A Survey of Techniques for Approximate Computing

SPARSH MITTAL, Oak Ridge National Laboratory

Approximate computing trades off computation quality with effort expended, and as rising performance demands confront plateauing resource budgets, approximate computing has become not merely attractive, but even imperative. In this article, we present a survey of techniques for approximate computing (AC). We discuss strategies for finding approximable program portions and monitoring output quality, techniques for using AC in different processing units (e.g., CPU, GPU, and FPGA), processor components, memory technologies, and so forth, as well as programming frameworks for AC. We classify these techniques based on several key characteristics to emphasize their similarities and differences. The aim of this article is to provide insights to researchers into working of AC techniques and inspire more efforts in this area to make AC the mainstream computing approach in future systems.

Categories and Subject Descriptors: A.2 [**General and Reference**]: Surveys and Overviews; B. [**Hardware**]: Power and Energy; C.0 [**Computer Systems Organization**]: Processors and Memory Architectures

General Terms: Design, Performance

Additional Key Words and Phrases: Review, classification, approximate computing technique (ACT), approximate storage, quality configurability, CPU, GPU, FPGA, neural networks

ACM Reference Format:
Sparsh Mittal. 2016. A survey of techniques for approximate computing. ACM Comput. Surv. 48, 4, Article 62 (March 2016), 33 pages.
DOI: http://dx.doi.org/10.1145/2893356

1. INTRODUCTION

As large-scale applications such as scientific computing, social media, and financial analysis gain prominence, the computational and storage demands of modern systems have far exceeded the available resources. It is expected that, in the coming decade, the amount of information managed by worldwide data centers will grow 50-fold, while the number of processors will increase only tenfold [Gantz and Reinsel 2011]. In fact, the electricity consumption of just the US data centers is expected to increase from 61 billion kWh (kilowatt hour) in 2006 [Mittal 2014a] and 91 billion kWh in 2013 to 140 billion kWh in 2020 [NRDC 2013]. It is clear that rising performance demands will soon outpace the growth in resource budgets; hence, overprovisioning of resources alone will not solve the conundrum that awaits the computing industry in the near future.

A promising solution for this dilemma is approximate computing (AC), which is based on the intuitive observation that, while performing exact computation or maintaining

Support for this work was provided by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research.
Author's address: 1 Bethel Valley Road, ORNL, TN, United States, 37830; email: mittals@ornl.gov.

DOI: http://dx.doi.org/10.1145/2893356

Paper organization

Fig. 1. Organization of the article by section.

peak-level service demand require a high amount of resources, allowing selective approximation or occasional violation of the specification can provide disproportionate gains in efficiency. For example, for a k-means clustering algorithm, up to 50× energy saving can be achieved by allowing a classification accuracy loss of 5 percent [Chippa et al. 2014]. Similarly, a neural approximation approach can accelerate an inverse kinematics application by up to 26× compared to the GPU execution, while incurring an error of less than 5 percent [Grigorian and Reinman 2015].

Approximate computing and storage approach leverages the presence of error-tolerant code regions in applications and perceptual limitations of users to intelligently trade off implementation, storage, and/or result accuracy for performance or energy gains. In brief, AC exploits the gap between the level of accuracy *required* by the applications/users and that *provided* by the computing system, for achieving diverse optimizations. Thus, AC has the potential to benefit a wide range of applications/frameworks, for example, data analytics, scientific computing, multimedia and signal processing, machine learning and MapReduce, and so forth.

However, although promising, AC is not a panacea. Effective use of AC requires judicious selection of approximable code/data portions and approximation strategy, since uniform approximation can produce unacceptable quality loss [Ranjan et al. 2015; Venkataramani et al. 2014; Tian et al. 2015; Sartori and Kumar 2013]. Even worse, approximation in control flow or memory access operations can lead to catastrophic results such as segmentation fault [Yetim et al. 2013]. Further, careful monitoring of output is required to ensure that quality specifications are met, since large loss makes the output unacceptable or necessitates repeated execution with precise parameters. Clearly, leveraging the full potential of AC requires addressing several issues. Recently, several techniques have been proposed to fulfill this need.

Contribution and article organization: In this article, we present a survey of techniques for approximate computing. Figure 1 shows the organization of this paper.

We first discuss the opportunities and obstacles in the use of AC in Section 2. We then present in Section 3 techniques for finding approximable program portions and monitoring output quality, along with the language support for expressing approximable

Table I. Terminology Used in Approximate Computing Research

(a) AC is synonymous with or has significant overlap with the ideas of :
Dynamic effort-scaling [Chippa et al. 2014], quality programmability/configurability [Venkataramani et al. 2013, 2014] and variable accuracy [Ansel et al. 2011]
(b) Applications or their code portions that are amenable to AC are called:
Approximable [Esmaeilzadeh et al. 2012b], relaxable [Yazdanbakhsh et al. 2015a], soft slices/ computations [Esmaeilzadeh et al. 2012a], best-effort computations [Chakradhar and Raghunathan 2010], noncritical/crucial [Shi et al. 2015], error-resilient/tolerant [Esmaeilzadeh et al. 2012b], error-acceptable [Kahng and Kang 2012], tunable [Sidiroglou et al. 2011], having 'forgiving nature' [Venkataramani et al. 2015] and being 'optional' (vs. guaranteed/mandatory) [Chakradhar and Raghunathan 2010]

variables/operations. We review the strategies for actually approximating these data in Section 4. In Section 5, we discuss research works to show how these strategies are used in many ACTs employed for different memory technologies, system components, and processing units.

In these sections, we organize the works in different categories to underscore their similarities and dissimilarities. Note that the works presented in these sections are deeply intertwined; while we study a work under a single group, several of these works belong to multiple groups. To show the spectrum of application of AC, we organize the works based on their workload or application domain in Section 6. Finally, Section 7 concludes this article with a discussion on future challenges.

Scope of the article: The scope of AC encompasses a broad range of approaches. For a concise presentation, we limit the scope of this article in the following manner. We focus on works that use an approximation strategy to trade off result quality/accuracy and not those that mainly focus on mitigation of hard/soft errors or other faults. We mainly focus on ACTs at architecture, programming, and system level, and briefly include design of inexact circuits. We do not typically include works on theoretical studies on AC. We believe that this article will be useful for computer architects, application developers, system designers, and other researchers[1].

2. PROMISES AND CHALLENGES OF APPROXIMATE COMPUTING

2.1. A Note on Terminology and Quality Metrics

Table I summarizes the terminology used for referring to AC and the code regions amenable to AC.

As we show in Sections 2.2 and 2.3, use of a suitable quality metric is extremely important to ensure correctness and balance quality loss with efficiency gain. For this reason, Table II shows the commonly used metrics for evaluating QoR of various applications/kernels (note that some of these applications may be internally composed of these kernels. Also, these metrics are not mutually exclusive). For several applications, multiple metrics can be used for evaluating quality loss, for example, both clustering accuracy and mean centroid distance can be used as metrics for k-means clustering [Chippa et al. 2013]. In essence, all of these metrics seek to compare some form of output (depending on the application, e.g., pixel values, body position, classification decision, execution time) in the approximate computation with that in

[1]We use the following acronyms throughout the article: bandwidth (BW), dynamic binary instrumentation (DBI), embedded DRAM (eDRAM), error-correcting code (ECC), finite impulse response (FIR), floating point (FP) unit (FPU), hardware (HW), instruction set architecture (ISA), multilayer perceptron (MLP), multilevel cell (MLC), neural network (NN), neural processing unit (NPU), nonvolatile memory (NVM), peak signal-to-noise ratio (PSNR), phase change memory (PCM), quality of result (QoR), resistive RAM (ReRAM), single instruction multiple data (SIMD), software (SW), solid state drive (SSD), spin transfer torque RAM (STT-RAM), structural similarity (SSIM).

Table II. Some Quality Metrics Used for Different Approximable Applications/Kernels

Quality metric(s)	Corresponding applications/kernels
Relative difference/ error from standard output	Fluidanimate, blackscholes, swaptions (PARSEC), Barnes, water, Cholesky, LU (Splash2), vpr, parser (SPEC2000), Monte Carlo, sparse matrix multiplication, Jacobi, discrete Fourier transform, MapReduce programs (e.g., page rank, page length, project popularity, and so forth), forward/inverse kinematics for 2-joint arm, Newton-Raphson method for finding roots of a cubic polynomial, n-body simulation, adder, FIR filter, conjugate gradient
PSNR and SSIM	H.264 (SPEC2006), x264 (PARSEC), MPEG, JPEG, rayshade, image resizer, image smoothing, OpenGL games (e.g., Doom 3)
Pixel difference	Bodytrack (PARSEC), eon (SPEC2000), raytracer (Splash2), particle filter (Rodinia), volume rendering, Gaussian smoothing, mean filter, dynamic range compression, edge detection, raster image manipulation
Energy conservation across scenes	Physics-based simulation (e.g., collision detection, constraint solving)
Classification/clustering accuracy	Ferret, streamcluster (PARSEC), k-nearest neighbor, k-means clustering, generalized learning vector quantization (GLVQ), MLP, convolutional neural networks, support vector machines, digit classification
Correct/incorrect decisions	Image binarization, jmeint (triangle intersection detection), ZXing (visual bar code recognizer), finding Julia set fractals, jMonkeyEngine (game engine)
Ratio of error of initial and final guess	3D variable coefficient Helmholtz equation, image compression, 2D Poisson's equation, preconditioned iterative solver
Ranking accuracy	Bing search, supervised semantic indexing (SSI) document search

exact computation. Some other quality metrics include *universal image quality index (UIQI)* for image processing, *satisfiability check* for SAT (Boolean satisfiability) solver, *difference in file size* for dedup (PARSEC), and so forth.

2.2. Motivation Behind and Scope For Approximate Computing

In many scenarios, use of AC is unavoidable, or the opportunity for AC arises inherently, while in other scenarios, AC can be proactively used for efficiency optimization. We now discuss the motivations behind, and opportunity for, AC.

2.2.1. Inherent Scope or Need For Approximation. In the case of inherently noisy input [Bornholt et al. 2014], limited data precision, a defect in HW, sudden additional load, or hard real-time constraints, use of AC becomes unavoidable. For example, AC is natural for FP computations in which rounding off happens frequently. For many complex problems, an exact solution may not be known, while an inexact solution may be efficient and sufficient.

2.2.2. Error-Resilience of Programs and Users. The perceptual limitations of humans provide scope for AC in visual and other computing applications. Similarly, many programs have noncritical portions, and small errors in these do not affect QoR significantly. For example, Esmaeilzadeh et al. [2012a] note that in a 3D raytracer application, 98% of FP operations and 91% of data accesses are approximable. Similarly, since the lower-order bits have smaller significance than the higher-order bits, approximating them may have only a minor impact on QoR [Cho et al. 2014; Ganapathy et al. 2015; Ranjan et al. 2015; Sampson et al. 2013; Tian et al. 2015; Rahimi et al. 2015; Gupta et al. 2011].

In several iterative refinement algorithms, running extra iterations with reduced precision of intermediate computations can still provide the same QoR [Khudia et al. 2015; Lopes et al. 2009; Tian et al. 2015; Raha et al. 2015; Chippa et al. 2014]. In some scenarios, for example, search engines, no unique answer exists, but multiple answers are admissible. Similarly, redundancy due to spatial/temporal correlation provides a scope for AC [Raha et al. 2015; Sartori and Kumar 2013; Sutherland et al. 2015; Yazdanbakhsh et al. 2015b; Samadi et al. 2013].

2.2.3. Efficiency Optimization. In the image processing domain, a PSNR value greater than 30dB, and in typical error-resilient applications, errors less than 10%, are generally considered acceptable [Rahimi et al. 2015]. By exploiting this margin, AC can aggressively improve performance and energy efficiency. For example, by intelligently reducing the eDRAM/DRAM refresh rate or SRAM supply voltage, the energy consumed in storage and memory access can be reduced with minor loss in precision [Mittal 2012, 2014b]. Similarly, the AC approach can alleviate the scalability bottleneck [Yeh et al. 2007], improving performance by early loop termination [Sidiroglou et al. 2011; Chippa et al. 2014], skipping memory accesses [Yazdanbakhsh et al. 2015b], offloading computations to an accelerator [Moreau et al. 2015], improving yield [Ganapathy et al. 2015], and much more.

2.2.4. Quality Configurability. AC can provide knobs to trade off quality with efficiency; thus, instead of executing every computation to full fidelity, the user needs to expend only as much effort (e.g., area, energy) as dictated by the QoR requirement [Chakradhar and Raghunathan 2010]. For example, an ACT can use different precisions for data storage/processing, program versions of different quality (see Table IV), different refresh rates in eDRAM/DRAM (see Table V), and so on, to just fulfill the QoR requirement.

2.3. Challenges in Approximate Computing

As we show here, AC also presents several challenges, which need to be addressed to fully realize its potential.

2.3.1. Limited Application Domain and Gains of AC. Due to their nature, some applications are not amenable to approximation, for example, cryptography and hard real-time applications. Some approximation strategies are valid only in a certain range, for example, Zhang et al. [2014] approximate inverse operation ($y = 1/x$) using the function $y = 2.823 - 1.882x$, which produces low errors only when $x \in [0.5, 1]$. Similarly, approximating `sin` function using an NN on an unbounded range is intractably hard [Eldridge et al. 2014]. Further, the gains of AC are bounded, for example, inexact storage does not reduce the number of operations on the data, and vice versa.

2.3.2. Correctness Issues. Aggressive ACTs may prevent program termination (e.g., a matrix computation kernel in which AC can lead to an unsolvable problem), or lead to corrupt output that may not even be detected by the quality metric used [Akturk et al. 2015]. For example, approximating a compression program may lead to corrupt output, which may go undetected on using output file size as the quality metric. This necessitates choosing an accurate, yet lightweight, quality metric.

Further, AC may interfere with synchronization and memory ordering, and may lead to nondeterministic output, which makes debugging difficult [Khudia et al. 2015]. Avoiding this may require executing a higher number of iterations or tightening convergence criteria in iterative methods or comparing with single-threaded exact execution.

2.3.3. Finding Application-Specific Strategies. A naïve approximation approach such as uniform approximation is unlikely to be efficient, and while some strategies for approximation such as precision scaling, memoization, and the like, are known, no strategy can be universally applied to all approximable applications. Hence, the approximation strategies need to be determined on a per-application basis by the user or a sophisticated program module [Ansel et al. 2011].

2.3.4. Overhead and Scalability. Several ACTs may have large implementation overhead; for example, voltage scaling may require voltage shifters for moving data between different voltage domains [Esmaeilzadeh et al. 2012a]. Similarly, analog NN implementations [Li et al. 2015; Amant et al. 2014] require conversion of signals between digital and analog domains. Other techniques may require the programmer to write multiple approximate versions of a program or annotate the source code, which do not scale to complex programs and legacy software. Similarly, some ACTs require ISA extension [Esmaeilzadeh et al. 2012a; Ranjan et al. 2015; Venkataramani et al. 2013; Esmaeilzadeh et al. 2012b; Sampson et al. 2011; Keramidas et al. 2015; Yazdanbakhsh et al. 2015b]; hence, implementing them on existing platforms may be challenging.

2.3.5. Providing High Quality and Configurability. ACTs must maintain the QoR to a desired level and also provide tunable knob(s) to trade off quality with efficiency [Venkataramani et al. 2014]. If QoR falls below a threshold, the application may have to be executed precisely, which increases design and verification cost and may even nullify the gains from approximation. Also, apart from average error, worst-case error needs to be bounded to maintain high QoR [Khudia et al. 2015; Grigorian and Reinman 2014]. These facts necessitate monitoring the QoR and adaptively changing the knobs to reach a desired QoR. However, this may be infeasible or prohibitively expensive in several ACTs.

The techniques proposed in the next several sections aim to address these challenges.

3. IDENTIFYING APPROXIMABLE PORTIONS AND EXPRESSING THIS AT LANGUAGE LEVEL

Finding approximable variables and operations is the crucial initial step in every ACT. While this is straightforward in several cases (e.g., approximating lower-order bits of graphics data), in other cases, it may require insights into program characteristics or error-injection to find the portions that can be approximated with little impact on QoR (Section 3.1). Closely related to it is the output-monitoring step, which verifies adherence to the quality constraint and triggers parameter adjustment or precise execution in the case of unacceptable quality loss (Section 3.2).

Further, once relaxable portions are identified, conveying this to the software or compiler requires source-code annotations; several programming frameworks provide support for this (Section 3.3). This source-code annotation can be in the form of OpenMP-style pragma directives (Section 3.4), which provides several benefits, for example, nonintrusive and incremental program transformation, and easy debugging or comparison with exact code by disabling the approximation directives with a single compiler flag. Table III classifies the techniques based on these factors. We now discuss several of these techniques.

3.1. Automatically Finding Approximable Code/Data

Roy et al. [2014] present a SW framework for automatically discovering approximable data in a program by using statistical methods. Their technique first collects the variables of the program and the range of values that they can take. Then, using binary instrumentation, the values of the variables are perturbed and the new output is measured. By comparing this against the correct output, which fulfills the acceptable QoS threshold, the contribution of each variable in the program output is measured. Based on this, the variables are marked as approximable or nonapproximable. Thus, their framework obviates the need of a programmer's involvement or source-code annotations for AC. They show that, compared to a baseline with type-qualifier annotations by the programmer [Sampson et al. 2011] (see Section 3.3), their approach achieves nearly 85% accuracy in determining the approximable data. The limitation of their technique is that some variables that are marked as nonapproximable in the

Table III. A Classification Based on Implementation Approach and Other Features

Classification	References
Error-injection	[Chippa et al. 2013, 2014; Cho et al. 2014; Esmaeilzadeh et al. 2012a; Ganapathy et al. 2015; Liu et al. 2012; Misailovic et al. 2014; Roy et al. 2014; Venkataramani et al. 2013; Xu and Huang 2015; Yetim et al. 2013]
Use of DBI	[Chippa et al. 2013; Düben et al. 2015; Liu et al. 2012; Miguel et al. 2014; Roy et al. 2014; Venkataramani et al. 2013]
Output quality monitoring	[Grigorian et al. 2015; Grigorian and Reinman 2014; Hegde and Shanbhag 1999; Khudia et al. 2015; Mahajan et al. 2015; Ringenburg et al. 2014, 2015; Roy et al. 2014; Samadi and Mahlke 2014; Yeh et al. 2007; Zhang et al. 2014]
Annotating approximable program portions	[Amant et al. 2014; Ansel et al. 2011; Carbin et al. 2013; Esmaeilzadeh et al. 2012a, 2012b; Liu et al. 2012; McAfee and Olukotun 2015; Misailovic et al. 2014; Moreau et al. 2015; Ringenburg et al. 2014; Sampson et al. 2011, 2015; Shi et al. 2015; Shoushtari et al. 2015; Vassiliadis et al. 2015; Yazdanbakhsh et al. 2015a, 2015b]
Use of OpenMP-style pragma	[Rahimi et al. 2013; Vassiliadis et al. 2015]
Use of compiler	[Amant et al. 2014; Ansel et al. 2011; Baek and Chilimbi 2010; Esmaeilzadeh et al. 2012a, 2012b; Mahajan et al. 2015; McAfee and Olukotun 2015; Mishra et al. 2014; Moreau et al. 2015; Rahimi et al. 2013; Ringenburg et al. 2015; Samadi et al. 2013; Sampson et al. 2015; Sartori and Kumar 2013; Sidiroglou et al. 2011; Vassiliadis et al. 2015; Yetim et al. 2013]

programmer-annotated version may be marked as approximable by their technique, which can lead to errors.

Chippa et al. [2013] present a technique for automatic resilience characterization of applications. Their technique works in two steps. In the resilience identification step, their technique considers innermost loops that occupy more than 1% of application execution time as atomic kernels. As the application runs with the input dataset provided, random errors are introduced into the output variables of a kernel using the Valgrind DBI tool. If the output does not meet the quality criterion or if the application crashes, the kernel is marked as sensitive; otherwise, it is potentially resilient. In the resilience characterization step, potentially resilient kernels are further explored to see the applicability of various approximation strategies. In this step, errors are introduced in the kernels using Valgrind based on the approximation models (and not randomly). To quantify resilience, they propose an ACT-independent model and an ACT-specific model for approximation. The ACT-independent approximation model studies the errors introduced due to ACT using a statistical distribution that shows the probability, magnitude, and predictability of errors. From this model, the application quality profile is generated as a function of these three parameters. The ACT-specific model may use different ACTs, such as precision scaling, inexact arithmetic circuits, and loop perforation (see Section 4). Their experiments show that several applications show high resilience to errors, and many parameters—such as the scale of input data, choice of output quality metric, and granularity of approximation—have a significant impact on application resilience.

Raha et al. [2015] present two techniques for selecting approximable computations for a reduce-and-rank kernel. A reduce-and-rank kernel (e.g., k-nearest neighbor) performs reduction between an input vector and each reference vector, then ranks the outputs to find the subset of top reference vectors for that input. Their first technique decomposes vector reductions into multiple partial reductions and interleaves them with the rank computation. Then, based on intermediate reduction results and ranks, this technique identifies whether a particular reference vector is expected to appear in the final subset. Based on this, future computations that can be relaxed with little impact on the output are selected. The second technique leverages the temporal or spatial correlation of inputs. Depending on the similarity between current and previous input, this technique approximates or entirely skips processing parts of the current inputs. Approximation is achieved using precision scaling and loop perforation strategies. They also design a runtime framework for dynamically adapting the parameters of their two techniques for minimizing energy consumption while meeting the QoR target.

Ansel et al. [2011] present language extensions and an accuracy-aware compiler for facilitating writing of configurable-accuracy programs. The compiler performs autotuning using a genetic algorithm to explore the search-space of possible algorithms and accuracy levels for dealing with recursion and subcalls to other configurable-accuracy code. Initially, the population of candidate algorithms is maintained, which is expanded using mutators and later pruned to allow more optimal algorithms (i.e., fastest K algorithms for a given accuracy level) to evolve. Thus, the user needs to specify only accuracy requirements and does not need to understand algorithm-specific parameters, while the library writer can write a portable code by simply specifying ways to search the space of parameter and algorithmic choices. To limit computation time, the number of tests performed for evaluating possible algorithms needs to be restricted. This, however, can lead to the choice of suboptimal algorithms and errors; hence, the number of tests performed needs to be carefully chosen. Their experiments show that exposing algorithmic and accuracy choices to the compiler for autotuning programs is effective and provides large speedups with bounded loss of accuracy.

3.2. Ensuring Quality of Approximate Computations Through Output Monitoring

We here discuss ACTs that especially focus on ensuring quality by efficiently monitoring application output.

Grigorian and Reinman [2014] note that, for several computation-intensive applications, although finding a solution may incur high overheads, checking the solution quality may be easy. Based on this, they propose decoupling error analysis of approximate accelerators from application quality analysis by using application-specific metrics called light weight checks (LWCs). For example, the Boolean satisfiability problem, which determines whether a set of variable assignments satisfies a Boolean formula, is NP-complete. However, checking whether a given solution satisfies the formula is easy; hence, this forms an example of LWC for the satisfiability problem. LWCs are directly integrated into the application, which enables compatibility with any ACT. By virtue of being lightweight, LWCs can be used dynamically for analyzing and adapting application-level errors. Only when testing with LWCs indicates unacceptable quality loss, exact computation needs to be performed for recovery. Otherwise, the approximation is considered acceptable. This saves energy without compromising reliability. Their approach guarantees bounding worst-case error and obviates the need of statically designed error models.

Khudia et al. [2015] note that, even when the average value of errors due to AC may be small, the worst-case value of errors may be high, which may affect the whole user experience. To address this, they present an output-quality monitoring and management technique that can ensure meeting a given output quality. Based on the observation that simple prediction approaches, for example, linear estimation, moving average and decision tree, can accurately predict approximation errors, they use a low-overhead error-detection module that tracks predicted errors to find the elements that need correction. Using this information, the recovery module, which runs in parallel to the detection module, re-executes the iterations that lead to high errors. This becomes possible since the approximable functions or codes are generally those that simply read inputs and produce outputs without modifying any other state, such as map and stencil patterns. Hence, these code sections can be safely re-executed without incurring high overhead or side effects. The recovery module trades off performance/energy improvements with output quality and ascertains how many iterations need to be re-executed. They show that, compared to an unchecked approximator, their technique reduces output error significantly with only a small reduction in energy saving and no reduction in performance improvement achieved from AC.

Ringenburg et al. [2015] present two offline debugging mechanisms and three online monitoring mechanisms for approximate programs. Among the offline mechanisms, the first one identifies the correlation between QoR and each approximate operation by tracking the execution and error frequencies of different code regions over multiple program executions with varying QoR values. The second mechanism tracks which approximate operations affect any approximate variable and memory location. This mechanism is helpful for deciding whether the energy saving from approximation is large enough to justify the accompanying loss in QoR. The online mechanisms complement the offline ones, and they detect and compensate QoR loss while maintaining the energy gains of approximation. The first mechanism compares the QoR for precise and approximate variants of the program for a *random subset of executions*. This mechanism is useful for programs for which QoR can be assessed by sampling a few outputs, but not for those that require bounding the worst-case errors. The second mechanism uses programmer-supplied verification functions that can *check* a result with much lower overhead than *computing* the result. For example, in video decoding, the similarity between current and past frames can be used as a check for QoR. The third mechanism stores past inputs and outputs of the checked code and estimates the output for the current execution based on interpolation of the previous executions with *similar* inputs. They show that their offline mechanisms help in effectively identifying the root of a quality issue instead of merely confirming the existence of an issue, and the online mechanisms help in controlling QoR while maintaining high energy gains.

Mahajan et al. [2015] present a quality-control technique for inexact accelerator-based platforms. They note that an accelerator may not always provide acceptable results; hence, blindly invoking the accelerator in all cases leads to quality loss and waste of energy and execution time. They propose a predictor that guesses whether that invocation of accelerator will lead to unacceptable quality degradation. If yes, their technique instead invokes the precise code. The predictor uses only the information local to a specific accelerator-invocation, hence, it does not require maintaining history information. They study a table-based and an NN-based design for the predictor. Both these designs have a training phase and a runtime phase. For the table-based design, in the training phase, the index for the table is generated by hashing the accelerator inputs, which is used for filling the predictor contents. At runtime, the prediction table is accessed for a decision and is indexed by hashing the accelerator inputs. Hash collisions are avoided by using a carefully designed hash function and employing multiple prediction tables. The neural predictor uses an MLP that is trained at compile time using a set of representative training input datasets. Using this, prediction is made at runtime. They show that the neural predictor yields higher accuracy than the table-based predictor, although both provide similar speedup and energy saving due to the higher overhead of neural predictor.

3.3. Programming Language Support for Approximate Computing

Sampson et al. [2011] propose using type qualifiers for specifying approximate data and separating precise and approximate portions in the program. For the variables marked with `approximate` qualifier, the storage, computing, and algorithm constructs used can all be approximate. Correctness is guaranteed for precise data, while only "best effort" is promised for the approximate data. Any flow of information from approximate to precise data needs to be explicitly specified by the programmer. This ensures careful handling of approximate data and makes the programming model safe. It also obviates the need of dynamic checking, which reduces the runtime overhead. For experiments, they assume that the programs run on a processor with inexact storage and inexact operations, using refresh rate reduction for DRAM main memory, precision scaling for FP operations, and voltage scaling for SRAM data cache, SRAM registers, and functional units. Inexact data are stored in inexact portions of main memory and cache, and in inexact registers; the opposite is true for precise data. Also, inexact functional units perform operations on inexact data. They demonstrate their approach using an extension to Java, and show that it saves a large amount of energy with small accuracy loss.

Yazdanbakhsh et al. [2015a] present annotations for providing suitable syntax and semantics for approximate HW design and reuse in Verilog. They allow the designer to specify both critical (precise) and approximable portions of the design. For example, `relax(arg)` can be used to implicitly approximate `arg` while `restrict(arg)` specifies that any design element affecting `arg` must be precise, unless specifically declared relaxable by `relax` annotation. Their approach allows reuse of approximate modules in different designs having different accuracy requirements without requiring reimplementation. For this, they provide annotations that delineate which outputs have approximate semantics and which inputs cannot be driven by approximate wires unless annotated explicitly. They also use relaxability inference analysis (RIA), which provides a formal safety guarantee of accurately identifying approximable circuit elements based on the designer's annotations. RIA begins with annotated wires and iteratively traverses the circuit for identifying the wires that must have precise semantics. All remaining wires can be approximated and the gates that immediately drive these wires can be approximated in the synthesis. Their approach allows applying approximation in the synthesis process while abstracting away these details from the designer. Their experiments show that the concise nature of language annotations and automated RIA enable their approach to safely approximate even large-scale designs.

Carbin et al. [2013] present a programming language, named Rely, that allows programmers to determine the quantitative reliability of a program, in contrast with other approaches (e.g., Sampson et al. [2011]) that allow only a binary accurate/approximate distinction for program variables. In the Rely language, quantitative reliability can be specified for function results; for example, in `int<0.99*R(arg)> FUNC(int arg, int x)` code, `0.99*R(arg)` specifies that the reliability of return value of `FUNC` must be at least 99% of reliability of `arg` when the function was invoked. Rely programs can run on a processor with potentially unreliable memory and unreliable logical/arithmetic operations. The programmer can specify that a variable can be stored in unreliable memory and/or an unreliable operation (e.g., add) can be performed on the variables. Integrity of memory access and control flow are maintained by ensuring reliable computations for the corresponding data. By running both error-tolerant programs and checkable programs (those for which an efficient checker can be used for dynamically verifying result correctness), they show that Rely allows determination of integrity (i.e., correctness of execution and validity of results) and QoR of the programs.

Misailovic et al. [2014] present a framework for accuracy (defined as the difference between accurate and inexact result) and reliability (defined as the probability of obtaining an acceptably accurate result) aware optimization of inexact kernels running on inexact HW. The user provides accuracy and reliability specifications for the inexact kernels along with these specifications for individual instructions and memory of the inexact HW platform. Their technique models the problem of selecting inexact instructions and variables allocated in inexact memories using an integer linear program that minimizes the energy consumption of the kernel while satisfying reliability/accuracy constraints. Their technique finds whether a given instruction can be inexact and whether a variable can be stored in inexact memory. They also provide a sensitivity profiler that uses the error-injection approach to estimate the contribution of a kernel in final output. They show that, by selecting inexact kernel operations for synthesizing inexact computations in an effective manner, their technique saves significant energy while also maintaining reliability guarantees.

Ringenburg et al. [2014] present a SW tool for prototyping, profiling, and autotuning the quality of programs designed to run on approximate HW. Their tool has three layers. The approximation layer simulates an approximate HW with customizable energy cost and approximation model. The user can annotate precise and approximate code regions. Approximation is allowed only in arithmetic operations, comparisons, and loads from data arrays and not in control flow and memory allocation operations. The profiling layer monitors both efficiency gain from approximation and quality loss, based on an application-specific QoR-measurement function. The autotuning layer explores the search space of possible approximate/precise decompositions of user code blocks by using strategies such as selectively marking an approximate code region in original code as precise (but *never* making precise code into approximate code). By profiling alternative configurations, it finds Pareto-optimal frontier of efficiency-quality trade-offs. They demonstrate their approach by building a prototype tool in OCaml language.

3.4. Using OpenMP-Style Directives For Marking Approximable Portions

Rahimi et al. [2013] note that timing errors due to static and dynamic variations (e.g., process variation and voltage variation) necessitate recovery that incurs a large overhead in FP pipelines. They design accuracy-configurable and variation-resilient FPUs for detecting and correcting online timing errors. They quantify variability of the FP pipeline in terms of the fraction of cycles in which the pipeline sees a timing error. An SW scheduler ranks the FPUs based on their variability to find the most suitable FPUs for the target accuracy. Using OpenMP `#pragma` directives, approximable program regions are annotated. At design time, acceptable error significance and error rate are identified by profiling program regions. At runtime, based on `accurate` or `approximate` directives, the FPUs are promoted/demoted to accurate/approximate mode to match program region requirements. In approximate mode, timing errors on least significant K bits of the fraction are ignored, while in accurate mode, all timing errors are detected and corrected. This approach avoids the cost of timing error correction for inexact program regions if the error rate is below an application-specific threshold. They show that their technique maintains acceptable quality loss in error-tolerant applications and reduces recovery overhead in error-intolerant applications, while providing significant energy savings in both types of applications.

Vassiliadis et al. [2015] present a programming model and runtime system for AC. In their technique, depending on the impact of a task on the final output quality, a programmer can express its significance using `#pragma` compiler directives. The programmer can also optionally provide a low-overhead inexact version of a task. Further, the acceptable quality loss is specified in terms of fraction of tasks to be executed precisely. Based on this, the runtime system employs inexact versions of less-important tasks or drops them completely. They show that their technique provides significant energy and performance gains compared to both fully accurate execution and loop perforation techniques.

4. STRATEGIES FOR APPROXIMATION

Once approximable variables and operations have been identified, they can be approximated using a variety of strategies, such as reducing their precision; skipping tasks, memory accesses, or some iterations of a loop; performing an operation on inexact

Table IV. Some Approximation Strategies Used in Different Works

Classification	References
Precision scaling	[Anam et al. 2013; Chippa et al. 2014, 2013; Düben et al. 2015; Hsiao et al. 2013; Keramidas et al. 2015; Lopes et al. 2009; Raha et al. 2015; Rahimi et al. 2013; Sampson et al. 2011; Shim et al. 2004; Tian et al. 2015; Venkataramani et al. 2013; Yeh et al. 2007; Zhang et al. 2015]
Loop perforation	[Baek and Chilimbi 2010; Chippa et al. 2013, 2014; Samadi and Mahlke 2014; Shi et al. 2015; Sidiroglou et al. 2011]
Load value approximation	[Miguel et al. 2014; Sutherland et al. 2015; Yazdanbakhsh et al. 2015b]
Memoization	[Alvarez et al. 2005; Keramidas et al. 2015; Rahimi et al. 2013, 2015; Ringenburg et al. 2015; Samadi et al. 2014; Yeh et al. 2007]
Task dropping/skipping	[Byna et al. 2010; Chakradhar and Raghunathan 2010; Goiri et al. 2015; Raha et al. 2015; Samadi et al. 2013; Sidiroglou et al. 2011; Vassiliadis et al. 2015]
Memory access skipping	[Samadi et al. 2013;Yazdanbakhsh et al. 2015b; Zhang et al. 2015]
Data sampling	[Ansel et al. 2011; Goiri et al. 2015; Samadi et al. 2014]
Using program versions of different accuracy	[Ansel et al. 2011; Baek and Chilimbi 2010; Goiri et al. 2015; Vassiliadis et al. 2015]
Using inexact or faulty HW	[Carbin et al. 2013; Chakradhar and Raghunathan 2010; Chippa et al. 2013; Du et al. 2014; Ganapathy et al. 2015; Gupta et al. 2011; Hegde and Shanbhag 1999; Kahng and Kang 2012; Kulkarni et al. 2011; Misailovic et al. 2014; Rahimi et al. 2013; Sampson et al. 2011; Shoushtari et al. 2015; Varatkar and Shanbhag 2008; Xu and Huang 2015; Yeh et al. 2007; Yetim et al. 2013; Zhang et al. 2014, 2015]
Voltage scaling	[Chippa et al. 2014; Esmaeilzadeh et al. 2012a; Gupta et al. 2011; Hegde and Shanbhag 1999; Rahimi et al. 2015; Sampson et al. 2011; Shim et al. 2004; Shoushtari et al. 2015; Varatkar and Shanbhag 2008; Venkataramani et al. 2013]
Refresh rate reduction	[Cho et al. 2014; Liu et al. 2012]
Inexact reads/writes	[Fang et al. 2012; Ranjan et al. 2015]
Reducing divergence in GPU	[Grigorian and Reinman 2015; Sartori and Kumar 2013]
Lossy compression	[Samadi et al. 2013; Yetim et al. 2013]
Use of neural network	[Amant et al. 2014; Du et al. 2014; Eldridge et al. 2014; Esmaeilzadeh et al. 2012b; Grigorian et al. 2015; Grigorian and Reinman 2014, 2015; Khudia et al. 2015; Li et al. 2015; Mahajan et al. 2015; McAfee and Olukotun 2015; Moreau et al. 2015; Sampson et al. 2015; Venkataramani et al. 2014, 2015; Zhang et al. 2015]

hardware; and so forth. Table IV summarizes the strategies used for approximation. Note that the ideas used in these strategies are not mutually exclusive. We now discuss these strategies, in context of the ACTs in which they are used.

4.1. Using Precision Scaling

Several ACTs work by changing the precision (bit-width) of input or intermediate operands to reduce storage/computing requirements.

Yeh et al. [2007] propose dynamic precision scaling for improving efficiency of physics-based animation. Their technique finds the minimum precision required by performing profiling at design time. At runtime, the energy difference between consecutive simulation steps is measured and compared with a threshold to detect whether the simulation is becoming unstable. In the case of instability, the precision is restored to the maximum; as simulation stabilizes, the precision is progressively reduced until it reaches the minimum value. They show that precision reduction provides three optimization opportunities. First, it can turn an FP operation into a trivial operation (e.g., multiplication by one), which would not require use of an FPU. Second, it increases the locality between similar items in similar scenes and between iterations during constraint relaxation. Thus, precision reduction combines close values to a single value, which increases the coverage of memoization technique and even allows using a lookup table for performing FP multiply and add operations. Third, precision scaling allows use of smaller and faster FPUs for several computations. Based on this, they propose a hierarchical FPU architecture in which a simple core-local FPU is used at the L1 level and full precision FPUs are shared at the L2 level to save area for allowing more cores to be added. An operation that requires higher precision than that provided by L1 FPU is executed in the L2 FPU. They show that their technique improves performance and energy efficiency compared to a baseline without FPU sharing.

Tian et al. [2015] present a technique for scaling precision of off-chip data accesses for saving energy. They apply their technique to a mixed-model-based clustering problem, which requires accessing a large amount of off-chip data. They note that, in a clustering algorithm, a functional error happens only when a sample is assigned to a wrong cluster. Based on this, the precision can be lowered as long as the *relative distances* between clusters and samples are still in correct order, so that no functional error happens. Further, since the clustering algorithm works in an iterative manner, samples can be relabeled in later iterations; thus, by using higher precision in those iterations, correctness can be ensured. Based on this, their technique selects the precision in each iteration depending on when a functional error will begin manifesting and how many functional errors can be tolerated by the application. Based on the precision, the memory controller decides the bit-width of data to be fetched from off-chip memory. Since use of approximation can lead to fluctuation in membership, on detecting such a situation, their technique increases the precision of data. To facilitate fetching the most significant bits, the data in off-chip memory is organized in a manner such that the bits of same significance in different words are placed together. They show that, compared to fully precise off-chip access, their technique saves significant energy with a negligible loss in accuracy.

4.2. Using Loop Perforation

Some techniques use a loop perforation approach, which works by skipping some iterations of a loop to reduce computational overhead.

Sidiroglou et al. [2011] identify several global computational patterns that work well with loop perforation, such as the Monte Carlo simulation, iterative refinement, and search space enumeration. For example, in search space enumeration, perforated computation skips some items and returns one of the remaining items from the search space. For exploring performance versus accuracy trade-off, they study two algorithms. The first algorithm exhaustively explores all combinations of tunable loops (those loops whose perforation produces efficient and still acceptable computations) at given perforation rates (i.e., fraction of iterations to skip) on all training inputs. The combinations producing error are discarded and those that are Pareto-optimal in performance and accuracy are selected. The second algorithm works on a greedy strategy. It uses a heuristic metric for prioritizing loop/perforation rate pairs and seeks to maximize performance within an accuracy loss bound. They show the performance advantage of their technique, along with its capability to allow performance–accuracy trade-off exploration.

4.3. Using Load Value Approximation

On a load miss in a cache, the data must be fetched from the next-level cache or main memory, which incurs large latency. Load value approximation (LVA) leverages the approximable nature of applications to estimate load values, thus allows a processor to progress without stalling for a response. This hides the cache miss latency. We now discuss some LVA-based techniques.

Miguel et al. [2014] present an LVA technique for graphics applications. Compared to the traditional load-value predictors, in which a block needs to be fetched *on every cache miss* to confirm correctness of prediction, their technique fetches the blocks *occasionally* just to train the approximator. Thus, fetching a cache block on each cache miss is not required, which reduces the memory accesses significantly. Further, since graphics applications can tolerate errors, if the values estimated by LVA do not match the exact value, rollbacks are not required. Their technique uses confidence estimation to make approximations only when the accuracy of the approximator is reasonably high. Also, the error-tolerance property allows approximating for a higher number of loads than that in traditional predictors. They show that, with negligible degradation in output quality, their technique provides significant speedup and energy saving.

Yazdanbakhsh et al. [2015b] present an ACT for offsetting both latency and BW constraints in GPUs and CPUs. Based on the programmer's code annotations, loads that do not deal with memory accesses and control flow are identified. Of these, the loads that cause the largest fraction of misses are selected. By individually approximating each of these loads, their impact on quality is measured and the loads leading to smaller degradation than a threshold are selected for approximation. When these loads miss in the cache, the requested values are predicted. However, no check for misprediction or recovery is performed, which avoids pipeline flush overheads. Also, a fraction of cache misses are dropped for mitigating BW bottleneck, which is especially helpful in GPUs. Removal of the cache-missing loads from the critical program path avoids long memory stalls and, by controlling the drop rate, a balance between quality and efficiency is obtained. Every data request in a GPU is a SIMD load, which produces values for multiple concurrent threads. Since predicting value for each thread *separately* incurs large overhead, they leverage the value similarity across accesses in adjacent threads to design a multivalue predictor that has only two parallel specialized predictors, one for threads 0 to 15 and another for threads 16 to 31. Use of special strategies for the GPU environment, such as use of a multivalue predictor, distinguishes their technique from that of Miguel et al. [2014]. They show that their technique improves performance and energy efficiency with bounded QoR loss in both the GPU and CPU.

Sutherland et al. [2015] use LVA strategy to reduce the memory stalls in GPUs. Their ACT uses the texture HW to generate inexact values, which obviates the need of fetching exact values from global memory. The texture fetch units lying between threads and texture cache (TC) are capable of interpolating between multiple neighboring cache entries. Using this interpolation feature, their technique uses FP data as indices for TC. Based on the observation that spatially or temporally correlated data also shows strong value locality, they use the difference (delta) between two successively read global memory values for making approximation. Thus, the approximate value is obtained as the sum of last exact value and delta approximation retrieved from TC. The delta approximations are preloaded in the TC based on analysis of data access patterns of every thread in the training dataset. For generating these delta values with the training set, they store last K values read from global memory in a buffer. On a memory access, they record the delta between two most recent elements in the buffer, along with the delta between the most recent buffer entry and the value returned from the memory access. They show that their technique improves performance with negligible quality loss and is suited for a wide range of data-intensive applications.

4.4. Using Memoization

The memoization approach works by storing the results of functions for later reuse with *identical* function/input. By reusing the results for *similar* functions/inputs, the scope of memoization can be enhanced at the cost of possible approximation. This approach is used by several ACTs.

Rahimi et al. [2013] note that SIMD architecture exposes the value locality of a parallel program to all its lanes. Based on this, they propose a technique that reuses the result of an instruction across different parallel lanes of the SIMD architecture to reduce their high timing error recovery overhead. Thus, their technique uses spatial memoization, as opposed to temporal memoization exploited by other techniques. Their technique memoizes the result of an error-free execution on a data item and reuses the memoized result to correct, either precisely or approximately, an errant execution on same or adjacent data items. Precise correction happens when the inputs of the instructions being compared match bit-by-bit, while approximate correction happens when inputs match only after masking a certain number of least significant fraction bits. Their SIMD architecture consists of a single strong lane and multiple weak lanes. The result of an exact FP instruction on a strong lane is memoized and is reused in case any weak lane sees an error. Thus, by leveraging instruction reuse, their technique avoids timing recovery for a large fraction of errant instructions, while keeping the quality loss due to approximation within bounds.

Keramidas et al. [2015] note that modern graphics applications perform high-precision computations; hence, memoizing the outcomes of a few instructions does not provide sufficient opportunities for value reuse. They propose using "approximate" value reuse for increasing the amount of successful value reuses. For this, a value cache is used that performs partial matches by reducing the accuracy of input parameters. They use approximation in fragment shaders, which compute the final color value of the pixel using arithmetic operations and texture fetches. They note that relaxing the precision of *arithmetic operations* leads to a negligible effect on perceptual quality, while doing this in *texture fetches* leads to a significance impact on output since texture coordinates are indices into an array. For this reason, the precision of arithmetic operations can be reduced more aggressively than that of texture fetches (e.g., dropping 12b vs. 4b). To maximize value reuse at runtime, they propose policies that reduce the precision of value cache progressively and monitor the resulting error. If a predefined number of errors have been detected, the precision of the value cache is increased again. They show that their technique reduces the operations executed with negligible perceptual quality loss.

4.5. Skipping Tasks and Memory Accesses

Several ACTs selectively skip memory references, tasks, or input portions to achieve efficiency with bounded QoR loss.

Samadi et al. [2014] present a SW-based ACT that works by identifying common patterns in data-parallel programs and using a specific approximation strategy for each pattern. They study approximation in six patterns, which are suitable for execution on multi/manycore architectures, (e.g., CPU and GPU) and are found in a wide range of applications. For scatter/gather and map patterns, they use memoization, whereby computations are substituted by memory accesses. For reduction patterns, sampling is used whereby output is computed by applying reduction on a subset of data. For scan patterns, the actual scan is performed on only a subset of input array, based on which the result is predicted for the remaining array. For stencil and partition patterns, neighboring locations in input array generally have similar values; based on this, an approximate version of the array is constructed by replicating a subset of values in the array. Based on a data-parallel kernel implemented in CUDA or OpenCL, their technique creates an approximate kernel version that is tuned at runtime to exercise a trade-off between performance and quality based on the user's QoR specification. Thus, their technique enables the user to write programs once and run them on multiple hardware platforms without manual code optimization. They show that, compared to precise execution, their technique achieves significant performance gains with acceptable quality loss.

Goiri et al. [2015] present general mechanisms for approximating the MapReduce framework. They use approximation strategies that are applicable for several MapReduce applications, that is, input data sampling (i.e., processing only a subset of data), task dropping (i.e., executing a fraction of tasks) and utilizing user-supplied approximate task code. They also use statistical theories to bound errors when approximating applications that employ aggregation reduce and extreme value reduce operations. They show the implementation of their technique in the Hadoop framework. They provide approximate classes that can be used by the programmer in place of regular Hadoop classes. These classes perform approximation, collect information for error estimation, perform reduce operations, predict final values and their confidence intervals, and output the results. As the job runs, their technique decides task dropping and/or data sampling ratios to meet user-specified error bounds at a particular confidence level. Alternatively, users can specify these ratios themselves; then, their technique computes the error bounds. They show that their technique allows trading accuracy for improving performance and energy saving.

4.6. Using Multiple Inexact Program Versions

We now discuss some ACTs that utilize multiple versions of application code with different trade-offs between accuracy and overhead.

Samadi et al. [2013] present an ACT for GPUs that allows trading off performance with accuracy based on the user-specified metric. Their technique works in two phases. In the offline compilation phase, a static compiler is used to create multiple versions of CUDA kernels with varying accuracy levels. In the runtime kernel management phase, a greedy algorithm is used to adjust parameters of approximate kernels for finding configurations that provide high performance and quality to meet the desired quality target. Use of a greedy algorithm avoids evaluation of all kernels, thus reducing the cost. To create approximate kernels, their technique uses three GPU-hardware-specific optimization approaches. In the first optimization, those atomic operations (used in kernels that write to a shared variable) are selectively skipped, which cause frequent conflicts and lead to poor performance on thread serialization. Thus, atomic operations that lead to mostly serial execution are eliminated, and those with little or no conflicts are left untouched. In the second optimization, the number of bits used to store input arrays is decreased to reduce the number of high-latency memory operations. In the third optimization, thread computations are selectively avoided by fusing adjacent threads into one and replicating one thread's output. Only threads that do not share data are candidates for fusion. To account for the runtime change in behavior of approximate kernels, periodic calibration is performed by running both the exact and approximate kernels on the GPU. Based on output quality and performance, the kernel configuration is updated to maintain a desired level of quality.

Baek and Chilimbi [2010] present a programming model framework that approximates functions and loops in a program and provides statistical guarantees of meeting the user-specified QoS target. The user provides one or more approximate versions of the function and the loops are approximated using loop perforation. Loss in QoS is measured by either a user-provided function or measuring difference in return value of the precise and approximate function versions. In the training phase, their technique uses the training input dataset to build a model for correlating performance and energy efficiency improvement from approximation to the resultant QoS loss. In the operational phase, approximation decisions are made based on this model and the user-specified QoS target. Since the difference in training inputs and actual inputs affects the accuracy of the model, their technique periodically measures the QoS loss seen at runtime and updates the approximation decision logic in order to provide statistical guarantees for meeting the QoS target. They show that their technique provides performance and energy gains with small and bounded QoS loss.

4.7. Using Inexact or Faulty Hardware

In this section, we first discuss design of a few inexact circuits, then discuss some ACTs that allow use of inexact/faulty circuits at the architecture level.

Kahng and Kang [2012] present the design of an inexact adder. For an N-bit inexact adder, (N/k - 1) subadders (each of which is a 2k-bit adder) are used to perform partial summations. The inexact adder avoids the carry chain to reduce critical-path delay, which can be used to improve performance and/or energy efficiency. When a carry input needs to be propagated to the result, the output of all (except the last) subadders becomes incorrect. With increasing k value, the probability of correct result increases but the dynamic power consumption and minimum clock period of the inexact adder also increase. Thus, by changing the value of k, the accuracy of the inexact adder can be controlled.

Kulkarni et al. [2011] present the design of an inexact 2x2 multiplier that represents the multiplication of $3_{10} * 3_{10}$ with 7_{10}, instead of 9_{10}. In other words, $11_2 * 11_2$ is represented with 111_2 instead of 1001_2, which uses three bits instead of four. For the remaining 15 input combinations, the multiplier provides correct output. Thus, while still providing a correct output for 15 out of 16 input combinations, their inexact multiplier reduces the area by half compared to the exact multiplier, leading to a shorter and faster critical path. By adding the shifted partial products from the 2x2 block, arbitrarily large multipliers can be built. By choosing between accurate and inexact versions of the 2x2 block in a large multiplier, a trade-off between error rate and power saving can be achieved. Further, for error-intolerant applications, they enhance their multiplier to detect the error magnitude and add it to the inexact output to produce the correct output.

Venkataramani et al. [2012] present an approach for designing general inexact circuits based on register transfer level (RTL) specification of the circuit and QoR metric (such as relative error). Their technique uses a quality function that determines whether a circuit meets the QoR requirement. Based on this, the input values can be found for which the quality function is insensitive to an output of inexact circuit. By leveraging these *approximation don't cares*, logic that generates that output can be simplified using conventional *don't care*-based synthesis approaches. Their technique iteratively performs this analysis for each output bit of inexact circuit; after each iteration, the inexact circuit setup is updated to account for the latest changes. They also discuss strategies for speeding up their technique, which allows its use for large circuits. They show that their technique can approximate both simple (e.g., adders, multipliers) and complex (e.g., discrete cosine transform, butterfly structure, FIR filter) circuits while providing area and energy advantages.

Ganapathy et al. [2015] present a technique for *minimizing the magnitude* of errors when using unreliable memories, which is in contrast to the ECC technique that actually *corrects* the errors. On each write, the data-word is circularly shifted for storing the least significant bits in the faulty cells of the memory. This skews the errors toward low-order bits, which reduces the magnitude of output error. To strike a balance between quality and performance/energy/area, their technique allows adaptation of the granularity of bit-shuffling. They show that, compared to using ECC, their technique achieves significant improvement in latency, power, and area. Also, use of their technique allows tolerating a limited number of faults, which reduces the manufacturing cost compared to the conventional zero-failure yield constraint.

Yetim et al. [2013] study the minimum error protection required from microarchitecture for mitigating control, memory addressing and I/O access faults for enabling approximate execution on a faulty processor. They use a macro instruction sequencer (MIS), a memory fence unit (MFU) and a streamed I/O approach, all of which seek to constrain execution depending on profiling information from the application. Based on the observation that a single-threaded streaming application can be logically divided into coarse-grained chunks of computations, the MIS constrains the control flow by limiting the allowed number of operations per chunk and by storing information about possible or legal series of chunks. The MFU checks whether accesses lie within the legal address range prescribed for a given chunk or instruction. For an out-of-range read/write access, the MFU either skips the memory instruction or references a dummy location to silence the fault. For an out-of-range fetch from instruction memory, silencing the instruction does not work since it can advance the program counter to yet another illegal instruction. Instead, for such faults, the MFU indicates the MIS to end the current chunk and start recovery from a known point. The streamed I/O approach allows only fixed-size streamed read/write operations and also limits the I/O operation count allowed per chunk or file. Bounding I/O to sequential access in this manner restricts the error-prone processor from accessing arbitrary addresses or data structures in the file system. They show that, even with reasonably frequent errors in video and audio applications, their technique still provides good output quality and avoids crashes and hangs.

4.8. Using Voltage Scaling

Voltage scaling reduces energy consumption of circuits at the cost of possible errors [Mittal 2015; Mittal and Vetter 2015]. For example, reducing SRAM supply voltage saves leakage energy but also increases probability of read upset (flipping of a bit during read operation) and write failure (writing a wrong bit) [Sampson et al. 2011]. Several ACTs use voltage scaling while accounting for these trade-offs.

Chippa et al. [2014] present an ACT that uses approximation at multiple levels of abstraction. For example, for k-means clustering, at *algorithm* level, early termination and convergence-based pruning are used. The former strategy, instead of terminating on full convergence, stops when the number of points changing clusters in successive iterations falls below a threshold. The latter strategy considers the points that have not changed their clusters for a predefined number of iterations as converged, and eliminates them from further computations. At *architecture* level, both input and intermediate variables are represented and operated on with scaled precision. This leaves some bit slices in the data path unutilized that are power-gated for saving energy. Alternatively, multiple data can be packed in the same word, and a single HW can process them simultaneously. At *circuit* level, voltage overscaling is used, without scaling the clock frequency. The adder circuit is segmented into adders of smaller bit-width. Based on voltage scaling, carry propagation across segmentation points is adaptively controlled and errors due to ignored carry values are reduced by using a low-cost correction circuit. They show that their approach provides significant energy saving with minor QoR loss and using approximation across the levels provides much larger improvement than using approximation only at a single level.

Rahimi et al. [2015] present an ACT for saving energy in GPUs. With each FPU in the GPU, they use a storage module composed of a ternary content addressable memory (TCAM) and a ReRAM block. Based on profiling, frequent redundant computations are identified in the GPU kernels, which are stored in the module. Reusing these values avoids later re-execution by the FPU; thus, this module performs a part of the functionality of the FPU. Further, under voltage overscaling, the error pattern of the module remains controllable; for instance, on reducing the voltage from 1V to 0.725V, the Hamming distance between an input (query) item and a computation stored in the module still remains 0, 1, or 2. For error-resilient GPU applications, this approach saves energy while still providing acceptable quality. Further, error-free storage of sign and exponent bits can be ensured by always using high voltage for them.

4.9. Reducing Branch Divergence in SIMD Architectures

On SIMD architectures, multiple threads executing the same set of instructions can diverge on a branch instruction (e.g., if-else and for/while loops). Some works seek to limit or avoid such divergence, which improves performance but introduces approximation.

Grigorian and Reinman [2015] present a technique for addressing the branch divergence issue on SIMD architectures. By characterizing the data-dependent control flow present in application, they identify the kernels responsible for highest performance loss due to branch divergence. Then, NNs are trained in an offline manner to approximate these kernels. Afterwards, the NNs are injected into the code itself by substituting the kernels. This entirely avoids the divergence problem by removing the control-flow-based code, at the cost of quality loss. Direct code modification obviates the need of costly HW modifications. They also provide a software framework for automating the entire technique and optimizations specific to different divergence patterns. Their experiments with GPUs using several divergent applications show that their technique provides energy and performance gains with minor quality loss.

Sartori and Kumar [2013] present two techniques: data herding and branch herding for reducing memory and control divergence in error-resilient GPU applications. In GPUs, a load instruction for a warp that creates requests for multiple memory regions leads to memory divergence. Memory coalescing finds the unique memory requests for satisfying individual scalar loads of a vector load instruction and multiple scalar loads in a warp are coalesced into one request only if they access consecutive addresses. Uncoalesced loads lead to divergence and BW wastage. To address this, their data-herding technique finds the most popular memory block to which the majority of loads are mapped, then maps all loads to that block. For implementing this, the number of loads coalescing into every potential memory request are compared; then, except for the most popular block, requests for all other blocks are discarded. Control divergence happens when different threads of a warp have different outcomes while evaluating the Boolean condition for a branch. Branch herding addresses this by finding the outcome of the majority of threads and then forcing all the threads to follow this outcome. They also propose a compiler framework and static analysis for avoiding data and control errors. They show that their techniques improve performance significantly with acceptable and controlled quality loss.

4.10. Use of Neural Network-Based Accelerators

Neural networks (NNs) expose significant parallelism and can be efficiently accelerated by dedicated hardware (NPU) to gain performance/energy benefits. We now discuss ACTs that work by mapping approximable code regions to NNs.

Esmaeilzadeh et al. [2012b] present an ACT that works by learning how an approximable code region works. In their programming model, programmers identify approximable imperative code regions. Then, an algorithmic program transformation is used that selects and trains an NN to mimic such code regions. Based on this learned model, the compiler replaces the original code with an invocation of low-power NPU. Their technique adds ISA extensions to configure and invoke the NPU. By virtue of automatically discovering and training NNs that are effective in approximating imperative codes, their technique extends applicability of NNs to a broad range of applications. Their experimental results confirm the performance and energy advantage of their technique.

McAfee and Olukotun [2015] present a SW-only approach for emulating and accelerating applications using NNs. Based on the programming language of Ansel et al. [2011], their technique generates a hierarchical application structure, for example, an 8×8 matrix can be decomposed into multiple 4×4 or 2×2 matrices. During compilation, their technique searches the subtask space of the application and generates a set of hierarchical task graphs that represent the application's functionality at varying resolution levels. Then, using a greedy approach, their technique chooses the subtasks to approximate along with the granularity of approximation, such that performance is maximized with minimal error. The emulation model is constantly updated using denoising autoencoders, so that the model may suit the current input well. Instead of modeling the entire application with a single complex NN, their technique emulates the application using multiple 2-layer linear NNs. NNs benefit greatly from highly optimized vector libraries and facilitate *learning* the model instead of requiring *explicit programming* by the user, which reduces the design cost. They show that, with bounded error, their technique achieves large speedup over precise computation.

Eldridge et al. [2014] propose MLP NN-based accelerators for approximating FP transcendental functions, that is, `cos`, `sin`, `exp`, `log`, and `pow`. They use a 3-stage internal neuron pipeline for multiplication of weight inputs and accumulation. They train the NN-based accelerator on limited input range (e.g., $[0, \pi/4]$ for `sin` function), then use mathematical identities to compute function value for any input value. They show that, compared to the conventional *glibc* (GNU C library) implementation, their implementation provides two orders of magnitude improvement in energy-delay-product with negligible loss in accuracy. They also build a SW library for invoking these accelerators from any application and show its use for the PARSEC benchmark suite.

Amant et al. [2014] present a technique for accelerating approximable code regions using limited-precision analog hardware through the NN approach. Using an algorithmic transformation, their technique automatically converts approximable code regions from a von Neumann representation to an analog neural representation. Use of the analog approach requires addressing challenges related to noise, circuit imprecision, limited accuracy of computation, limited range of encoded values, limitations on feasible NN topologies, and so on. To address these, they propose solutions at different layers of computing stacks. At the circuit level, a mixed-signal neural HW is used for multilayer perceptrons. The programmer marks approximable code regions using code annotations. The compiler mimics approximable code regions with an NN that can be executed on the mixed-signal neural HW. The error due to limited analog range is reduced by using a continuous-discrete learning method. Also, analog execution is kept limited to a single neuron, and communication between neurons happens in the digital domain. Limited analog range also restricts the bit weights used by neurons and the number of inputs to them. To reduce errors due to topology restrictions, a resilient backpropagation training algorithm is used instead of a conventional backpropagation algorithm. They show that their technique leads to improved performance and energy efficiency.

Li et al. [2015] propose a framework that uses a ReRAM-based AC unit (ACU) to accelerate approximate computations. The ACU is based on HW implementation of a 3-layer NN. Conceptually, the NN approximator performs matrix-vector multiplication of network weights and input variations and sigmoid activation function. Of these, they map matrix-vector multiplication to a ReRAM crossbar array and realize the sigmoid function using an NMOS/PMOS circuit [Li et al. 2015]. Several such ACUs are used together to perform algebraic calculus and achieve high performance. Digital signals are converted into analog signals for processing by ACUs and their outputs are again converted into digital format for further processing. For each task, these ACUs need to be trained; this is achieved by adjusting the network weights. Then, these weights are mapped to suitable conductance states of ReRAM devices in the crossbar arrays; the ReRAM devices are programmed to these states. They note that several complex tasks that require thousands of cycles in x86-64 architecture can be performed in a few cycles using the ACU. Their experiments show that their framework improves performance and power efficiency with acceptable quality loss.

4.11. Approximating Neural Networks

Based on the observation that NNs are typically used in error-tolerant applications and are resilient to many of their constituent computations, some researchers propose techniques to approximate them.

Venkataramani et al. [2014] present a technique for transforming a given NN into an approximate NN (AxNN) for allowing energy-efficient implementation. They use a backpropagation technique, which is used for training NNs, to quantify the impact of approximating any neuron to the overall quality. Afterwards, the neurons that have the least impact on network quality are replaced by their approximate versions to create an AxNN. By modulating the input precision and neuron weights, different approximate versions are created that allow trading off energy for accuracy. Since training is by itself an error-healing process, after creating the AxNN, they progressively retrain the network while using approximations to recover the quality loss due to AC. Thus, retraining allows further approximation for the same output quality. They also propose a neuromorphic processing engine for executing AxNNs with any weights, topologies, and degrees of approximation. This programmable HW platform uses neural computation units and activation function units together to execute AxNNs and exercise precision-energy trade-off at runtime.

Zhang et al. [2015] present a technique for approximating NNs. They define a neuron as critical (or resilient, respectively) if a small perturbation on its computation leads to large (or small) degradation in final output quality. They present a theoretical approach for finding the criticality factor of neurons in each output and hidden layer after the weights have been found in the training phase. The least critical neurons are candidates for approximation. However, due to the tight interconnection between neurons, the criticality ranking changes after approximation of each neuron. Hence, they use an iterative procedure, whereby the neurons for approximation are selected based on the quality budget. With successive iterations (i.e., moving toward convergence), the quality budget reduces; hence, the number of neurons selected in each iteration also reduces. In addition to finding the number of neurons, their technique also finds the amount of approximation performed (using three strategies, i.e., precision scaling, memory access skipping, and approximate multiplier circuits), by selecting a configuration that provides the largest energy saving for a given quality loss.

Du et al. [2014] propose the design of an inexact HW NN accelerator based on the observation that NNs allow retraining which allows suppressing the impact of neurons producing the largest amount of error. They assume a 2-layer feed-forward NN composed primarily of circuits for multiplying synaptic weight and neuron output and for adding the neuron inputs. Given the large number of possible ways of approximating NNs, they consider strategies for finding fruitful inexact configurations. Given the small HW cost of adders, approximating them yields only marginal returns; hence, they introduce approximation in synaptic weight multipliers only. Further, the output layer of an NN generally has a small number of neurons and since there is no synaptic weight after these neurons, lowering the errors in these neurons through retraining is difficult. Hence, they introduce approximation in synaptic weight multipliers of the hidden layers only. They show that, for applications that use HW NN accelerators, using their inexact NN accelerator provides significant savings in area, delay, and energy.

5. APPROXIMATE COMPUTING IN VARIOUS DEVICES AND COMPONENTS

Different memory technologies, processor components,and processing units offer different trade-offs, and design of effective ACTs requires accounting for their properties and trade-offs. Table V classifies different ACTs based on these factors. This table also organizes the ACTs based on their optimization target, which highlights that AC allows a designer to optimize multiple metrics of interest at the cost of a small loss in QoR. A few techniques perform power-gating of noncritical or faulty HW for saving energy, which has also been highlighted in Table V.

It is also noteworthy that different research works have used different evaluation platforms/approaches, such as simulators (e.g., Sampson et al. [2013], Miguel et al. [2014]), analytical models [Düben et al. 2015], actual CPU [Sampson et al. 2015], GPU [Samadi et al. 2013] and FPGA [Moreau et al. 2015; Sampson et al. 2015]. Further, as for search/optimization heuristics, researchers have used greedy algorithms [Samadi et al. 2013; Zhang et al. 2015; Sidiroglou et al. 2011; McAfee and Olukotun 2015; Ringenburg et al. 2014], divide and conquer [Venkataramani et al. 2012], gradient descent search [Ranjan et al. 2015], genetic algorithms [Ansel et al. 2011] and integer linear programming [Misailovic et al. 2014]. In what follows, we briefly discuss several ACTs.

Different memory technologies have different limitations, for example, SRAM and (e)DRAM consume high leakage and refresh power, respectively, and NVMs have high write energy/latency and low write endurance [Vetter and Mittal 2015; Mittal et al. 2015]. To reduce energy consumption of these memories and improve the lifetime of NVMs, approximate storage techniques sacrifice data integrity by reducing supply voltage in SRAM (Section 5.1) and refresh rate in (e)DRAM (Section 5.2) and by relaxing read/write operation in NVMs (Section 5.3).

5.1. Approximating SRAM Memory

Shoushtari et al. [2015] present an ACT for saving cache energy in error-tolerant applications. They assume that, in a cache, some ways are fault-free (due to use of suitable voltage and ECC), while other ways (called relaxed ways) may have faults due to use of lower supply voltage (V_{dd}). A cache block with more than a threshold (say K) number of faulty bits is disabled and power-gated. Thus, the values of V_{dd} and K together decide the active portion of the cache, and by selecting their values, a designer can relax guard bands for most of the cache ways while bounding the overall application error. They assume that the software programmer identifies noncritical data structures using suitable annotations, and the virtual addresses for them are kept in a table. On a cache miss, if the missed data block is noncritical, a victim block is selected from the relaxed ways. A critical data item is always stored in a fault-free block. Thus, voltage scaling and power-gating lead to leakage energy saving, and quality loss is kept minimal by approximating only the noncritical data.

5.2. Approximating eDRAM and DRAM Memories

Cho et al. [2014] present a technique for saving refresh energy in eDRAM-based frame buffers in video applications. They note that a change in higher-order bits of video data is more easily detected by the human eye than the change in lower-order bits, although completely discarding the lower-order bits makes the video lossy. Based on this, their technique trades off pixel data accuracy for saving energy in frame buffers. The memory array is divided into different segments, each of which can be refreshed at different periods. The pixel bits are arranged at a subpixel granularity and the highest-order bits of a pixel are allocated to the most reliable memory segment. Further, based on application characteristics and user preference, the number of segments and their refresh rates can be changed. They show that their technique saves significant refresh power without incurring loss in visual perception quality of the video.

Table V. A Classification Based on Processing Unit, Processor Component, Memory Technology and Optimization Objective

Classification	References
Memory technology	
NVM	Flash [Xu and Huang 2015], PCM [Fang et al. 2012; Sampson et al. 2013], STT-RAM [Ranjan et al. 2015], ReRAM [Li et al. 2015; Rahimi et al. 2015]
DRAM/eDRAM	[Cho et al. 2014; Liu et al. 2012; Sampson et al. 2011]
SRAM	[Esmaeilzadeh et al. 2012a; Ganapathy et al. 2015; Sampson et al. 2011; Shoushtari et al. 2015]
Processor Component	
Cache	[Düben et al. 2015; Esmaeilzadeh et al. 2012a; Keramidas et al. 2015; Misailovic et al. 2014; Sampson et al. 2011; Shoushtari et al. 2015; Sutherland et al. 2015]
Main memory	[Carbin et al. 2013; Düben et al. 2015; Liu et al. 2012; Misailovic et al. 2014; Sampson et al. 2011, 2013]
Secondary storage	[Sampson et al. 2013; Xu and Huang 2015]
Functional unit	[Carbin et al. 2013; Esmaeilzadeh et al. 2012a; Misailovic et al. 2014; Ringenburg et al. 2014; Sampson et al. 2011]
Floating-point unit	[Rahimi et al. 2013; Yeh et al. 2007]
Scratchpad	[Ranjan et al. 2015]
Processing unit or accelerator	
GPU	[Byna et al. 2010; Grigorian and Reinman 2015; Hsiao et al. 2013; Keramidas et al. 2015; Rahimi et al. 2013, 2015; Samadi et al. 2013, 2014; Samadi and Mahlke 2014; Sartori and Kumar 2013; Sutherland et al. 2015; Yazdanbakhsh et al. 2015b; Zhang et al. 2014]
FPGA	[Lopes et al. 2009; Moreau et al. 2015; Sampson et al. 2015]
ASIC	[Grigorian et al. 2015]
CPU	Nearly all others
Study/optimization objective and approach	
Energy saving	[Amant et al. 2014; Anam et al. 2013; Baek and Chilimbi 2010; Chakradhar and Raghunathan 2010; Chippa et al. 2014; Du et al. 2014; Düben et al. 2015; Eldridge et al. 2014; Esmaeilzadeh et al. 2012a, 2012b; Fang et al. 2012; Ganapathy et al. 2015; Goiri et al. 2015; Grigorian and Reinman 2015; Gupta et al. 2011; Hegde and Shanbhag 1999; Hsiao et al. 2013; Kahng and Kang 2012; Khudia et al. 2015; Kulkarni et al. 2011; Li et al. 2015; Liu et al. 2012; Mahajan et al. 2015; Miguel et al. 2014; Moreau et al. 2015; Raha et al. 2015; Rahimi et al. 2013; Rahimi et al. 2015; Ranjan et al. 2015; Ringenburg et al. 2015; Sampson et al. 2011, 2013, 2015; Shim et al. 2004; Shoushtari et al. 2015; Tian et al. 2015; Varatkar and Shanbhag 2008; Vassiliadis et al. 2015; Venkataramani et al. 2012, 2013, 2014, 2015; Xu and Huang 2015; Yazdanbakhsh et al. 2015a, 2015b; Yeh et al. 2007; Zhang et al. 2014, 2015]
Performance	[Amant et al. 2014; Anam et al. 2013; Ansel et al. 2011; Baek and Chilimbi 2010; Byna et al. 2010; Chakradhar and Raghunathan 2010; Du et al. 2014; Eldridge et al. 2014; Esmaeilzadeh et al. 2012b; Ganapathy et al. 2015; Goiri et al. 2015; Grigorian and Reinman 2015; Kahng and Kang 2012; Li et al. 2015; Mahajan et al. 2015; McAfee and Olukotun 2015; Miguel et al. 2014; Moreau et al. 2015; Rahimi et al. 2013; Samadi et al. 2013, 2014; Sampson et al. 2013, 2015; Sartori and Kumar 2013; Shi et al. 2015; Sidiroglou et al. 2011; Sutherland et al. 2015; Vassiliadis et al. 2015; Xu and Huang 2015; Yazdanbakhsh et al. 2015b; Yeh et al. 2007]
NVM lifetime	[Fang et al. 2012; Sampson et al. 2013]
Lowering error correction overhead	[Ganapathy et al. 2015; Rahimi et al. 2013; Shi et al. 2015; Xu and Huang 2015]
Power/clock-gating	[Chippa et al. 2014; Esmaeilzadeh et al. 2012a; Shoushtari et al. 2015; Venkataramani et al. 2013]

Liu et al. [2012] use a SW-based ACT for saving refresh power in DRAM memories. In their technique, the programmer identifies critical and noncritical data. At runtime, these data are allocated in different memory modules. The critical data are refreshed at regular rates, while the noncritical data are refreshed at much lower rate. Their results show that their technique saves significant refresh energy and, by virtue of introducing errors in only noncritical data, the degradation in application quality is kept minimal.

5.3. Approximating Nonvolatile Memories

Ranjan et al. [2015] present a technique for exploring quality–energy trade-off in STT-RAM, which introduces a small probability of errors in read/write operations for gaining large improvements in energy efficiency. They use three mechanisms for approximation: (1) lowering the current used to sense (read) the bit cells, which increases the probability of erroneous reads; (2) lowering the sensing duration and simultaneously increasing the read current, which increases the odds of flipping the bit cells on a read; and (3) lowering either write duration or write current or both, which may lead to occasional unsuccessful writes. Using these mechanisms, they design an adaptive-quality memory array in which reads/writes can be performed at different quality levels using additional peripheral circuits that can adjust the read/write duration and current magnitude. To control the numerical significance of errors, along with error probability, their technique allows specifying error-probability for different groups of bits. Using a device-level simulator, they study the trade-off between quality and bit-cell-level energy consumption. Further, they evaluate this memory array as a scratchpad for a vector processor [Venkataramani et al. 2013] (see Section 5.4) and utilize gradient descent search for determining minimum quality for each load/store instruction such that overall energy is minimized for a given output quality.

Sampson et al. [2013] present two ACTs for improving lifetime, density, and performance of NVMs in error-tolerant applications. The density of MLC NVM increases with the rising number of levels, although this also reduces the access speed of the memory due to the need of iterative programming. Their first technique reduces the number of programming pulses used to write the MLC memory. This can improve performance and energy efficiency for a given MLC memory at the cost of approximate writes. Alternatively, this can be used to improve density for a fixed power budget or performance target. Their second technique improves memory lifetime by storing approximate data in those blocks that have exhausted their HW error correction resources. Also, for reducing the effect of failed bits on the final result, higher priority is given to correction of higher-order bits compared to lower-order bits. They show that approximate writes in MLC PCM are much faster than precise writes, and using faulty blocks improves the lifetime with bounded quality loss.

Fang et al. [2012] propose a technique for reducing the number of writes to PCM by utilizing the error-tolerance property of video applications. When the new data to be written are same as the existing stored data, their technique cancels the write operation and takes the existing data themselves as the new data. While incurring only negligible reduction in video quality, their technique provides significant energy saving and lifetime improvement of the PCM-based memory.

We now discuss ACTs designed for different processor components (Section 5.4) and processing units, such as GPU (Section 5.5) and FPGA (Section 5.6), which take into account their architecture, operation characteristics, and criticality.

5.4. Using Approximation in Various Processor Components

In general-purpose cores, control units such as instruction fetch, decode, and retire consume a large fraction of energy. Since control operations are not easily approximable, general-purpose cores present limited opportunity for approximation. To address this, Venkataramani et al. [2013] present a quality-programmable processor (QPP), which allows specification of the desired quality (or accuracy) in the ISA itself, and thus allows AC to be applied to larger portions of the application. The microarchitecture of QPP leverages instruction-level quality specification for saving energy. Further, the actual error incurred in every instruction execution is exposed to the software, based on which quality specification for upcoming instructions is modulated. Their QPP features three different types of processing elements—APE, MAPE, and CAPE, which refer to approximate-, mixed accuracy- and completely accurate processing elements, respectively. APEs perform vector–vector reduction operations, which are commonly found in error-tolerant applications. MAPEs perform both control and arithmetic operations, and CAPEs perform operations related to control flow. These 3 elements provide different levels of quality versus energy trade-offs by using precision scaling with error monitoring and compensation. They show that QPP saves a significant amount of energy while incurring little loss in accuracy.

Esmaeilzadeh et al. [2012a] present extensions to ISA and corresponding microarchitecture for mapping AC to HW. Their ISA provides approximate versions of all FP and integer arithmetic and bitwise operation instructions provided by the original ISA. These instructions do not provide formal guarantee of accuracy, but only carry informal *expectation* of inexact adherence to the behavior of original instruction. They also define approximation granularity for setting the precision of cache memory. They logically divide the microarchitectural components into two groups: data movement/processing components (e.g., cache, register file, functional unit, load-store queue) and instruction control components (those dealing with fetching and decoding of instructions, among other things). The data-movement components are approximated only for approximate instructions, using lower supply voltage for such memory structures (e.g., cache) and logic, which saves energy at the cost of timing errors. By comparison, instruction control components are supplied with normal supply voltage for precise operation, which avoids catastrophic events (e.g., crashes) due to control flow violation. Their experiments show that their approach provides larger energy savings in in-order cores than in out-of-order cores, which is due to the fact that instruction control components contribute a much larger fraction of total energy in out-of-order cores than in in-order cores and their approach saves energy in only data-movement components.

5.5. Approximate Computing Techniques For GPUs

Hsiao et al. [2013] note that reducing the precision of FP representation and using the fixed-point representation are two commonly used strategies for reducing energy consumption of a GPU shader. Of these, reduced-precision FP provides wider numerical range but also consumes higher latency and energy; the opposite is true for fixed-point representation. They propose an automatic technique that intelligently chooses between these two strategies. Their technique performs runtime profiling to record precision information and determine feasible precisions for both fragment and vertex shading. Then both these rendering strategies are evaluated with selected precisions to find the more energy-efficient strategy for the current application. Finally, the winning strategy with its precision is used for the successive frames, except that memory access-related and other critical operations are executed in full-precision FP. They show that their technique provides higher energy saving and quality than using either strategy alone.

Zhang et al. [2014] note that the FPU and special function units are used only in arithmetic operations (but not in control/memory operations) and they contribute a large fraction of GPU power consumption. Thus, using inexact HW for them can provide large energy savings at bounded quality loss without affecting correctness. Based on this, their technique uses linear approximation within a reduced range for functions such as square root, reciprocal, log, FP multiplication and division, for example, $y = 1/\sqrt{x}$ is approximated as $y = 2.08 - 1.1911x$ in the range $x \in [0.5, 1]$. They build the functional models of inexact HW and import them in a GPU simulator that can also model GPU power consumption. In the simulator, each inexact HW unit can be activated or deactivated and their parameters can be tuned. They first obtain the reference (exact) output, then run a functional simulation with inexact units to obtain the inexact output. By comparing the reference and inexact output using an application-specific metric, the quality loss is estimated. If the loss exceeds a threshold, then either an inexact unit is disabled or its parameters are adjusted, depending on the program-specific error sensitivity characterization. The simulation is performed again with an updated configuration and this process is repeated until the quality loss becomes lower than the threshold. They show that their technique allows exercising trade-off between quality and system power and provides large power savings for several compute-intensive GPU programs.

Byna et al. [2010] present an ACT for accelerating a supervised semantic indexing (SSI) algorithm, which is used for organizing unstructured text repositories. Due to the data dependencies between the iterations of SSI, parallelism can only be exploited within individual iterations. Hence, for small datasets, GPU implementation of SSI does not fully utilize the GPU HW resources. They note that SSI is an error-tolerant algorithm and the spatial locality of writes between different iterations of SSI is low, (i.e., these iterations rarely update the same part of the model). Also, after some initial iterations, only a few iterations perform any updates. Using these properties, dependencies between iterations can be intelligently relaxed, then multiple iterations can be run in parallel. Also, noncritical computation is avoided, for example, processing of common words, such as “a,” “of,” and “the,” are avoided since it does not affect the accuracy. They show that their technique improves performance compared to both a baseline GPU implementation and a multicore CPU implementation.

5.6. Approximate Computing Techniques For FPGAs

Moreau et al. [2015] present a technique for neural acceleration of approximable codes on a programmable system on chip (SoC), that is, an FPGA. Their technique can be used as either a high-level mechanism in which approximable codes are offloaded to FPGA using a compiler, or at low-level in which experienced programmers can exercise fine-grained control using an instruction-level interface. Since use of programmable logic such as an FPGA faces the challenges of large communication latency between the CPU and FPGA and large differences in their speeds, they propose throughput-oriented operation of an FPGA-based accelerator, such that invocations of NNs are grouped and sent together to the FPGA-based accelerator. Thus, the accelerator does not block program execution since numerous invocations keep the accelerator busy and individual invocations need not complete immediately. While other works (e.g., Amant et al. [2014] and Esmaeilzadeh et al. [2012b]) implement NPU in fully custom logic and integrate it with the host CPU pipeline, their technique implements NPU in an off-the-shelf FPGA, without closely integrating it with the CPU. This avoids changes to ISA of the processor and enables using neural acceleration in devices that are already available commercially. Instead of configuring the programmable logic (FPGA), their technique configures the NN topology and weights themselves. Thus, while requiring expertise much lower than that required for programming FPGAs and even using high-level synthesis tools, their technique can accelerate a broad range of applications. Their experiments show that their approach provides speedup and energy saving with bounded degradation of QoR.

Lopes et al. [2009] note that, for achieving a desired final QoR with iterative solvers, a user can lower the precision of intermediate computations and run more iterations or vice versa, as opposed to direct solvers, for which final QoR depends only on the precision of intermediate computations. On FPGA, for a fixed area constraint, lowering the precision of the iterative solver allows greater parallelism. Based on this, they present an ACT for accelerating the solution of a system of linear equations on an FPGA. They plot the variation of computation time and iteration-count (for convergence) with different mantissa widths. From these two plots, the optimum mantissa-width (i.e., precision) that minimizes the computation time for a desired QoR is found. Thus, by balancing the operation precision and iteration count, they achieve a large performance improvement over a double-precision implementation, while achieving the same final QoR.

5.7. Using Scalable Effort Design For Approximate Computing

In several cases, the level of effort required for performing different tasks of an application may be different. Several ACTs use this feature to tune the effort expended on a per-task basis.

Grigorian et al. [2015] present a technique that uses both precise and NN-based approximate accelerators, while enforcing accuracy constraints using error analysis. The execution starts with computationally easier approximations, continues onto more complex ones and ends with a precise computation. After each stage (except the last one), application-specific LWCs (see Section 3.2) are performed to measure output quality. The tasks with acceptable output quality are committed and only remaining tasks go to the next stage. Most tasks are expected to be committed in early stages, which reduces the overall overhead. Further, by choosing the LWCs suitably, user-specified accuracy can be achieved at runtime. NN models allow training for different functionality without the need to modify topology, which obviates the need of design flexibility provided by FPGAs; hence, they implement all the accelerators in digital ASIC technology. They show that, compared to using either precise accelerators alone or software-based computation, their technique provides significant energy and performance advantage.

Venkataramani et al. [2015] present a technique for improving energy efficiency of supervised machine-learning classifiers. They note that, in real-world datasets, different inputs demand different classification efforts, and only a small fraction of inputs need full computational capability of the classifier. Using this, their technique allows the dynamic tuning of the computational efforts based on difficulty of input data. They use a chain of classifiers with increasing complexity and classification accuracy. Each stage in the chain has multiple biased classifiers that can detect a single class with high precision. Depending on the consensus between outputs of different classifiers, confidence in a classification is judged based on which the decision to terminate at a stage is made. Thus, only hard inputs go through multiple stages, while simpler inputs get processed in only a few stages. Classifiers of different complexity and accuracy are created by modulating their algorithm parameters, for example, neuron and layer count in an NN. Their technique also allows trading off the number of states, their complexity and input fraction classified at every stage to optimize the overall classification efforts. They show that their technique reduces the number of operations per input and energy consumption of the classification process.

5.8. Reducing Error-Correction Overhead Using Approximate Computing

Conventional error-correction mechanisms such as redundant execution and use of ECC incur high overhead [Mittal and Vetter 2015]. Some ACTs leverage the error-tolerance property of applications to reduce this overhead.

Shi et al. [2015] present a technique for trading application accuracy with soft-error resilience overhead, while achieving 100% soft-error coverage. The programmer identifies the noncritical code such as perforable loops. Based on this, redundant execution for soft-error protection is applied only to the critical code. When redundant execution is turned off for the noncritical code, low-overhead resilience mechanisms, such as checking of store instructions by HW-level redundant address calculation, are used for ensuring application correctness. Due to soft errors, results of certain iterations in a perforable loop may have to be dropped in the worst case. This, however, only affects accuracy and not correctness. For a multicore processor, resilience is independently modulated for each core. They show that their technique reduces the performance overhead of resilience.

Xu and Huang [2015] leverage error-tolerance capability of datacentric applications for reducing ECC overhead in Flash-based SSDs. Using error-injection experiments, they show that different data have different impacts on output quality, and their applications can tolerate a much higher error rate than that targeted by Flash SSDs. They design a framework for monitoring and estimating soft-error rates of Flash SSD at runtime. Based on the error rate of the SSD obtained from this framework and the error rate that can be tolerated by the application, their technique dynamically lowers the ECC protection or avoids using ECC altogether. They show that their approach provides significant performance and energy efficiency gains with an acceptable QoR.

6. APPLICATION AREAS OF APPROXIMATE COMPUTING

To show the spectrum of application of AC, Table VI roughly classifies the techniques based on their application/workload domain. Note that these categories are not mutually exclusive and may include others as subcategories. Of the benchmark suites, PARSEC (e.g., Sidiroglou et al. [2011]), SciMark (e.g., Sampson et al. [2011]), MediaBench (e.g., Liu et al. [2012]), PhysicsBench (e.g., Yeh et al. [2007]), UCI machine-learning repository (e.g., Du et al. [2014]), Caltech 101 computer vision dataset (e.g., Rahimi et al. [2015]), SPEC benchmark (e.g., Roy et al. [2014]), MiBench (e.g., Roy et al. [2014]), and more, have been frequently utilized. These domains and benchmark suites find application in or represent many other real-life problems also, such as robotics, artificial intelligence, and fluid dynamics, which clearly shows the growing importance of AC.

7. CONCLUSION AND FUTURE CHALLENGES

In this article, we surveyed the techniques proposed for approximate computing. We highlighted the opportunities and obstacles in the use of AC, then organized the techniques in several groups to provide a bird's eye view of the research field. We conclude this article with a brief mention of challenges that lie ahead in this field.

Most existing ACTs have focused on multimedia applications and iterative algorithms. However, these error-resilient workloads comprise only a fraction of the computational workloads. As these "low-hanging fruits" gradually vanish, researchers will have to turn their attention to general-purpose applications, thus extend the scope of AC to the entire spectrum of computing applications.

Several current large-scale software packages have been written in conventional languages that assume precise operations and storage. Facilitating development of code that fully exploits the inexact hardware and approximation strategies while also meeting quality expectations requires a powerful and, yet, intuitive and simple programming language. Significant work is required to transform today's research-stage programming frameworks for AC into mature and robust code development platforms of tomorrow.

Using approximation in a single-processor component alone can have an unforeseen effect on the operation of other components and is likely to lead to erroneous or myopic conclusions. Going forward, a comprehensive evaluation of the effect of AC on the entire system and use of AC in multiple components is required. Further, since existing systems use several management schemes such as data compression, prefetching, and

Table VI. A Classification Based on Research Fields, Frameworks or Applications Where AC is Used

Classification	References
Image processing or multimedia	[Amant et al. 2014; Ansel et al. 2011; Baek and Chilimbi 2010; Chippa et al. 2013; Cho et al. 2014; Esmaeilzadeh et al. 2012a, 2012b; Fang et al. 2012; Goiri et al. 2015; Grigorian et al. 2015; Grigorian and Reinman 2014, 2015; Gupta et al. 2011; Hsiao et al. 2013; Keramidas et al. 2015; Khudia et al. 2015; Kulkarni et al. 2011; Li et al. 2015; Liu et al. 2012; Mahajan et al. 2015; McAfee and Olukotun 2015; Miguel et al. 2014; Misailovic et al. 2014; Mishra et al. 2014; Moreau et al. 2015; Raha et al. 2015; Rahimi et al. 2013, 2013, 2015; Ringenburg et al. 2014, 2015; Roy et al. 2014; Samadi et al. 2013, 2014; Samadi and Mahlke 2014; Sampson et al. 2011, 2013; Sartori and Kumar 2013; Shoushtari et al. 2015; Sidiroglou et al. 2011; Sutherland et al. 2015; Vassiliadis et al. 2015; Xu and Huang 2015; Yazdanbakhsh et al. 2015a; Yeh et al. 2007; Yetim et al. 2013; Zhang et al. 2014]
Signal processing	[Amant et al. 2014; Esmaeilzadeh et al. 2012a, 2012b; Grigorian and Reinman 2014; Gupta et al. 2011; Hegde and Shanbhag 1999; Li et al. 2015; McAfee and Olukotun 2015; Samadi et al. 2014; Varatkar and Shanbhag 2008; Venkataramani et al. 2012; Yazdanbakhsh et al. 2015a]
Machine learning	[Amant et al. 2014; Ansel et al. 2011; Baek and Chilimbi 2010; Chakradhar and Raghunathan 2010; Chippa et al. 2014; Du et al. 2014; Esmaeilzadeh et al. 2012b; Goiri et al. 2015; Khudia et al. 2015; Li et al. 2015; Moreau et al. 2015; Raha et al. 2015; Ranjan et al. 2015; Samadi et al. 2013; Samadi et al. 2014; Shi et al. 2015; Tian et al. 2015; Vassiliadis et al. 2015; Venkataramani et al. 2013; Venkataramani et al. 2014; Xu and Huang 2015; Zhang et al. 2015]
Scientific computing	[Carbin et al. 2013; Eldridge et al. 2014; Esmaeilzadeh et al. 2012a; Grigorian et al. 2015; Grigorian and Reinman 2015; Khudia et al. 2015; McAfee and Olukotun 2015; Misailovic et al. 2014; Moreau et al. 2015; Ringenburg et al. 2015; Roy et al. 2014; Sampson et al. 2011, 2013; Xu and Huang 2015; Yazdanbakhsh et al. 2015a]
Financial analysis	[Amant et al. 2014; Baek and Chilimbi 2010; Grigorian and Reinman 2014, 2015; Khudia et al. 2015; Mahajan et al. 2015; Miguel et al. 2014; Misailovic et al. 2014; Moreau et al. 2015; Ringenburg et al. 2015; Samadi et al. 2014; Shi et al. 2015; Sidiroglou et al. 2011; Zhang et al. 2015]
Database search	[Baek and Chilimbi 2010; Byna et al. 2010; Chippa et al. 2013; Düben et al. 2015; Miguel et al. 2014; Sidiroglou et al. 2011; Venkataramani et al. 2013]
MapReduce	[Goiri et al. 2015; Xu and Huang 2015]

dynamic voltage/frequency scaling, ensuring synergy of ACTs with these schemes is important for smooth integration of AC in commercial systems.

As the quest for performance confronts resource constraints, major breakthroughs in computing efficiency are expected to come from unconventional approaches. We hope for a promising near future in which approximate computing helps in continuing the historic trends of performance scaling and becomes *the* mainstream computing approach for all classes of processors.

REFERENCES

Ismail Akturk, Karen Khatamifard, and Ulya R. Karpuzcu. 2015. On quantification of accuracy loss in approximate computing. *Workshop on Duplicating, Deconstructing and Debunking.*

Carlos Alvarez, Jesus Corbal, and Mateo Valero. 2005. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computers* 54, 7, 922–927.

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 *International Symposium on Computer Architecture*. 505–516.

Mohammad Ashraful Anam, Paul Whatmough, and Yiannis Andreopoulos. 2013. Precision-energy-throughput scaling of generic matrix multiplication and discrete convolution kernels via linear projections. In *Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia'13)*. 21–30.

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 *International Symposium on Code Generation and Optimization*. 85–96.

Woongki Baek and Trishul M. Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. In *ACM SIGPLAN Notices*, Vol. 45. 198–209.

James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. 2014. Uncertain<T>: A first-order type for uncertain data. *ACM SIGARCH Computer Architecture News* 42, 1, 51–66.

Surendra Byna, Jiayuan Meng, Anand Raghunathan, Srimat Chakradhar, and Srihari Cadambi. 2010. Best-effort semantic document search on GPUs. In *General-Purpose Computation on Graphics Processing Units*. 86–93.

Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *ACM SIGPLAN Notices*, Vol. 48. 33–52.

Srimat T. Chakradhar and Anand Raghunathan. 2010. Best-effort computing: Re-thinking parallel software and hardware. In *Design Automation Conference*. 865–870.

Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and characterization of inherent application resilience for approximate computing. In *Design Automation Conference*. 113.

Vinay K. Chippa, Debabrata Mohapatra, Kaushik Roy, Srimat T. Chakradhar, and Anand Raghunathan. 2014. Scalable effort hardware design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 9, 2004–2016.

Kyungsang Cho, Yongjun Lee, Young H. Oh, Gyoo-cheol Hwang, and Jae W. Lee. 2014. eDRAM-based tiered-reliability memory with applications to low-power frame buffers. In *International Symposium on Low Power Electronics and Design*. 333–338.

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 *Asia and South Pacific Design Automation Conference (ASP-DAC'14)*. 201–206.

Peter Düben, Jeremy Schlachter, Parishkrati, Sreelatha Yenugula, John Augustine, Christian Enz, K. Palem, and T. N. Palmer. 2015. Opportunities for energy efficient computing: A study of inexact general purpose processors for high-performance and big-data applications. In *Design, Automation & Test in Europe*. 764–769.

Schuyler Eldridge, Florian Raudies, David Zou, and Ajay Joshi. 2014. Neural network-based accelerators for transcendental function approximation. In *Great Lakes Symposium on VLSI*. 169–174.

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012a. Architecture support for disciplined approximate programming. In *ACM SIGPLAN Notices*, Vol. 47. 301–312.

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012b. Neural acceleration for general-purpose approximate programs. In *IEEE/ACM International Symposium on Microarchitecture*. 449–460.

Yuntan Fang, Huawei Li, and Xiaowei Li. 2012. SoftPCM: Enhancing energy efficiency and lifetime of phase change memory in video applications via approximate write. In *IEEE Asian Test Symposium (ATS)*. 131–136.

Shrikanth Ganapathy, Georgios Karakonstantis, Adam Shmuel Teman, and Andreas Peter Burg. 2015. Mitigating the impact of faults in unreliable memories for error-resilient applications. In *Design Automation Conference*.

John Gantz and David Reinsel. 2011. Extracting Value from Chaos. Retrieved February 25, 2016 from http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf.

Íñigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. 2015. ApproxHadoop: Bringing approximations to MapReduce frameworks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 383–397.

Beayna Grigorian, Nazanin Farahpour, and Glenn Reinman. 2015. BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing. In *International Symposium on High Performance Computer Architecture (HPCA'15)*. 615–626.

Beayna Grigorian and Glenn Reinman. 2014. Dynamically adaptive and reliable approximate computing using light-weight error analysis. In *Conference on Adaptive Hardware and Systems (AHS'14)*. 248–255.

Beayna Grigorian and Glenn Reinman. 2015. Accelerating divergent applications on SIMD architectures using neural networks. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 1, 2.

Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. 2011. IMPACT: Imprecise adders for low-power approximate computing. In *International Symposium on Low Power Electronics and Design*. 409–414.

Rajamohana Hegde and Naresh R. Shanbhag. 1999. Energy-efficient signal processing via algorithmic noise-tolerance. In *International Symposium on Low Power Electronics and Design*. 30–35.

Chih-Chieh Hsiao, Slo-Li Chu, and Chen-Yu Chen. 2013. Energy-aware hybrid precision selection framework for mobile GPUs. *Computers and Graphics* 37, 5, 431–444.

Andrew B. Kahng and Seokhyeong Kang. 2012. Accuracy-configurable adder for approximate arithmetic designs. In *Design Automation Conference*. 820–825.

Georgios Keramidas, Chrysa Kokkala, and Iakovos Stamoulis. 2015. Clumsy value cache: An approximate memoization technique for mobile GPU fragment shaders. *Workshop on Approximate Computing (WAPCO'15).*

Daya S. Khudia, Babak Zamirai, Mehrzad Samadi, and Scott Mahlke. 2015. Rumba: An online quality management system for approximate computing. In *International Symposium on Computer Architecture*. 554–566.

Parag Kulkarni, Puneet Gupta, and Milos Ercegovac. 2011. Trading accuracy for power with an underdesigned multiplier architecture. In *International Conference on VLSI Design (VLSI Design'11)*. 346–351.

B. Li, P. Gu, Y. Shan, Y. Wang, Y. Chen, and H. Yang. 2015. RRAM-based analog approximate computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.*

Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2012. Flikker: Saving DRAM refresh-power through critical data partitioning. *ACM SIGPLAN Notices* 47, 4, 213–224.

Antonio Roldao Lopes, Amir Shahzad, George Constantinides, Eric C. Kerrigan, and others. 2009. More flops or more precision? Accuracy parameterizable linear equation solvers for model predictive control. In *Symposium on Field Programmable Custom Computing Machines (FCCM'09)*. 209–216.

Divya Mahajan, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, and Hadi Esmaeilzadeh. 2015. Prediction-based quality control for approximate accelerators. *Workshop on Approximate Computing Across the System Stack.*

Lawrence McAfee and Kunle Olukotun. 2015. EMEURO: A framework for generating multi-purpose accelerators via deep learning. In *International Symposium on Code Generation and Optimization*. 125–135.

Joshua San Miguel, Mario Badr, and Enright Natalie Jerger. 2014. Load value approximation. *MICRO.*

Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *International Conference on Object Oriented Programming Systems Languages and Applications*. 309–328.

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).*

Sparsh Mittal. 2012. A survey of architectural techniques for DRAM power management. *International Journal of High Performance Systems Architecture* 4, 2, 110–119.

Sparsh Mittal. 2014a. *Power Management Techniques for Data Centers: A Survey*. Technical Report ORNL/TM-2014/381. Oak Ridge National Laboratory, Oak Ridge, TN.

Sparsh Mittal. 2014b. A survey of architectural techniques for improving cache power efficiency. *Sustainable Computing: Informatics and Systems* 4, 1, 33–43.

Sparsh Mittal. 2015. A survey of architectural techniques for near-threshold computing. *ACM Journal on Emerging Technologies in Computing Systems* 12, 4, 46:1–46:26.

Sparsh Mittal and Jeffrey Vetter. 2015. A survey of techniques for modeling and improving reliability of computing systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS).*

Sparsh Mittal, Jeffrey S. Vetter, and Dong Li. 2015. A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 26, 6, 1524–1537.

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 *International Symposium on High Performance Computer Architecture (HPCA'15)*. 603–614.

NRDC. 2015. Retrieved February 25, 2016 from http://www.nrdc.org/energy/data-center-efficiency-assessment.asp.

Azar Rahimi, Luca Benini, and Rajesh K. Gupta. 2013. Spatial memoization: Concurrent instruction reuse to correct timing errors in SIMD architectures. *IEEE Transactions on Circuits and Systems II: Express Briefs* 60, 12, 847–851.

Abbas Rahimi, Amirali Ghofrani, Kwang-Ting Cheng, Luca Benini, and Rajesh K. Gupta. 2015. Approximate associative memristive memory for energy-efficient GPUs. In *Design, Automation and Test in Europe*. 1497–1502.

Arnab Raha, Swagath Venkataramani, Vijay Raghunathan, and Anand Raghunathan. 2015. Quality configurable reduce-and-rank for energy efficient approximate computing. In *Design, Automation and Test in Europe*. 665–670.

Abbas Rahimi, Andrea Marongiu, Rajesh K. Gupta, and Luca Benini. 2013. A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters. In *International Conference on Hardware/Software Codesign and System Synthesis*. 35.

Ashish Ranjan, Swagath Venkataramani, Xuanyao Fong, Kaushik Roy, and Anand Raghunathan. 2015. Approximate storage for energy efficient spintronic memories. In *Design Automation Conference*. 195.

Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. 2015. Monitoring and debugging the quality of results in approximate programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 399–411.

Michael F. Ringenburg, Adrian Sampson, Luis Ceze, and Dan Grossman. 2014. Profiling and autotuning for energy-aware approximate programming. In *Workshop on Approximate Computing Across the System Stack (WACAS'14)*.

Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. 2014. ASAC: Automatic sensitivity analysis for approximate computing. In *Conference on Languages, Compilers and Tools For Embedded Systems*. 95–104.

Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based approximation for data parallel applications. In *ACM SIGARCH Computer Architecture News*, Vol. 42. 35–50.

Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning approximation for graphics engines. In *International Symposium on Microarchitecture*. 13–24.

Mehrzad Samadi and Scott Mahlke. 2014. CPU-GPU collaboration for output quality monitoring. In *Workshop on Approximate Computing Across the System Stack*. 1–3.

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*. Technical Report UW-CSE-15-01-01. University of Washington, Seattle, WA.

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 *ACM SIGPLAN Notices*, Vol. 46. 164–174.

Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. 2013. Approximate storage in solid-state memories. In *International Symposium on Microarchitecture*. 25–36.

John Sartori and Ravindra Kumar. 2013. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. *IEEE Transactions on Multimedia* 15, 2, 279–290.

Qinfeng Shi, Henry Hoffmann, and Omar Khan. 2015. A HW-SW multicore architecture to tradeoff program accuracy and resilience overheads. *Computer Architecture Letters*.

Byonghyo Shim, Srinivasa R. Sridhara, and Naresh R. Shanbhag. 2004. Reliable low-power digital signal processing via reduced precision redundancy. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12, 5, 497–510.

Majid Shoushtari, Abbas BanaiyanMofrad, and Nikil Dutt. 2015. Exploiting partially-forgetful memories for approximate computing. *IEEE Embedded Systems Letters* 7, 1, 19–22.

Stelios Sidiroglou, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 124–134.

Mark Sutherland, Joshua San Miguel, and Natalie Enright Jerger. 2015. Texture cache approximation on GPUs. *Workshop on Approximate Computing Across the Stack*.

Ye Tian, Qian Zhang, Ting Wang, Feng Yuan, and Qiang Xu. 2015. ApproxMA: Approximate memory access for dynamic precision scaling. In *ACM Great Lakes Symposium on VLSI*. 337–342.

Girish Vishnu Varatkar and Naresh R. Shanbhag. 2008. Error-resilient motion estimation architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16, 10, 1399–1412.

Vassilis Vassiliadis, Konstantinos Parasyris, Charalambos Chaliosy, Christos D. Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendoncky, and Dimitrios S. Nikolopoulos. 2015. A programming model and runtime system for significance-aware energy-efficient computing. In *1st Workshop On Approximate Computing (WAPCO'15)*.

Swagath Venkataramani, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2015. Approximate computing and the quest for computing efficiency. In *Design Automation Conference*. 120.

Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Quality programmable vector processors for approximate computing. In *International Symposium on Microarchitecture*. 1–12.

Swagath Venkataramani, Anand Raghunathan, Jie Liu, and Mohammed Shoaib. 2015. Scalable-effort classifiers for energy-efficient machine learning. In *Design Automation Conference*. 67.

Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. 2014. AxNN: Energy-efficient neuromorphic systems using approximate computing. In *International Symposium on Low Power Electronics and Design*. 27–32.

Swagath Venkataramani, Amit Sabne, Vivek Kozhikkottu, Kaushik Roy, and Anand Raghunathan. 2012. SALSA: Systematic logic synthesis of approximate circuits. In *Design Automation Conference*. 796–801.

Jeffrey S. Vetter and Sparsh Mittal. 2015. Opportunities for nonvolatile memory systems in extreme-scale high performance computing. *Computing in Science and Engineering* 17, 2, 73–82.

Xin Xu and H. Howie Huang. 2015. Exploring data-level error tolerance in high-performance solid-state drives. *IEEE Transactions on Reliability* 64, 1, 15–30.

Amir Yazdanbakhsh, Divya Mahajan, Bradley Thwaites, Jongse Park, Anandhavel Nagendrakumar, Sindhuja Sethuraman, Kartik Ramkrishnan, Nishanthi Ravindran, Rudra Jariwala, Abbas Rahimi, Hadi Esmaeilzadeh, and Kia Bazargan. 2015a. Axilog: Language support for approximate hardware design. In *Design, Automation and Test in Europe Conference and Exhibition*. 812–817.

Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Taesoo Kim, Onur Mutlu, and Todd C. Mowry. 2015b. *RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads*. Technical Report. Georgia Institute of Technology, Atlanta, GA.

Thomas Y. Yeh, Petros Faloutsos, Milos Ercegovac, Sanjay J. Patel, and Glenn Reinman. 2007. The art of deception: Adaptive precision reduction for area efficient physics acceleration. In *International Symposium on Microarchitecture*. 394–406.

Yavuz Yetim, Margaret Martonosi, and Sharad Malik. 2013. Extracting useful computation from error-prone processors for streaming applications. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'13)*. 202–207.

Hang Zhang, Mateja Putic, and John Lach. 2014. Low power GPGPU computation with imprecise hardware. In *Design Automation Conference (DAC'14)*. 1–6.

Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. 2015. ApproxANN: An approximate computing framework for artificial neural network. In *Design, Automation and Test in Europe*. 701–706.

Received July 2015; revised January 2016; accepted January 2016