

Approximate Computing Survey, Part I: Terminology and Software & Hardware Approximation Techniques

VASILEIOS LEON, National Technical University of Athens, Greece

MUHAMMAD ABDULLAH HANIF, New York University Abu Dhabi, United Arab Emirates

GIORGOS ARMENIAKOS, National Technical University of Athens, Greece

XUN JIAO, Villanova University, United States

MUHAMMAD SHAFIQUE, New York University Abu Dhabi, United Arab Emirates

KIAMAL PEKMESTZI, National Technical University of Athens, Greece

DIMITRIOS SOUDRIS, National Technical University of Athens, Greece

The rapid growth of demanding applications in domains applying multimedia processing and machine learning has marked a new era for edge and cloud computing. These applications involve massive data and compute-intensive tasks, and thus, typical computing paradigms in embedded systems and data centers are stressed to meet the worldwide demand for high performance. Concurrently, over the last 15 years, the semiconductor industry has established power efficiency as a first-class design concern. As a result, the community of computing systems is forced to find alternative design approaches to facilitate high-performance and power-efficient computing. Among the examined solutions, *Approximate Computing* has attracted an ever-increasing interest, which has resulted in novel approximation techniques for all the layers of the traditional computing stack. More specifically, during the last decade, a plethora of approximation techniques in software (programs, frameworks, compilers, runtimes, languages), hardware (circuits, accelerators), and architectures (processors, memories) have been proposed in the literature. The current article is Part I of a comprehensive survey on Approximate Computing. It reviews its motivation, terminology and principles, as well it classifies the state-of-the-art software & hardware approximation techniques, presents their technical details, and reports a comparative quantitative analysis.

CCS Concepts: • General and reference → Surveys and overviews; • Software and its engineering → Software notations and tools; • Hardware → Integrated circuits; • Computer systems organization → Architectures.

Additional Key Words and Phrases: Inexact Computing, Approximation Method, Approximate Programming, Approximation Framework, Approximate Circuit, Approximate Arithmetic, Error Resilience, Accuracy

ACM Reference Format:

Vasileios Leon, Muhammad Abdullah Hanif, Giorgos Armeniakos, Xun Jiao, Muhammad Shafique, Kiamal Pekmestzi, and Dimitrios Soudris. “Approximate Computing Survey, Part I: Terminology and Software & Hardware Approximation Techniques”. *ACM Computing Surveys*, vol. 57, no. 7, pp. 1–36, 2025. <https://doi.org/10.1145/3716845>

Vasileios Leon, National Technical University of Athens, School of Electrical and Computer Engineering, Greece; Muhammad Abdullah Hanif, New York University Abu Dhabi, Division of Engineering, United Arab Emirates; Giorgos Armeniakos, National Technical University of Athens, School of Electrical and Computer Engineering, Greece; Xun Jiao, Villanova University, Department of Electrical and Computer Engineering, United States; Muhammad Shafique, New York University Abu Dhabi, Division of Engineering, United Arab Emirates; Kiamal Pekmestzi, National Technical University of Athens, School of Electrical and Computer Engineering, Greece; Dimitrios Soudris, National Technical University of Athens, School of Electrical and Computer Engineering, Greece.

© Owner/Authors 2025. This is the authors' version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in ACM Computing Surveys, <https://doi.org/10.1145/3716845>.



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](#).

1 INTRODUCTION

The proliferation of emerging technologies such as Artificial Intelligence (AI), Machine Learning (ML), Digital Signal Processing (DSP), big data analytics, cloud computing and Internet of Things (IoT) is driving the growing demand for computational power and storage requirements. The International Data Corporation (IDC) reported that the global data sphere is expected to grow from 33 zettabytes (2018) to 175 zettabytes by 2025 with a Compound Annual Growth Rate (CAGR) of 61% [41], highlighting the pressing need for more efficient computing solutions. This problem is intensified, especially when considering resource-restricted systems and/or battery-driven devices, such as smartphones and wearables [16].

Historically, the industry of computing systems was driven for more than 40 years by two fundamental principles: Moore’s Law [130] and Dennard’s Law [46]. Today, even though the number of transistors integrated per area is still increasing (Moore’s Law), the supply voltage cannot be scaled according to Dennard’s Law, and thus, the power density is increased. The end of Dennard’s scaling combined with other factors (e.g., the cooling technology and the natural limits of silicon) led us to the “Dark Silicon” era [48]. In this era, the power efficiency is a critical issue for computing systems, either they are placed at the edge (embedded systems) or on the cloud (data centers). Concurrently, the compute-intensive workloads of novel AI/ML and DSP applications challenge their deployment in terms of performance (speed). As a result, the industry of computing systems is forced to find new design/computing approaches that will improve the power efficiency while providing the desired performance.

I: Need for Low-Power/Energy Computing. With the continuous shrinking of the transistor size into deep nanometer regime, the power/energy consumption has become a critical issue and a top priority to consider in the design of computing systems. Actually, with the current trend, scientists have predicted that by the year 2040 computers will need more electricity than the world’s energy resources can generate, unless radical improvements are made in the computer design [143]. The ever-increasing deployment of IoT devices [52, 67], the exploding “Big Data” from all kinds of sources (e.g., videos and images), and the growth of supporting cyberinfrastructure such as data centers, are all exaggerating this situation.

This challenge is present in computing devices of all sizes – from low-power edge devices to high-performance data centers. For example, for mobile/edge devices used intensively in IoT endpoints, low-power/energy computing must be achieved to increase the battery life. On the other hand, data centers must achieve low-power/energy to reduce the costs in electricity and cooling, and achieve reliable operations. In certain applications (e.g., autonomous driving), the high power consumption could lead to an increased temperature of the computing chips, which will adversely affect the reliability of the chips and cause severe consequences. Recently, the increasing workload (in terms of data size and computational demands) from the AI, ML, and DSP domains are all worsening the issue of power/energy consumption.

II: Need for Accelerated Computing. Many practical application domains, such as autonomous driving, robotics, and space, require real-time processing of data streams. However, the very nature of existing powerful algorithms pose significant challenges to the hardware implementations. A specific example is the recent massive deployment of AI/ML methods with millions of parameters, like in the case of Deep Neural Networks (DNNs). Typically speaking, the execution of DNN models requires a huge amount of computing operations, such as additions/multiplications and transformations, as well as intensive memory accesses, which may lead to a significant delay in processing data streams. This can cause compromised quality of results, especially in resource-constrained edge devices that have real-time latency requirements.

How Can Approximate Computing Help? As Dennard’s Law expired in the mid-2000s and Moore’s Law is declining, the transistor scaling is increasingly less effective in improving performance, energy efficiency, and robustness. Therefore, alternative computing paradigms are urgently needed as we look to the future of the computing industry. *Approximate Computing (AxC)* has recently arisen as a promising candidate for resolving this challenge due to its success in many compute-intensive applications (e.g., image processing, object classification, and bio-signal analysis). Such applications show an inherent error tolerance, i.e., they do not require completely accurate computations for delivering acceptable output quality. For example, in image processing, a few pixel drops do not affect how images are perceived by human eyes; AI/ML may not need precise model parameters to get accurate results in classification and detection; communication systems are resilient against occasional noise. This paves the way for new optimization opportunities. By introducing a new design dimension – “accuracy” – to the overall design optimization, it is possible to trade off accuracy and lead to *less power used, less time consumed, and fewer computing resources required*. This leads to the promising novel design paradigm of Approximate Computing. Some examples of accuracy/quality metrics are peak signal-to-noise ratio (multimedia applications), relative difference (numerical analysis), and classification accuracy (machine learning).

The idea of leveraging imprecise computation for improved design dates back several decades in real-time system scheduling, where imprecise computation was used to enhance its dependability [104]. Another related research field is Fault Tolerance, which seeks to continue to provide the required functionality despite occasional failures by hiding the errors [141]. Compared to this field, Approximate Computing “intentionally” seeks to design imperfect hardware and software systems (*induce errors*) for improved performance and/or power/area efficiency. Specifically, researchers have built approximate integrated circuits, software programs, and architectures that outperform their conventional “accurate” counterparts in terms of resources (power, area, and/or performance). Approximate Computing has achieved tremendous success in many application domains and targets among other image processing [2, 4], computer vision [1, 15], computer graphics [14, 15], machine learning [9, 14], signal processing [8, 14], financial analysis [1, 14], database search [3, 14], and scientific computing [11, 18]. The error resilience of such application domains and the relaxed constraints for the quality of the results constitute Approximate Computing as an applicable design paradigm. They originate from:

- (1) The user’s intention to accept inaccuracies and results of lower quality.
- (2) The limited human perception, e.g., in multimedia applications.
- (3) The lack of perfect/golden results for validation, e.g., in data mining applications.
- (4) The lack of a unique answer/solution, e.g., in machine learning applications.
- (5) The application’s self-healing property, i.e., its capability to absorb/compensate errors by default.
- (6) The application’s inherent approximate nature, e.g., in probabilistic calculations, iterative algorithms, and learning systems.
- (7) The application’s analog/noisy real-world input data, e.g., in multimedia/signal processing.

The prominent outcome of approximate systems, in parallel with the ever-increasing demand for efficient and sustainable computing, has attracted a vast research interest. In this context, approximation techniques are applied at different design layers, i.e., from software/programs to hardware/circuits. Motivated by the benefits of Approximate Computing, as well as the great momentum it has gained over the last years, we conduct a survey that is presented in two parts. The goal is to cover the entire spectrum of Approximate Computing. The contribution and the content of the two-part survey are analyzed in Section 2.

Table 1. Qualitative comparison of Approximate Computing surveys on the entire computing stack.

AxC Survey	Year	Coverage	Pages #	References #	SW Tech.	HW Tech.	Arch. Approx.	AI/ML	Memories	Frameworks & Tools	Metrics	Benchmarks	CPU/FPGA/GPU/ASIC	Terminology	Challenges	Quant. Analysis ²
[59]	2013	6	65	✓	✓	≈	≈	x	x	✓	x	≈	x	x	≈	
[127]	2015	33 ¹	84	✓	≈	✓	≈	✓	✓	✓	✓	✓	x	✓	x	
[207]	2015	15	59	✓	✓	✓	x	≈	x	x	x	≈	x	x	x	
[199]	2015	6	54	✓	✓	✓	x	x	✓	x	x	≈	x	x	x	
[175]	2016	6	47	✓	✓	✓	x	x	≈	x	x	≈	x	x	≈	
[131]	2017	4	40	✓	✓	x	≈	≈	x	x	x	≈	x	x	x	
[12]	2017	6	72	✓	✓	✓	≈	≈	≈	x	✓	≈	x	✓	x	
[182]	2020	39 ¹	235	≈	✓	✓	x	✓	≈	x	x	≈	x	✓	x	
This work	Pt. 1	2023	36 ¹	222	✓	✓			✓			✓	✓	✓		✓
	Pt. 2	2023	36 ¹	301	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

¹ Single-column pages.² Quantitative analysis involving count of works, frequencies and numerical assessments.

2 APPROXIMATE COMPUTING SURVEY

Scope and Contribution: The literature includes surveys on Approximate Computing that target specific areas, e.g., arithmetic circuits [71] and logic synthesis [169], or focus on a single approximation technique, e.g., precision scaling [35]. In [198] and [13], a thorough analysis of software and hardware approximation techniques, respectively, is provided, but they are both laser-focused on DNN applications. Similarly, the survey of [43] mainly reviews the impact of AxC on edge computing and none of the [13, 35, 43, 71, 169, 198] surveys cover approximations from system level down to circuit level. *In contrast, the proposed two-part survey covers the entire computing stack, reviewing and classifying all the state-of-the-art approximation techniques from the software, hardware, and architecture layers for a wide range of application domains.* Table 1 reports a list of surveys that can be classified along with the proposed one, showing a qualitative comparison and key aspects that characterize each work. In the current work, the AxC stack (pyramid of design layers) is formed as shown in Fig. 1, and design techniques & approaches from all the layers are analyzed. In fact, this is the traditional computing stack with the addition of various kind of approximation techniques across the design layers (from the application and software down to the hardware and device). It is worth mentioning that the proposed two-part survey is the first to report a quantitative analysis with numerical assessments at this extent.

More explicitly, the current survey constitutes a comprehensive and detailed guide that provides a step-by-step explanation of key concepts, techniques, and applications of the AxC paradigm. The reader will have a complete view on AxC principles and works that implement and evaluate software, hardware, application-specific and architectural approximations. This survey also acts as a tutorial on the state-of-the-art approximation techniques. The main objectives and contributions of the survey are: 1) to attribute definitions in key AxC aspects and explain the main terminology, 2) to analyze the state-of-the-art works, identify approximation categories and cluster the reviewed works with respect to the approximation type/approach, 3) to survey application domains of AxC including the impact of approximations on them, and 4) to identify and discuss open challenges and future directions as a step towards the realization of approximate applications.

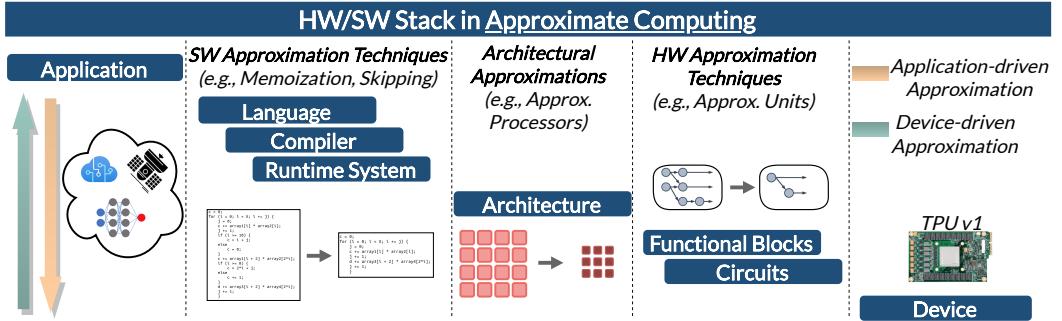


Fig. 1. The Approximate Computing stack: approximation techniques in the design abstraction layers.

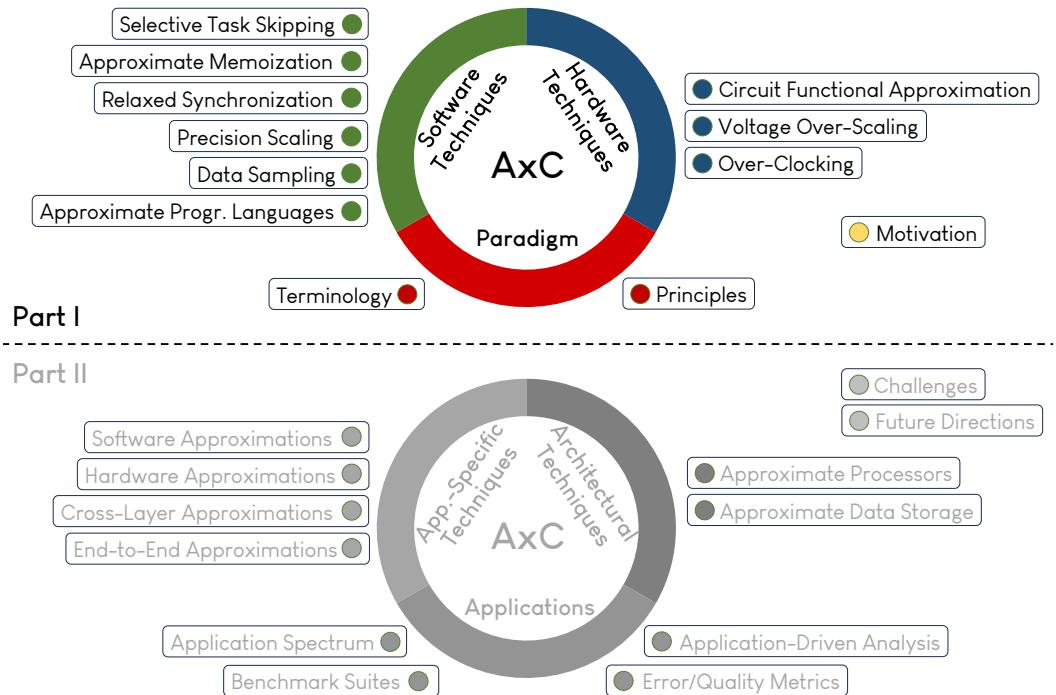


Fig. 2. Organization of the proposed two-part survey on Approximate Computing.

Organization: As shown in Fig. 2, the proposed survey is divided into two parts, which constitute standalone manuscripts focusing on different aspects/areas of Approximate Computing:

- Part I:** It is presented in the current paper, and it introduces the AxC paradigm (terminology and principles) and reviews software & hardware approximation techniques.
- Part II:** It is presented in [94], and it reviews application-specific & architectural approximation techniques and introduces the AxC applications (domains, quality metrics, benchmarks).

The remainder of the article (Part I of the survey) is organized as follows. Section 3 provides fundamental concepts of AxC, while the next two sections (Sections 4-5) review and classify software- and hardware-level works, respectively. Section 6 performs a quantitative analysis for all the reviewed approximation techniques. Finally, Section 7 concludes the survey.

3 THE APPROXIMATE COMPUTING PARADIGM

3.1 Terminology of Approximate Computing

Even though approximate computations have been examined since the 1960s (e.g., Mitchell’s logarithmic-based multiplication/division [125]), the first systematic efforts to define the Approximate Computing paradigm started in the late 2000s. Various terms have been used in the literature to describe strategies for delivering approximate architectures, programs, and circuits. Approximate Computing is synonymous or overlaps with these terms. Chakradhar et al. [27] define **Best-Effort Computing** as “*the approach of designing software/hardware computing systems with reduced workload, improved parallelization and/or approximate components towards enhanced efficiency and scalability*”. The term **Relaxed Programming** is introduced by Carbin et al. [23] to express “*the transformation of programs with approximation methods and relaxed semantics to enable greater flexibility in their execution*”. Chippa et al. [38] use the term **Scalable Effort Design** for “*the systematic approach that embodies the notion of scalable effort into the design process at different levels of abstraction, involving mechanisms to vary the computational effort and control knobs to achieve the best possible trade-off between energy efficiency and quality of results*”.

According to Mittal [127], “*Approximate Computing exploits the gap between the accuracy required by the applications/users and that provided by the computing system to achieve diverse optimizations*”. Han and Orshansky [59] distinguish Approximate Computing from Probabilistic/Stochastic Computing, stating that “*it does not involve assumptions on the stochastic nature of the underlying processes implementing the system and employs deterministic designs for producing inaccurate results*”. Another interesting point of view is expressed by Sampson [165], who claims that Approximate Computing is based on “*the idea that we are hindering the efficiency of the computer systems by demanding too much accuracy from them*”. The current article attributes the following definition:

Approximate Computing: *It constitutes a radical paradigm shift in the design and development of computing systems, circuits and programs, which is built on top of the error-resilient nature of various application domains and relies on disciplined methods to intentionally insert errors for providing valuable resource gains in exchange for tunable accuracy loss.*

Table 2 describes the most frequently used terms in Approximate Computing. The term *error* is used to indicate that the output result is different from the accurate result (produced with conventional computing). Error is distinguished from *fault*, which refers to an unexpected condition (e.g., stuck-at-logic in circuits, bit-flips in memories, faults in operating systems) that causes the system to unintentionally output erroneous results. Another significant term is *accuracy*, which is defined as the distance between the approximate and the accurate result and is measured using application-specific and general-computing error metrics. Accuracy is distinguished from *precision*, which expresses the differentiation between nearby discrete values and does not refer to errors of Approximate Computing but to quantization noise (inserted by the real-to-digital value mapping). Moreover, in Approximate Computing, the term *Quality-of-Service (QoS)* is used to describe the overall quality of the results regarding accuracy and errors.

3.2 Principles of Approximate Computing

To enable and realize significant efficiency gains through approximations, the design of approximate systems should be guided using the following steps/principles:

- *Application Analysis:* The quality requirements and metrics vary across applications. Therefore, it is essential to analyze the application in detail to identify the acceptable QoS and specify the error metrics that can truly quantify the output quality for evaluation and comparison.

Table 2. Terminology of Approximate Computing.

Term	Description
<i>Error-Resilient Application</i>	The application that allows computation errors and accepts results of lower quality.
<i>Quality of Service</i>	The quality of the results in terms of errors and accuracy.
<i>Accuracy Constraint</i>	The quality requirements that the results need to satisfy.
<i>Error Bound/Threshold</i>	The maximum error allowed in the results.
<i>Golden Result</i>	The result that is obtained from the original accurate computations.
<i>Acceptable Result</i>	The result that satisfies the accuracy constraints and error bounds of the application.
<i>Variable Accuracy</i>	The capability of providing different levels of accuracy.
<i>Non-Critical Task/Computation</i>	The task/computation that can be safely approximated due to its small impact on the quality of the output results.
<i>Error Analysis</i>	The study involving metrics, mathematics and simulations to examine the range, frequency, scaling, and/or propagation of the errors.
<i>Approximation Technique/Method</i>	The systematic and disciplined approach to insert computation errors in exchange for gains in power, energy, area, latency, and/or throughput.
<i>Approximation Degree/Strength</i>	The aggressiveness of the approximation technique in terms of errors inserted and tasks/computations approximated.
<i>Approximation Configuration</i>	An instance of the parameters and settings of the approximation technique.
<i>Frozen Approximation</i>	The approximation that is fixed and cannot be re-configured at a different degree.
<i>Dynamic Approximation Tuning</i>	The capability of adjusting the approximation degree at runtime to satisfy the desired error constraints.
<i>Cross-Layer Approximation</i>	The approximation that is applied at multiple design abstraction layers (software, hardware, architecture).
<i>Heterogeneous Approximation</i>	The approximation that applies concurrently multiple configurations of different degree within the same system.
<i>Application-Driven Approximation</i>	The approximation that is applied with respect to the error resilience and sensitivity of the targeted application.
<i>Device-Driven Approximation</i>	The approximation that is applied with respect to the targeted device/technology (e.g., CPU, GPU, ASIC, FPGA).
<i>Approximate Space Exploration</i>	The study involving error analysis and resource gain quantification to examine trade-offs and select the most suitable approximation techniques/configurations.
<i>Approximation Localization</i>	The systematic approach to locate the tasks/computations and design regions that are offered for approximation.
<i>Error Modeling</i>	The process of emulating the errors inserted by the approximations.
<i>Error Prediction</i>	The process of predicting errors before computing the final result.
<i>Error Detection</i>	The process of identifying an error occurrence.
<i>Error Compensation</i>	The process of modifying the erroneous result to reduce the error.
<i>Error Correction</i>	The process of replacing the erroneous result with the accurate one.

– *Workload Analysis*: Not all tasks/computations in an application can be approximated. Therefore, it is important to identify the non-critical tasks/computations (to be approximated) and isolate them from the critical ones. This is essential to enable disproportionate benefits, i.e., significant improvements in efficiency for a negligible loss in quality.

– *Development of Approximation Methodology*: To achieve ultra-high efficiency gains while ensuring that acceptable quality is maintained, approximations are required to be introduced systematically in the system. Moreover, the sources of disproportionate benefits are distributed across different layers of the computing stack. Therefore, to achieve a superior quality-efficiency trade-off, the development of sophisticated methodologies that can exploit the cross-layer knowledge of the system and deploy approximations systematically across various layers and sub-systems of the given system becomes an important step.

– *Development of Error Models*: Error estimation is vital for comparing different approximations. However, empirical evaluation of approximate implementations is time-consuming and costly (especially when inducing approximations at multiple design layers or at low-level hardware implementations). Therefore, it is essential to build models that can emulate the errors and examine the output quality of the system when approximated.

– *Design Space Exploration*: Typically, various types of approximations can be deployed in a system, where each sub-system/system module may have a completely different set of approximations that lead to disproportionate benefits. Therefore, design space exploration is performed to examine different approximation configurations, evaluate the approximation space and make decisions regarding the final approximate implementation that yields significant efficiency gains while meeting user-defined quality and performance constraints. These exploration methodologies are usually supported by error and performance models to efficiently search the approximation space.

– *Error Analysis*: Input distribution can have a profound impact on the resilience of the approximated system. Therefore, it is important to study the errors for different input distributions using appropriate error/QoS metrics.

– *Quantification of Results*: This is performed to prove the resilience of the application and ensure that constraints about QoS and/or resources are met.

4 SOFTWARE APPROXIMATION TECHNIQUES

This section classifies and presents approximation techniques that are applied at software level, i.e., the higher level of the design abstraction hierarchy. The goal of software Approximate Computing is to improve the execution time of the program and/or the energy consumption of the system. The techniques of the literature, illustrated in Fig. 3, can be categorized into five classes: (i) *Selective Task Skipping*, (ii) *Approximate Memoization*, (iii) *Relaxed Synchronization*, (iv) *Precision Scaling*, (v) *Data Sampling*, and (vi) *Approximate Programming Languages*. Typical software approximation techniques integrate some of the following features: approximation libraries/frameworks, compiler extensions, accuracy tuning tools, runtime systems, and language annotations. Moreover, numerous of these techniques allow the programmer to specify QoS constraints, provide approximate code variants, and mark the program regions/tasks for approximation.

The remainder of this section reports representative state-of-the-art works for software approximation techniques. The references of these works are summarized in Table 3. The literature also includes software approximation frameworks, such as ACCEPT [166] and OPPROX [126], which apply multiple state-of-the-art approximation techniques.

4.1 Selective Task Skipping

4.1.1 *Loop Perforation*. The loop perforation technique aims at skipping some of the loop iterations in a software program to provide performance/energy gains in exchange for QoS loss. Subsequently, several relevant works [15, 63, 80, 99, 122, 135, 178, 179, 185], which involve design space exploration on loop perforation with programming frameworks and profiling tools, are presented.

Starting with one of the first state-of-the-art works, the SpeedPress compiler [63] supports a wide range of loop perforation types, i.e., modulo, truncation, and randomized. It takes as input the original source code, a set of representative inputs, as well as a programmer-defined QoS acceptability model, and outputs a loop perforated binary. In the same context, Misailovic et al. [122] propose a QoS profiler to identify computations that can be approximated via loop perforation. The proposed profiling tool searches the space of loop perforation and generates results for multiple perforation configurations. In [179], the same authors propose a methodology to exclude critical loops, i.e., whose skipping results in unacceptable QoS, and perform exhaustive and greedy design space explorations to find the Pareto-optimal perforation configurations for a given QoS constraint.

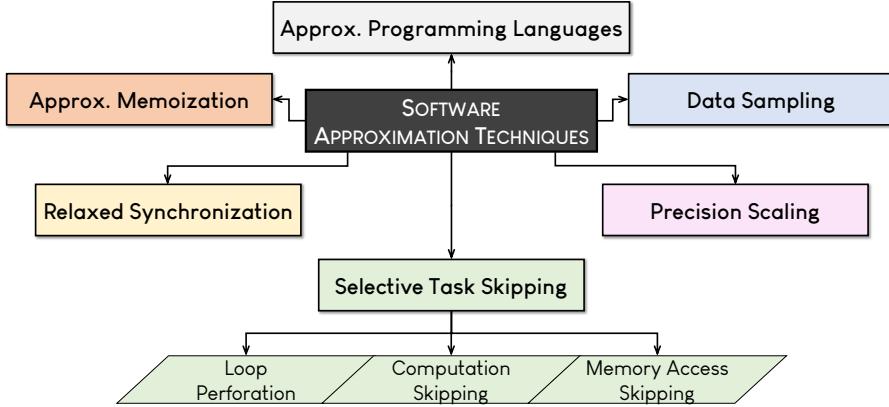


Fig. 3. Classification of SW approximation techniques in 6 classes: *Selective Task Skipping*, *Approximate Memoization*, *Relaxed Synchronization*, *Precision Scaling*, *Data Sampling*, and *Approximate Programming Languages*.

Table 3. Classification of software approximation techniques.

SW Approximation Class	References
Loop Perforation	[15, 63, 80, 99, 122, 135, 178, 179, 185]
Computation Skipping	[6, 21, 101, 149, 154, 155, 195, 196]
Memory Access Skipping	[82, 85, 119, 164, 211, 220]
Approximate Memoization	[8, 19, 28, 83, 110, 124, 148, 150, 163, 188, 218]
Relaxed Synchronization	[22, 116, 121, 123, 153, 156, 181, 183]
Precision Scaling	[20, 36, 37, 44, 45, 57, 81, 88–90, 102, 117, 158, 159, 171, 186, 213]
Data Sampling	[3, 9, 55, 64, 79, 86, 91, 139, 144, 145, 206, 221]
Approx. Programming Languages	[1, 11, 14, 17, 18, 23, 24, 50, 56, 76, 105, 114, 120, 137, 138, 167, 168, 180, 187]

In [178], the authors propose an architecture that employs a profiler to identify non-critical loops towards their perforation. To protect code segments that can be affected by the perforated loops, the architecture is equipped with HaRE, i.e., a hardware resilience mechanism. Another interesting work is GraphTune [135], which is an input-aware loop perforation scheme for graph algorithms. This approach analyzes the input dependence of graphs to build a predictive model that finds near-optimal perforation configurations for a given accuracy constraint. Li et al. [99] propose a compiling & profiling system, called Sculptor, to improve the conventional loop perforation, which skips a static subset of iterations. More specifically, Sculptor dynamically skips a subset of the loop instructions (and not entire iterations) that do not affect the output accuracy. More recently, the authors of [15] develop LEXACT, which is a tool for identifying non-critical code segments and monitoring the QoS of the program. LEXACT searches the loop perforation space, trying to find perforation configurations that satisfy pre-defined metrics.

The loop perforation technique has been also used in approximation frameworks for heterogeneous multi-core systems combining various approximation mechanisms. Tan et al. [185] propose a task scheduling algorithm that employs multiple approximate versions of the tasks with loops perforated. Kanduri et al. [80] target applications whose main computations are continuously repeated and tune the loop perforation at runtime.

4.1.2 Computation Skipping. This technique omits the execution of blocks of codes according to the acceptable QoS loss, programmer-defined constraints, and/or runtime predictions regarding the

output accuracy [6, 21, 101, 115, 149, 154, 155, 195, 196]. Compared to loop perforation, these techniques do not focus only on skipping loop iterations, but also skip higher-level computations/tasks e.g., an entire convolution operation. Most of the state-of-the-art works perform application-specific computation skipping.

Meng et al. [115] introduce a parallel template to develop approximate programs for iterative-convergence recognition & mining algorithms. The proposed programming template provides several strategies (implemented as libraries) for task dropping, such as convergence-based computation pruning, computation grouping in stages, and early termination of iterations. Another interesting work involving application-specific computation skipping is presented in [21]. The authors of this work study the error tolerance of the supervised semantic indexing algorithm to make approximation decisions. Regarding their task dropping approach, they choose to omit the processing of common words (e.g., “the”, “and”) after the initial iterations, as these computations have negligible impact on the output accuracy.

The authors of [149] propose two techniques to find computations with low impact on the QoS of the Reduce-and-Rank computation pattern, targeting to approximate or skip them completely. To identify these computations, the first technique uses intermediate reduction results and ranks, while the second one is based on the spatial or temporal correlation of the input data (e.g., adjacent image pixels or successive video frames). Similarly to the other state-of-the-art works, Vassiliadis et al. [195, 196] propose a programming environment that skips (or approximates) computations according to programmer-defined QoS constraints. More specifically, the programmer expresses the significance of the tasks using pragmas directives, optionally provides approximate variants of tasks, and specifies the desired task percentage to be executed accurately. Based on these constraints, the proposed system makes decisions at runtime regarding the approximation/skipping of the less significant tasks.

Rinard [154] builds probabilistic distortion models based on linear regression to study the impact of computation skipping on the output accuracy. In particular, the programmer partitions the computations into tasks, which are then marked as “critical” or “skippable” through random skip executions. The probabilistic models estimate the output distortion as a function of the skip rates of the skippable tasks. This approach is also applied in parallel programs [155], where probabilistic distortion models are employed to tune the early phase termination at barrier synchronization points, targeting to keep all the parallel cores busy.

Significant research has been also conducted on skipping the computations of Convolutional Neural Networks (CNNs). Lin et al. [101] introduce PredictiveNet to predict the sparse outputs of the nonlinear layers and skip a large subset of convolutions at runtime. The proposed technique, which does not require any modification in the original CNN structure, examines the most-significant part of the convolution to predict if the nonlinear layer output is zero, and then decides whether to skip the remaining least-significant part computations or not. In the same context, Akhlaghi et al. [6] propose SnaPEA, exploiting the convolution–activation algorithmic chain in CNNs (activation takes as input the convolution result and outputs zero if the latter is negative). This technique early predicts negative convolution results based on static re-ordering of the weights and monitoring of the partial sums’ sign bit, in order to skip the rest computations.

4.1.3 Memory Access Skipping.

Another approach to improve the execution time and energy consumption at the software level is the memory access skipping. Such techniques [82, 85, 119, 164, 211, 220] aim at avoiding high-latency memory operations, while they inherently reduce the number of computations.

Miguel et al. [119] exploit the approximate data locality to skip the required memory accesses due to L1 cache miss. They employ a load value approximator, which learns value patterns using a

global history buffer and an approximator table, to estimate the memory data values. RFVP [211] uses value prediction instead of memory accessing. When selected load operations miss in the cache memory, RFVP predicts the requested values without checking for misprediction or recovering the values. Thus, timing overheads from pipeline flushes and re-executions are avoided. Furthermore, a tunable rate of cache misses is dropped after the value prediction to eliminate long memory stalls. Similarly, the authors of [85] propose a framework that skips costly last-level cache misses according to a programmer-defined error constraint and an heuristic predicting skipped data.

To improve the performance of CUDA kernels on GPUs, Samadi et al. [164] propose a runtime approximation framework, called SAGE, which focuses on optimizing the memory operations among other functionalities. The approximations lie in skipping selective atomic operations (used by kernels to write shared variables) to avoid conflicts leading to performance decrease. Furthermore, SAGE reduces the number of memory accesses by packing the read-only input arrays, and thus, allowing to access more data with fewer requests. Karakoy et al. [82] propose a slicing-based approach to identify data (memory) accesses that can be skipped to deliver energy/performance gains within an acceptable error bound. The proposed method applies backward and forward code slicing to estimate the gains from skipping each output data. Moreover, the ‘0’ value is used for each data access that is not performed. The ApproxANN framework [220], besides performing approximate computations, skips memory accesses on neural networks according to the neuron criticality. More specifically, a theoretical analysis is adopted to study the impact of neurons on the output accuracy and characterize their criticality. The neuron approximation under a given QoS constraint is tuned by an iterative algorithm, which applies the approximations and updates the criticality of each neuron (it may change due to approximations in other neurons).

4.2 Approximate Memoization

The memoization technique stores results of previous calculations or pre-computed values in memory to use them instead of performing calculations. Namely, this memory functions as a look-up table that maps a set of data identifiers to a set of stored data. The current survey focuses on approximate memoization techniques [19, 28, 83, 124, 163, 188] relying on software frameworks, compilers and programmer’s decisions. Nevertheless, it is noted that there are also approaches [8, 110, 150, 218] requiring hardware modification to support memoization, as well as hardware-level look-up table approximation techniques (e.g., [148]). The latter work proposes a quantized look-up table, called qLUT, which replaces complex arithmetic functions. The qLUT table contains precomputed output values corresponding to a small input set that has been created from the original input data using their probability distributions.

Chadhuri et al. [28] propose an approximate memoization for computations in loops. Prior to executing an expensive function within a loop, this technique checks a look-up table to find if this computation was previously performed for similar input data. In this case, the cached result is used, otherwise, the function is executed and the new computation is stored in the look-up table. Paraprox [163] is a software framework for identifying common patterns in data-parallel programs and applying tailored approximations to them. For the Map & Scatter/Gather patterns, Paraprox uses memoization rather than performing computations. In particular, it fills a look-up table with pre-computed data, which are obtained from the execution of the Map & Scatter/Gather function for some representative inputs, and performs runtime look-up table queries instead of the conventional computations.

iACT [124] is another approximation framework that applies runtime memoization among other functionalities. The programmer uses pragmas to declare the functions for memoization and specify the error tolerance percentage. For each function call-site, the framework creates a global table to store pairs of function arguments and output results. In case the function arguments are

already stored in the table (within an error bound), the corresponding output results are returned. Otherwise, the function is accurately executed and the new input–output pairs are stored in the table. The ATM approach [19] performs runtime task memoization, relying on hashing functions to store the task inputs and an adaptive algorithm to automatically decide whether to use memoization or execute the task. The programmer needs to use pragmas to specify the tasks that are suitable for memoization. The authors of [83] introduce an approximate memoization mechanism for GPU fragment shading operations, which reduces the precision of the input parameters and performs partial matches. To identify approximate memoization opportunities, they characterize various fragment shader instructions in terms of memoization hits and output accuracy. Moreover, runtime policies are proposed to tune the precision according to the errors introduced.

Contrary to the aforementioned techniques, TAF-Memo [188] is an output-based function memoization technique, i.e., it memoizes function calls based on their output history. TAF-Memo checks for temporal locality by calculating the relative arithmetic difference of two consecutive output values from the same function call-site. In case this difference is below the acceptable error constraint, memoization is applied by returning the last computed output for the following function calls.

4.3 Relaxed Synchronization

The execution of parallel applications on multi/many-core systems requires time-consuming synchronization to either access shared data or satisfy data dependencies. Various techniques [22, 116, 121, 123, 153, 156, 181, 183] have been proposed to relax the conventional synchronization requirements that guarantee error-free execution, delivering speedup in exchange for QoS loss.

The authors of [153] propose the RaC methodology to systematically relax synchronization, while always satisfying a programmer-defined QoS constraint. Initially, the programmer specifies the parallel code segments, and then applies the four-step RaC methodology. This methodology identifies criteria for quantifying the acceptable QoS, selects the relaxation points, modifies the code to enable the execution of both the original and relaxed versions, and selects the suitable relaxation degree (i.e., which instances to relax for each synchronization point). Misailovic et al. [123] propose the Dubstep system, which relaxes the synchronization of parallelized programs based on a “find-transform-navigate” approach. More specifically, Dubstep performs a profiling-based analysis of the original program to find possible optimizations, inserts opportunistic synchronization and barriers, and finally, performs an exploration including accuracy, performance and safety analysis for the transformed program.

QuickStep [121] is a system for approximately parallelizing sequential programs, i.e., without preserving the semantics of the original program, within statistical accuracy bounds. Among other transformations, QuickStep replicates shared objects to eliminate the bottlenecks of synchronized operations on them. HELIX-UP [22] is another parallelizing compiler that selectively relaxes strict adherence to program semantics to tackle runtime performance bottlenecks, involving profiling and user interaction to tune QoS. The compiler also offers a synchronization-relaxing knob to decrease the inter-core communication overhead by synchronizing sequential segments with prior iterations. More recently, the authors of [183] introduce PANDORA, which is an approximate parallelizing framework based on symbolic regression machine learning and sampled outputs of the original function. To avoid timing bottlenecks, such as data movement and synchronization, and improve parallelism, PANDORA eliminates loop-carried dependencies using fitness functions and constraints regarding error and performance. In [181], the authors exploit the concept of approximate shared value locality to reduce synchronization conflicts in programs using optimistic synchronization. The reduction of conflicts on approximately local variables, detected for a given similarity constraint, is achieved through an arbitration mechanism that imprecisely shares the values between threads. The authors of [116] apply aggressive coarse-grained parallelism on recognition & mining algorithms

by relaxing or even ignoring data dependencies between different iterations. As a result, the timing overheads are reduced in comparison with the conventional parallel implementation, which also applies parallelization only within the iteration (iterations are executed serially). Rinard [156] introduces synchronization-free updates to shared data structures by eliminating the conventional use of mutual exclusion and dropping array elements at the worst scenario. Moreover, the same work applies relaxed barrier synchronization, allowing the threads to pass the barrier without stalling to wait for the other threads.

4.4 Precision Scaling

Precision scaling (tuning) refers to the discipline reduction of the numerical precision, resulting in improved processing speed and/or memory bandwidth [35]. The state-of-the-art software-level works [20, 36, 37, 44, 45, 57, 81, 88–90, 102, 117, 158, 159, 171, 186, 213] address several challenges, such as the scaling degree, scaling automation, mixed precision, and dynamic scaling.

Starting with works based on formal methods to reduce the precision and examine the errors, the Gappa tool [45] automates the study of rounding errors in elementary functions and floating-point calculations using interval arithmetic. An extended version of this tool is Gappa++ [102], which provides automated analysis of numerical errors in a wide range of computations, i.e., fixed-point, floating-point, linear and non-linear. This tool integrates several features, such as operation rewriting to facilitate the isolation of rounding errors, and affine arithmetic to accurately bound linear calculations with correlated errors. FPTuner [36] is a tool that performs formal error analysis based on symbolic Taylor expansions and quadratically constrained quadratic programming. It searches for precision allocations that satisfy constraints such as the number of operators at a given precision and the number of type casts. Rosa [44] is a source-to-source compiler that combines satisfiability modulo theories with interval arithmetic to bound the round-off errors of the fixed- and floating-point formats.

Several works of the literature employ heuristics and automated search to scale the precision of floating-point programs. Precimonious [159] searches all the program variables in their order of declaration using the delta-debugging algorithm, and lowers their precision according to an error constraint specified by the programmer. In the same context, HiFPTuner [57] firstly groups dependent variables that may require the same precision, and then performs a customized hierarchical search. Lam et al. [90] introduce a framework that employs the breadth-first search algorithm to identify code regions that can tolerate lower precision. Similarly to this technique, CRAFT [89] performs binary searches to initially determine the required program precision, and then truncate the results of some of the floating-point instructions. Towards the detection of large floating-point errors, the authors of [37] propose S³FP. This tool is based on an heuristic-guided search to find the inputs causing the largest errors. The Blame Analysis [158] combines concrete and shadow execution to generate a blame set for the program instructions, which contains the minimum precision requirements under a given error constraint. This approach can be also used in cooperation with the previous search-based works, and specifically, as pre-processing to reduce the search space. Schkufza et al. [171] treat the scaling of floating-point precision as a stochastic search problem. In particular, they repeatedly apply random program transformations and use a robust search to guarantee the maximum errors.

The concept of dynamic precision scaling, i.e., the tuning of the precision at runtime with respect to the input data and error sensitivity, has been studied in [213]. The dynamic scaling framework of this work integrates an offline application profiler, a runtime monitor to track workload changes, and an accuracy controller to adjust the precision accordingly. ApproxMA [186] dynamically scales the precision of data memory accesses in algorithms such as mixture model-based clustering. This framework integrates a runtime precision controller, which generates custom bit-widths according

to the QoS constraints, and a memory access controller, which loads the scaled data from memory. The custom bit-widths are generated by analyzing a subset of data and intermediate results and calculating metrics regarding the error appearance and the number of tolerable errors.

Mixed floating-point precision has been also studied in high-performance computing workloads. ADAPT [117] uses algorithmic differentiation, i.e., a technique for numerically evaluating the derivative of a function corresponding to the program, to estimate the output error of high-performance computing workloads. It provides a precision sensitivity profile to guide the development of mixed-precision programs. In the same context, the authors of [20] propose an instruction-based search that explores information about the dynamic program behaviour and the temporal locality.

To enable mixed floating-point precision in GPUs, the authors of [88] propose the GPUMixer tool, which relies on static analysis to find code regions where precision scaling improves the performance. Next, GPUMixer performs a dynamic analysis involving shadow computations to examine if the scaled program configurations satisfy the accuracy constraints. In the same context, PreScaler [81] is an automatic framework that generates precision-scaled OpenCL programs, considering both kernel execution and data transfer. Initially, it employs a system inspector to collect information about precision scaling on the target platform, and an application profiler to identify memory objects with floating-point elements for potential scaling. This information is exploited by a decision maker that finds the best scaling configuration using decision tree search on a minimized space.

4.5 Data Sampling

Approximate Computing is also exploited in big data analysis, in an effort to reduce the increased number of computations and storage requirements due to the large amount of input data. The key idea is to perform computations on a representative data sample rather than on the entire input dataset. Therefore, various data sampling methods [3, 9, 55, 64, 79, 86, 91, 139, 144, 145, 206, 221] are examined to provide real-time processing with error bounds in applications involving stream analytics, database search, and model training.

EARL [91] is an extension of Hadoop (i.e., a software framework that provides distributed storage and big data processing on clusters), which delivers early results with reliable error bounds. It applies statistics-based uniform sampling from distributed files. Goiri et al. [55] propose the ApproxHadoop framework to generate approximate MapReduce programs based on task dropping and multi-stage input sampling. They also bound the errors using statistical theories. The programmer tunes the approximation by specifying either the desired error bound or the task dropping and input sampling ratios. Similarly, ApproxSpark [64] performs sampling at multiple arbitrary points of long chains of transformations to facilitate the aggregation of huge amounts of data. This framework models the clustering information of transformations as a data provenance tree, and then computes the approximate aggregate values as well as error thresholds. Moreover, the sampling rates are dynamically selected according to programmer-specified error thresholds.

Sampling methods have been also examined in stream analytics. StreamApprox [145] is an approximate stream analytics system that supports both batched and pipelined stream processing. It employs two sampling techniques, i.e., stratified and reservoir sampling, to approximate the outputs with rigorous error bounds. IncApprox [86] combines approximate and incremental computations to provide stream analytics with bounded error. This system executes a stratified sampling algorithm that selects data for which the results have been memoized from previous runs, and adjusts the computations to produce an incrementally updated output. On the other hand, PrivApprox [144] combines sampling and randomized response to provide both approximate computations and privacy guarantee. This system integrates a query execution interface that enables the systematic exploration of the trade-off between accuracy and query budget. ApproxIoT [206]

facilitates approximate stream analytics at the edge by combining stratified reservoir sampling and hierarchical processing.

A variety of sampling methods have been employed in approximate query processing systems for databases. BlinkDB [3] performs approximate distributed query processing, supporting SQL-based aggregation queries with time and error constraints. It creates stratified samples based on past queries, and uses an heuristic-based profiler to dynamically select the sample that meets the query's constraints. Another system applying approximate big-data queries is Quickr [79], which integrates operators sampling multiple join inputs into a query optimizer, and then searches for an appropriate sampled query plan. Sapprox [221] is a distribution-aware system that employs the occurrences of sub-datasets to drive the online sampling. In particular, the exponential number of sub-datasets is reduced to a linear one using a probabilistic map, and then, cluster sampling with unequal probability theory is applied for sub-dataset sampling. Sapprox also determines the optimal sampling unit size in relation with approximation costs and accuracy.

Numerous works of the literature use data sampling to decrease the increased computational cost of model training in machine learning applications. Zombie [9] is a two-stage system that trains approximate models based on clustering and active learning. The first stage applies offline indexing to organize the dataset into index groups of similar elements. Subsequently, the stage of online querying uses the index groups that are likely to output useful features to creates the training subset of data. BlinkML [139] approximately trains a model on a small sample, while providing accuracy guarantees. The sample is obtained through uniform random sampling, however, in case of very large datasets, a memory-efficient algorithm is employed.

4.6 Approximate Programming Languages

The high-level approximation of software programs has been examined through approximate programming languages, i.e., language extensions that allow the programmer to systematically declare approximate code regions, variables, loops, and functions, insert randomness in the program, and/or specify error constraints. The literature involves numerous works [1, 11, 14, 17, 18, 23, 24, 50, 56, 76, 105, 114, 120, 137, 138, 167, 168, 180, 187] that enable approximate procedural, object-oriented, and probabilistic programming.

Ansel et al. [11] introduce a set of PetaBricks language extensions that allow the programmer to write code of variable accuracy. These extensions expose the performance–accuracy trade-off to the compiler, which automatically searches the algorithmic space to tune the program according to the programmer's accuracy constraints. Eon [180] is a programming language that allows the programmer to annotate program flows (paths) with different energy states. The Eon runtime system predicts the workload and energy of the system, and then adjusts the execution of flows according to the programmer's declarations and the energy constraints. In the same context, Baek and Chilimbi [14] propose Green, which is a two-phase programming framework providing language extensions to approximate expensive functions and loops. The programmer uses pragma-like annotations to specify approximate variants of functions. In the calibration phase, Green builds a model to quantify the QoS loss and the performance/energy gains. This model is then used in the operational phase to generate an approximate program satisfying the programmer's QoS constraint. DECAF [18] is a type-based approximate programming language that allows the programmer to specify the correctness probability for some of the program variables. The DECAF type system also integrates solver-aided type inference to automatically tune the type of the rest variables, code specialization, and dynamic typing. Flikker [105] provides language annotations to mark the program variables and partition the data into critical and non-critical regions (the latter are stored in unreliable memories). Topaz [1] is a task-based language that maps tasks onto approximate hardware and uses an outlier detector to find and re-execute the computations producing unacceptable results.

In [23], the authors introduce language constructs for generating approximate programs and proof rules for verifying the acceptability properties. Rely [24] is an imperative language that allows the programmer to introduce quantitative reliability specifications for generating programs with data stored in approximate memory and inexact arithmetic/logical operations. Chisel [120] automates the selection of Rely’s approximations while satisfying the programmer-defined reliability and accuracy constraints. To solve this optimization problem, Chisel employs an integer programming solver. All these works include safety analysis and program verification for sequential programs. In contrast, Parallely [50] is a programming language for approximating parallel programs through canonical sequentialization, i.e., a verification method that generates sequential programs capturing the semantics of parallel programs.

Targeting approximations in Java programs, the authors of [167] propose EnerJ, i.e., a language extension providing type qualifiers to specify data that can be approximately stored or computed. EnerJ guarantees isolation of the approximate computations. FlexJava [137] offers another set of language extensions to annotate approximate programs. Using an approximation safety analysis, FlexJava automates the approximation of data and operations while ensuring safety guarantees. ExpAX [138] allows the programmer to explicitly specify error expectations for a subset of Java. Based on an approximation safety analysis, it identifies operations that are candidate for approximation, and then, a heuristic-based framework approximates those that statistically satisfy the error expectations.

Significant research has also been conducted on probabilistic programming languages. Church [56] is a probabilistic language that inserts randomness on a deterministic function subset using stochastic functions. The Church semantics are defined in terms of evaluation histories and conditional distributions on the latter. Similarly, Venture [114] is another language that enables the specification of probabilistic models and inference problems. The Anglican [187] language and runtime system provides probabilistic evaluation model and functional representations, e.g., distributions and sequences of random variables.

Uncertain<T> [17] is a language abstraction that manipulates data as probability distributions. More specifically, random variables are declared as “uncertain” and a Bayesian network for representing computations is build, where nodes correspond to the variables and edges correspond to conditional variable dependencies. The Uncertain<T> runtime system performs hypothesis tests and sampling to evaluate the network. In the same context, Sampson et al. [168] use probabilistic assertions on random variables. Their tool, called MayHap, performs probabilistic evaluation by statically building a Bayesian representation network based on the input distribution and dynamically interpreting it via sampling. In the same context, AxProf [76] is a profiling-based framework for analyzing randomized approximate programs. The programmer specifies probabilistic predicates for the output, i.e., regarding the expectation of the output value and/or the probability that the output satisfies a condition, and AxProf generates approximate programs based on statistical tests.

5 HARDWARE APPROXIMATION TECHNIQUES

This section classifies and introduces the hardware approximation techniques, which are applied at the lower level of the design abstraction hierarchy. These techniques aim to improve the area, power consumption, and performance of the circuits i.e., the basic building blocks of accelerators, processors, and computing platforms. The hardware approximation techniques can be categorized into three classes: (i) *Circuit Functional Approximation (CFA)*, (ii) *Voltage Over-Scaling (VOS)*, and (iii) *Over-Clocking (OC)*. In approximate hardware, two types of errors are distinguished: the functional errors (produced by CFA) and the timing errors (produced by VOS and OC). Fig. 4 illustrates the hardware approximation techniques, including a further taxonomy to sub-classes.

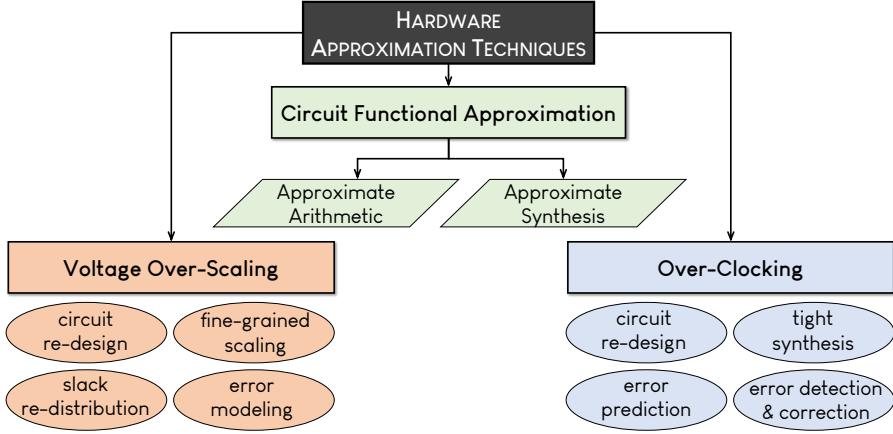


Fig. 4. Classification of HW approximation techniques in 3 classes: *Circuit Functional Approximation*, *Voltage Over-Scaling* and *Over-Clocking*.

Table 4. Classification of hardware approximation techniques.

HW Approximation Class	Technique/Approach
Adder Approximation	Use of Approximate Full Adder Cells [42, 58, 140, 209] Segmentation and Carry Prediction [5, 47, 65, 77, 84, 174, 208, 212]
Multiplier Approximation	Truncation and Rounding [51, 60, 93, 95, 98, 190, 214] Approximate Radix Encodings [69, 97, 107, 197, 204, 205, 222] Use of Approximate Compressors [4, 49, 129, 162, 184] Logarithmic Approximation [10, 108, 142, 160]
Divider Approximation	Bit-width Scaling [61, 70] Use of Approximate Adder/Subtractor Cells [2, 31, 32, 34] Simplification of Computations [66, 106, 161, 191, 215]
Approximate Synthesis	Structural Netlist Transformation [25, 103, 170, 200] Boolean Rewriting [62, 118, 152, 201] High-Level Approximate Description [26, 92, 133, 134, 210] Evolutionary Synthesis [132, 173, 192–194]
Voltage Over-Scaling	Slack Re-distribution [78] Circuit Re-design and Architecture Modification [29, 128, 219] Fine-Grained Scaling [136, 202, 217] Error Modeling [68, 74, 109, 146, 216]
Over-Clocking	Tight Synthesis [7] Circuit Re-design and Architecture Modification [151, 177, 203] Error Detection & Correction [39, 100, 147] Error Prediction [40, 72, 73, 75, 112, 157]

The remainder of this section presents state-of-the-art works, organized according to the proposed classification of Table 4. It is noted that, even though some works may belong to more than one sub-class, they are assigned to their prevalent one and their relevant features are highlighted.

5.1 Circuit Functional Approximation

Circuit functional approximation modifies the original accurate design by reducing its circuit complexity at logic level. Typical CFA approaches include: (i) the modification of the circuit's truth table, (ii) the use of an approximate version of the initial hardware algorithm, (iii) the use of small

inexact components as building blocks, and (iv) approximate circuit synthesis. The main target of CFA is the arithmetic circuits [71], as they constitute the key processing units of processors and accelerators, and thus, they inherently affect their power efficiency and performance. The literature provides several open-source libraries of approximate arithmetic circuits, such as ApproxAdderLib [174], EvoApprox8b [132] and SMAApproxLib [189]. This survey focuses on approximate adders, multipliers, and dividers. However, it is noted that numerous works design and evaluate other approximate arithmetic operations, such as circuits for Multiplication-and-Accumulation (MAC) [30, 54], square root [70], squaring [113], square-accumulate [53], and Coordinate Rotation Digital Computer (CORDIC) [33]. The literature also includes automated methods for generating approximate circuits, which are presented in the context of approximate logic synthesis. Moreover, there are works applying functional approximations based on the inputs such as [111], where the authors configure and assign different approximation operators for a given data flow program based on the input workload. It is also important to mention that methodologies for digital hardware design based on approximate arithmetic circuits and formats have been proposed [96].

5.1.1 Approximate Adders. Significant research has been conducted on the design of approximate area- and power-efficient adders. The approximation techniques for inexact adders involve: (i) *use of approximate full adder cells* [42, 58, 140, 209] and (ii) *segmentation and carry prediction* [5, 47, 65, 77, 84, 174, 208, 212]. Next, representative state-of-the-art works for approximate adders are presented.

The IMPACT adders are based on inexact full adders cells, which are approximated at the transistor level to deliver up to 45% area reduction [58]. Another transistor-level cell approximation is proposed in [209], where the AXA 4-transistor XOR/XNOR-based adders are implemented, delivering up to 31% gain in dynamic power consumption. Moreover, in [140], approximate reverse carry propagate full adders are used to build the hybrid RCPA adders. Targeting higher level approximations, the OLOCA adder splits the addition into accurate and approximate segments [42], and for the latter, it employs OR gates for the most-significant bit additions and outputs constant ‘1’ for the least-significant ones.

To reduce the worst-case carry propagation delay, Kim et al. [84] propose a carry prediction scheme leveraging the less-significant input bits, which is 2.4× faster than the conventional ripple-carry adder. Similarly, Hu et al. [65] introduce a carry speculating method to segment the carry chain in their design, which also performs error and sign correction. Compared to the accurate adder, the proposed design is 4.3× faster and saves 47% power.

The quality constraint of applications may vary during runtime, thus, research efforts have also focused on designing dynamically configurable adders that can tune their accuracy. In [77], the authors propose an accuracy-configurable adder, called ACA, which consists of several sub-adders and an error detection & correction module. The accuracy is controlled at runtime, while operation in accurate mode is also supported. Another dynamically configurable adder, called GDA, is proposed in [212], where multiplexers select the carry input either from the previous sub-adder or the carry prediction unit, providing a more graceful accuracy degradation. In the same direction, the GeAr adder [174] employs multiple sub-adders of equal length to variable approximation modes. This architecture supports accurate mode via a configurable error correction unit.

Akbari et al. [5] introduce the RAP-CLA adder, which splits the conventional carry look-ahead scheme into two segments, i.e., the approximate part and the augmenting part, supporting approximate and accurate mode. When operating at the approximate mode, the augmenting part is power-gated to reduce power consumption. Another carry-prediction-based approach supporting both modes is the SARA design [208]. This adder uses carry ripple sub-adders, and the carry prediction does not require a dedicated circuitry. Finally, the BSCA adder, which is based on a

block-based carry speculative approach [47], integrates an error recovery unit and non-overlapped blocks consisting of a sub-adder, a carry prediction unit, and a selection unit.

5.1.2 Approximate Multipliers. The multiplication circuits have attracted significant interest from the research community. The literature includes a plethora of inexact multipliers that can be categorized according to the prevailing approximation techniques: (i) *truncation and rounding* [51, 60, 93, 95, 98, 190, 214], (ii) *approximate radix encodings* [69, 97, 107, 197, 204, 205, 222], (iii) *use of approximate compressors* [4, 49, 129, 162, 184], and (iv) *logarithmic approximation* [10, 108, 142, 160]. Subsequently, the state-of-the-art works from each category are introduced.

Starting with the rounding and truncation techniques, the DRUM multiplier [60] dynamically reduces the input bit-width, based on the leading ‘1’ bits, to achieve 60% power gain in exchange for mean relative error of 1.47%. Zendegani et al. propose the RoBa multiplier [214], which rounds the operands to the nearest exponent-of-two and performs a shift-based multiplication in segments. In [98], the PR approximate multiplier perforates partial products and applies rounding to the remaining ones, delivering up to 69% energy gains. The same approximation technique is integrated in the mantissa multiplication of floating-point numbers to create the AFMU multiplier [95]. Vahdat et al. propose the TOSAM multiplier [190] that truncates the input operands according to their leading ‘1’ bit. To decrease the error, the truncated values are rounded to the nearest odd number. In [93], different rounding, perforation and encoding schemes are combined to extract the most energy-efficient multiplication circuits. Finally, Frustaci et al. [51] implement an alternative dynamic truncation with correction, along with an efficient mapping for the remaining partial product bits.

Next, multipliers that generate their partial products based on approximate radix encodings are presented. Liu et al. [107] modify the Karnaugh map of the radix-4 encoding to create approximate encoders for generating the least-significant partial product bits. A similar approach is followed in [197], where approximate radix-4 partial product generators are designed. Jiang et al. [69] use an approximate adder to generate the $\pm 3 \times$ multiplicand term in the radix-8 multiplier. In [204], the authors propose a hybrid low-radix encoding that encodes the most-significant bits with the accurate radix-4 encoding and the least-significant bits with the an approximate radix-8 encoding. A similar approach is used in [205], where the authors propose approximate radix-8 multipliers for FPGA-based design. Targeting to high-order radix, the authors of [97] propose the hybrid high-radix encoding, which applies both the accurate radix-4 and approximate radix- 2^k encodings. Correspondingly, in [222], a radix-256 encoder is proposed for approximate multiplication circuits.

Several works employ approximate compressors for the partial product accumulation. Momeni et al. [129] modify the truth table of the accurate 4:2 compressor to create two simplified designs and use them in the Dadda multiplier. The authors in [4] design 4:2 compressors, again for Dadda multipliers, which can switch between accurate and approximate mode at runtime, consuming 68% lower power. In [162], an approximate 4:2 compressor is implemented in FinFET based on a three-input majority gate, and then it is used in the Dadda architecture along truncation. Esposito et al. [49] introduce a new family of approximate compressors and assign them to each column of the partial product matrix according to their allocation algorithm. Another interesting work is the design of approximate compressors for multipliers using the stacking circuit concept [184].

Regarding the approximate logarithmic multipliers, Liu et al. [108] employ a truncated binary-logarithm converter and inexact adders for the mantissa addition to design the ALM family of multipliers. The logarithmic-based REALM multiplier [160] partitions the power-of-two intervals of the input operands into segments, and determines an error compensation factor for each one. The ILM multiplier [10] differentiates from the conventional design, as it rounds the input operands to their nearest power-of-two using a nearest ‘1’ bit detector. Pilipovic et al. [142] propose a two-stage

trimming logarithmic multiplier, which firstly, reduces the bit-width of the input operands, and then, the bit-width of the mantissas.

5.1.3 Approximate Dividers. The division circuits have received less attention than adders and multipliers. Nevertheless, the literature provides numerous works aiming to reduce the large critical paths of the conventional dividers. The approximation techniques for division circuits can be categorized as follows: (i) *bit-width scaling* [61, 70], (ii) *use of approximate adder/subtractor cells* [2, 31, 32, 34], and (iii) *simplification of computations* [66, 106, 161, 191, 215].

The first class of approximation techniques uses exact dividers with reduced bit-width. The approximate divider of [61] dynamically selects the most relevant bits from the input operands and performs accurate division at lower bit-width, providing up to 70% power gains in exchange for 3% average error. The design makes use of leading ‘1’ bit detectors, priority encoders, multiplexers, subtractor and barrel shifter. Similarly, the AAXD divider of [70] detects the leading ‘1’ bits and uses a pruning scheme to extract the bits that will be given as input to the divider. Additionally, the design integrates an error correction unit to form the final output.

Regarding the second class of approximation techniques, Chen et al. [31] perform the subtraction of the non-restoring array divider with inexact subtractor circuits employing pass transistor logic. For their divider, called AXDnr, the authors examine different schemes regarding which subtractions of the division array to approximate. Similarly, in the AXDr divider of [32], some of the subtractions of the restoring array divider are performed with inexact subtractor circuits. The use of inexact cells has also been examined in the high-radix SRT divider [34]. In this divider, called HR-AXD, the inexact cell is a signed-digit adder that is employed according to different replacement schemes, along with cell truncation and error compensation. Adams et al. [2] introduce two approximate division architectures, called AXRD-M1 and AXRD-M2, which deliver up to 46% area and 57% power gains, respectively, compared to the exact restoring divider. The first design replaces some of the restoring divider cells with inexact ones of simplified logic, while the second one involves the elimination of some rows of the divider.

Targeting to perform the division with alternative simplified computations, the SEERAD divider [215] rounds the divisor to a specific form based on the leading ‘1’ bit position, and thus, the division is transformed to shift-&-add multiplication. In the same context, Vahdat et al. [191] propose the TruncApp divider that multiplies the truncated dividend with the approximate inverse divisor. Targeting to model the division operation, the CADE divider of [66] performs the floating-point division by subtracting the input mantissas. To compensate a large error (estimated by analyzing the most-significant input bits), a pre-computed value is retrieved from memory. In [106], the proposed AXHD divider approximates the least-significant computations of the division using a non-iterative logarithmic approach that is based on leading ‘1’ bit detection and subtraction of the logarithmic mantissas. Finally, Saadat et al. [161] propose approximate integer and floating-point dividers with near-zero error bias, called INZeD and FaNZeD, respectively, by combining an error correction method with the classical approximate logarithmic divider.

5.1.4 Approximate Synthesis. An automated approach to generate inexact circuits is the approximate logic synthesis. This method provides increased approximation diversity, i.e., it generates multiple approximate circuit variants, without relying on the manual approximation inserted by the designer, such as in the case of the aforementioned arithmetic approximations. Another benefit of approximate synthesis is that several techniques generate the approximate variant that leads to the maximum hardware gains for a given approximation/error constraint. The state-of-the-art techniques can be categorized as follows [169]: (i) *structural netlist transformation* [25, 103, 170, 172, 200], (ii) *Boolean rewriting* [62, 118, 152, 201], (iii) *high-level approximate description* [26, 92, 133, 134, 210], and (iv) *evolutionary synthesis* [132, 173, 192–194].

Several works of the literature employ a direct acyclic graph to represent the circuit netlist, where each node corresponds to a gate. In this context, the GLP technique [172] prunes nodes with an iterative greedy approach according to their impact on the final output and their toggle activity. In contrast, the CC framework [170] performs an exhaustive exploration of all possible node subsets that can be pruned without surpassing the error constraint. Venkataramani et al. [200] propose SASIMI, which is based on a greedy heuristic to find signal pairs assuming the same value and substitute one with the other. This automatic synthesis framework guarantees that the user-defined quality constraint is satisfied, and generates accuracy-configurable circuits. To apply stochastic netlist transformation, the SCALS framework [103] maps an initial gate-level network to the targeted technology (standard-cell or FPGA), and then iteratively extracts sets of sub-netlists and inserts random approximations in them. These sub-netlists are evaluated using statistical hypothesis testing. Castro-Codinez et al. [25] propose the AxLS framework, which converts the Verilog netlist to XML format and then applies typical transformation techniques, e.g., gate pruning, with respect to an error threshold.

The second category includes techniques that apply approximations in a formal Boolean representation of the circuit before it is synthesized. The SALSA approach [201] encodes the error constraints into a quality logic function, which compares the outputs of the accurate and approximate circuits. Towards logic simplification, SALSA computes the “observability don’t cares” for each output of the approximate circuit, i.e., the set of input values for which the output is insensitive. In the same direction, but for sequential circuits, Ranjan et al. introduce ASLAN [152]. This framework generates several approximate variants of the combinational blocks, and then identifies the best approximations for the entire sequential circuit based on a gradient-descent approach. Miao et al. [118] use a two-phase Boolean minimization algorithm to address the problem of approximate synthesis. The first phase solves the problem under a given constraint for error magnitude, and the second phase iteratively finds a solution that also satisfies the error frequency constraint. In an iterative manner, the BLASYS methodology [62] partitions the circuit into smaller circuits, and for each one, it generates an approximate truth table based on Boolean matrix factorization. The approximate sub-circuits are synthesized and the trade-off between error and power/area efficiency for the entire circuit is evaluated.

Regarding approximations introduced at the hardware description level, Yazdanbakhsh et al. [210] propose the Axilog language annotations, which provide syntax and semantics for approximate design and reuse in Verilog. Axilog allows the designer to partition the design into accurate and approximate segments. ABACUS [134] is another interesting work that parses the behavioral Verilog description of the design to create its abstract syntax tree. Next, a set of diverse transformations is applied to the tree to create approximate variants, which are then written in Verilog. An expanded version of ABACUS is introduced in [133], where sorting-based evolutionary algorithms are employed for design space exploration. Moreover, the new ABACUS version focuses on approximations in critical paths to facilitate the reduction of the supply voltage. Lee et al. [92] generate approximate designs in Verilog from C accurate descriptions. The proposed framework computes data statistics and mobility information for the given design, and employs an heuristic solver for optimizing the energy-quality trade-off. Targeting to high-level synthesis, the AxHLS approach [26] performs a design space exploration based on analytical models to identify the best arithmetic approximations for a given error constraint. Starting from a C description, AxHLS adopts scheduling and binding operations to apply the approximations provided by the exploration and generate the Verilog code.

The fourth class of techniques for automated synthesis of approximate circuits is based on evolutionary algorithms, i.e., heuristic-based search algorithms that treat circuit approximation as multi-objective optimization problem and generate a set of solutions. In this context, Sekanina et al.

[173] use Cartesian genetic programming to minimize the error in adders considering the number of logic gates as constraint. This approach is extended in [193], where approximate multipliers and median filters are evolved through randomly seeded Cartesian genetic programming. Based on the same utilities, the authors of [132] propose the EvoApprox8b library of approximate adders and multipliers. This library is generated by examining various trade-offs between accuracy and hardware efficiency, and offers different approximation variants and circuit architectures. In [194], a search-based technique for evolutionary circuit synthesis for FPGAs is proposed. In particular, this approach represents the circuit as a directed acyclic graph, and re-synthesizes approximate configurations based on Cartesian genetic programming. Vasicek et al. [192] adjust the approximation degree with respect to the significance of the inputs. To do so, they adopt a weighted error metric to determine the significance of each input vector and use Cartesian genetic programming to minimize the circuit's area while satisfying a threshold.

5.2 Voltage Over-Scaling

Voltage over-scaling aims to reduce the circuit's supply voltage below its nominal value, while keeping the clock frequency constant. The circuit operation at a lower voltage value produces timing errors due to the failure of the critical paths to meet the delay constraints. Nevertheless, considering that power consumption depends on the voltage value, VOS techniques are continuously examined in the literature. An exploration and quantification of the benefits and overheads of VOS is presented in [87]. Research involving VOS can be classified in the following categories: (i) *slack re-distribution* [78], (ii) *circuit re-design and architecture modification* [29, 128, 219], (iii) *fine-grained scaling* [136, 202, 217], and (iv) *error modeling* [68, 74, 109, 146, 216].

Kahng et al. [78] shift the timing slack of the frequently executed near-critical paths through slack redistribution, and thus, reduce the minimum voltage at which the error rate remains acceptable. The proposed technique is based on post-layout cell resizing to deliver the switching activity-aware slack redistribution. More specifically, a heuristic finds the voltage satisfying the desired error rate, and then increases the transistor width of the cells to optimize the frequently executed paths.

In [128], the authors optimize building blocks for more graceful degradation under VOS, using two techniques, i.e., dynamic segmentation & error compensation and delay budgeting of chained datapath. The first technique bit-slices the datapath of the adder and employs a multi-cycle error correction circuitry that tracks the carries. The second technique adds transparent latches between chained arithmetic units to distribute the clock period. To facilitate VOS, Chen et al. [29] build their designs on the residue number system, which provides shorter critical paths than conventional arithmetic. They also employ the reduced precision redundancy scheme to eliminate the timing errors. Another interesting work is Thundervolt [219], which provides error recovery in the MAC units of systolic arrays. To detect timing errors, Thundervolt employs Razor shadow flip-flops. In case an error occurs in a MAC, a multiplexer forwards the previous MAC's accurate partial sum (stored in the Razor flip-flop) to the next MAC.

Targeting fine-grained VOS solutions, i.e., the use of different voltages across the same circuit architecture, Pandey et al. propose GreenTPU [136]. This technique stores input sequences producing timing errors in MACs. As a result, when such an input sequence pattern is identified, the voltage of the MAC is scaled accordingly to prevent timing errors. In the same context, the authors of [202] propose NN-APP. This framework analyzes the error propagation in neural networks to model the impact of VOS on accuracy. Based on this analysis, as well as an error resilience study for the neurons, NN-APP uses a voltage clustering method to assign the same voltage to neurons with similar error resilience. Another fine-grained VOS approach is proposed in [217]. This framework provides voltage heterogeneity by using a greedy algorithm to solve the optimization problem of grouping and assigning the voltage of arithmetic units to different islands.

The analysis of errors in circuits under VOS is considered a key factor, as it guides the aggressiveness of voltage scaling towards the acceptable error margins. In [109], an analytical method to study the errors in voltage over-scaled arithmetic circuits is proposed. Similarly, the authors of [68] introduce a probabilistic approach to model the errors of the critical paths. In the same category, works relying on simulations to analyze the errors of VOS can be included. Ragavan et al. [146] characterize arithmetic circuits in terms of energy efficiency and errors using transistor-level SPICE simulation for various voltages. Based on this characterization, they propose a statistical model to simulate the behavior of arithmetic operations in VOS systems. By exploiting machine learning methods, Jiao et al. [74] propose LEVAX to model voltage over-scaled functional units. This input-aware model is trained on data from gate-level simulations to predict the timing error rate for each output bit. To provide accurate VOS-aware gate-level simulation, Zervakis et al. propose VOSsim [216]. This framework performs an offline characterization of the flip-flop for timing violations, and calculates the cell delays for the targeted voltage, enabling gate-level simulation under VOS.

5.3 Over-Clocking

Over-clocking (or frequency over-scaling) aims to operate the circuit/system at higher clock frequencies than those that respect the critical paths. As a result, timing errors are induced in exchange for increased performance. A trade-off analysis between accuracy and performance when over-clocking FPGA-based designs is presented in [176]. In the same work, the authors show that OC outperforms the traditional bit truncation for the same error constraint. The analysis of the current survey considers that the state-of-the-art works of the domain focus on the following directions: (i) *tight synthesis* [7], (ii) *circuit re-design and architecture modification* [151, 177, 203], (iii) *error detection & correction* [39, 100, 147], and (iv) *error prediction* [40, 72, 73, 75, 112, 157].

The first approach towards the reduction of timing errors caused by OC optimizes the critical paths of the design. In this context, the SlackHammer framework [7] synthesizes circuits with tight delay constraints to reduce the number of near-critical paths, and thus, decrease the probability of timing errors when frequency is over-scaled. At first, SlackHammer isolates the paths and identifies potential delay optimizations. Based on the isolated path analysis, the framework performs an iterative synthesis with tighter constraints for the primary outputs with negative slack.

The second class of techniques aims at modifying the conventional circuit architecture to facilitate frequency OC and increase the resilience to timing errors. The retiming technique [151] re-defines the boundaries of combinational logic by moving the flip-flops backward or forward between the stages. Based on this circuit optimization, the synthesis is relaxed by ignoring the paths that are bottleneck to minimum period retiming. Targeting different circuit architectures, Shi et al. [177] adopt an alternative arithmetic, called Online, and show that online-based circuits are more resilient to the timing errors of OC than circuits with traditional arithmetic. The modification of the initial neural network model to provide resilience in timing errors has also attracted research interest. In this direction, Wang et al. [203] propose an iterative reclocking-and-retraining framework for operating neural network circuits at higher frequencies under a given accuracy constraint. The clock frequency is gradually increased, and the network's weights are updated through back-propagation training until to find the maximum frequency for which the timing errors are mitigated and the accuracy constraint is satisfied.

Several works propose circuits for timing error detection & correction, enabling the use of over-clocking. These techniques either improve the frequency value of the first failure, i.e., the first timing error, or reduce the probability of timing errors. TIMBER [39] masks timing errors by borrowing time from successive pipeline stages. According to this approach, the use of discrete time-borrowing flip-flops and continuous time-borrowing latches slows down the appearance

of timing errors with respect to the frequency scaling. Ragavan et al. [147] detect and correct timing errors by employing a dynamic speculation window on the double-sampling scheme. This technique adds an extra register, called shadow and clocked by a second “delayed” clock, at the end of the pipelined path to sample the output data at two different time instances. This approach also uses an online slack measurement to adaptively over-clock the design. The TEAI approach [100] is based on the locality of the timing errors in software-level instructions, i.e., the tendency of specific instructions to produce timing errors. TEAI identifies these instructions at runtime, and sends error alarms to hardware, which is equipped with error detection & correction circuits.

Significant research has also been conducted on predicting the timing errors in advance, allowing to over-scale the frequency according to the acceptable error margins. In [157], the authors introduce an instruction-level error prediction system for pipelined micro-processors, which stalls the pipeline when critical instructions are detected. Their method is based on gate-level simulations to find the critical paths that are sensitized during the program execution. Similarly, Constantin et al. [40] obtain the maximum delays for each arithmetic instruction through gate-level simulations, and dynamically exploit timing margins to apply frequency over-scaling.

In addition to instruction-level prediction models, there are numerous works that build models based on machine learning and simulations of functional units. A representative work of this approach is WILD [72], which builds a workload-dependent prediction model using logistic regression. In the same direction, SLoT [73] is a supervised learning model that predicts timing errors based on the inputs and the clock frequency. At first, SLoT performs gate-level simulation to extract timing class labels, i.e., “timing error” or “no timing error”, for different inputs and frequencies. These classes are then used, along with features extracted from random data pre-processing, to train the error prediction model. Towards the same approach, TEVoT [75] uses machine learning to build a timing error prediction model that can predict the timing errors under different clock speeds and operating conditions, which are used to estimate the output quality of error-tolerant applications (e.g., image processing). DEVoT [112] is an extension of TEVoT that formulates the timing error prediction as a circuit dynamic delay prediction problem, saving significant prediction resources.

6 COMPARATIVE QUANTITATIVE ANALYSIS OF APPROXIMATION TECHNIQUES

This section reports a quantitative analysis for the software and hardware approximation classes. Due to their very large volume, diversity and differentiation, direct intra- and cross-layer comparisons are not performed. However, significant outcomes can be extracted for each class (e.g., type and size of workloads, acceptable accuracy loss, targeted resource gains). It is also noted that additional comparisons are reported in Part II of the survey [94], where an application-driven analysis of case studies is presented, involving both software and hardware techniques.

6.1 Software Approximation Techniques

Table 5 reports remarkable software-level works from each approximation class, along with key numerical results for resource gains and errors. Based on the literature’s review, each software approximation technique is favored in specific workloads (e.g., precision scaling in high-performance floating-point programs and data sampling in large queries), even though there are also more general techniques (e.g., loop perforation). Most works evaluate the approximations for various levels of quality degradation, e.g., 2%, 5%, and 10%, with the latter being widely considered as the largest acceptable threshold for the workloads of Table 5. Another significant outcome is that the combination of approximation knobs, such as in the case of approximate programming languages, delivers more resource gains than the application of a single approximation technique. For example, in comparison with approximate memoization, the approximate programming language of Table 5 provides 8%–28% more energy gain for workloads of similar type and size.

Table 5. Quantitative analysis of software approximation techniques.

Approximation Class	Workloads	Resource Gains	Accuracy Metric	Ref.
Loop Perforation	<i>BlackScholes, KMeans</i>	32%–36% energy	rel.err=3.5%, PSNR=36dB	[15]
	<i>X264, Streamcluster</i>	2×–8.5× speedup	error=5%, 10%	[99]
Computation Skipping	<i>GoogLeNet, VGGNet</i>	2.2×–1.9× speedup	class.accur.loss≤3%	[6]
	<i>MPEG, KNN</i>	~2× energy	qual.degrad=2.5%, 5%	[149]
Memory Access Skipping	<i>FluidAnimate, BodyTrack</i>	6%–28% speedup	error<10%	[119]
	<i>Gaussian, MatMul</i>	1.45×–2.4× speedup	qual.degrad=10%	[211]
Approximate Memoization	<i>BoxMuller, GammaCorr</i>	1.5×–3.2× speedup	qual.degrad=10%	[163]
	<i>BodyTrack, Sobel</i>	22% energy	qual.degrad=4%–10%	[124]
Relaxed Synchronization	<i>Graph500, KMeans</i>	3×–15× speedup	qual.degrad=0	[153]
	<i>Barnes-Hut, VolRender</i>	6× speedup	qual.degrad=0	[123]
Precision Scaling	<i>Lulesh, JetEngine</i>	1.2×–1.4× speedup	error=10 ⁻¹³ –10 ⁻¹¹	[117]
	<i>Bessel, FFT</i>	11.5×–43.4× speedup	error=10 ⁻¹⁰ –10 ⁻⁴	[57]
Data Sampling	<i>WikiLength</i>	21% speedup	error=0.34%	[55]
	<i>Conviva, TPC-H</i>	10×–100× speedup	accur.loss=2%–10%	[3]
Approx. Programm. Languages	<i>PageRank, Sobel</i>	1.1×–1.7× speedup	e2e.error=10 ⁻⁸ –10 ⁻¹⁶	[50]
	<i>MonteCarlo, RayTracer</i>	30%–50% energy	error<0.2	[167]

6.2 Hardware Approximation Techniques

The respective results for all the hardware approximation techniques analyzed in the previous sections are summarized in Fig. 5. To generate Fig. 5, we identified the most commonly used workloads among each hardware approximation family and categorized them into two arbitrary levels: *small* and *heavy*. Workloads requiring more than 50M operations (such as in deep neural networks) are classified as heavy, while those below this threshold are considered small. Then, we created a decision tree that helps the reader to identify the works having remarkable results with respect to the complexity of the workload, the baseline bit precision and the desired accuracy loss constraint. For the accuracy loss, two thresholds are considered, i.e., small accuracy loss (less than 1%) and moderate accuracy loss (less than 5%). For only few reported works, we define low accuracy loss as an MRED of less than 5%, while medium accuracy loss falls between 5% and 15%.

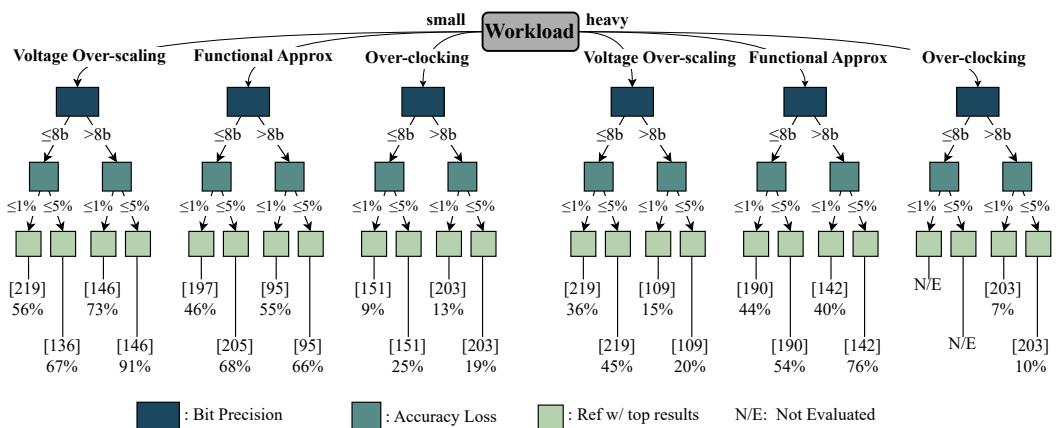


Fig. 5. Classification and comparative analysis of the most remarkable hardware approximation works with respect to their workload complexity, baseline bit precision and an accuracy loss threshold (1% and 5%). The leafs present the corresponding work along with their top results in energy reduction.

The leaves represent the works that achieve the highest energy reduction in each case. Overall, it seems that both Voltage Over-scaling and Circuit Functional Approximation appear among the top energy reduction results, outperforming Over-clocking. It is worth noting that Over-clocking shows a significant lack of energy efficiency compared to other techniques when processing heavy workloads. On the other hand, approximation techniques in small workloads can achieve substantial energy savings (e.g., 68%, 91%), even when using reduced precision and considering low accuracy loss thresholds. However, the high difference between small and heavy workloads highlights the need for continued research into hardware approximation techniques.

7 CONCLUSION

This article presented Part I of a comprehensive survey on Approximate Computing, focusing on key aspects of this novel design paradigm (motivation, terminology, and principles) and reviewing the state-of-the-art software and hardware approximation techniques. The review and classification was performed in both coarse-grained and fine-grained manners: each software/hardware-level technique was assigned to a higher-level approximation class (e.g., precision scaling, voltage over-scaling), as well as to a lower-level class with respect to its technical/implementation details (e.g., radix encoding, error prediction). Finally, a quantitative analysis of the approximation techniques was reported, involving the most commonly used workloads and key numerical results. Part II of the survey reviews the state-of-the-art software & hardware application-specific approximation techniques and architecture-level approximations in processors and memories. It also presents the application spectrum of Approximate Computing, including an analysis of use cases with remarkable results per technique and application domain, as well as well-established benchmark suites and error metrics for Approximate Computing.

ACKNOWLEDGEMENT

This research is partially supported by ASPIRE, the technology program management pillar of Abu Dhabi’s Advanced Technology Research Council (ATRC), via the ASPIRE Awards for Research Excellence.

REFERENCES

- [1] Sara Achour and Martin C. Rinard. 2015. Approximate Computation with Outlier Detection in Topaz. In *ACM SIGPLAN Int'l. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 711–730.
- [2] Elizabeth Adams, Suganthi Venkatachalam, and Seok-Bum Ko. 2020. Approximate Restoring Dividers Using Inexact Cells and Estimation From Partial Remainders. *IEEE Trans. on Computers* 69, 4 (2020), 468–474.
- [3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*. 29–42.
- [4] Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. 2017. Dual-Quality 4:2 Compressors for Utilizing in Dynamic Accuracy Configurable Multipliers. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 25, 4 (2017), 1352–1361.
- [5] Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. 2018. RAP-CLA: A Reconfigurable Approximate Carry Look-Ahead Adder. *IEEE Trans. on Circuits and Systems II: Express Briefs* 65, 8 (2018), 1089–1093.
- [6] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K. Gupta, and Hadi Esmaeilzadeh. 2018. SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks. In *ACM/IEEE Int'l. Symposium on Computer Architecture (ISCA)*. 662–673.
- [7] Tanfer Alan and Jörg Henkel. 2018. SlackHammer: Logic Synthesis for Graceful Errors Under Frequency Scaling. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2802–2811.

- [8] Carlos Alvarez, Jesus Corbal, and Mateo Valero. 2005. Fuzzy Memoization for Floating-Point Multimedia Applications. *IEEE Trans. on Computers* 54, 7 (2005), 922–927.
- [9] Michael R. Anderson and Michael Cafarella. 2016. Input Selection for Fast Feature Engineering. In *IEEE Int'l. Conference on Data Engineering (ICDE)*. 577–588.
- [10] Mohammad Saeed Ansari, Bruce F. Cockburn, and Jie Han. 2021. An Improved Logarithmic Multiplier for Energy-Efficient Neural Computing. *IEEE Trans. on Computers* 70, 4 (2021), 614–625.
- [11] 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 *IEEE/ACM Int'l. Symposium on Code Generation and Optimization (CGO)*. 85–96.
- [12] Alexander Aponte-Moreno, Alejandro Moncada, Felipe Restrepo-Calle, and Cesar Pedraza. 2018. A Review of Approximate Computing Techniques Towards Fault Mitigation in HW/SW Systems. In *IEEE Latin-American Test Symposium (LATS)*. 1–6.
- [13] Giorgos Armeniakos, Georgios Zervakis, Dimitrios Soudris, and Jörg Henkel. 2022. Hardware Approximate Techniques for Deep Neural Network Accelerators: A Survey. *Comput. Surveys* 55, 4 (2022), 1–36.
- [14] Woongki Baek and Trishul M. Chilimbi. 2010. Green: A Framework for Supporting Energy-Conscious Programming Using Controlled Approximation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 198–209.
- [15] Farshad Baharvand and Seyed Ghassem Miremadi. 2020. LEXACT: Low Energy N-Modular Redundancy Using Approximate Computing for Real-Time Multicore Processors. *IEEE Trans. on Emerging Topics in Computing* 8, 2 (2020), 431–441.
- [16] Nathaniel Bleier, Calvin Lee, Francisco Rodriguez, Antony Sou, Scott White, and Rakesh Kumar. 2022. FlexiCores: Low Footprint, High Yield, Field Reprogrammable Flexible Microprocessors. In *ACM/IEEE Int'l. Symposium on Computer Architecture (ISCA)*. 831–846.
- [17] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. 2014. Uncertain<T>: A First-Order Type for Uncertain Data. In *ACM Int'l. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 51–66.
- [18] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. 2015. Probability Type Inference for Flexible Approximate Programming. In *ACM SIGPLAN Int'l. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 470–487.
- [19] Iulian Brumar, Marc Casas, Miquel Moreto, Mateo Valero, and Gurindar S. Sohi. 2017. ATM: Approximate Task Memoization in the Runtime System. In *IEEE Int'l. Parallel and Distributed Processing Symposium (IPDPS)*. 1140–1150.
- [20] Hugo Brunie, Costin Iancu, Khaled Z. Ibrahim, Philip Brisk, and Brandon Cook. 2020. Tuning Floating-Point Precision Using Dynamic Program Information and Temporal Locality. In *ACM/IEEE SC, Int'l. Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.
- [21] Surendra Byna, Jiayuan Meng, Anand Raghunathan, Srimat Chakradhar, and Srihari Cadambi. 2010. Best-Effort Semantic Document Search on GPUs. In *Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*. 86–93.
- [22] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. 2015. HELIX-UP: Relaxing Program Semantics to Unleash Parallelization. In *IEEE/ACM Int'l. Symposium on Code Generation and Optimization (CGO)*. 235–245.
- [23] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. 2012. Proving Acceptability Properties of Relaxed Nondeterministic Approximate Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 169–180.
- [24] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In *ACM SIGPLAN Int'l. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 33–52.
- [25] Jorge Castro-Godínez, Humberto Barrantes-García, Muhammad Shafique, and Jörg Henkel. 2021. AxLS: A Framework for Approximate Logic Synthesis Based on Netlist Transformations. *IEEE Trans. on Circuits and Systems II: Express Briefs* 68, 8 (2021), 2845–2849.
- [26] Jorge Castro-Godínez, Julián Mateus-Vargas, Muhammad Shafique, and Jörg Henkel. 2020. AxHLS: Design Space Exploration and High-Level Synthesis of Approximate Accelerators using Approximate Functional Units and Analytical Models. In *Int'l. Conference On Computer Aided Design (ICCAD)*. 1–9.
- [27] Srimat T. Chakradhar and Anand Raghunathan. 2010. Best-Effort Computing: Re-thinking Parallel Software and Hardware. In *Design Automation Conference (DAC)*. 865–870.
- [28] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. 2011. Proving Programs Robust. In *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering (FSE)*. 102–112.
- [29] Jianan Chen and Jianhao Hu. 2013. Energy-Efficient Digital Signal Processing via Voltage-Overscaling-Based Residue Number System. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 21, 7 (2013), 1322–1332.

- [30] Ke Chen, Linbin Chen, Pedro Reviriego, and Fabrizio Lombardi. 2019. Efficient Implementations of Reduced Precision Redundancy (RPR) Multiply and Accumulate (MAC). *IEEE Trans. on Computers* 68, 5 (2019), 784–790.
- [31] Linbin Chen, Jie Han, Weiqiang Liu, and Fabrizio Lombardi. 2015. Design of Approximate Unsigned Integer Non-Restoring Divider for Inexact Computing. In *Great Lakes Symposium on VLSI (GLSVLSI)*. 51–56.
- [32] Linbin Chen, Jie Han, Weiqiang Liu, and Fabrizio Lombardi. 2016. On the Design of Approximate Restoring Dividers for Error-Tolerant Applications. *IEEE Trans. on Computers* 65, 8 (2016), 2522–2533.
- [33] Linbin Chen, Jie Han, Weiqiang Liu, and Fabrizio Lombardi. 2017. Algorithm and Design of a Fully Parallel Approximate Coordinate Rotation Digital Computer (CORDIC). *IEEE Trans. on Multi-Scale Computing Systems* 3, 3 (2017), 139–151.
- [34] Linbin Chen, Jie Han, Weiqiang Liu, Paolo Montuschi, and Fabrizio Lombardi. 2018. Design, Evaluation and Application of Approximate High-Radix Dividers. *IEEE Trans. on Multi-Scale Computing Systems* 4, 3 (2018), 299–312.
- [35] Stefano Cherubin and Giovanni Agosta. 2020. Tools for Reduced Precision Computation: A Survey. *Comput. Surveys* 53, 2 (2020), 1–35.
- [36] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. 2017. Rigorous Floating-Point Mixed-Precision Tuning. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 300–315.
- [37] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient Search for Inputs Causing High Floating-Point Errors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 43–52.
- [38] Vinay Kumar Chippa, Debabrata Mohapatra, Kaushik Roy, Srimat T. Chakradhar, and Anand Raghunathan. 2014. Scalable Effort Hardware Design. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 22, 9 (2014), 2004–2016.
- [39] Mihir R. Choudhury, Vikas Chandra, Robert C. Aitken, and Kartik Mohanram. 2014. Time-Borrowing Circuit Designs and Hardware Prototyping for Timing Error Resilience. *IEEE Trans. on Computers* 63, 2 (2014), 497–509.
- [40] Jeremy Constantin, Lai Wang, Georgios Karakostantis, Anupam Chattopadhyay, and Andreas Burg. 2015. Exploiting Dynamic Timing Margins in Microprocessors for Frequency-Over-Scaling with Instruction-Based Clock Adjustment. In *Design, Automation & Test in Europe (DATE)*. 381–386.
- [41] Tom Coughlin. 2023. 175 Zettabytes By 2025. (2023). <https://www.forbes.com/sites/tomcoughlin/2018/11/27/175-zettabytes-by-2025/?sh=254787045459>
- [42] Ayad Dalloo, Ardalan Najafi, and Alberto Garcia-Ortiz. 2018. Systematic Design of an Approximate Adder: The Optimized Lower Part Constant-OR Adder. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 26, 8 (2018), 1595–1599.
- [43] Hans Jakob Damsgaard, Aleksandr Ometov, and Jari Nurmi. 2023. Approximation Opportunities in Edge Computing Hardware: A Systematic Literature Review. *Comput. Surveys* 55, 12 (2023), 1–49.
- [44] Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. on Programming Languages and Systems* 39, 2 (2017), 1–28.
- [45] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. 2011. Certifying the Floating-Point Implementation of an Elementary Function Using Gappa. *IEEE Trans. on Computers* 60, 2 (2011), 242–253.
- [46] R. H. Dennard, F. H. Gaenslen, Hwa-Nien Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. 1974. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (1974), 256–268.
- [47] Farhad Ebrahimi-Azandaryani, Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. 2020. Block-Based Carry Speculative Approximate Adder for Energy-Efficient Applications. *IEEE Trans. on Circuits and Systems II: Express Briefs* 67, 1 (2020), 137–141.
- [48] Hadi Esmaeilzadeh, Emily Blehm, Renée St. Amant, Karthikkeyan Sankaralingam, and Doug Burger. 2013. Power Challenges May End the Multicore Era. *Commun. ACM* 56, 2 (2013), 93–102.
- [49] Darjin Esposito, Antonio Giuseppe Maria Strollo, Ettore Napoli, Davide De Caro, and Nicola Petra. 2018. Approximate Multipliers Based on New Approximate Compressors. *IEEE Trans. on Circuits and Systems I: Regular Papers* 65, 12 (2018), 4169–4182.
- [50] Vimuth Fernando, Keyur Joshi, and Sasa Misailovic. 2019. Verifying Safety and Accuracy of Approximate Parallel Programs via Canonical Sequentialization. *Proceedings of the ACM on Programming Languages* 3, 119 (2019), 1–29.
- [51] Fabio Frustaci, Stefania Perri, Pasquale Corsonello, and Massimo Alioto. 2020. Approximate Multipliers With Dynamic Truncation for Energy Reduction via Graceful Quality Degradation. *IEEE Trans. on Circuits and Systems II: Express Briefs* 67, 12 (2020), 3427–3431.
- [52] Gartner. 2018. *Gartner Identifies Top 10 Strategic IoT Technologies and Trends*. <https://www.gartner.com/en/newsroom/press-releases/2018-11-07-gartner-identifies-top-10-strategic-iot-technologies-and-trends>
- [53] Ghayoor A. Gillani, Muhammad Abdullah Hanif, M. Krone, Sabih H. Gerez, Muhammad Shafique, and Andre B. J. Kokkeler. 2018. SquASH: Approximate Square-Accumulate With Self-Healing. *IEEE Access* 6 (2018), 49112–49128.
- [54] Ghayoor A. Gillani, Muhammad Abdullah Hanif, Bart Verstoep, Sabih H. Gerez, Muhammad Shafique, and Andre B. J. Kokkeler. 2019. MACISH: Designing Approximate MAC Accelerators With Internal-Self-Healing. *IEEE Access* 7

(2019), 77142–77160.

- [55] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. 2015. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *ACM Int'l. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 383–397.
- [56] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Conference on Uncertainty in Artificial Intelligence (UAI)*. 220–229.
- [57] Hui Guo and Cindy Rubio-González. 2018. Exploiting Community Structure for Floating-Point Precision Tuning. In *ACM SIGSOFT Int'l. Symposium on Software Testing and Analysis (ISSTA)*. 333–343.
- [58] Vaibhav Gupta, Debabrata Mohapatra, Anand Raghunathan, and Kaushik Roy. 2013. Low-Power Digital Signal Processing Using Approximate Adders. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 32, 1 (2013), 124–137.
- [59] Jie Han and Michael Orshansky. 2013. Approximate Computing: An Emerging Paradigm for Energy-Efficient Design. In *IEEE European Test Symposium (ETS)*. 1–6.
- [60] Soheil Hashemi, R. Iris Bahar, and Sherief Reda. 2015. DRUM: A Dynamic Range Unbiased Multiplier for Approximate Applications. In *Int'l. Conference on Computer-Aided Design (ICCAD)*. 418–425.
- [61] Soheil Hashemi, R. Iris Bahar, and Sherief Reda. 2016. A Low-Power Dynamic Divider for Approximate Applications. In *Design Automation Conference (DAC)*. 1–6.
- [62] Soheil Hashemi, Hokchhay Tann, and Sherief Reda. 2018. BLASYS: Approximate Logic Synthesis Using Boolean Matrix Factorization. In *Design Automation Conference (DAC)*. 1–6.
- [63] Henry Hoffmann, Sasa Misailovic, Stelios Sidiropoulos, Anant Agarwal, and Martin C. Rinard. 2009. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures. *Massachusetts Institute of Technology Technical Report MIT-CSAIL-TR-2009-042* (2009), 1–21.
- [64] Guangyan Hu, Sandro Rigo, Desheng Zhang, and Thu Nguyen. 2019. Approximation with Error Bounds in Spark. In *IEEE Int'l. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 61–73.
- [65] Junjun Hu and Weikang Qian. 2015. A New Approximate Adder with Low Relative Error and Correct Sign Calculation. In *Design, Automation & Test in Europe (DATE)*. 1449–1454.
- [66] Mohsen Imani, Ricardo Garcia, Andrew Huang, and Tajana Rosing. 2019. CADE: Configurable Approximate Divider for Energy Efficiency. In *Design, Automation & Test in Europe (DATE)*. 586–589.
- [67] IoT Analytics. 2023. *State of IoT 2023*. <https://iot-analytics.com/number-connected-iot-devices/>
- [68] Dongsuk Jeon, Mingoo Seok, Zhengya Zhang, David Blaauw, and Dennis Sylvester. 2012. Design Methodology for Voltage-Overscaled Ultra-Low-Power Systems. *IEEE Trans. on Circuits and Systems II: Express Briefs* 59, 12 (2012), 952–956.
- [69] Honglan Jiang, Jie Han, Fei Qiao, and Fabrizio Lombardi. 2016. Approximate Radix-8 Booth Multipliers for Low-Power and High-Performance Operation. *IEEE Trans. on Computers* 65, 8 (2016), 2638–2644.
- [70] Honglan Jiang, Leibo Liu, Fabrizio Lombardi, and Jie Han. 2019. Low-Power Unsigned Divider and Square Root Circuit Designs Using Adaptive Approximation. *IEEE Trans. on Computers* 68, 11 (2019), 1635–1646.
- [71] Honglan Jiang, Francisco Javier Hernandez Santiago, Hai Mo, Leibo Liu, and Jie Han. 2020. Approximate Arithmetic Circuits: A Survey, Characterization, and Recent Applications. *Proc. IEEE* 108, 12 (2020), 2108–2135.
- [72] Xun Jiao, Yu Jiang, Abbas Rahimi, and Rajesh K. Gupta. 2016. WILD: A Workload-Based Learning Model to Predict Dynamic Delay of Functional Units. In *IEEE Int'l. Conference on Computer Design (ICCD)*. 185–192.
- [73] Xun Jiao, Yu Jiang, Abbas Rahimi, and Rajesh K. Gupta. 2017. SLoT: A Supervised Learning Model to Predict Dynamic Timing Errors of Functional Units. In *Design, Automation & Test in Europe (DATE)*. 1183–1188.
- [74] Xun Jiao, Dongning Ma, Wanli Chang, and Yu Jiang. 2020. LEVAX: An Input-Aware Learning-Based Error Model of Voltage-Scaled Functional Units. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 39, 12 (2020), 5032–5041.
- [75] Xun Jiao, Dongning Ma, Wanli Chang, and Yu Jiang. 2020. TEVoT: Timing Error Modeling of Functional Units under Dynamic Voltage and Temperature Variations. In *Design Automation Conference (DAC)*. 1–6.
- [76] Keyur Joshi, Vimuth Fernando, and Sasa Misailovic. 2019. Statistical Algorithmic Profiling for Randomized Approximate Programs. In *ACM/IEEE Int'l. Conference on Software Engineering (ICSE)*. 608–618.
- [77] Andrew B. Kahng and Seokhyeong Kang. 2012. Accuracy-Configurable Adder for Approximate Arithmetic Designs. In *Design Automation Conference (DAC)*. 820–825.
- [78] Andrew B. Kahng, Seokhyeong Kang, Rakesh Kumar, and John Sartori. 2010. Slack Redistribution for Graceful Degradation Under Voltage Overscaling. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*. 825–831.
- [79] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex Ad-Hoc Queries in Big Data Clusters. In *ACM SIGMOD Int'l. Conference on Management of Data (MOD)*. 631–646.

- [80] Anil Kanduri, Antonio Miele, Amir M. Rahmani, Pasi Liljeberg, Cristiana Bolchini, and Nikil Dutt. 2018. Approximation-Aware Coordinated Power/Performance Management for Heterogeneous Multi-cores. In *Design Automation Conference (DAC)*. 1–6.
- [81] Seokwon Kang, Kyunghwan Choi, and Yongjun Park. 2020. PreScaler: An Efficient System-Aware Precision Scaling Framework on Heterogeneous Systems. In *IEEE/ACM Int'l. Symposium on Code Generation and Optimization (CGO)*. 280–292.
- [82] Mustafa Karakoy, Orhan Kislar, Xulong Tang, Mahmut Taylan Kandemir, and Meenakshi Arunachalam. 2019. Architecture-Aware Approximate Computing. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 2 (2019), 1–24.
- [83] Georgios Keramidas, Chrysa Kokkala, and Iakovos Stamoulis. 2015. Clumsy Value Cache: An Approximate Memoization Technique for Mobile GPU Fragment Shaders. In *Workshop on Approximate Computing (WAPCO)*. 1–6.
- [84] Yongtae Kim, Yong Zhang, and Peng Li. 2013. An Energy Efficient Approximate Adder with Carry Skip for Error Resilient Neuromorphic VLSI Systems. In *Int'l. Conference on Computer-Aided Design (ICCAD)*. 130–137.
- [85] Orhan Kislar and Mahmut T. Kandemir. 2018. Data Access Skipping for Recursive Partitioning Methods. *Elsevier Computer Languages, Systems & Structures* 53 (2018), 143–162.
- [86] Dhanya R. Krishnan, Do Le Quoc, Pramod Bhatotia, Christof Fetzer, and Rodrigo Rodrigues. 2016. IncApprox: A Data Analytics System for Incremental Approximate Computing. In *Int'l. Conference on World Wide Web (WWW)*. 1133–1144.
- [87] Fadi J. Kurdahi, Ahmed Eltawil, Kang Yi, Stanley Cheng, and Amin Khajeh. 2010. Low-Power Multimedia System Design by Aggressive Voltage Scaling. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 18, 5 (2010), 852–856.
- [88] Ignacio Laguna, Paul C. Wood, Ranvijay Singh, and Saurabh Bagchi. 2019. GPUMixer: Performance-Driven Floating-Point Tuning for GPU Scientific Applications. In *ISC Int'l. Conference on High Performance Computing (HPC)*. 227–246.
- [89] Michael O. Lam and Jeffrey K. Hollingsworth. 2018. Fine-Grained Floating-Point Precision Analysis. *SAGE Int'l. Journal of High Performance Computing Applications* 32, 2 (2018), 231–245.
- [90] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. 2013. Automatically Adapting Programs for Mixed-Precision Floating-Point Computation. In *ACM Int'l. Conference on Supercomputing (ICS)*. 369–378.
- [91] Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. 2012. Early Accurate Results for Advanced Analytics on MapReduce. *Proceedings of the VLDB Endowment* 5, 10 (2012), 1028–1039.
- [92] Seogoo Lee, Lizy K. John, and Andreas Gerstlauer. 2017. High-Level Synthesis of Approximate Hardware under Joint Precision and Voltage Scaling. In *Design, Automation & Test in Europe (DATE)*. 187–192.
- [93] Vasileios Leon, Konstantinos Asimakopoulos, Sotirios Xydias, Dimitrios Soudris, and Kiamal Pekmestzi. 2019. Cooperative Arithmetic-Aware Approximation Techniques for Energy-Efficient Multipliers. In *Design Automation Conference (DAC)*. 1–6.
- [94] Vasileios Leon, Muhammad Abdullah Hanif, Giorgos Armeniakos, Xun Jiao, Muhammad Shafique, Kiamal Pekmestzi, and Dimitrios Soudris. 2025. Approximate Computing Survey, Part II: Application-Specific & Architectural Approximation Techniques and Applications. *Comput. Surveys* 57, 7 (2025), 1–36.
- [95] Vasileios Leon, Theodora Paparouni, Evangelos Petrongonas, Dimitrios Soudris, and Kiamal Pekmestzi. 2021. Improving Power of DSP and CNN Hardware Accelerators Using Approximate Floating-Point Multipliers. *ACM Trans. on Embedded Computing Systems* 20, 5 (2021), 1–21.
- [96] Vasileios Leon, Kiamal Pekmestzi, and Dimitrios Soudris. 2021. Exploiting the Potential of Approximate Arithmetic in DSP & AI Hardware Accelerators. In *Int'l. Conference on Field-Programmable Logic and Applications (FPL)*. 263–264.
- [97] Vasileios Leon, Georgios Zervakis, Dimitrios Soudris, and Kiamal Pekmestzi. 2018. Approximate Hybrid High Radix Encoding for Energy-Efficient Inexact Multipliers. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 26, 3 (2018), 421–430.
- [98] Vasileios Leon, Georgios Zervakis, Sotirios Xydias, Dimitrios Soudris, and Kiamal Pekmestzi. 2018. Walking through the Energy-Error Pareto Frontier of Approximate Multipliers. *IEEE Micro* 38, 4 (2018), 40–49.
- [99] Shikai Li, Sunghyun Park, and Scott Mahlke. 2018. Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation. In *ACM Int'l. Conference on Supercomputing (ICS)*. 341–351.
- [100] Zhan-Hui Li, Tao-Tao Zhu, Zhi-Jian Chen, Jian-Yi Meng, Xiao-Yan Xiang, and Xiao-Lang Yan. 2017. Eliminating Timing Errors Through Collaborative Design to Maximize the Throughput. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 25, 2 (2017), 670–682.
- [101] Yingyan Lin, Charbel Sakr, Yongjune Kim, and Naresh Shanbhag. 2017. PredictiveNet: An Energy-Efficient Convolutional Neural Network via Zero Prediction. In *IEEE Int'l. Symposium on Circuits and Systems (ISCAS)*. 1–4.
- [102] Michael D. Linderman, Matthew Ho, David L. Dill, Teresa H. Meng, and Garry P. Nolan. 2010. Towards Program Optimization through Automated Analysis of Numerical Precision. In *IEEE/ACM Int'l. Symposium on Code Generation and Optimization (CGO)*. 230–237.

- [103] Gai Liu and Zhiru Zhang. 2017. Statistically Certified Approximate Logic Synthesis. In *Int'l. Conference on Computer-Aided Design (ICCAD)*. 344–351.
- [104] Jane WS Liu, Kwei-Jay Lin, Riccardo Bettati, David Hull, and Albert Yu. 1994. Use of Imprecise Computation to Enhance Dependability of Real-Time Systems. In *Springer Foundations of Dependable Computing*. 157–182.
- [105] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. Flikker: Saving DRAM Refresh-Power through Critical Data Partitioning. In *ACM Int'l. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 213–224.
- [106] Weiqiang Liu, Jing Li, Tao Xu, Chenghua Wang, Paolo Montuschi, and Fabrizio Lombardi. 2018. Combining Restoring Array and Logarithmic Dividers into an Approximate Hybrid Design. In *IEEE Symposium on Computer Arithmetic (ARITH)*. 92–98.
- [107] Weiqiang Liu, Liangyu Qian, Chenghua Wang, Honglan Jiang, Jie Han, and Fabrizio Lombardi. 2017. Design of Approximate Radix-4 Booth Multipliers for Error-Tolerant Computing. *IEEE Trans. on Computers* 66, 8 (2017), 1435–1441.
- [108] Weiqiang Liu, Jiahua Xu, Danye Wang, Chenghua Wang, Paolo Montuschi, and Fabrizio Lombardi. 2018. Design and Evaluation of Approximate Logarithmic Multipliers for Low Power Error-Tolerant Applications. *IEEE Trans. on Circuits and Systems I: Regular Papers* 65, 9 (2018), 2856–2868.
- [109] Yang Liu, Tong Zhang, and Keshab K. Parhi. 2010. Computation Error Analysis in Digital Signal Processing Systems With Overscaled Supply Voltage. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 18, 4 (2010), 517–526.
- [110] Zhenhong Liu, Amir Yazdanbakhsh, Dong Kai Wang, Hadi Esmaeilzadeh, and Nam Sung Kim. 2019. AxMemo: Hardware-Compiler Co-Design for Approximate Code Memoization. In *ACM/IEEE Int'l. Symposium on Computer Architecture (ISCA)*. 685–697.
- [111] Dongning Ma, Rahul Thapa, Xingjian Wang, Xun Jiao, and Cong Hao. 2021. Workload-Aware Approximate Computing Configuration. In *Design, Automation & Test in Europe (DATE)*. 920–925.
- [112] Dongning Ma, Xinqiao Zhang, Ke Huang, Yu Jiang, Wanli Chang, and Xun Jiao. 2022. DEVoT: Dynamic Delay Modeling of Functional Units Under Voltage and Temperature Variations. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 41, 4 (2022), 827–839.
- [113] K. Manikanta Reddy, M. H. Vasantha, Y. B. Nithin Kumar, and Devesh Dwivedi. 2020. Design of Approximate Booth Squarer for Error-Tolerant Computing. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 28, 5 (2020), 1230–1241.
- [114] Vikash K. Mansinghka, Daneil Selsam, and Yura N. Perov. 2014. Venture: A Higher-Order Probabilistic Programming Platform with Programmable Inference. *CoRR* abs/1404.0099 (2014), 1–78.
- [115] Jiayuan Meng, Srimat Chakradhar, and Anand Raghunathan. 2009. Best-Effort Parallel Execution Framework for Recognition and Mining Applications. In *IEEE Int'l. Symposium on Parallel Distributed Processing (IPDPS)*. 1–12.
- [116] Jiayuan Mengt, Anand Raghunathan, Srimat Chakradhar, and Surendra Byna. 2010. Exploiting the Forgiving Nature of Applications for Scalable Parallel Execution. In *IEEE Int'l. Symposium on Parallel Distributed Processing (IPDPS)*. 1–12.
- [117] Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. 2018. ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning. In *ACM/IEEE SC, Int'l. Conference for High Performance Computing, Networking, Storage and Analysis*. 614–626.
- [118] Jin Miao, Andreas Gerstlauer, and Michael Orshansky. 2013. Approximate Logic Synthesis under General Error Magnitude and Frequency Constraints. In *Int'l. Conference on Computer-Aided Design (ICCAD)*. 779–786.
- [119] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load Value Approximation. In *IEEE/ACM Int'l. Symposium on Microarchitecture (MICRO)*. 127–139.
- [120] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and Accuracy-Aware Optimization of Approximate Computational Kernels. In *ACM SIGPLAN Int'l. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 309–328.
- [121] Sasa Misailovic, Deokhwan Kim, and Martin C. Rinard. 2013. Parallelizing Sequential Programs with Statistical Accuracy Tests. *ACM Trans. on Embedded Computing Systems* 12, 2s (2013), 1–26.
- [122] Sasa Misailovic, Stelios Sidiropoulos, Henry Hoffmann, and Martin C. Rinard. 2010. Quality of Service Profiling. In *ACM/IEEE Int'l. Conference on Software Engineering (ICSE)*, Vol. 1. 25–34.
- [123] Sasa Misailovic, Stelios Sidiropoulos, and Martin C. Rinard. 2012. Dancing with Uncertainty. In *ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*. 51–60.
- [124] 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*. 1–6.
- [125] John N. Mitchell. 1962. Computer Multiplication and Division Using Binary Logarithms. *IRE Trans. on Electronic Computers* EC-11, 4 (1962), 512–517.

- [126] Subrata Mitra, Manish K. Gupta, Sasa Misailovic, and Saurabh Bagchi. 2017. Phase-Aware Optimization in Approximate Computing. In *IEEE/ACM Int'l. Symposium on Code Generation and Optimization (CGO)*. 185–196.
- [127] Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *Comput. Surveys* 48, 4 (2016), 1–33.
- [128] Debabrata Mohapatra, Vinay K. Chippa, Anand Raghunathan, and Kaushik Roy. 2011. Design of Voltage-Scalable Meta-Functions for Approximate Computing. In *Design, Automation & Test in Europe (DATE)*. 1–6.
- [129] Amir Momeni, Jie Han, Paolo Montuschi, and Fabrizio Lombardi. 2015. Design and Analysis of Approximate Compressors for Multiplication. *IEEE Trans. on Computers* 64, 4 (2015), 984–994.
- [130] Gordon E. Moore. 1965. Cramming More Components onto Integrated Circuits. *IEEE Solid-State Circuits Society Newsletter* 38, 8 (1965), 1–4.
- [131] Thierry Moreau, Joshua San Miguel, Mark Wyse, James Bornholt, Armin Alaghi, Luis Ceze, Natalie Enright Jerger, and Adrian Sampson. 2018. A Taxonomy of General Purpose Approximate Computing Techniques. *IEEE Embedded Systems Letters* 10, 1 (2018), 2–5.
- [132] Vojtech Mrazek, Radek Hrbacek, Zdenek Vasicek, and Lukas Sekanina. 2017. EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods. In *Design, Automation & Test in Europe (DATE)*. 258–261.
- [133] Kumud Nepal, Soheil Hashemi, Hokchhay Tann, R. Iris Bahar, and Sherief Reda. 2019. Automated High-Level Generation of Low-Power Approximate Computing Circuits. *IEEE Trans. on Emerging Topics in Computing* 7, 1 (2019), 18–30.
- [134] Kumud Nepal, Yuetong Li, R. Iris Bahar, and Sherief Reda. 2014. ABACUS: A Technique for Automated Behavioral Synthesis of Approximate Computing Circuits. In *Design, Automation & Test in Europe (DATE)*. 1–6.
- [135] Hamza Omar, Masab Ahmad, and Omer Khan. 2017. GraphTuner: An Input Dependence Aware Loop Perforation Scheme for Efficient Execution of Approximated Graph Algorithms. In *IEEE Int'l. Conference on Computer Design (ICCD)*. 201–208.
- [136] Pramesh Pandey, Prabal Basu, Koushik Chakraborty, and Sanghamitra Roy. 2019. GreenTPU: Improving Timing Error Resilience of a Near-Threshold Tensor Processing Unit. In *Design Automation Conference (DAC)*. 1–6.
- [137] Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. 2015. FlexJava: Language Support for Safe and Modular Approximate Programming. In *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering (FSE)*. 745–757.
- [138] Jongse Park, Xin Zhang, Kangqi Ni, Hadi Esmaeilzadeh, and Mayur Naik. 2014. ExpAX: A Framework for Automating Approximate Programming. *Georgia Institute of Technology Technical Report GT-CS-14-05* (2014), 1–17.
- [139] Yongjoo Park, Jingyi Qing, Xiaoyang Shen, and Barzan Mozafari. 2019. BlinkML: Efficient Maximum Likelihood Estimation with Probabilistic Guarantees. In *ACM SIGMOD Int'l. Conference on Management of Data (MOD)*. 1135–1152.
- [140] Masoud Pashaeifar, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. 2018. Approximate Reverse Carry Propagate Adder for Energy-Efficient DSP Applications. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 26, 11 (2018), 2530–2541.
- [141] William H Pierce. 2014. *Failure-Tolerant Computer Design*. Academic Press.
- [142] Ratko Pilipović, Patricio Bulić, and Uroš Lotrič. 2021. A Two-Stage Operand Trimming Approximate Logarithmic Multiplier. *IEEE Trans. on Circuits and Systems I: Regular Papers* 68, 6 (2021), 2535–2545.
- [143] Powermag. 2016. *Computers May Need More Power than the World Can Generate by 2040*. <https://www.powermag.com/computers-may-need-more-power-than-the-world-can-generate-by-2040/>
- [144] Do Le Quoc, Martin Beck, Pramod Bhatotia, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe. 2017. PrivApprox: Privacy-Preserving Stream Analytics. In *USENIX Annual Technical Conference (ATC)*. 659–672.
- [145] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. 2017. StreamApprox: Approximate Computing for Stream Analytics. In *ACM/IFIP/USENIX Int'l. Middleware Conference*. 185–197.
- [146] Rengarajan Ragavan, Benjamin Barrois, Cedric Killian, and Olivier Senteys. 2017. Pushing the Limits of Voltage Over-Scaling for Error-Resilient Applications. In *Design, Automation & Test in Europe (DATE)*. 476–481.
- [147] Rengarajan Ragavan, Cedric Killian, and Olivier Senteys. 2016. Adaptive Overclocking and Error Correction Based on Dynamic Speculation Window. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 325–330.
- [148] Arnab Raha and Vijay Raghunathan. 2017. qLUT: Input-Aware Quantized Table Lookup for Energy-Efficient Approximate Accelerators. *ACM Trans. on Embedded Computing Systems* 16, 5s (2017), 130:1–130:23.
- [149] Arnab Raha, Swagath Venkataramani, Vijay Raghunathan, and Anand Raghunathan. 2015. Quality Configurable Reduce-and-Rank for Energy Efficient Approximate Computing. In *Design, Automation & Test in Europe (DATE)*. 665–670.
- [150] Abbas Rahimi, Luca Benini, and Rajesh K. Gupta. 2013. Spatial Memoization: Concurrent Instruction Reuse to Correct Timing Errors in SIMD Architectures. *IEEE Trans. on Circuits and Systems II: Express Briefs* 60, 12 (2013), 847–851.
- [151] Shankar Ganesh Ramasubramanian, Swagath Venkataramani, Adithya Parandhaman, and Anand Raghunathan. 2013. Relax-and-Retime: A Methodology for Energy-Efficient Recovery Based Design. In *Design Automation Conference*

- (DAC). 1–6.
- [152] Ashish Ranjan, Arnab Raha, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. 2014. ASLAN: Synthesis of Approximate Sequential Circuits. In *Design, Automation & Test in Europe (DATE)*. 1–6.
 - [153] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. 2012. Programming with Relaxed Synchronization. In *ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*. 41–50.
 - [154] Martin C. Rinard. 2006. Probabilistic Accuracy Bounds for Fault-Tolerant Computations That Discard Tasks. In *ACM Int'l. Conference on Supercomputing (ICS)*. 324–334.
 - [155] Martin C. Rinard. 2007. Using Early Phase Termination to Eliminate Load Imbalances at Barrier Synchronization Points. In *ACM SIGPLAN Int'l. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 369–386.
 - [156] Martin C. Rinard. 2012. Un同步ized Techniques for Approximate Parallel Computing. In *ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*. 1–7.
 - [157] Sanghamitra Roy and Koushik Chakraborty. 2012. Predicting Timing Violations through Instruction-Level Path Sensitization Analysis. In *Design Automation Conference (DAC)*. 1074–1081.
 - [158] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. 2016. Floating-Point Precision Tuning Using Blame Analysis. In *IEEE/ACM Int'l. Conference on Software Engineering (ICSE)*. 1074–1085.
 - [159] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning Assistant for Floating-Point Precision. In *SC13: Int'l. Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12.
 - [160] Hassaan Saadat, Haris Javaid, Aleksandar Ignjatovic, and Sri Parameswaran. 2020. REALM: Reduced-Error Approximate Log-based Integer Multiplier. In *Design, Automation & Test in Europe (DATE)*. 1366–1371.
 - [161] Hassaan Saadat, Haris Javaid, and Sri Parameswaran. 2019. Approximate Integer and Floating-Point Dividers with Near-Zero Error Bias. In *Design Automation Conference (DAC)*. 1–6.
 - [162] Farnaz Sabetzadeh, Mohammad Hosseini Moaiyeri, and Mohammad Ahmadinejad. 2019. A Majority-Based Imprecise Multiplier for Ultra-Efficient Approximate Image Multiplication. *IEEE Trans. on Circuits and Systems I: Regular Papers* 66, 11 (2019), 4200–4208.
 - [163] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-Based Approximation for Data Parallel Applications. In *ACM Int'l. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 35–50.
 - [164] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-Tuning Approximation for Graphics Engines. In *IEEE/ACM Int'l. Symposium on Microarchitecture (MICRO)*. 13–24.
 - [165] Adrian Sampson. 2015. Hardware and Software for Approximate Programming. *University of Washington PhD Dissertation* (2015), 1–212.
 - [166] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. 2015. ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing. *University of Washington Technical Report UW-CSE-15-01-01* (2015), 1–14.
 - [167] 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 Conference on Programming Language Design and Implementation (PLDI)*. 164–174.
 - [168] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and Verifying Probabilistic Assertions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 112–122.
 - [169] Ilaria Scarabottolo, Giovanni Ansaldi, George A. Constantinides, Laura Pozzi, and Sherief Reda. 2020. Approximate Logic Synthesis: A Survey. *Proc. IEEE* 108, 12 (2020), 2195–2213.
 - [170] Ilaria Scarabottolo, Giovanni Ansaldi, and Laura Pozzi. 2018. Circuit Carving: A Methodology for the Design of Approximate Hardware. In *Design, Automation & Test in Europe (DATE)*. 545–550.
 - [171] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic Optimization of Floating-Point Programs with Tunable Precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 53–64.
 - [172] Jeremy Schlachter, Vincent Camus, Krishna V. Palem, and Christian Enz. 2017. Design and Applications of Approximate Circuits by Gate-Level Pruning. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 25, 5 (2017), 1694–1702.
 - [173] Lukas Sekanina and Zdenek Vasicek. 2013. Approximate Circuit Design by Means of Evolvable Hardware. In *IEEE Int'l. Conference on Evolvable Systems (ICES)*. 21–28.
 - [174] Muhammad Shafique, Waqas Ahmad, Rehan Hafiz, and Jörg Henkel. 2015. A Low Latency Generic Accuracy Configurable Adder. In *Design Automation Conference (DAC)*. 1–6.

- [175] Muhammad Shafique, Rehan Hafiz, Semeen Rehman, Walaa El-Harouni, and Jörg Henkel. 2016. Cross-Layer Approximate Computing: From Logic to Architectures. In *Design Automation Conference (DAC)*. 1–6.
- [176] Kan Shi, David Boland, and George A. Constantinides. 2013. Accuracy-Performance Tradeoffs on an FPGA through Overclocking. In *IEEE Int'l. Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 29–36.
- [177] Kan Shi, David Boland, Edward Stott, Samuel Bayliss, and George A. Constantinides. 2014. Datapath Synthesis for Overclocking: Online Arithmetic for Latency-Accuracy Trade-offs. In *Design Automation Conference (DAC)*. 1–6.
- [178] Qingchuan Shi, Henry Hoffmann, and Omer Khan. 2015. A Cross-Layer Multicore Architecture to Tradeoff Program Accuracy and Resilience Overheads. *IEEE Computer Architecture Letters* 14, 2 (2015), 85–89.
- [179] Stelios Sidiropoulos-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin C. Rinard. 2011. Managing Performance vs. Accuracy Trade-Offs with Loop Perforation. In *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering (FSE)*. 124–134.
- [180] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. 2007. Eon: A Language and Runtime System for Perpetual Systems. In *ACM Int'l. Conference on Embedded Networked Sensor Systems (SenSys)*. 161–174.
- [181] Jaswanth Sreeram and Santosh Pande. 2010. Exploiting Approximate Value Locality for Data Synchronization on Multi-Core Processors. In *IEEE Int'l. Symposium on Workload Characterization (IISWC)*. 1–10.
- [182] Phillip Stanley-Marbell et al. 2020. Exploiting Errors for Efficiency: A Survey from Circuits to Applications. *Comput. Surveys* 53, 3 (2020), 1–39.
- [183] Greg Stitt and David Campbell. 2020. PANDORA: An Architecture-Independent Parallelizing Approximation-Discovery Framework. *ACM Trans. on Embedded Computing Systems* 19, 5 (2020), 1–17.
- [184] Antonio Giuseppe Maria Strollo, Ettore Napoli, Davide De Caro, Nicola Petra, and Gennaro Di Meo. 2020. Comparison and Extension of Approximate 4-2 Compressors for Low-Power Approximate Multipliers. *IEEE Trans. on Circuits and Systems I: Regular Papers* 67, 9 (2020), 3021–3034.
- [185] Cheng Tan, Thannirmalai Somu Muthukaruppan, Tulika Mitra, and Ju Lei. 2015. Approximation-Aware Scheduling on Heterogeneous Multi-Core Architectures. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*. 618–623.
- [186] Ye Tian, Qian Zhang, Ting Wang, Feng Yuan, and Qiang Xu. 2015. ApproxMA: Approximate Memory Access for Dynamic Precision Scaling. In *Great Lakes Symposium on VLSI (GLSVLSI)*. 337–342.
- [187] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *Symposium on Implementation and Application of Functional Programming Languages (IFL)*. 1–12.
- [188] Georgios Tzantzioulis, Nikos Hardavellas, and Simone Campanoni. 2018. Temporal Approximate Function Memoization. *IEEE Micro* 38, 4 (2018), 60–70.
- [189] Salim Ullah, Sanjeev Sripadraj Murthy, and Akash Kumar. 2018. SMAApproxLib: Library of FPGA-based Approximate Multipliers. In *Design Automation Conference (DAC)*. 1–6.
- [190] Shaghayegh Vahdat, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. 2019. TOSAM: An Energy-Efficient Truncation- and Rounding-Based Scalable Approximate Multiplier. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 27, 5 (2019), 1161–1173.
- [191] Shaghayegh Vahdat, Mehdi Kamal, Ali Afzali-Kusha, Massoud Pedram, and Zainalabedin Navabi. 2017. TruncApp: A Truncation-Based Approximate Divider for Energy Efficient DSP Applications. In *Design, Automation & Test in Europe (DATE)*. 1635–1638.
- [192] Zdenek Vasicek, Vojtech Mrazek, and Lukas Sekanina. 2019. Automated Circuit Approximation Method Driven by Data Distribution. In *Design, Automation & Test in Europe (DATE)*. 96–101.
- [193] Zdenek Vasicek and Lukas Sekanina. 2015. Evolutionary Approach to Approximate Digital Circuits Design. *IEEE Trans. on Evolutionary Computation* 19, 3 (2015), 432–444.
- [194] Zdenek Vasicek and Lukas Sekanina. 2016. Search-based Synthesis of Approximate Circuits Implemented into FPGAs. In *Int'l. Conference on Field Programmable Logic and Applications (FPL)*. 1–4.
- [195] Vassilis Vassiliadis, Konstantinos Parasyris, Charalambos Chalias, Christos D. Antonopoulos, Spyros Lalidis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2015. A Programming Model and Runtime System for Significance-Aware Energy-Efficient Computing. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 275–276.
- [196] Vassilis Vassiliadis, Jan Riehme, Jens Deussen, Konstantinos Parasyris, Christos D. Antonopoulos, Nikolaos Bellas, Spyros Lalidis, and Uwe Naumann. 2016. Towards Automatic Significance Analysis for Approximate Computing. In *IEEE/ACM Int'l. Symposium on Code Generation and Optimization (CGO)*. 182–193.
- [197] Suganthi Venkatachalam, Elizabeth Adams, Hyuk Jae Lee, and Seok-Bum Ko. 2019. Design and Analysis of Area and Power Efficient Approximate Booth Multipliers. *IEEE Trans. on Computers* 68, 11 (2019), 1697–1703.

- [198] Swagath Venkataramani et al. 2020. Efficient AI System Design With Cross-Layer Approximate Computing. *Proc. IEEE* 108, 12 (2020), 2232–2250.
- [199] Swagath Venkataramani, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2015. Approximate Computing and the Quest for Computing Efficiency. In *Design Automation Conference (DAC)*. 1–6.
- [200] Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. 2013. Substitute-and-Simplify: A Unified Design Paradigm for Approximate and Quality Configurable Circuits. In *Design, Automation & Test in Europe (DATE)*. 1367–1372.
- [201] Swagath Venkataramani, Amit Sabne, Vivek Kozhikkottu, Kaushik Roy, and Anand Raghunathan. 2012. SALSA: Systematic Logic Synthesis of Approximate Circuits. In *Design Automation Conference (DAC)*. 796–801.
- [202] Jing Wang, Xin Fu, Xu Wang, Shubo Liu, Lan Gao, and Weigong Zhang. 2020. Enabling Energy-Efficient and Reliable Neural Network via Neuron-Level Voltage Scaling. *IEEE Trans. on Computers* 69, 10 (2020), 1460–1473.
- [203] Ying Wang, Jiachao Deng, Yuntan Fang, Huawei Li, and Xiaowei Li. 2017. Resilience-Aware Frequency Tuning for Neural-Network-Based Approximate Computing Chips. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 25, 10 (2017), 2736–2748.
- [204] Haroorn Waris, Chenghua Wang, and Weiqiang Liu. 2020. Hybrid Low Radix Encoding-Based Approximate Booth Multipliers. *IEEE Trans. on Circuits and Systems II: Express Briefs* 67, 12 (2020), 3367–3371.
- [205] Haroorn Waris, Chenghua Wang, Weiqiang Liu, and Fabrizio Lombardi. 2021. AxBMs: Approximate Radix-8 Booth Multipliers for High-Performance FPGA-Based Accelerators. *IEEE Trans. on Circuits and Systems II: Express Briefs* 68, 5 (2021), 1566–1570.
- [206] Zhenyu Wen, Do Le Quoc, Pramod Bhatotia, Ruichuan Chen, and Myungjin Lee. 2018. ApproxIoT: Approximate Analytics for Edge Computing. In *IEEE Int'l. Conference on Distributed Computing Systems (ICDCS)*. 411–421.
- [207] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. 2016. Approximate Computing: A Survey. *IEEE Design & Test* 33, 1 (2016), 8–22.
- [208] Wenbin Xu, Sachin S. Sapatnekar, and Jiang Hu. 2018. A Simple Yet Efficient Accuracy-Configurable Adder Design. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 26, 6 (2018), 1112–1125.
- [209] Zhixi Yang, Ajaypat Jain, Jinghang Liang, Jie Han, and Fabrizio Lombardi. 2013. Approximate XOR/XNOR-Based Adders for Inexact Computing. In *IEEE Int'l. Conference on Nanotechnology (NANO)*. 690–693.
- [210] Amir Yazdanbakhsh et al. 2015. Axilog: Language Support for Approximate Hardware Design. In *Design, Automation & Test in Europe (DATE)*. 812–817.
- [211] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmailzadeh, Onur Mutlu, and Todd C. Mowry. 2016. RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads. *ACM Trans. on Architecture and Code Optimization* 12, 4 (2016), 1–26.
- [212] Rong Ye, Ting Wang, Feng Yuan, Rakesh Kumar, and Qiang Xu. 2013. On Reconfiguration-Oriented Approximate Adder Design and Its Application. In *Int'l. Conference on Computer-Aided Design (ICCAD)*. 48–54.
- [213] Serif Yesil, Ismail Akturk, and Ulya R. Karpuzcu. 2018. Toward Dynamic Precision Scaling. *IEEE Micro* 38, 4 (2018), 30–39.
- [214] Reza Zendegani, Mehdi Kamal, Milad Bahadori, Ali Afzali-Kusha, and Massoud Pedram. 2017. RoBA Multiplier: A Rounding-Based Approximate Multiplier for High-Speed yet Energy-Efficient Digital Signal Processing. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 25, 2 (2017), 393–401.
- [215] Reza Zendegani, Mehdi Kamal, Arash Fayyazi, Ali Afzali-Kusha, Saeed Safari, and Massoud Pedram. 2016. SEERAD: A High Speed yet Energy-Efficient Rounding-Based Approximate Divider. In *Design, Automation & Test in Europe (DATE)*. 1481–1484.
- [216] Georgios Zervakis, Fotios Ntouskas, Sotirios Xydis, Dimitrios Soudris, and Kiamal Pekmestzi. 2018. VOSSim: A Framework for Enabling Fast Voltage Overscaling Simulation for Approximate Computing Circuits. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 26, 6 (2018), 1204–1208.
- [217] Georgios Zervakis, Sotirios Xydis, Dimitrios Soudris, and Kiamal Pekmestzi. 2019. Multi-Level Approximate Accelerator Synthesis Under Voltage Island Constraints. *IEEE Trans. on Circuits and Systems II: Express Briefs* 66, 4 (2019), 607–611.
- [218] Guowei Zhang and Daniel Sanchez. 2018. Leveraging Hardware Caches for Memoization. *IEEE Computer Architecture Letters* 17, 1 (2018), 59–63.
- [219] Jeff Zhang, Kartheek Rangineni, Zahra Ghodsi, and Siddharth Garg. 2018. ThUnderVolt: Enabling Aggressive Voltage Underscaling and Timing Error Resilience for Energy Efficient Deep Learning Accelerators. In *Design Automation Conference (DAC)*. 1–6.
- [220] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. 2015. ApproxANN: An Approximate Computing Framework for Artificial Neural Network. In *Design, Automation & Test in Europe (DATE)*. 701–706.
- [221] Xuhong Zhang, Jun Wang, and Jiangling Yin. 2016. Sapprox: Enabling Efficient and Accurate Approximations on Sub-Datasets with Distribution-Aware Online Sampling. *Proceedings of the VLDB Endowment* 10, 3 (2016), 109–120.

- [222] Feiyu Zhu, Shaowei Zhen, Xilin Yi, Haoran Pei, Bowen Hou, and Yajuan He. 2022. Design of Approximate Radix-256 Booth Encoding for Error-Tolerant Computing. *IEEE Trans. on Circuits and Systems II: Express Briefs* 69, 4 (2022), 2286–2290.

Received 18 July 2023; Revised 13 September 2024; Accepted 4 February 2025; Published 5 March 2025