs.

A first step when implementing an application in an error-efficient way is to determine which parts of the application are resilient to errors and which are not [172, 173]. Different applications allow for error-efficient computing to various degrees. For instance, some algorithms can tolerate higher error rates but lower error magnitudes and vice versa [41].

Profiling has traditionally been used to identify performance-intensive portions of a program. A quality of service profiler [141] takes into account quality of the results in addition to performance and can thus identify resilient portions of an application. A similar idea has been explored by the ARC framework [41], which profiles an application while injecting errors, derives a statistical error-resilience profile, and identifies the best error-efficient technique for the given application. The statistical error-resilience profile has also been explored for iterative workloads [68] to identify the number of resilient iterations. Passert [182] and AxProf [96] use statistical testing to determine the confidence of specifications for randomized approximate programs.

Once resiliency of an application is established, a developer or a compiler can apply various transformations. There are several interesting directions here:

- In arithmetic, Precimonious [175] assigns differing floating-point precisions across the variables in a program. STOKE [188], on the other hand, generates entirely new implementations of floating-point functions. Both Precimonious and STOKE ensure that on a given test set a user-defined quality bound is satisfied.
- Since loops usually make up the bulk of running time of a program, loop perforation [87, 141, 199] selectively skips entire loop iterations. Adaptive perforations have been subsequently explored in image processing [129], neural networks [63], and at a finer-level of granularity [121, 144].
- Approximate memoization interpolates skipped data using the already computed one, thus reducing error. Variants of approximate memoization have been proposed by several works [34, 143, 177, 178, 224], exploiting both temporal and spatial locality of data. Neural networks can also be used to replace blocks of imperative code [58] and can provide a performance benefit when coupled with a dedicated neural processing unit.
- Synchronization is another expensive part of many applications, and several research efforts have observed that some synchronization primitives can be removed without impacting quality significantly [28, 49, 140, 142, 171, 172]. Quickstep [140] explores nondeterminism as a technique for trading resource usage for correctness techniques, by parallelizing a sequential program such that data races can occasionally occur. STAS [49] proposes generating alternative data in parallel, to prevent waiting at synchronization points and to improve thread-level parallelism.

Another approach to exploiting resilient applications is to let a user define several application components with different resource-correctness tradeoffs and to provide tool support to select between these candidates to obtain a final implementation [8, 9, 12, 52, 61]. The search can be optimized with sensitivity analysis [227]. The Intel open-source approximate computing toolkit (iACT) [143] provides a simulation-based testbed for different approximations, such as precision scaling and approximate memoization.

Although not all resource versus correctness tradeoffs are suitable for all application domains, most of the techniques discussed above are application-independent. Chippa *et al.* [42] and Venkataramani *et al.* [230] present application-specific approaches for machine learning classifiers which exploit the fact that many instances are easy to classify. These easy-to-classify instances are handled by simpler classifiers, while harder-to-classify instances use increasingly more complex classifiers. IRA [115] tunes approximation fully at runtime using canary inputs—a small representation of the full input (e.g., a thumbnail of an image). The optimization can tailor the approximation to the input properties, while the overhead is small. This approach has been extended to video processing [236].

Several techniques emerged for optimizing data-parallel and heterogeneous applications, beyond CPU. SAGE [178] and Paraprox [177] pioneered the approximation of data-parallel kernels running on GPUs and provide specialized approximate versions of common idioms, such as maps and reductions. Kernel perforation [132] skips loading chunks of input data from global memory, and reconstructs it in the local memory to improve accuracy, thus saving on data transfers. For composing multiple approximations in heterogenous systems, ApproxHPVM [194] tunes the computation (e.g., a DNN layer) executing it at each processing unit (with its own set of approximation knobs).

7.4 Summary

The techniques discussed above are first steps towards addressing the need for automated tool support for developers (Challenge 3 in Section 11) but they remain limited because each addresses one particular point in the design space. More comprehensive tools and ways to combine the existing techniques are necessary. One solution might be for researchers to make their program analyses and program transformations available as passes for the LLVM compiler infrastructure.

Today, many techniques employ a simplified model of the underlying hardware and these models are rarely based on characterization of real hardware systems. In the future, error models will need to be consistent with the errors observed at the hardware level. In addition to these extensions of the way software-level techniques are evaluated today, end-to-end evaluation platforms could provide increased confidence in research results (Challenges 2 and 8 in Section 11).

8 OPERATING SYSTEM AND RUNTIME TECHNIQUES

Operating system (OS) and runtime techniques for trading correctness for efficiency dynamically monitor a running system and adapt its accuracy to a changing environment. These systems may take explicit input from a program, such as through an application programming interface (API) or system call interface, or might be driven based on user input.

8.1 Notation

For OS and runtime techniques, measuring, sampling, and quantizing signals from the physical world are already completed by the lower layers of the system. Following Section 4, computation is a nondeterministic function $f^* : \mathbb{Q}^m \rightarrow \mathbb{Q}^o$ with nondeterminism introduced by the need to multiplex processes over a shared resource (the processor) in the presence of asynchronous input and output events, user interaction, and time-varying power supply limitations. Actuation typically takes the form of I/O (e.g., network, peripherals, displays).

At the OS/runtime level, the computation specification relation, $f \subseteq \mathbb{I} \times \mathbb{O}$ takes the form of guarantees provided by the system. These may be guarantees and the resulting definition of correctness in terms of the numeric behavior of the computation, or may be guarantees on timeliness of operations in real-time and interactive computing systems. At this layer, faults and errors typically refer to the failure of a component from the architecture level and its manifestation in a difference in machine state respectively.

8.2 Runtime systems: computation

Trading timeliness guarantees for reduced resource usage was heavily explored in the 1990s, in research efforts on *imprecise realtime systems* [11, 91, 123, 124, 195]. Much like the recent resurgence of interest in trading correctness for resource usage, these earlier efforts were targeted at an application domain (embedded systems) where the relaxation of correctness requirements was motivated by the inherent nondeterminism of their operating environments.

The Eon system [200] provides a declarative language which allows users to annotate components with different energy specifications, which are then used at runtime to select suitable components. Green [12] builds a quality of service model during a calibration phase based on approximations supplied by the programmer. This model is used at runtime to select suitable approximations and occasionally to recalibrate by running the approximate and exact subcomputations side-by-side. Hoffman *et al.* [88] turn static configuration parameters into dynamic knobs which can adjust the accuracy and energy usage of a system at runtime; An off-line calibration pass minimizes monitoring overhead at runtime. Follow-up work presents an optimization in the general tradeoff space between performance, energy, and accuracy [86]. Chippa *et al.* [40] propose a general framework which phrases the dynamic management as a feedback system and further suggest different quality measurements at the circuit, architecture, and algorithm level which serve as the feedback signal. Capri [219] proactively determines the configuration of the approximate program before each run, by formulating and solving a constrained optimization problem, which minimizes cost (performance) subject to error (accuracy loss).

To manage noisy data during execution (e.g., from sensors), the Uncertain<T> system [18] encapsulates probability distributions as types. Uncertain data are stored and manipulated as probability distributions, in a way that is transparent to the developer. At each conditional, Uncertain<T> infers the number of samples necessary to make a high-confidence decision.

Most previous runtime approaches consider average errors or only check the errors occasionally during execution, and can thus miss large outliers. Rumba [106] checks all results with light-weight checks and proposes an approximate correction mechanism, which is specific to data-parallel applications. Topaz [1] also verifies every result, but at a higher granularity, by decomposing a computation into tasks. Topaz checks each task's output with lightweight checks provided by the user. If the error is too large, Topaz automatically re-executes the corresponding task.

8.3 Runtime systems: sensors, actuation, and displays

All measurements have some amount of measurement uncertainty and as a result, sensing systems provide many opportunities for trading errors for improved efficiency. These range from trading accuracy and reliability in sensors in the Lax system [213], to trading precision for fidelity in imaging sensors (cameras) [25], to trading the fidelity of display colors and shapes for reduced display panel power dissipation for OLED displays in Crayon [207].

8.4 Summary

OS and runtime techniques provide a unique opportunity to exploit dynamic information about running programs. Unlike circuit-level, microarchitectural, architectural, or language-level techniques, they can exploit information about a user's environment such as remaining energy store in a mobile device or activation of a low-power mode on the device. OS and runtime techniques also have the opportunity to learn across program executions. Hardware platforms for exploring the end-to-end benefits of the techniques presented in this survey (Challenges 2 and 8 in Section 11) may however be necessary for a meaningful evaluation of real-world benefits.

9 TAXONOMY

Table 1 highlights techniques for trading correctness for resource usage discussed throughout this survey. The table focuses on publications that present a specific technique as opposed to publications discussed in the survey to provide context. Table 1 classifies techniques by three primary categories: (1) *error type*, (2) *property traded for errors*, and (3) *affected resources*:

- **The error type** refers to the nature of the error that gets introduced into a system. Given the same input and set of initial conditions, a technique is deterministic if it will always cause the same outcome and a technique is nondeterministic if the outcome can differ.
- **The property traded for errors** is one of *energy*, *time*, and *data density*. These are cost functions that a system designer optimizes for. In the context of this survey, we consider trading an improvement in one or more of these properties for increased occurrence of errors.
- **The affected resources** are the hardware subsystems that are impacted by the tradeoffs. In practice, these will be the subsystems in which errors occur.

Although the table places publications in discrete categories, many techniques lie somewhere in a spectrum. For example, when voltage overscaling (Section 5.6) is performed at a coarse level (e.g., in steps of 500 mV for a device with a supply voltage range of 1.8 V to 3.3 V), it could be seen as a deterministic technique where some voltage levels always lead to repeatable failures. On the other hand, if voltage overscaling is performed at a fine granularity of voltages (e.g., 50 mV), there will likely be one or more voltage levels where nondeterministic failures occur, resulting from the interplay between devices operating right at the threshold of the minimum voltage for reliable operation, and falling below that threshold due to power supply noise or thermal noise.

Table 1. Highlights from the techniques covered in this survey, from circuit-level techniques, to architecture-level techniques, to algorithmic and programming-language-level techniques, to operating system techniques. We classify the techniques under the three broad categories of (1) *error type*, (2) *property traded for errors*, and (3) *affected resources*. (PL: Programming language; OS: Operating system.)

Layer	**Technique**	**Examples**	**Error Type**		**Property Traded for Errors**			**Affected Resource**		
			Deterministic	*Non-Deterministic*	*Energy*	*Runtime*	*Data Density*	*Computation*	*Data Storage/Movement*	*Physical World I/O*
Circuit	Sensor value approximation	[25, 213]		•	•					•
	Probabilistic sensor comms.	[209]	•		•					•
	Probabilistic computing	[31, 35, 67, 107, 156]		•	•			•		
	Stochastic computing (non-det.)	[65, 112]		•		•		•		
	Stochastic computing (det.)	[75, 151]	•			•		•		
	Voltage overscaling	[81, 84, 100, 102, 114]		•	•			•		
	Logic pruning	[136, 189, 197]	•		•	•	•	•		
	Approximate addition	[76, 99, 131, 137, 243]	•		•	•	•	•		
	Approximate multiplication	[80, 95, 113, 170]	•		•	•	•	•		
	RTL approximations	[152, 166, 231]	•		•	•	•	•		
	Approx. high-level synthesis	[118, 120]	•		•	•	•	•		
	Voltage overscaled SRAM	[19, 32, 66, 198]		•	•				•	
	NVRAM noise margins	[167]		•	•				•	
Architecture	Deterministic lossy I/O	[109, 208, 214]	•		•				•	
	Voltage overscaling	[58]		•	•			•	•	
	Analog neural acceleration	[204]		•	•	•		•		
	Digital neural acceleration	[59, 150, 238]	•		•	•		•		
	Anytime computation	[138]	•			•		•		
	Approximate reads	[167]		•	•	•			•	
	Approximate writes	[181]		•	•	•			•	
	Reuse of failed data blocks	[181]		•			•		•	
	Variable redundancy	[64, 74, 94]		•			•		•	
	Approx. cache de-duplication	[183, 184]	•				•		•	
	Load-value approximation	[185, 221]	•		•	•			•	
	Low-refresh DRAM	[98, 125, 165]		•	•				•	
	In-network lossy compression	[22]	•				•		•	
PL	Floating-point optimization	[39, 48, 175]	•			•		•		
	Neural approximation	[58]	•		•	•		•		
	Relaxed parallelization	[28, 140, 142, 171, 172]		•		•		•		
	Compiling to approx. hardware	[29, 139]		•	•			•		
	Data-parallel kernel approx.	[177]	•			•		•		
	Isolation of approx. data	[158, 180]	•	•	•			•	•	
	Algorithm approximation	[93, 188]	•			•		•		
	Loop perforation	[87, 141, 199]	•			•		•		
OS	Display color approximation	[207, 217]	•		•					•
	Drivers for approx. sensors	[213]		•	•					•
	Dynamic accuracy adaptation	[12, 88, 200]	•		•			•		
	Task-level approximation	[1]		•	•			•		

At the circuit level, most techniques in the research literature to date have focused on trading errors for energy efficiency and to a lesser extent, performance and data storage density. At this level of the system stack, the focus has been overwhelmingly on computation resources (e.g., arithmetic/logic units (ALUs)) as the *Affected Resources* columns in Table 1 show.

Most of the architectural techniques in Table 1 target computation at a coarse grain (e.g., analog and digital neural accelerators). A majority of the architectural techniques listed in Table 1 apply to data movement and storage such as on- and off-chip memories, memory hierarchy data traffic, and on- and off-chip I/O links.

Programming language techniques have largely focused only on techniques that affect computation, as the *Affected Resources* columns in Table 1 show. This is unsurprising, since most programming languages focus on providing primitives and abstractions for computations (as opposed to, say, communication). There is potentially an unexplored opportunity to investigate techniques for trading errors for efficiency applied to language-level constructs for communication such as the channels in Hoare's communicating sequential processes (CSP). One early investigation of this direction is the M language, which provided language-level error, erasure, and latency tolerance constraints [210] on CSP-style language-level channels.

Techniques implemented at the operating system (OS) level, such as application programming interfaces (APIs), standard system libraries, device drivers, and so on, have the unique vantage point of seeing all system processes. Techniques at the OS-level often have a global view of the system hardware, and visibility into application behavior beyond a single execution instance. OS-level techniques also have access to information about user preferences, such as activation of a low-power mode on a mobile device. Despite these potential advantages of OS-level techniques for trading errors for efficiency, there have been relatively few techniques implemented at this level of abstraction. The techniques which Table 1 lists target improving energy efficiency and do so primarily by trading the use of sensors, displays, and coarse-grained application level error behavior for improved efficiency.

10 FUNDAMENTAL LIMITS OF RESOURCE VERSUS CORRECTNESS TRADEOFFS

Section 5 through Section 8 presented concrete techniques for trading resource usage for correctness at levels of abstraction ranging from the device-, gate-, and circuit-level, to the operating system. For techniques at each of these levels of abstraction, this article formulated the resource usage versus correctness tradeoff in terms of a computational problem, its implementation in an algorithm, and a distance function d between the state representations of a computation's correct and resource-reduced variants. That relation between a computation's input and output or between a computation's state prior to and subsequent to computation has parallels to communication systems. We can draw an analogy between the state transformation performed by an algorithm which must consume resources (time, energy, die area) to achieve the exact behavior specified by the computational problem which it implements, and source- and channel-coding for communication over a channel: Source- and channel-coding which likewise consume resources in order to maximize the mutual information between the transmitter and receiver over a channel. Von Neumann [233], Berger and Gibson [17], Evans [60], Maggs [43], Elias [56], Spielman [201], and Shanbhag [83], among others, have previously drawn similar analogies between resource usage versus correctness tradeoffs and communication channels. And doing so provides a useful lens through which to study the fundamental limits of resource usage versus correctness tradeoffs in computing systems.

10.1 From information and coding theory to coded computation

The study of fault-tolerant systems dates back to von Neumann's investigation [233] of building reliable systems from unreliable components. Fault-tolerant systems research has focused more heavily on a coarser-grained view. In contrast, *information theory* focuses on the mathematical study of communication over noisy channels [193] while *coding theory* studies methods for judiciously trading redundancy in data representations for either reduced transmission time (source coding) or improved end-to-end reliability in transmission over a noisy channel (channel coding).

In contrast to channel coding techniques whose objective is to counteract the effect of noise, Chen *et al.* [36] exploit the presence of noise to improve image processing tasks, demonstrating how adding Gaussian noise to quantized images can improve the output quality of signal processing tasks. This observation that noise can improve a computing system's performance has parallels to randomized algorithms (see, e.g., Section 9).

Classical information and coding theory rely on the assumption that noise only occurs in communication, rather than in computation. In contrast, recent research has begun to study the fundamental limits of encoders [237] and decoders [226, 240] built on top of hardware implementations that are, like the communication channel, susceptible to noise. Similarly, recent research has investigated techniques for executing computation on encoded representations in order to obtain exact or approximate results in the presence of noise. These methods have been referred to in the research literature as *coded computation* [73, 163]. One plausible direction for future research is to identify computing abstractions that unify the above techniques via new computational operators that execute on encoded representations. Stochastic computing [4], hyper-dimensional computing [161], and deep embedded representations (deep learning) offer promising examples.

10.2 Theoretical bounds

Recent research has used information theory as a foundation to investigate theoretical bounds on performance [239], efficiency [206], energy consumption [33], Shannon-style channel capacity and storage bounds [206, 226] for computing and communication systems which trade resource usage for correctness. Varshney [226] demonstrates Shannon-style bounds on storage capacity in the context of noisy LDPC iterative decoders. Stanley-Marbell [206] derives best-case efficiency bounds for encoding techniques which limit the deviations of values in the presence of logic upsets. Chatterjee *et al.* [33] present lower bounds on energy consumption for achieving a desired level of reliability in computation of an n-input Boolean function and Yazdi *et al.* [239] formulate an optimization problem to produce a noisy Gallager B LDPC decoder that achieves minimal bit error rate, by treating unreliable hardware components as communication channels as in stochastic computing (see Section 5 for coverage of stochastic computing). These recent research efforts demonstrate that information and coding theory can provide a baseline to derive bounds on efficiency, capacity, energy consumption, and performance in the systems of interest in this survey: computing systems which trade resource usage for correctness.

10.3 Application-aware source and channel coding across the hardware stack

Mitigating the effects of errors across the stack will ultimately require encoding techniques, applied across the layers of the stack that are designed to take advantage of application characteristics. Early examples of such *application-aware codes* can be found in the work of Huang *et al.* [90] which proposes a redundancy-free adaptive code that can correct errors in data retrieved from potentially faulty cells. The technique relies on an application-specific cost function and an *input-adaptive coding scheme* that pairs a source encoder that introduces modest distortion, with a channel encoder that adds redundant bits to protect the distorted data against errors. Adaptive coding can greatly reduce output quality degradation in the presence of noise, compared to naïve implementations where noise is allowed to traverse the stack unchecked.

10.4 Summary

Information and coding theory today form the basis for techniques to analyze and model noisy communication and storage systems as well as techniques to counteract the effects of noise. With the emergence of approximate computing, there is an opportunity to investigate new approaches to source and channel coding. These new approaches could explicitly take into account the specific noise distributions observed in practice and could explicitly take into account the requirements of the applications consuming the data in question. These new challenges require a unifying mathematical theory to reason about errors, efficiency, and capacity bounds.

11 CHALLENGES

We identify eight challenges facing both research and applications of techniques to trade correctness for resource usage.

Challenge 1: Holistic cross-layer approaches. A whole-system view to trading errors for efficiency requires expertise in the target application domain and in multiple levels of the computing stack. Most of the existing approximation and error-handling mechanisms are designed in the context of a single layer in the stack. This is likely to be suboptimal. Techniques in different layers can easily negate each other, where gains reported in isolation may not translate into real system-level benefits in the end. At the same time, techniques in different layers may complement each other, where significant opportunities for cross-layer optimizations can be expected. A full-stack view of error-efficient system design requires less insular approaches. A cross-layer approach will however significantly increase the size of the design space and could introduce significant additional design complexity. Early proposals for "approximating outside the processor" [211] and recent proposals for "approximating beyond the processor" [164] are promising directions for holistic approximate systems.

Challenge 2: Hardware models, hardware platforms, and measurement data. Most software-level techniques employ models or abstractions of the errors and performance of the underlying hardware in order to achieve modularity and scalability. Examples of hardware error models assumed today include assumptions about the distribution of locations and values of errors caused by voltage overscaling in microarchitectural structures and memories, or assumptions about the distribution of errors in DRAMs that are not refreshed as regularly as they should be. Similarly, lower levels of the software stack may expose higher-level models to, e.g., application developers. Today, different research efforts often use different models, which makes comparisons between research results difficult and raises questions about the validity of reported resource savings. Error and performance models which have been validated by the hardware community, e.g., by hardware measurements and which are suitable for the software levels of the stack would be an invaluable contribution to the research directions described in this survey. In order to make credible claims about across-the-stack approximations, the proposed techniques need to be evaluated end-to-end either on actual state-of-the-art hardware platforms or with realistic simulations. Such end-to-end evaluations with an agreed-upon platform is missing today. Early examples in this direction include measurement results from open hardware platforms explicitly designed to expose accuracy, precision, performance, and energy efficiency tradeoffs [216, 218].

Challenge 3: Hardware emulation/simulation, software tools, languages, and compiler infrastructure. Applying error versus resource tradeoffs in software requires tools that help programmers and systems builders take advantage of techniques in a systematic and controlled way. It also requires hardware simulators or emulators that help bridge the gap between the fidelity of hardware prototypes and the flexibility of software simulation. On the hardware simulation side, these tools would ideally provide support for end-to-end evaluation of entire systems, to be used in comparing different proposed techniques. Language and compiler tools would include those to support testing, debugging, and dynamic quality monitoring. First steps in this direction for compiler tools include ACCEPT [179].

Challenge 4: Application domains and algorithmic patterns. Today, there is insufficient consensus on well-defined classes of application domains and algorithmic patterns that constitute a preferential target for relaxations. First steps include the definition of "Recognition, Mining, and Synthesis" application classes [54]. A standard benchmark set which has been agreed upon by the wider community would increase progress and comparability of different techniques, like it has done for SAT/SMT solving. Such a benchmark set should ideally cover different domains and include also real-world 'challenge applications' which cannot be solved today, but which would convincingly demonstrate the viability of error-efficient computing.

Challenge 5: Large-scale user studies to provide empirical characterization of acceptability. User studies with thousands of participants will be necessary to provide quantitative data [126], which researchers can use when proposing techniques that exploit tolerance of human observers to deviations from correctness of program results. Initial steps in this direction include the "Specimen" dataset of color perception data used in color approximation techniques [27, 217].

Challenge 6: Metrics. When applying techniques to an application, it would be useful to have a reliable error metric to guide the optimization process. In the ideal case, that error metric would be binary: "correct" and "not correct." But in reality, correctness and its boundaries are not well known for many applications. The metrics in question might be broadly applicable to many systems, or might be application-specific metrics used to measure Pareto-optimality. Early work in this direction includes the work of Akturk *et al.* [3].

Challenge 7: Studies of the theoretical bounds on resource usage. Theoretical upper and lower bounds are invaluable in guiding research as they set the bar for what is achievable. Such bounds are needed for a given formally-defined specification of deviation from correctness. One promising direction are bounds on encoding overhead for communication encoding techniques which trade communication energy use or performance (data rate) for integer deviations in communicated values. Examples of first steps in this direction include work on the bounds of encoding efficiency for deterministic and probabilistic approximate communication techniques [206, 212].

Challenge 8: Reproducibility and deployment of techniques. This survey describes a broad range of techniques, from circuits to algorithms. For many of the research results across these layers, it can be challenging to replicate the setup, tools, or benchmarks employed in evaluations. Beyond good scientific practice of describing experiments in sufficient detail to be reproducible, because there is today no common consensus even on many aspects of terminology, it is challenging to compare, replicate, and build upon research results. This survey attempted to address the challenge of terminology with a consistent set of formal definitions across the layers of a system (Section 4). Further progress is however needed. Opportunities include greater availability of open-source libraries. First steps for low-power and high-performance approximate arithmetic components include synthesizable Verilog/VHDL files and behavioral models in C/MATLAB [190, 191].

12 RETROSPECTIVE AND FUTURE DIRECTIONS

Computing systems use resources such as time, energy, and hardware to transform their inputs to outputs. For many years, the primary driver of efficiency improvements in computing were a combination of improved semiconductor process technology and better algorithms. In the last decade, two important shifts have forced a fundamental re-evaluation of the hardware driver of efficiency improvements. First, with the cessation of constant-field Dennard scaling and the stagnation of device supply voltages, process technology scaling no longer provides the improvements in energy efficiency that it once did. Second, in contrast to traditional computing applications such as financial transaction processing and office productivity, the dominant computing system applications are increasingly driven by inputs from the physical world (audio, images) with outputs for consumption by humans. Applications driven by data from the physical world have essentially unbounded input datasets, and this has partly motivated a resurgence of interest in machine learning approaches to learning functions from large datasets. The stagnation in hardware device-level improvements coupled with increasingly ever more abundant sensor-data-centric workloads has led to a need for new ways of improving computing system performance. This survey explored techniques to address this challenge of computing on data when less-than-perfect outputs are acceptable for a computing system's users.

ACKNOWLEDGEMENTS

This article is the result of a four-day workshop held in April 2017 organized by Phillip Stanley-Marbell, Adrian Sampson, and Babak Falsafi. Many of the ideas in the survey resulted from discussions among the entire group of participants (in alphabetical order): Sara Achour, Armin Alaghi, Mattia Cacciotti, Michael Carbin, Alexandros Daglis, Eva Darulova, Lara Dolecek, Mario Drumond, Natalie Enright Jerger, Babak Falsafi, Andreas Gerstlauer, Ghayoor Gillani, Djordje Jevdjic, Sasa Misailovic, Thierry Moreau, Adrian Sampson, Phillip Stanley-Marbell, Radha Venkatagiri, Naveen Verma, Marina Zapater, and Damien Zufferey. The workshop was funded by the Swiss National Science Foundation (SNF) under grant *IZ32Z0_173393/1: "International Exploratory Workshop on Theory and Practice for Error-Efficient Computing"*, and by the EPFL EcoCloud Research Center. Ghayoor Gillani thanks A.B.J. Kokkeler (CAES group, University of Twente) for his encouragement and suggestions to contribute towards this article. Phillip Stanley-Marbell is supported by an Alan Turing Institute award TU/B/000096 under EPSRC grant EP/N510129/1, by Royal Society grant RG170136, and by EPSRC grants EP/P001246/1 and EP/R022534/1.

REFERENCES

[1] Sara Achour and Martin C. Rinard. 2015. Approximate computation with outlier detection in topaz. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 711–730. **(Cited on pages 23 and 24.)**

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the European Conference on Computer Systems*. 29–42. **(Cited on page 5.)**

[3] Ismail Akturk, Karen Khatamifard, and Ulya R Karpuzcu. 2015. On quantification of accuracy loss in approximate computing. In *Workshop on Duplicating, Deconstructing and Debunking (WDDD)*. 15. **(Cited on page 28.)**

[4] Armin Alaghi and John P. Hayes. 2013. Survey of stochastic computing. *ACM Transactions on Embedded computing systems (TECS)* 12, 2s (2013), 92:1–92:19. **(Cited on pages 13 and 26.)**

[5] Armin Alaghi and John P. Hayes. 2014. Fast and accurate computation using stochastic circuits. In *Proceedings of the Design, Automation Test in Europe Conference (DATE)*. 1–4. **(Cited on page 13.)**

[6] Armin Alaghi, Weikang Qian, and John P. Hayes. 2017. The promise and challenge of stochastic computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017). **(Cited on page 13.)**

[7] Rajeevan Amirtharajah and Anantha P. Chandrakasan. 2004. A micropower programmable DSP using approximate signal processing based on distributed arithmetic. *IEEE Journal of Solid-State Circuits* 39, 2 (2004), 337–347. **(Cited on pages 2 and 4.)**

[8] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A language and compiler for algorithmic choice. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 38–49. **(Cited on page 21.)**

[9] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 85–96. **(Cited on page 21.)**

[10] Alexander Aponte-Moreno, Alejandro Moncada, Felipe Restrepo-Calle, and Cesar Pedraza. 2018. A review of approximate computing techniques towards fault mitigation in HW/SW systems. In *Proceedings of the Latin-American Test Symposium (LATS)*. 1–6. **(Cited on page 3.)**

[11] Hakan Aydın, Rami Melhem, and Daniel Mossé. 1999. Incorporating error recovery into the imprecise computation model. In *Proceedings of the International Conference on Real-Time Computing Systems and Applications (RTCSA)*. 348–355. **(Cited on page 22.)**

[12] Woongki Baek and Trishul M. Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 198–209. **(Cited on pages 21, 22, and 24.)**

[13] Nilanjan Banerjee, Georgios Karakonstantis, and Kaushik Roy. 2007. Process variation tolerant low power DCT architecture. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–6. **(Cited on page 14.)**

[14] Benjamin Barrois, Olivier Sentieys, and Daniel Menard. 2017. The hidden cost of functional approximation against careful data sizing—A case study. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*. 181–186. **(Cited on page 15.)**

[15] Luiz A. Barroso and Urs Hölzle. 2007. The case for energy-proportional computing. *IEEE Computer* 40, 12 (2007), 33–37. **(Cited on page 15.)**

[16] Charles H Bennett and Rolf Landauer. 1985. The fundamental physical limits of computation. *Scientific American* 253, 1 (1985), 48–56. **(Cited on page 2.)**

[17] Berger and Gibson. 1998. Lossy Source Coding. *IEEE Transactions on Information Theory* 44 (1998). **(Cited on page 25.)**

[18] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. 2014. Uncertain<T>: A first-order type for uncertain data. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 51–66. **(Cited on page 23.)**

[19] Daniele Bortolotti, Hossein Mamaghanian, Andrea Bartolini, Maryam Ashouei, Jan Stuijt, David Atienza, Pierre Vandergheynst, and Luca Benini. 2014. Approximate compressed sensing: ultra-low power biosignal processing via aggressive voltage scaling on a hybrid memory multi-core processor. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 45–50. **(Cited on pages 15 and 24.)**

[20] Brett Boston, Zoe Gong, and Michael Carbin. 2018. Leto: verifying application-specific hardware fault tolerance with programmable execution models. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30. **(Cited on page 19.)**

[21] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. 2015. Probability type inference for flexible approximate programming. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 470–487. **(Cited on page 20.)**

[22] Rahul Boyapati, Jiayi Huang, Pritam Majumder, Ki Hwan Yum, and Eun Jung Kim. 2017. APPROX-NoC: A data approximation framework for network-on-chip architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 666–677. **(Cited on pages 18 and 24.)**

[23] Melvin Breuer. 2005. Multi-media applications and imprecise computation. In *Proceedings of the Euromicro Conference on Digital System Design*. 2–7. **(Cited on page 2.)**

[24] Michael Bromberger, Wolfgang Karl, and Vincent Heuveline. 2015. Exploiting approximate computing methods in FPGAs to accelerate stereo correspondence algorithms. In *Workshop On Approximate Computing*. **(Cited on page 4.)**

[25] Mark Buckler, Suren Jayasuriya, and Adrian Sampson. 2017. Reconfiguring the imaging pipeline for computer vision. In *Proceedings of the International Conference on Computer Vision (ICCV)*. **(Cited on pages 23 and 24.)**

[26] Richard L. Burden, J. Douglas Faires, and Annette M. Burden. 2015. *Numerical Analysis*. Brooks Cole Pub Co. **(Cited on page 4.)**

[27] Jose Cambronero, Phillip Stanley-Marbell, and Martin Rinard. 2018. Incremental color quantization for color-vision-deficient observers using mobile gaming data. *CoRR* abs/1803.08420 (2018). arXiv:1803.08420 **(Cited on page 28.)**

[28] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. 2015. HELIX-UP: Relaxing Program Semantics to Unleash Parallelization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE/ACM, 235–245. **(Cited on pages 21 and 24.)**

[29] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. 2012. Proving Acceptability Properties of Relaxed Nondeterministic Approximate Programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 169–180. **(Cited on pages 19 and 24.)**

[30] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 33–52. **(Cited on page 20.)**

[31] Lakshmi N Chakrapani, Pinar Korkmaz, Bilge ES Akgul, and Krishna V Palem. 2007. Probabilistic system-on-a-chip architectures. *ACM Transactions on Design Automation of Electronic Systems* 12, 3 (2007), 29. **(Cited on pages 13 and 24.)**

[32] Ik Joon Chang, Debabrata Mohapatra, and Kaushik Roy. 2011. A priority-based 6T/8T hybrid SRAM architecture for aggressive voltage scaling in video applications. *IEEE transactions on circuits and systems for video technology* 21, 2 (2011), 101–112. **(Cited on pages 15 and 24.)**

[33] Avhishek Chatterjee and Lav R Varshney. 2016. Energy-reliability limits in nanoscale circuits. In *Proceedings of Information Theory and Applications Workshop (ITA '16)*. 1–6. **(Cited on page 26.)**

[34] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. 2011. Proving Programs Robust. In *Proceedings of the Symposium and the European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. 102–112. **(Cited on pages 20 and 21.)**

[35] Suresh Cheemalavagu, Pinar Korkmaz, Krishna V Palem, Bilge ES Akgul, and Lakshmi N Chakrapani. 2005. A probabilistic CMOS switch and its realization by exploiting noise. In *Proceedings of International Conference on VLSI*. IFIP, 535–541. **(Cited on pages 13 and 24.)**

[36] Hao Chen, Lav R Varshney, and Pramod K Varshney. 2014. Noise-enhanced information systems. *Proc. IEEE* 102, 10 (2014), 1607–1621. **(Cited on page 25.)**

[37] Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. 2015. Compressing Neural Networks with the Hashing Trick. *CoRR* abs/1504.04788 (2015). arXiv:1504.04788 **(Cited on page 5.)**

[38] Wei-Chung Cheng and Massoud Pedram. 2001. Memory bus encoding for low power: a tutorial. In *Proceedings of International Symposium on Quality Electronic Design (ISQED '01)*. IEEE, 199–204. **(Cited on page 18.)**

[39] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous Floating-point Mixed-precision Tuning. In *Proceedings of Principles of Programming Languages (POPL)*. **(Cited on pages 20 and 24.)**

[40] Vinay Chippa, Anand Raghunathan, Kaushik Roy, and Srimat Chakradhar. 2011. Dynamic effort scaling: Managing the quality-efficiency tradeoff. In *Design Automation Conference (DAC '11)*. ACM/EDAC/IEEE, 603–608. **(Cited on page 22.)**

[41] Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the Design Automation Conference (DAC '13)*. ACM/EDAC/IEEE, 113. **(Cited on page 20.)**

[42] Vinay K. Chippa, Debabrata Mohapatra, Anand Raghunathan, Kaushik Roy, and Srimat T. Chakradhar. 2010. Scalable Effort Hardware Design: Exploiting Algorithmic Resilience for Energy Efficiency. In *Proceedings of the Design Automation Conference (DAC '10)*. ACM/EDAC/IEEE, 555–560. **(Cited on page 21.)**

[43] R. J. Cole, B. M. Maggs, and R. K. Sitaraman. 1997. Reconfiguring arrays with faults part I: worst-case faults. *SIAM J. Comput.* 26, 6 (December 1997), 1581–1611. **(Cited on page 25.)**

[44] Christian Constanda. 2017. *Numerical Methods. In: Differential Equations. Springer Undergraduate Texts in Mathematics and Technology*. Springer. **(Cited on page 4.)**

[45] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *CoRR* abs/1511.00363 (2015). arXiv:1511.00363 **(Cited on page 5.)**

[46] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*. 3123–3131. **(Cited on page 8.)**

[47] Germund Dahlquist and Åke Björk. 2008. *Numerical Methods in Scientific Computing*. Society for Industrial and Applied Mathematics. **(Cited on page 4.)**

[48] Eva Darulova, Einar Horn, and Saksham Sharma. 2018. Sound Mixed-precision Optimization with Rewriting. In *Proceedings of the International Conference on Cyber-Physical Systems (ICCPS)*. **(Cited on pages 20 and 24.)**

[49] Enrico A Deiana, Vincent St-Amour, Peter A Dinda, Nikos Hardavellas, and Simone Campanoni. 2018. Unconventional parallelization of nondeterministic applications. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 432–447. **(Cited on page 21.)**

[50] Emily Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. 2014. Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation. *CoRR* abs/1404.0736 (2014). arXiv:1404.0736 **(Cited on page 5.)**

[51] J. A. Dickson, R. D. McLeod, and H. C. Card. 1993. Stochastic arithmetic implementations of neural networks with in situ learning. In *Proceedings of the International Conference on Neural Networks*. IEEE, 711–716. **(Cited on page 13.)**

[52] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman P. Amarasinghe. 2015. Autotuning algorithmic choice for input sensitivity. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 379–390. **(Cited on page 21.)**

[53] Zidong Du, Avinash Lingamneni, Yunji Chen, Krishna Palem, Olivier Temam, and Chengyong Wu. 2014. Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators. In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC '14)*. IEEE, 201–206. **(Cited on page 8.)**

[54] Pradeep Dubey. 2005. Recognition, mining and synthesis moves computers to the era of tera. *Technology@ Intel Magazine* 9, 2 (2005), 1–10. **(Cited on page 27.)**

[55] Bo Einarsson. 2005. *Accuracy and Reliability in Scientific Computing*. Society for Industrial and Applied Mathematics. **(Cited on page 4.)**

[56] Peter Elias. 1958. Computation in the presence of noise. *IBM Journal of Research and Development* 2, 4 (1958), 346–353. **(Cited on page 25.)**

[57] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. 2003. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *Proceedings of the International Symposium on Microarchitecture (MICRO-36)*. IEEE/ACM. **(Cited on pages 15 and 16.)**

[58] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, 301–312. **(Cited on pages 16, 17, 21, and 24.)**

[59] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the International Symposium on Microarchitecture (MICRO-45)*. 449–460.

(**Cited on pages 16 and 24.**)

[60] William Evans and Nicholas Pippenger. 1998. On The Maximum Tolerable Noise for Reliable Computation by Formulas. *IEEE Transactions on Information Theory* 44, 3 (1998), 1299–1305. (**Cited on page 25.**)

[61] Shuangde Fang, Zidong Du, Yuntan Fang, Yuanjie Huang, Yang Chen, Lieven Eeckhout, Olivier Temam, Huawei Li, Yunji Chen, and Chengyong Wu. 2014. Performance Portability Across Heterogeneous SoCs Using a Generalized Library-Based Approach. *ACM Trans. Archit. Code Optim.* 11, 2, Article 21 (June 2014), 21:1–21:25 pages. (**Cited on page 21.**)

[62] Vimuth Fernando, Keyur Joshi, and Sasa Misailovic. 2019. Verifying safety and accuracy of approximate parallel programs via canonical sequentialization. *Proceedings of the ACM on Programming Languages* OOPSLA (2019), 1–29. (**Cited on page 20.**)

[63] Mikhail Figurnov, Aizhan Ibraimova, Dmitry P Vetrov, and Pushmeet Kohli. 2016. PerforatedCNNs: Acceleration through elimination of redundant convolutions. In *Advances in Neural Information Processing Systems*. 947–955. (**Cited on page 21.**)

[64] Daichi Fujiki, Kiyo Ishii, Ikki Fujiwara, Hiroki Matsutani, Hideharu Amano, Henri Casanova, and Michihiro Koibuchi. 2017. High-Bandwidth Low-Latency Approximate Interconnection Networks. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'17)*. 469–480. (**Cited on pages 18 and 24.**)

[65] Brian R Gaines. 1967. Stochastic computing. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 149–156. (**Cited on pages 13 and 24.**)

[66] Shrikanth Ganapathy, Georgios Karakonstantis, Adam Teman, and Andreas Burg. 2015. Mitigating the Impact of Faults in Unreliable Memories for Error-Resilient Applications. In *Proceedings of the Design Automation Conference (DAC'15)*. ACM, 1–6. (**Cited on pages 15 and 24.**)

[67] Jason George, Bo Marr, Bilge ES Akgul, and Krishna V Palem. 2006. Probabilistic Arithmetic and Energy Efficient Embedded Signal Processing. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*. ACM, 158–168. (**Cited on pages 13 and 24.**)

[68] GA Gillani and Andre BJ Kokkeler. 2017. Improving Error Resilience Analysis Methodology of Iterative Workloads for Approximate Computing. In *Proceedings of the Computing Frontiers Conference (CF'17)*. ACM, 374–379. (**Cited on page 20.**)

[69] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. 2015. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, 383–397. (**Cited on page 5.**)

[70] Oded Goldreich. 2017. *Introduction to property testing*. Cambridge University Press. (**Cited on page 5.**)

[71] Gene H. Golub and James M. Ortega. 2014. *Scientific Computing and Differential Equations: an Introduction to Numerical Methods*. Elsevier. (**Cited on page 4.**)

[72] Minglun Gong, Ruigang Yang, Liang Wang, and Mingwei Gong. 2007. A Performance Study on Different Cost Aggregation Approaches used in Real-Time Stereo Matching. *International Journal of Computer Vision* 75, 2 (2007), 283–296. (**Cited on page 4.**)

[73] Pulkit Grover. 2014. Is "Shannon-capacity of noisy computing" zero?. In *International Symposium on Information Theory (ISIT'14)*. IEEE, 2854–2858. (**Cited on page 26.**)

[74] Qing Guo, Karin Strauss, Luis Ceze, and Henrique S Malvar. 2016. High-Density Image Storage Using Approximate Memory Cells. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. 413–426. (**Cited on pages 16, 17, and 24.**)

[75] Prabhat K. Gupta and Ramdas Kumaresan. 1988. Binary multiplication with PN sequences. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 36, 4 (April 1988), 603–606. (**Cited on pages 13 and 24.**)

[76] Vaibhav Gupta, Debabrata Mohapatra, Anand Raghunathan, and Kaushik Roy. 2013. Low-Power Digital Signal Processing using Approximate Adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 1 (2013), 124–137. (**Cited on pages 14 and 24.**)

[77] Jie Han and Michael Orshansky. 2013. Approximate Computing: An Emerging Paradigm for Energy-Efficient Design. In *European Test Symposium (ETS) (ETS'13)*. IEEE, 1–6. (**Cited on page 3.**)

[78] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the International Symposium on Computer Architecture (ISCA '16)*. IEEE, 243–254. (**Cited on page 9.**)

[79] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015). arXiv:1510.00149 (**Cited on page 5.**)

[80] Soheil Hashemi, R Iris Bahar, and Sherief Reda. 2015. DRUM: A dynamic range unbiased multiplier for approximate applications. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 418–425. (**Cited on pages 14 and 24.**)

[81] Ku He, Andreas Gerstlauer, and Michael Orshansky. 2013. Circuit-Level Timing-Error Acceptance for Design of Energy-Efficient DCT/IDCT-Based Systems. *Transactions on Circuits and Systems for Video Technology* 23, 6 (2013), 961–974. **(Cited on pages 14 and 24.)**

[82] Shaobo He, Shuvendu K. Lahiri, and Zvonimir Rakamarić. 2018. Verifying Relative Safety, Accuracy, and Termination for Program Approximations. *Journal of Automated Reasoning* (2018), 23–42. **(Cited on page 20.)**

[83] Rajamohana Hegde and Naresh R. Shanbhag. 1999. Energy-efficient Signal Processing via Algorithmic Noise-tolerance. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '99)*. ACM, 30–35. **(Cited on pages 4 and 25.)**

[84] Rajamohana Hegde and Naresh R. Shanbhag. 2001. Soft Digital Signal Processing. *IEEE Transactions on VLSI Systems* 9, 6 (2001), 813–823. **(Cited on pages 14 and 24.)**

[85] Nicholas Higham. 2002. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics. **(Cited on page 4.)**

[86] Henry Hoffmann. 2015. JouleGuard: energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 198–214. **(Cited on page 22.)**

[87] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. 2009. Using code perforation to improve performance, reduce energy consumption, and respond to failures. *MIT-CSAIL-TR-2009-042* (2009). **(Cited on pages 21 and 24.)**

[88] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic Knobs for Responsive Power-aware Computing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, 199–212. **(Cited on pages 22 and 24.)**

[89] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 **(Cited on page 5.)**

[90] Chu-Hsiang Huang, Yao Li, and Lara Dolecek. 2015. ACOCO: Adaptive Coding for Approximate Computing on Faulty Memories. *IEEE Transactions on Communications* 63, 12 (2015), 4615–4628. **(Cited on pages 5 and 26.)**

[91] David Hull and Jane Liu. 1993. ICS: A System for Imprecise Computations. In *Proc. AIAA Computing in Aerospace*, Vol. 9. **(Cited on page 22.)**

[92] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). arXiv:1602.07360 **(Cited on page 5.)**

[93] Anastasiia Izycheva, Eva Darulova, and Helmut Seidl. 2019. Synthesizing Efficient Low-Precision Kernels. In *Automated Technology for Verification and Analysis - 17th International Symposium (ATVA)*. 294–313. **(Cited on pages 20 and 24.)**

[94] Djordje Jevdjic, Karin Strauss, Luis Ceze, and Henrique S Malvar. 2017. Approximate Storage of Compressed and Encrypted Videos. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. **(Cited on pages 16, 17, and 24.)**

[95] Honglan Jiang, Cong Liu, Naman Maheshwari, Fabrizio Lombardi, and Jie Han. 2016. A comparative evaluation of approximate multipliers. In *IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*. IEEE, 191–196. **(Cited on pages 14 and 24.)**

[96] Keyur Joshi, Vimuth Fernando, and Sasa Misailovic. 2019. Statistical algorithmic profiling for randomized approximate programs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 608–618. **(Cited on page 21.)**

[97] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *CoRR* abs/1704.04760 (2017). arXiv:1704.04760 **(Cited on page 5.)**

[98] Matthias Jung, Deepak M. Mathew, Christian Weis, and Norbert Wehn. 2016. Approximate Computing with Partially Unreliable Dynamic Random Access Memory - Approximate DRAM. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, Article 100, 4 pages. **(Cited on pages 15 and 24.)**

[99] Andrew B. Kahng and Seokhyeong Kang. 2012. Accuracy-configurable Adder for Approximate Arithmetic Designs. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. ACM, 820–825. **(Cited on pages 14 and 24.)**

[100] Andrew B. Kahng, Seokhyeong Kang, Rakesh Kumar, and John Sartori. 2010. Slack Redistribution for Graceful Degradation Under Voltage Overscaling. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference (ASPDAC '10)*. IEEE Press, 825–831. (**Cited on pages 14 and 24.**)

[101] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 631–646. (**Cited on page 5.**)

[102] Georgios Karakonstantis, Debabrata Mohapatra, and Kaushik Roy. 2009. System Level DSP Synthesis Using Voltage Overscaling, Unequal Error Protection and Adaptive Quality Tuning. In *IEEE Workshop on Signal Processing Systems (SIPS)*. (**Cited on pages 14 and 24.**)

[103] Walter J Karplus and Walter W Soroka. 1959. *Analog Methods: Computation and Simulation*. McGraw-Hill. (**Cited on page 12.**)

[104] R Baker Kearfott, Mitsuhiro T Nakao, Arnold Neumaier, Siegfried M Rump, Sergey P Shary, and Pascal van Hentenryck. 2010. Standardized notation in interval analysis. *Computational Technologies* 15, 1 (2010), 7–13. (**Cited on page 4.**)

[105] Robert W Keyes. 1985. What makes a good computer device? *Science* 230, 4722 (1985), 138–144. (**Cited on page 2.**)

[106] Daya S. Khudia, Babak Zamirai, Mehrzad Samadi, and Scott Mahlke. 2015. Rumba: An Online Quality Management System for Approximate Computing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, 554–566. (**Cited on page 23.**)

[107] Jaeyoon Kim and Sandip Tiwari. 2014. Inexact computing using probabilistic circuits: Ultra low-power digital processing. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 10, 2 (2014), 16. (**Cited on pages 13 and 24.**)

[108] Sung Kim, Patrick Howe, Thierry Moreau, Armin Alaghi, Luis Ceze, and Sathe Visvesh. 2018. MATIC: Learning Around Errors for Efficient Low-Voltage Neural Network Accelerators. In *Design Automation and Test in Europe Conference (DATE)*. (**Cited on page 9.**)

[109] Younghyun Kim, Setareh Behroozi, Vijay Raghunathan, and Anand Raghunathan. 2017. AXSERBUS: A quality-configurable approximate serial bus for energy-efficient sensing. In *Low Power Electronics and Design (ISLPED, 2017 IEEE/ACM International Symposium on*. IEEE, 1–6. (**Cited on pages 18 and 24.**)

[110] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2015. Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications. *CoRR* abs/1511.06530 (2015). arXiv:1511.06530 (**Cited on page 5.**)

[111] Young-Chul Kim and M. A. Shanblatt. 1995. Architecture and statistical model of a pulse-mode digital multilayer neural network. *IEEE Transactions on Neural Networks* 6, 5 (Sep 1995), 1109–1118. (**Cited on page 13.**)

[112] Phil Knag, Wei Lu, and Zhengya Zhang. 2014. A Native Stochastic Computing Architecture Enabled by Memristors. *IEEE Transactions on Nanotechnology* 13, 2 (March 2014), 283–293. (**Cited on pages 13 and 24.**)

[113] Parag Kulkarni, Puneet Gupta, and Milos Ercegovac. 2011. Trading accuracy for power with an underdesigned multiplier architecture. In *VLSI Design (VLSI Design), 2011 24th International Conference on*. IEEE, 346–351. (**Cited on pages 14 and 24.**)

[114] F. Kurdahi, A. Eltawil, K. Yi, S. Cheng, and A. Khajeh. 2010. Low-Power Multimedia System Design by Aggressive Voltage Scaling. *IEEE Transactions on VLSI Systems* 18, 5 (2010), 852–856. (**Cited on pages 14 and 24.**)

[115] Michael A Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. 2016. Input responsiveness: using canary inputs to dynamically steer approximation. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 161–176. (**Cited on page 21.**)

[116] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. (**Cited on page 8.**)

[117] Yann LeCun, John S Denker, and Sara A Solla. 1990. Optimal brain damage. In *Advances in neural information processing systems*. 598–605. (**Cited on page 8.**)

[118] Seogoo Lee, Lizy K. John, and Andreas Gerstlauer. 2017. High-level synthesis of approximate hardware under joint precision and voltage scaling. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE '17)*. IEEE, 187–192. (**Cited on pages 14 and 24.**)

[119] Vincent T. Lee, Armin Alaghi, John P. Hayes, Visvesh Sathe, and Luis Ceze. 2017. Energy-efficient hybrid stochastic-binary neural networks for near-sensor computing. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE '17)*. IEEE, 13–18. (**Cited on page 13.**)

[120] Chaofan Li, Wei Luo, Sachin S Sapatnekar, and Jiang Hu. 2015. Joint precision optimization and high level synthesis for approximate computing. In *52nd ACM/EDAC/IEEE Design Automation Conference (DAC '15)*. IEEE, 1–6. (**Cited on pages 14 and 24.**)

[121] Shikai Li, Sunghyun Park, and Scott Mahlke. 2018. Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation. In *Proceedings of the 2018 International Conference on Supercomputing*. 341–351. (**Cited on page 21.**)

[122] Avinash Lingamneni, Christian Enz, Krishna Palem, and Christian Piguet. 2011. Parsimonious circuits for error-tolerant applications through probabilistic logic minimization. In *Proceedings of the 21st international Conference on Integrated Circuit and System Design: Power and Timing modeling, Optimization, and Simulation (PATMOS '11)*. Springer, 204–213. (**Cited on page 14.**)

[123] Jane W. S. Liu, Kwei-Jay Lin, Wei-Kuan Shih, Albert Chuang-shi Yu, Jen-Yao Chung, and Wei Zhao. 1991. Algorithms for scheduling imprecise computations. *Computer* 24, 5 (1991), 58–68. (**Cited on page 22.**)

[124] Jane W. S. Liu, Wei-Kuan Shih, Kwei-Jay Lin, Ricardo Bettati, and Jen-Yao Chung. 1994. Imprecise Computations. *Proc. IEEE* 82, 1 (1994), 83–94. (**Cited on page 22.**)

[125] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin Zorn. 2011. Flikker: Saving DRAM refresh-power through critical data partition. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. ACM. (**Cited on pages 16, 17, and 24.**)

[126] Jeffrey W. Lockhart, Gary M. Weiss, Jack C. Xue, Shaun T. Gallagher, Andrew B. Grosner, and Tony T. Pulickal. 2011. Design considerations for the WISDM smart phone-based sensor mining architecture. In *Proceedings of the Fifth International Workshop on Knowledge Discovery from Sensor Data (SensorKDD '11)*. ACM, 25–33. (**Cited on page 28.**)

[127] Debasmita Lohar, Eva Darulova, Sylvie Putot, and Eric Goubault. 2018. Discrete Choice in the Presence of Numerical Uncertainties. *IEEE Trans. on CAD of Integrated Circuits and Systems* 37, 11 (2018), 2381–2392. (**Cited on page 20.**)

[128] Debasmita Lohar, Milos Prokop, and Eva Darulova. 2019. Sound Probabilistic Numerical Error Analysis. In *15th International Conference on Integrated Formal Methods (IFM)*. 322–340. (**Cited on page 20.**)

[129] Li-ming Lou, Paul Nguyen, Jason Lawrence, and Connelly Barnes. 2016. Image Perforation: Automatically Accelerating Image Pipelines by Intelligently Skipping Samples. *ACM Trans. Graph.* 35, 5 (2016), 153:1–153:14. (**Cited on page 21.**)

[130] Bruce J MacLennan. 2009. Analog computation. In *Encyclopedia of complexity and systems science*. Springer, 271–294. (**Cited on page 12.**)

[131] Hamid Reza Mahdiani, Ali Ahmadi, Sied Mehdi Fakhraie, and Caro Lucas. 2010. Bio-inspired imprecise computational blocks for efficient VLSI implementation of soft-computing applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* 57, 4 (2010), 850–862. (**Cited on pages 14 and 24.**)

[132] Daniel Maier, Biagio Cosenza, and Ben Juurlink. 2018. Local memory-aware kernel perforation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 278–287. (**Cited on page 21.**)

[133] Igor L. Markov. 2014. Limits on fundamental limits to computation. *Nature* 512 (08 2014), 147–54. (**Cited on page 2.**)

[134] Mark M. Meerschaert. 2013. *Mathematical modeling*. Academic Press. (**Cited on page 4.**)

[135] Judicaël Menant, Muriel Pressigout, Luce Morin, and Jean-Francois Nezan. 2014. Optimized fixed point implementation of a local stereo matching algorithm onto C66x DSP. In *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP '14)*. IEEE, 1–6. (**Cited on page 4.**)

[136] Jin Miao, Andreas Gerstlauer, and Michael Orshansky. 2013. Approximate logic synthesis under general error magnitude and frequency constraints. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '13)*. IEEE, 779–786. (**Cited on pages 14 and 24.**)

[137] Jin Miao, Ku He, Andreas Gerstlauer, and Michael Orshansky. 2012. Modeling and synthesis of quality-energy optimal approximate adders. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '12)*. IEEE, 728–735. (**Cited on pages 14 and 24.**)

[138] Joshua San Miguel and Natalie Enright Jerger. 2016. The anytime automaton. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. 545–557. (**Cited on page 24.**)

[139] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, 309–328. (**Cited on pages 20 and 24.**)

[140] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. 2013. Parallelizing sequential programs with statistical accuracy tests. *ACM Transactions on Embedded Computer Systems* 12, 2s, Article 88 (2013), 88:1–88:26 pages. (**Cited on pages 21 and 24.**)

[141] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. 2010. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, 25–34. (**Cited on pages 20, 21, and 24.**)

[142] Sasa Misailovic, Stelios Sidiroglou, and Martin C. Rinard. 2012. Dancing with uncertainty. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES '12)*. ACM, 51–60. (**Cited on pages 21 and 24.**)

[143] Asit K. Mishra, Rajkishore Barik, and Somnath Paul. 2014. iACT: A software-hardware framework for understanding the scope of approximate computing. In *Workshop on Approximate Computing Across the System Stack (WACAS '14)*. (**Cited on page 21.**)

[144] Subrata Mitra, Manish K Gupta, Sasa Misailovic, and Saurabh Bagchi. 2017. Phase-aware optimization in approximate computing. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 185–196. **(Cited on page 21.)**

[145] Sparsh Mittal. 2016. A survey of techniques for approximate computing. *Comput. Surveys* 48, 4, Article 62 (2016), 62:1–62:33 pages. **(Cited on page 3.)**

[146] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press. **(Cited on page 5.)**

[147] Debabrata Mohapatra, Georgios Karakonstantis, and Kaushik Roy. 2009. Significance driven computation: a voltage-scalable, variation-aware, quality-tuning motion estimator. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '09)*. ACM. **(Cited on page 14.)**

[148] Thierry Moreau, Felipe Augusto, Patrick Howe, Armin Alaghi, and Luis Ceze. 2017. Exploiting quality-energy tradeoffs with arbitrary quantization. In *Proceedings of the Twelfth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '17)*. ACM, Article 30, 30:1–30:2 pages. **(Cited on page 12.)**

[149] Thierry Moreau, Joshua San Miguel, Mark Wyse, James Bornholt, Armin Alaghi, Luis Ceze, Natalie Enright Jerger, and Adrian Sampson. 2018. A taxonomy of general purpose approximate computing techniques. *IEEE Embedded System Letters* 10, 1 (2018), 2–5. **(Cited on page 3.)**

[150] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. 2015. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '15)*. 603–614. **(Cited on pages 16 and 24.)**

[151] M. Hassan Najafi, Devon Jenson, David J. Lilja, and Marc D. Riedel. 2019. Performing Stochastic Computation Deterministically. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 12 (Dec 2019), 2925–2938. https://doi.org/10.1109/TVLSI.2019.2929354 **(Cited on pages 13 and 24.)**

[152] Kumud Nepal, Soheil Hashemi, Hokchhay Tann, R. Iris Bahar, and Sherief Reda. 2016. Automated high-level generation of low-power approximate computing circuits. *IEEE Transactions on Emerging Topics in Computing* (2016). **(Cited on pages 14 and 24.)**

[153] Frank Olken and Doron Rotem. 1986. Simple Random Sampling from Relational Databases. In *Proceedings of the 12th International Conference on Very Large Data Bases*. 160–169. **(Cited on page 5.)**

[154] Frank Olken and Doron Rotem. 1990. Random Sampling from Database Files: A Survey. In *Proceedings of the 5th International Conference on Statistical and Scientific Database Management (SSDBM'1990)*. Springer-Verlag, 92–111. **(Cited on page 5.)**

[155] Daniele Jahier Pagliari, Enrico Macii, and Massimo Poncino. 2016. Serial T0: Approximate bus encoding for energy-efficient transmission of sensor signals. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, 14. **(Cited on page 18.)**

[156] Krishna V. Palem. 2003. Energy aware algorithm design via probabilistic computing: from algorithms and models to Moore's law and novel (semiconductor) devices. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems (CASES '03)*. ACM, 113–116. **(Cited on pages 13 and 24.)**

[157] Krishna V. Palem. 2005. Energy Aware Computing through Probabilistic Switching: A Study of Limits. *IEEE Trans. Comput.* 54 (September 2005), 1123–1137. Issue 9. **(Cited on page 2.)**

[158] Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. 2015. FlexJava: language support for safe and modular approximate programming. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. 745–757. **(Cited on pages 7, 19, and 24.)**

[159] John G. Proakis and Dimitris G. Manolakis. 1996. *Digital Signal Processing (3rd Ed.): Principles, Algorithms, and Applications*. Prentice-Hall, Inc. **(Cited on page 12.)**

[160] W. Qian, Xin Li, Marc D. Riedel, Kia Bazargan, and David J. Lilja. 2011. An Architecture for Fault-Tolerant Computation with Stochastic Logic. *IEEE Trans. Comput.* 60, 1 (Jan 2011), 93–105. **(Cited on page 13.)**

[161] Jan Rabaey, Abbas Rahimi, Sohum Datta, Miles Rusch, Pentti Kanerva, and Bruno Olshausen. 2017. Human-centric computing – The case for a Hyper-Dimensional approach. In *International Workshop on Advances in Sensors and Interfaces (IWASI)*. 29–29. **(Cited on page 26.)**

[162] Jan M. Rabaey. 1996. *Digital Integrated Circuits: A Design Perspective*. Prentice-Hall, Inc. **(Cited on page 13.)**

[163] Eric Rachlin and John E. Savage. 2008. A framework for coded computation. In *Information Theory, 2008. ISIT 2008. IEEE International Symposium on*. IEEE, 2342–2346. **(Cited on page 26.)**

[164] Arnab Raha and Vijay Raghunathan. 2018. Approximating beyond the processor: Exploring full-system energy-accuracy tradeoffs in a smart camera system. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 12 (2018), 2884–2897. **(Cited on page 27.)**

[165] Arnab Raha, Soubhagya Sutar, Hrishikesh Jayakumar, and Vijay Raghunathan. 2017. Quality Configurable Approximate DRAM. *IEEE Transactions on Computers (TC)* 66, 7 (2017), 1172–1187. **(Cited on pages 15 and 24.)**

[166] Ashish Ranjan, Arnab Raha, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. 2014. ASLAN: Synthesis of approximate sequential circuits. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*. IEEE, 1–6. **(Cited on pages 14 and 24.)**

[167] Ashish Ranjan, Swagath Venkataramani, Xuanyao Fong, Kaushik Roy, and Anand Raghunathan. 2015. Approximate storage for energy efficient spintronic memories. In *Proceedings of the 52nd Annual Design Automation Conference (DAC '15)*. **(Cited on pages 15, 17, and 24.)**

[168] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*. Springer, 525–542. **(Cited on page 8.)**

[169] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling Low-power, Highly-accurate Deep Neural Network Accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, 267–278. **(Cited on page 9.)**

[170] Semeen Rehman, Walaa El-Harouni, Muhammad Shafique, Akash Kumar, and Jörg Henkel. 2016. Architectural-Space Exploration of Approximate Multipliers. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '16)*. **(Cited on pages 14 and 24.)**

[171] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. 2012. Programming with Relaxed Synchronization. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES '12)*. ACM, 41–50. **(Cited on pages 21 and 24.)**

[172] M. Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the International Conference on Supercomputing (ICS)*. 324–334. **(Cited on pages 20, 21, and 24.)**

[173] Martin C. Rinard. 2007. Using Early Phase Termination to Eliminate Load Imbalances at Barrier Synchronization Points. In *Proceedings of the Conference on Object-oriented Programming Systems and Applications (OOPSLA)*. ACM, 369–386. **(Cited on page 20.)**

[174] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. 2014. FitNets: Hints for Thin Deep Nets. *CoRR* abs/1412.6550 (2014). arXiv:1412.6550 **(Cited on page 5.)**

[175] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning Assistant for Floating-point Precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, Article 27, 12 pages. **(Cited on pages 21 and 24.)**

[176] T. Sakurai and A.R. Newton. 1990. Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas. *IEEE Journal of Solid-State Circuits* 25, 2 (1990), 584–594. **(Cited on page 13.)**

[177] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based Approximation for Data Parallel Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, 35–50. **(Cited on pages 21 and 24.)**

[178] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning Approximation for Graphics Engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, 13–24. **(Cited on page 21.)**

[179] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. 2015. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01* 1 (2015). **(Cited on page 27.)**

[180] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-power Computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, 164–174. **(Cited on pages 7, 19, and 24.)**

[181] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. 2013. Approximate Storage in Solid-State Memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. **(Cited on pages 15, 16, 17, and 24.)**

[182] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and verifying probabilistic assertions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 112–122. **(Cited on page 21.)**

[183] Joshua San Miguel, Jorge Albericio, and Natalie Enright Jerger. 2016. The Bunker Cache for spatio-value approximation. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. **(Cited on pages 16, 17, and 24.)**

[184] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. 2015. Doppelganger: A Cache for Approximate Computing. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*. **(Cited on pages 17 and 24.)**

[185] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load Value Approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. **(Cited on pages 16, 17, and 24.)**

[186] Rahul Sarpeshkar. 1998. Analog versus Digital: Extrapolating from Electronics to Neurobiology. *Neural Computation* 10, 7 (1998), 1601–1638. **(Cited on page 13.)**

[187] Daniel Scharstein and Richard Szeliski. 2002. A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms. *International Journal of Computer Vision* 47, 1-3 (2002), 7–42. **(Cited on page 4.)**

[188] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic Optimization of Floating-Point Programs with Tunable Precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. **(Cited on pages 21 and 24.)**

[189] Jeremy Schlachter, Vincent Camus, Krishna V Palem, and Christian Enz. 2017. Design and Applications of Approximate Circuits by Gate-Level Pruning. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 5 (2017), 1694–1702. **(Cited on pages 14 and 24.)**

[190] Muhammad Shafique, Waqas Ahmad, Rehan Hafiz, and Jörg Henkel. 2015. A Low Latency Generic Accuracy Configurable Adder. In *Proceedings of the Design Automation Conference (52nd DAC)*. **(Cited on page 28.)**

[191] Muhammad Shafique, Rehan Hafiz, Semeen Rehman, Walaa El-Harouni, and Jörg Henkel. 2016. Invited-Cross-layer Approximate Computing: From Logic to Architectures. In *Proceedings of the 53rd Annual Design Automation Conference (53rd DAC)*. **(Cited on pages 3 and 28.)**

[192] Naresh R. Shanbhag. 2002. Reliable and Energy-efficient Digital Signal Processing. In *Proceedings of the 39th Annual Design Automation Conference (DAC '02)*. ACM, 830–835. **(Cited on page 4.)**

[193] Claude E. Shannon. 1959. Coding Theorems for a Discrete Source with a Fidelity Criterion. *IRE National Convention Record* 7, 4 (1959), 142–163. **(Cited on pages 2 and 25.)**

[194] Hashim Sharif, Prakalp Srivastava, Muhammad Huzaifa, Maria Kotsifakou, Keyur Joshi, Yasmin Sarita, Nathan Zhao, Vikram S Adve, Sasa Misailovic, and Sarita Adve. 2019. ApproxHPVM: a portable compiler IR for accuracy-aware optimizations. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30. **(Cited on page 21.)**

[195] Wei-Kuan Shih and Jane W. S. Liu. 1995. Algorithms for Scheduling Imprecise Computations with Timing Constraints to Minimize Maximum Error. *IEEE Trans. Comput.* 44, 3 (1995), 466–471. **(Cited on page 22.)**

[196] Byonghyo Shim and Naresh R Shanbhag. 2006. Energy-Efficient Soft Error-Tolerant Digital Signal Processing. *IEEE Transactions on VLSI Systems* 14, 4 (2006), 336–348. **(Cited on page 14.)**

[197] Doochul Shin and Sandeep K Gupta. 2010. Approximate Logic Synthesis for Error Tolerant Applications. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*. **(Cited on pages 14 and 24.)**

[198] Majid Shoushtari, Abbas BanaiyanMofrad, and Nikil Dutt. 2015. Exploiting Partially-Forgetful Memories for Approximate Computing. *IEEE Embedded Systems Letters* 7, 1 (2015), 19–22. **(Cited on pages 15 and 24.)**

[199] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. **(Cited on pages 21 and 24.)**

[200] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. 2007. Eon: A Language and Runtime System for Perpetual Systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys '07)*. **(Cited on pages 22 and 24.)**

[201] Daniel Alan Spielman. 1996. Highly Fault-Tolerant Parallel Computation. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS '96)*. **(Cited on page 25.)**

[202] Vilas Sridharan and David R Kaeli. 2009. Eliminating Microarchitectural Dependency from Architectural Vulnerability. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture (HPCA '09)*. **(Cited on pages 15 and 16.)**

[203] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 1 (2014), 1929–1958. **(Cited on page 5.)**

[204] Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. 2014. General-purpose Code Acceleration with Limited-precision Analog Computation. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. 505–516. **(Cited on pages 16 and 24.)**

[205] Mircea R. Stan and Wayne P. Burleson. 1995. Bus-invert Coding for Low-power I/O. *IEEE TVLSI* 3, 1 (1995), 49–58. **(Cited on page 18.)**

[206] Phillip Stanley-Marbell. 2009. Encoding Efficiency of Digital Number Representations under Deviation Constraints. In *Proceedings of the IEEE Information Theory Workshop*. 203–207. **(Cited on pages 26 and 28.)**

[207] Phillip Stanley-Marbell, Virginia Estellers, and Martin Rinard. 2016. Crayon: Saving Power Through Shape and Color Approximation on Next-generation Displays. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. 11:1–11:17. **(Cited on pages 4, 23, and 24.)**

[208] Phillip Stanley-Marbell, Pier Andrea Francese, and Martin Rinard. 2016. Encoder Logic for Reducing Serial I/O Power in Sensors and Sensor Hubs. In *Proceedings of the 28th IEEE Hot Chips Symposium (HotChips 28)*. 1–2. **(Cited on pages 7, 18, and 24.)**

[209] Phillip Stanley-Marbell and Paul Hurley. 2018. Probabilistic Value-Deviation-Bounded Integer Codes for Approximate Communication. *Computing Research Repository (CoRR)* abs/1804.02317 (2018). arXiv:1804.02317 **(Cited on pages 4, 7, 18, and 24.)**

[210] P. Stanley-Marbell and D. Marculescu. 2006. A Programming Model and Language Implementation for Concurrent Failure-Prone Hardware. In *Proceedings of the 2nd Workshop on Programming Models for Ubiquitous Parallelism, PMUP '06*. 44–49. **(Cited on page 25.)**

[211] Phillip Stanley-Marbell and Martin Rinard. 2015. Approximating outside the processor. In *Proc. Workshop Approx. Comput. Across Syst. Stack*. Citeseer, 1–3. **(Cited on page 27.)**

[212] Phillip Stanley-Marbell and Martin Rinard. 2015. Efficiency Limits for Value-Deviation-Bounded Approximate Communication. *IEEE Embedded Systems Letters* 7, 4 (2015), 109–112. **(Cited on pages 4, 18, and 28.)**

[213] Phillip Stanley-Marbell and Martin Rinard. 2015. Lax: Driver Interfaces for Approximate Sensor Device Access. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. **(Cited on pages 4, 7, 23, and 24.)**

[214] Phillip Stanley-Marbell and Martin Rinard. 2016. Reducing Serial I/O Power in Error-tolerant Applications by Efficient Lossy Encoding. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. 62:1–62:6. **(Cited on pages 4, 7, 18, and 24.)**

[215] Phillip Stanley-Marbell and Martin Rinard. 2017. Error-Efficient Computing Systems. *Foundations and Trends in Electronic Design Automation* 11, 4 (2017), 362–461. **(Cited on page 2.)**

[216] Phillip Stanley-Marbell and Martin Rinard. 2018. A Hardware Platform for Efficient Multi-Modal Sensing with Adaptive Approximation. *ArXiv e-prints* (2018). arXiv:1804.09241 **(Cited on page 27.)**

[217] P. Stanley-Marbell and M. Rinard. 2018. Perceived-Color Approximation Transforms for Programs that Draw. *IEEE Micro* 38, 4 (Jul 2018), 20–29. https://doi.org/10.1109/MM.2018.043191122 **(Cited on pages 24 and 28.)**

[218] P. Stanley-Marbell and M. Rinard. 2020. Warp: A Hardware Platform for Efficient Multi-Modal Sensing with Adaptive Approximation. *IEEE Micro* 40, 1 (Jan 2020), 57–66. **(Cited on page 27.)**

[219] Xin Sui, Andrew Lenharth, Donald S Fussell, and Keshav Pingali. 2016. Proactive control of approximate programs. *ACM SIGPLAN Notices* 51, 4 (2016), 607–621. **(Cited on page 22.)**

[220] Olivier Temam. 2012. A Defect-tolerant Accelerator for Emerging High-performance Applications. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. 356–367. **(Cited on page 16.)**

[221] Bradley Thwaites, Gennady Pekhimenko, Hadi Esmaeilzadeh, Amir Yazdanbakhsh, Onur Mutlu, Jongse Park, Girish Mururu, and Todd Mowry. 2014. Rollback-Free Value Prediction with Approximate Loads. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*. **(Cited on pages 17 and 24.)**

[222] Federico Tombari, Stefano Mattoccia, Luigi Di Stefano, and Elisa Addimanda. 2008. Classification and Evaluation of Cost Aggregation Methods for Stereo Correspondence. In *Conference on Computer Vision and Pattern Recognition (CVPR '08)*. IEEE, 1–8. **(Cited on page 4.)**

[223] Sergio L. Toral, Jose M. Quero, and Leopoldo G. Franquelo. 2000. Stochastic Pulse Coded Arithmetic. In *IEEE International Symposium on Circuits and Systems.*, Vol. 1. 599–602. **(Cited on page 13.)**

[224] Georgios Tziantzioulis, Nikos Hardavellas, and Simone Campanoni. 2018. Temporal Approximate Function Memoization. *IEEE Micro* 38, 4 (2018), 60–70. **(Cited on page 21.)**

[225] Girish V. Varatkar and Naresh R. Shanbhag. 2006. Energy-efficient Motion Estimation Using Error Tolerance. In *International Symposium on Low Power Electronics and Design (ISLPED '06)*. **(Cited on page 14.)**

[226] Lav Varshney. 2011. Performance of LDPC Codes Under Faulty Iterative Decoding. *IEEE Transactions on Information Theory* (2011), 4427–4444. **(Cited on page 26.)**

[227] Vassilis Vassiliadis, Jan Riehme, Jens Deussen, Konstantinos Parasyris, Christos D Antonopoulos, Nikolaos Bellas, Spyros Lalis, and Uwe Naumann. 2016. Towards automatic significance analysis for approximate computing. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 182–193. **(Cited on page 21.)**

[228] Vijay V Vazirani. 2013. *Approximation algorithms*. Springer Science & Business Media. **(Cited on page 5.)**

[229] Swagath Venkataramani, Vinay Chippa, Srimat Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Quality Programmable Vector Processors for Approximate Computing. In *46th Annual International Symposium on Microarchitecture*. ACM, 1–12. **(Cited on page 15.)**

[230] Swagath Venkataramani, Anand Raghunathan, Jie Liu, and Mohammed Shoaib. 2015. Scalable-Effort Classifiers for Energy-Efficient Machine Learning. In *52nd Annual Design Automation Conference (DAC '15)*. ACM, 67:1–67:6. **(Cited on page 21.)**

[231] Swagath Venkataramani, Amit Sabne, Vivek Kozhikkottu, Kaushik Roy, and Anand Raghunathan. 2012. Salsa: Systematic Logic Synthesis of Approximate Circuits. In *49th Annual Design Automation Conference (DAC '12)*. ACM, 796–801. **(Cited on pages 14 and 24.)**

[232] Ajay Verma, Philip Brisk, and Paolo Ienne. 2008. Variable Latency Speculative Addition: A New Paradigm for Arithmetic Circuit Design. In *Conference on Design, Automation and Test in Europe (DATE '08)*. ACM, 1250–1255. (**Cited on page** 14.)

[233] John von Neumann. 1956. Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components. *Automata Studies* (1956), 43–98. (**Cited on pages** 2 **and** 25.)

[234] David P Williamson and David B Shmoys. 2011. *The design of approximation algorithms*. Cambridge university press. (**Cited on page** 5.)

[235] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. 2016. Approximate Computing: A Survey. *IEEE Design and Test* (2016), 8–22. (**Cited on page** 3.)

[236] Ran Xu, Jinkyu Koo, Rakesh Kumar, Peter Bai, Subrata Mitra, Sasa Misailovic, and Saurabh Bagchi. 2018. Videochef: efficient approximation for streaming video processing pipelines. In *USENIX Annual Technical Conference (USENIX ATC 18)*. 43–56. (**Cited on page** 21.)

[237] Yaoqing Yang, Pulkit Grover, and Soummya Kar. 2014. Can a Noisy Encoder Be Used to Communicate Reliably?. In *52nd Annual Allerton Conference on Communication, Control, and Computing*. IEEE, 659–666. (**Cited on page** 26.)

[238] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. 2015. Neural Acceleration for GPU Throughput Processors. In *48th International Symposium on Microarchitecture (MICRO '48)*. 482–493. (**Cited on pages** 16 **and** 24.)

[239] SM Sadegh Tabatabaei Yazdi, Chu-Hsiang Huang, and Lara Dolecek. 2012. Optimal design of a Gallager B noisy decoder for irregular LDPC codes. *IEEE Communications Letters* 16, 12 (2012), 2052–2055. (**Cited on page** 26.)

[240] Sadegh Tabatabaei Yazdi, Hyungmin Cho, and Lara Dolecek. 2013. Gallager B Decoder on Noisy Hardware. *Transactions on Communications* 61 (2013), 1660–1673. (**Cited on page** 26.)

[241] Yavuz Yetim, Sharad Malik, and Margaret Martonosi. 2015. CommGuard: Mitigating Communication Errors in Error-Prone Parallel Execution. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. 311–323. (**Cited on page** 18.)

[242] Neil Zhao. 2010. Full-Featured Pedometer Design Realized with 3-Axis Digital Accelerometer. *Analog Dialogue* 44 (2010). (**Cited on page** 6.)

[243] Ning Zhu, Wang Ling Goh, Weija Zhang, Kiat Seng Yeo, and Zhi Hui Kong. 2010. Design of Low-Power High-Speed Truncation-Error-Tolerant Adder and its Application in Digital Signal Processing. *IEEE Transactions on Very Large Scale Integration Systems* (2010), 1225–1229. (**Cited on pages** 14 **and** 24.)

[244] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. 2012. Randomized Accuracy-Aware Program Transformations for Efficient Approximate Computations. In *39th Annual Symposium on Principles of Programming Languages (POPL '12)*. ACM, 441–454. (**Cited on page** 20.)

Received June 2018; accepted February 2020