Approximate Computing: A Survey

Qiang Xu
The Chinese University of Hong Kong

Nam Sung Kim
University of Illinois at Urbana-Champaign

Todd Mytkowicz
Microsoft Research

Editor's notes:
As one of the most promising energy-efficient computing paradigms, approximate computing has gained a lot of research attention in the past few years. This paper presents a survey of state-of-the-art work in all aspects of approximate computing and highlights future research challenges in this field.

—Jie Han, University of Alberta

■ **Despite the advances** in semiconductor technologies and development of energy-efficient design techniques, the overall energy consumption of computer systems is still rapidly growing at an alarming rate in order to process an ever-increasing amount of information. In particular, as computer systems become pervasive, they are increasingly used to interact with the physical world, and process a large amount of data from various sources. Moreover, we expect them to be context aware and present natural human interfaces. Consequently, a large number of applications, commonly referred to as recognition, mining, and synthesis (RMS) applications, have emerged and they account for a significant portion of computational resources across the computing spectrum, from mobile and Internet of Things (IoT) devices to large-scale data centers [1].

It is essential to dramatically improve the energy efficiency for these emerging workloads in order to keep pace with the growth of information that needs to be processed. Fortunately, such applications usually feature an intrinsic error-resilience property [2]. They process noisy and redundant data from nontraditional input sources such as various types of sensors (inexact inputs) and the associated algorithms are often stochastic in nature (e.g., iterative algorithms). Moreover, these applications usually do not require to compute a unique or golden numerical result ("acceptable" instead of precise outputs). For example, in multimedia processing, due to the limited perceptual capability of humans, occasional errors such as dropping a particular frame or a small image quality loss often rarely affect a user's satisfaction. As another example in data analytics, consider two different classifiers that produce similar classification results on a set of example objects. It is very difficult, if not impossible, to tell which one is "better" for the classification of new objects.

On the other hand, it is increasingly energy inefficient to ensure fault-free computations as semiconductor technology advances to nanometer regime. This is because circuits are more sensitive to parameter variations and faults at advanced technology node with low supply voltage and ever-increasing integration density [3]. Conventional fault-free computing requires guardbands and redundancies at various levels of design hierarchy

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.
Digital Object Identifier 10.1109/MDAT.2015.2505723
Date of publication: 07 December 2015; date of current version: 18 January 2016.

for variation tolerance and error correction, causing significant energy overhead.

Motivated by the above challenges, one promising solution, known as approximate computing, has attracted significant traction from both academia and industry. By relaxing the numerical equivalence between the specification and implementation of error-tolerant applications, approximate computing deliberately introduces "acceptable errors" into the computing process and promises significant energy-efficiency gains. Considering the fact that traditional Dennard's scaling provides diminishing returns with technology advancement, leveraging the new source of energy efficiency provided by approximate computing is increasingly important.

Over the years, various approximate computing techniques at almost all computing layers have been presented in the literature. The objectives of this article are twofold. First, we put existing approximate computing work in perspective and consolidate existing results and insights by presenting a survey of the literature. Note that we do not aim at a complete survey of the literature, but rather a thought-provoking portrait of key aspects of the state-of-the-art knowledge. Second, we highlight open challenges and discuss future research directions. When compared to a recent survey [4], our taxonomy and organization are completely different and we take a much broader viewpoint that spans all computing layers.

The remainder of this article is organized as follows. The approximate computing paradigm section overviews the approximate computing paradigm. In the approximate software, approximate architecture, and approximate circuit section, we survey and consolidate existing techniques in approximate software development, approximate architecture exploration, and approximate circuit design, respectively. Finally, we outline challenges and future research directions in approximate computing in the Challenges and future research directions section.

The approximate computing paradigm

Computing has entered the era of approximation, in which hardware and software generate and reason about estimates. Navigation applications turn maps and location estimates from hardware global positioning system (GPS) sensors into driving directions; speech recognition turns an analog signal into a likely sentence; and search turns queries into information. These complex systems require sophisticated algorithms to deliver good enough answers quickly, at scale, and with energy efficiency, and approximation is often the only way to meet these competing goals. In this section, we briefly review the approximate computing paradigm.

For a given error-tolerant application, the overall framework to apply approximate computing is shown in Figure 1. The key components of the framework are discussed as follows.

Approximate kernels. Approximate kernels refer to those techniques used to realize approximation for energy-efficiency gains. Generally speaking, there are two types of approximate kernels.

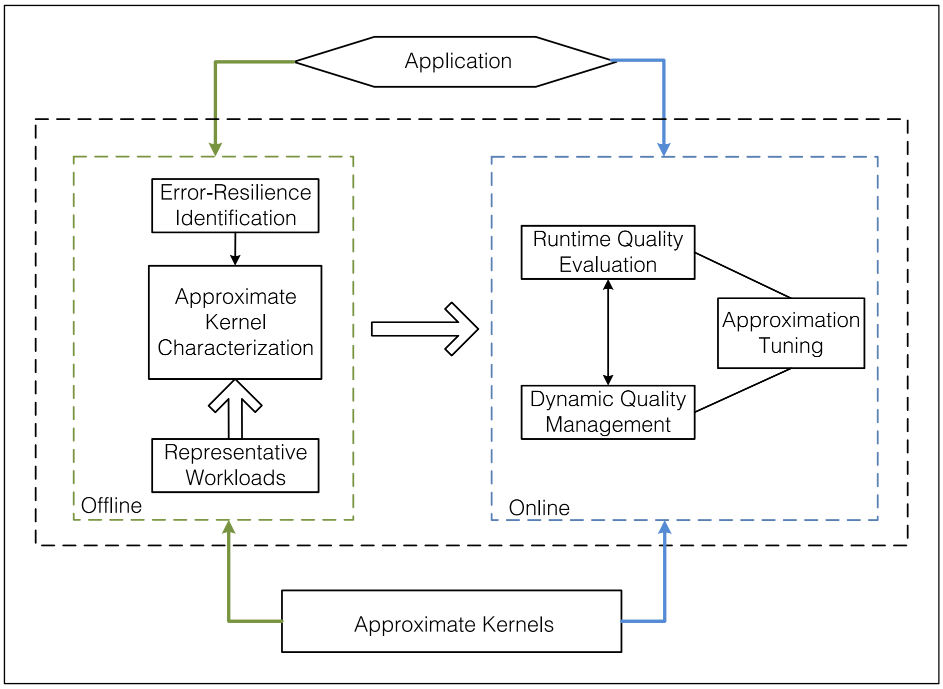
Application
Error-Resilience Identification
Approximate Kernel Characterization
Representative Workloads
Offline
Runtime Quality Evaluation
Approximation Tuning
Dynamic Quality Management
Online
Approximate Kernels

Figure 1. Overall framework of approximate computing.

Table 1 **Approximate kernels at different layers.**

	Approximation Technique
Program	Loop perforation [5], Code perforation [6], Thread fusion [11], Tunable kernels [11], Pattern reduction [37]
Architecture	Approximate storage [19], ISA extensions [7], Approximate accelerators [14]
Circuit	Imprecise logic [51], Voltage overscaling [48], Analog computation [15], Precision scaling [18]

At the hardware layer, we could either use a less accurate yet more energy-efficient circuit for computation or purposely reduce the supply voltage for certain hardware components to tradeoff energy and accuracy. At the software layer, we could selectively ignore certain computations and/or memory accesses that are not critical to the final quality of the application to achieve the same objective.

A few representative approximate kernels spanning from application level to circuit level are shown in Table 1.

Resilience identification and characterization. Even for error-tolerant applications, there exist error-sensitive parts for which using inexact computations may cause fatal errors. Consequently, one critical step to employ approximate computing is to identify those error-resilient parts that can be approximated.

One way to identify resiliency is to annotate applications and/or to use dedicated approximate data types to express where approximation is feasible and how it may impact the results (e.g., [7] and [8]). Alternatively, we could divide applications into segments and conduct sensitivity analysis for each of them. Some segments can be easily determined to be sensitive (e.g., control flow) or resilient (e.g., iterative methods [10]). For the remaining ones, it is possible to inject errors and perform simulations with representative workloads to evaluate the impact of approximation [2]. If the application crashes or produces unacceptable results when injecting errors to a specific segment, it is marked as error sensitive; otherwise, it is resilient. However, such a method with an insufficient amount of simulation could be risky because it might fail to capture cases where the effect of corrupted data is not immediately manifested.

After resilience identification, we characterize the impact of different approximate kernels on the resilient parts, typically by performing simulations on representative workloads. Note that, as the actual workloads running on the system would be different from simulated ones, it is usually necessary to perform dynamic quality management to guarantee the final application quality.

Quality management. In dynamic quality management, we evaluate the intermediate computation quality (preferably with application-specific lightweight checker) and determine when to use approximate kernels and their corresponding approximation modes (if any) at runtime.

SAGE [11] and GREEN [22] check the output quality once in every N invocations of the approximate kernel, in which they compare the quality between approximate computation and exact computation and adjust the approximate modes for the subsequent computations accordingly. Obviously, with such a quality sampling strategy, the more often the quality is checked, the higher confidence for the final quality of the application, at the cost of more energy overhead. On the other hand, it is unrealistic to monitor all invocations using this scheme because it violates the very purpose of approximation.

One critical problem with the above quality sampling scheme is that we can never guarantee the output qualities for those invocations that are not checked. Consequently, continuous quality monitoring is preferred. No doubt to say, only low overhead checkers can be used in order not to offset the gains from approximation. Inspired by the large body of speculative prediction work (e.g., branch prediction and value prediction), a few quality predictors for approximate computing were presented in the literature. In [21], Ringenburg et al. introduced the so-called fuzzy memoization technique, which records previous inputs and outputs of the checked code and predicts the output of the current execution from past executions with similar inputs. The quality of the approximation is estimated by checking how different the current output is from the predicted one. Rumba [23] constructs several simple error prediction models by observing the inputs or the outputs of approximate accelerators. If large approximation errors are predicted, exact recomputation is performed for error correction.

In the following sections, we discuss the approximation techniques at different computing layers in detail.

Approximate software

A good programming language should enable programmers to be productive. It should allow programmers to quickly express their ideas as programs while at the same time allow a compiler or runtime system to optimize their program's execution. In many ways, a programming language and its implementation balance programmer productivity with a system's efficiency.

The mechanism by which programming languages provide this balance is "abstraction." An abstraction allows the programmer to say what to do in order to accomplish a task, not how to do it. The "how to do it"—or the implementation part of the abstraction—is left to the compiler or runtime system.

Building sophisticated applications is difficult and requires expertise across the system stack, and in this era of approximation, that expertise includes statistics (or some other approximation-aware reasoning) in addition to application-specific domain knowledge. It is no wonder, then, that many researchers have proposed abstractions to help programmers with this daunting task. These abstractions help programmers express their ideas in code (i.e., approximate-aware programming languages), let analysis engines reason about the correctness of such programs (i.e., approximate-aware analyses), and let compilers generate machine code (i.e., approximate-aware compilers).

This section reviews recent academic and industrial work from these areas of approximate-aware programming languages, approximate-aware analyses, and approximate-aware compilers.

Approximation-aware programming languages

Approximation-aware programming languages let programmers express where and how randomness can impact their results. In 1979, Kozen recognized the need for both a syntax and semantics for programs that use randomness during execution [30]. The goal of this early work was to use a programming language to specify the meaning of a program with probabilistic constructions. That work provides two semantics for a simple probabilistic language—one that models sampling and one that computes on probability distributions directly—and proves them equivalent. This result showed that follow-on work could use sampling to approximate computing with probability distributions directly.

Expressing randomness in programs. There has been significant follow-on work wherein languages let programmers express how randomness impacts their program. For example, programming languages such as Eon [41], EnerJ [7], and Rely [25] expose approximation to the programmer through language syntax. Consider EnerJ and Rely, which let a programmer annotate their program with approximate type information. In EnerJ, the programmer must explicitly cast from an approximate type to a precise one, while in Rely the compiler tries to reason statically about how an approximate type flows through to others.

Libraries, such as Uncertain $\langle T \rangle$ [8], provide abstractions that encapsulate approximate data within standard object-oriented programming languages and propagates approximate data through a program's variables at runtime. When a program needs to act on that approximate data (i.e., at a conditional) the Uncertain $\langle T \rangle$ runtime uses hypothesis tests to make statistically correct branch choices.

The key challenges to approximation-aware programming languages is to automate as much reasoning as possible so the programmer need not understand many of the details of how approximation affects the accuracy of her program.

Inference via probabilistic programming. A significant body of work tries to address this challenge by building on the theoretical constructs introduced by Kozen. Probabilistic programming languages augment existing programming languages with probabilistic primitives (see [28] for a summary). The major goal of these languages is the efficient implementation of probabilistic inference, which combines a model (written in the probabilistic programming language) with observed evidence to infer a distribution over variables in the program in light of that evidence. The canonical probabilistic programming example answers, "given that the grass is wet, was it due to rain or the sprinkler?" As a probabilistic program, the programmer would describe a probabilistic model, or the relationship between variables in a program (i.e., the likelihood of rain and that when it is raining, someone is unlikely to use his sprinkler), then specifies evidence (i.e., the grass is wet), and the probabilistic programming runtime infers properties of the model (i.e., was it the rain or a sprinkler that more likely caused grass to be wet) in light of this evidence.

These languages abstract the details of inference, and hence are frequently used by machine learning experts when building their models. Probabilistic programming has made significant strides in democratizing probabilistic inference; they let machine learning experts encode models and then ask complicated and computationally demanding queries via probabilistic inference, of those models. In general, probabilistic inference is $\mathtt{NP-Hard}$ and so much of the research in probabilistic programming languages is about how to make that inference efficient.

Some probabilistic programming languages such as Church [27] can perform inference on any distribution they can represent. Other probabilistic programming languages trade expressivity for performance and restrict the distributions they allow thus making inference more tractable and efficient. Infer.NET [24], [33] uses various approximate and exact inference engines, each of which has different restrictions.

As a concrete example, consider a simple probabilistic programming implementation in LINQ wherein a developer expresses both a probabilistic program and inference over that program as a LINQ query. The first program in Listing 1a provides an implementation of a binomial distribution (over three coin flips) while the second demonstrates how to add constraints—or evidence—via a LINQ `where` clause. These "programs" use LINQ to enumerate all possible flips of a fair coin, and it is the job of `Inference` (shown in Listing 1b) to find the marginal distribution that maps program outputs to likelihoods. For example, running `Inference (Binomial())` would result in $[(0, 0.125), (1, 0.375), (2, 0.375), (3, 0.175)]$ while `Inference (ConstrainedBinomial())` would result in $[(0, 0.25), (1, 0.5), (2, 0.25)]$.

This simple example clearly shows some of the difficulty in probabilistic inference: enumerating all possible paths of a program is intractable for all but small programs (this implementation of inference is exponential in the number of random primitives). Much of the insights behind probabilistic programming is in: 1) letting programmers easily express a probabilistic program such that 2) a runtime can efficiently perform inference on such a program without enumerating all possible paths of a program. To the extent to which the

```
IEnumerable<int> Binomial() {
  var flip = new[] { 0, 1 };
  var program = from a in flip
                from b in flip
                from c in flip
                let output = a + b + c
                select ouptut;
  return program;
}

IEnumerable<int> ConstrainedBinomial() {
  var flip = new[] { 0, 1 };
  var program = from a in flip
                from b in flip
                from c in flip
                where a == 0
                let path = a + b + c
                select path;
  return program;
}
```
(a)

```
IEnumerable<Tuple<T, double>>
    Inference<T>(IEnumerable<T> program) {
  // run the program and aggregate output
  var groups =
    from output in program
    group output by output into summary
    select summary;
  // # of program paths is normalization
  var numpaths = (double) program.Count();
  // normalize probabilities by # of paths
  var infer =
    from pair in groups
    let output     = pair.Key
    let outputProb = pair.Count() / numpaths
    select Tuple.Create(output, outputProb);
  return infer;
}
```
(b)

Listing 1. (a) Two probabilistic programs in LINQ. (b) Probabilistic inference in LINQ.

probabilistic programming community makes strides toward this goal, inference could become one abstraction that unites an approximate architecture with a program written for that architecture. In other words, given a program inference may let us infer the range of allocable approximation an architecture must adhere to in order to guarantee certain correctness properties and quality constraints in the program.

Approximation-aware analyses

The goal of an analyses is to model or understand the semantics of part, or all, of a program. Probabilistic modeling is widely used in the design and analysis of computer systems, and has been rapidly gaining in importance in recent years. This section reviews recent work in this area.

Model checking. Probabilistic model checking is an automatic procedure for establishing whether a desired property holds in a probabilistic system (see Legay and Delahaye's survey [32]). Conventional model checkers let a programmer formally describe a state-transition system (usually expressed in temporal logic) and the model checker provides formal guarantees about properties of the system. In contrast, a probabilistic model checker requires the programmer to specify state transitions with probabilities (i.e., instead of the existence of such a transition) and thus provides probabilistic guarantees. The popular tools MRMC [29] and PRISM [31] have been used in diverse fields ranging from analyzing the source code of wireless protocols for faults to modeling human driver behavior for use in autonomous driving systems.

Probabilistic accuracy. Likewise, some research adapts more traditional program analyses to the probabilistic domain. For example, probabilistic static program analyses compute conservative bounds on the probability of large output deviations given two inputs that are close to each other. These analyses reason about programs 1) that operate on approximate hardware [34], 2) are transformed using accuracy-aware transformations [26], and 3) operate on uncertain inputs [38]. Likewise, dynamic program analyses estimate the probability of large output deviations by running programs on representative inputs [36].

Approximation, be it either from approximate hardware, approximate compiler transformations, or others, let a runtime balance an application's accuracy with its performance and or energy consumption. Notable recent work searches for optimizations and or transformations that maximize energy savings subject to accuracy constraints. This exploration is done via dynamic testing [5], [22], [35], [36] or statically reducing to an optimization problem and then using linear or integer mathematical programming solvers [34].

Approximation-aware compilers

Approximation-aware compilers can use approximations (inferred or explicitly expressed), in addition to the approximation-aware analyses, to automatically transform and thus change the semantics of programs to trade the accuracy of the program's result for the improved performance and/or energy consumption. For instance, loop perforation is a software-only technique that modifies the program to execute fewer loop iterations and therefore make the program run faster [5]. A compiler can also automate the placement of operations that execute on approximate hardware [34]. Likewise, Stoke [39] reduces the bit width of floating-point operations at compile time, trading accuracy for performance. Sampson et al. demonstrate how to lift the semantics of a program from its concrete operations on values to operations over distributions of values [38]. As a consequence, they apply transformations which do not obey the semantics of the original program but statistically maintain the semantics of the lifted one, and show these transformations significantly speed up the runtime of the program. Finally, Schkufza et al. introduce "stochastic superoptimization," a technique that uses Monte Carlo Markov chain inference to search the space of optimizations for loop-free 86 code [40]. Their compiler does not introduce approximations into generated code but instead uses an approximate search method to generate fast code without requiring the compiler to know anything about the semantics of the x86 code they are optimizing, just how to introduce, modify, and delete individual x86 statements.

Approximate architecture

A computer system's key hardware components are processor, memory, and storage. For these hardware components, computer architects aim to balance performance with energy efficiency under various constraints imposed by a given technology such as chip area and power. For memory and storage it also attempts to balance density (or the cost per bit) with performance.

However, it is challenging to simultaneously improve the performance, energy efficiency, and density of these components, because improving one often sacrifices the others. For example, an aggressive out-of-order speculation technique improves processor performance, but in concert drastically increases processor energy consumption because of significantly increased circuit and microarchitecture complexities. Besides, the higher the density of memory and storage is, the longer time it takes to store and retrieve information.

The advent of approximating computing introduces another tradeoff aspect—quality of computing (or data) to the traditional tradeoff among performance, energy efficiency, and density; slightly sacrificing computing quality can improve either performance, energy efficiency, or density.

This section reviews recent work proposing processor, memory, and storage subsystem architectures that exploit approximate computing to trade the energy efficiency and/or performance with computing quality.

Approximate processor architectures

The processor architectures supporting approximate computing can be categorized into two distinct groups. The first group aims to support approximate computing for traditional code running on general-purpose processors enhanced to energy-efficiently execute some chosen instructions or code segments in approximate mode. The second group transforms approximable segments of traditional code into a neurally inspired algorithm running on accelerators. Both groups rely on a compiler or a programmer to identify or annotate approximable code segment (cf., the Approximate software section).

Approximate computing with enhanced general-purpose processors. In approximate computing with general-purpose processors, traditional code is analyzed and its precise and approximable instructions or segments are identified or annotated, as reviewed in the Approximate software section. Subsequently, various microarchitectural choices can be made to energy-efficiently execute the approximable instructions or segments; the approximable instructions and segments can be executed by energy efficient but unreliable data paths and cores in fine- and coarse-grained manners.

For fine-grained approximate computing, an instruction set architecture (ISA) may define a set of special instructions that allow the compiler to convey what can be approximated without specifying how. To avoid the overheads of dynamic correctness checks and error recovery, the ISA is targeted for a disciplined style of approximate programming. Then microarchitecture can freely choose from a range of approximation techniques without exposing them. In general, approximation techniques can be classified into runtime and design-time techniques. For example, a processor can be provided with both precise and approximate data paths, as depicted in Figure 2. The approximate data path can be a data path operating at an aggressive voltage level [13] or a specially designed data path with limited precision.

For coarse-grained approximate computing, approximable segments can be offloaded to dedicated energy-efficient but somewhat unreliable cores in a processor, and architectural choices similar to fine-grained approximate computing can be made for energy-efficient but unreliable cores at runtime and design time. For instance, a processor can comprise homogeneous cores, some of which operate at an aggressive voltage to run approximable segments, or heterogeneous cores, some of which are designed to be energy efficient at the cost of reduced precision or occasional compute errors [42].

These discussed processor architectures can be easily adopted for traditional applications containing approximable instructions and segments, but their energy-efficiency benefit can be limited due to the fundamental limitation of traditional von-Neumann processor architecture—instruction fetch and control, data transfer across the processor memory hierarchy, and synchronous clock distribution to every timing elements in a processor. Although the energy consumption of these components can be reduced by allowing approximation and/or occasional faults, they still constitute most of total energy consumption in modern general-purpose processors.

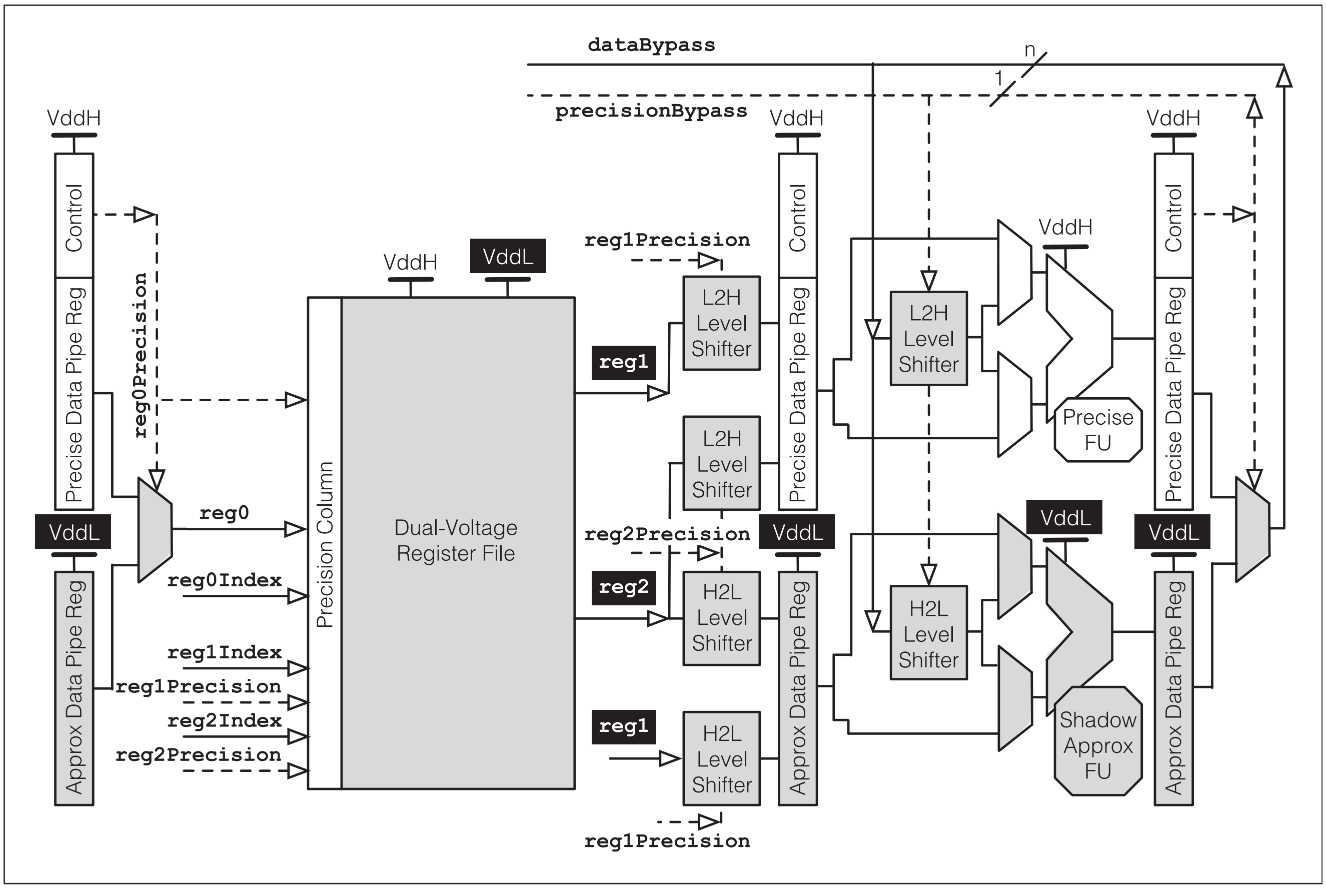

Figure 2. Processor microarchitecture and architecture integrated with energy efficient but unreliable data paths and cores to support approximate computing, respectively.

Approximate computing through algorithmic transformation. A data/compute-intensive approximable code segment can be converted to a neurally inspired algorithm such as artificial neural network (ANN) through a parrot transformation for more efficient processing [14]. The transformed code is typically offloaded to neural accelerators that are coupled with a host processor, as illustrated in Figure 3. The compiler automatically performs the neural transformation and replaces the approximable segment with an invocation of a neural hardware that accelerates the execution of

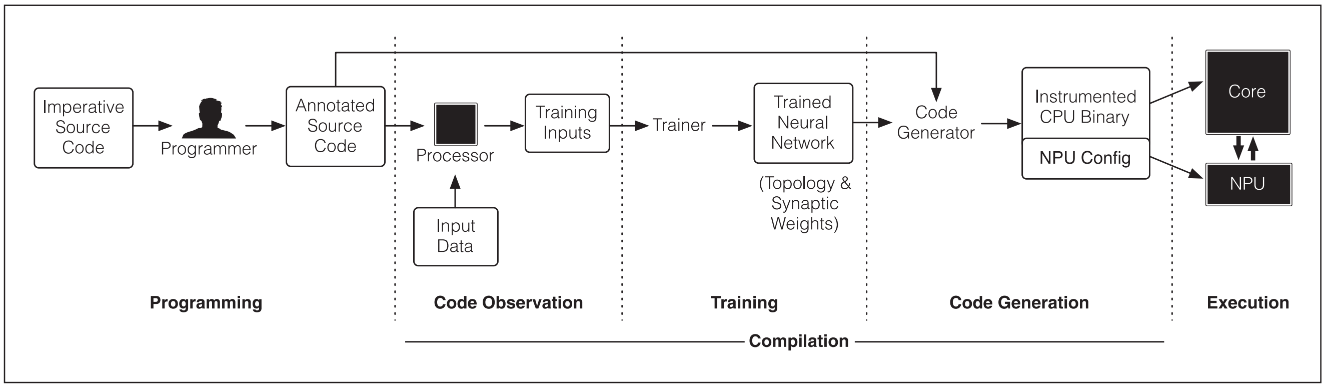

Figure 3. Compiler automatically performs the neural transformation and replaces the approximable segment with an invocation of a neural hardware that accelerates the execution of the thread.

the thread. Since neurally inspired algorithms are inherently error resilient, the neural accelerators can be architected and optimized in various ways to maximize energy efficiency. The Neural accelerators section discusses various low-level implementation choices along with their advantages and disadvantages.

Approximate memory and storage architectures

Approximate memory and storage designs trade the quality of data for: 1) smaller on-chip memory with lower operating voltage (e.g., [43]) or stronger protection from particle strikes (e.g., [44]); 2) lower DRAM energy (e.g., [45]); 3) faster solid-state memory with longer lifetime (e.g., [46]); and 4) higher memory channel bandwidth [47].

Memory. Some on-chip SRAM cells begin to fail as the operating voltage decreases below a certain level, determining the minimum operating voltage (Vmin) of the entire chip and limiting the maximum power efficiency. Furthermore, SRAM cells are vulnerable to particle strikes. Thus, larger SRAM cells with more transistors (e.g., eight or ten transistors instead of six transistors) or larger transistors are required for lower operating voltage and/or stronger protection from particle strikes. Yet, constructing the entire on-chip memory with large SRAM cells is very costly in many aspects. To cost-effectively offer on-chip memory, small SRAM cells can be judiciously used for least significant bits of the on-chip memory if the precision requirement can be relaxed for noncritical data. Such a design technique can notably reduce the space and energy penalties of on-chip memory with negligible quality loss while providing the same Vmin and/or protection from particle strikes.

The DRAM refresh operations may constitute a significant fraction of total DRAM energy consumption, while the refresh rate is often determined by a small number of weak cells. To reduce DRAM energy consumption, a much longer refresh rate is applied to DRAM rows storing error-resilient (or approximable) data. Since only a smaller number of cells will fail even with a significantly longer refresh rate, the impact on the computing quality is negligible while a considerable amount of DRAM refresh energy is reduced.

Storage. The solid-state memory allows us to store/retrieve multiple states to/from a cell [i.e., multilevel cell (MLC) mode]. However, to precisely store/retrieve more states to/from a cell, increasingly more effort (or longer latency) is required. On the other hand, if the precision requirement can be relaxed and some imprecision can be occasionally tolerated, the states can be stored/retrieved with much less effort. Such a characteristic of the solid-state memory can be exploited to store/retrieve error-resilient data with much lower latency [19]. Such a technique can significantly reduce memory latency at the cost of some data quality loss. Furthermore, error-resilient data can be stored in blocks that are worn out and contain uncorrectable faulty bits by prioritizing the exhausted error-correction resources for more significant defective bits [19]. Such a technique can extend the lifetime with negligible quality loss.

Interconnect. Emerging applications are data intensive and thus demand high memory bandwidth. To increase the bandwidth, the data transfer rate and/or the number of memory channels must be increased, but doing so is increasingly more challenging due to stringent package pin and power constraints. To use given bandwidth more efficiently, data can be compressed/uncompressed before/after they are transferred through the memory channel. Such an approach typically employs a lossless compression algorithm but its benefit (i.e., effective data transfer rate) is often limited by how much given data can be compressed (i.e., compression ratio). To further improve the compression ratio and thus effective data transfer rate, a lossy compression algorithm can be applied to error-resilient data since some precision loss in such data can be tolerated.

Approximate circuit

For a given circuit, we could achieve reduced energy consumption by lowering its supply voltage without reducing the corresponding operational frequency. One could use this so-called overscaling technique for approximate computing. However, overscaling-induced timing errors occurring on critical paths often lead to large computational errors, unless the circuit is designed in a scalability-friendly manner [48]. But even so, the energy-efficiency gain for such approximation method is relatively small. Consequently, the majority of approximate circuit proposals appearing in the literature resort to functional approximation, in which the circuit's original function is deliberately modified for energy-accuracy tradeoff.

In this section, we first introduce various approximate arithmetic unit designs, which are usually handcrafted due to their wide applications and well-researched structure. Next, we discuss approximate synthesis solutions for general logic circuit approximation. Finally, we introduce neural accelerators that are able to achieve significant energy-efficiency gains.

Approximate arithmetic units

Since adder is one of the key arithmetic circuits used in RMS applications, many approximate adder designs have been proposed in the literature. Jiang et al. gave a comparative review on recent work in this domain [12]. We describe a few representative approximate adder designs in the following.

In [49], a transistor-level approximation is utilized to construct approximate adder cells. When compared with a conventional mirror-adder cell, the layout area of the most aggressive approximate implementation is only about one-third of it. In order to maintain a reasonable output quality, however, such approximate adder cells can only be applied for the least significant bits (LSBs). In [50], Kim et al. proposed a carry skip adder (CSA), which uses a segmentation method to truncate the carry propagation chain and the computation of each segment is based on the carry signal speculated using the bits in a lower segment.

As the quality requirements of applications may vary significantly at runtime, accuracy-configurable approximate arithmetic unit designs are preferable. One could cut the original adder into several sub-adders and combine partial summations of sub-adders to generate final results. An error detection and correction unit is introduced for error mitigation, and we could configure runtime accuracy by controlling the amount of corrections used in the calculation procedure. For example, Ye et al. [51] presented a configurable approximate adder design that degrades computation quality gracefully, in which approximation errors can be corrected from MSBs to LSBs. As depicted in Figure 4, it consists of several k-bit subadders, whose carry-ins can be either selected from less significant subadder units or carry-in prediction components. By setting the length of the subadders and the control signals of the multiplexers, the accuracy of the adder is not only configurable, but more importantly, it degrades gracefully, thereby enabling a better quality-effort tradeoff.

Another important arithmetic unit used in computing is the multiplier. There was some recent work in this domain and the basic design principle is similar to approximate adders. A power-efficient multiplier constructed with 2 × 2 approximate multiplier blocks was proposed in [52]. In [53], approximate adders are leveraged to build approximate multiplier with configurable accuracy.

Approximate circuit synthesis

While we could manually craft approximate arithmetic units with their well-known structures, for general logic circuits, it is preferable to resort to automated synthesis techniques to create their approximate versions to save design efforts.

Unlike arithmetic units, there does not exist well-accepted error model for arbitrary circuits. Consequently, the key challenge in approximate circuit synthesis is how to effectively represent quality constraints and bring them into the synthesis procedure efficiently.

Early attempts in this domain adopt rather simple error models. The synthesis problem is formulated as finding the approximate design with minimum area under a given error rate constraint

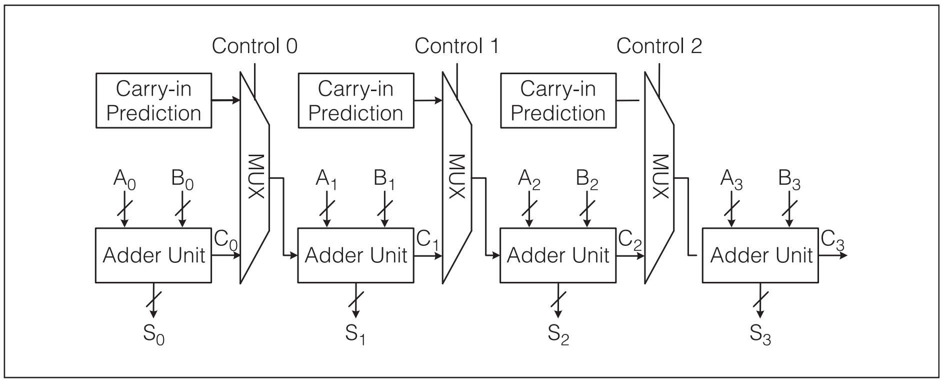

Figure 4. Accuracy-configurable adder in [51].

[54]. Later, both error rate and error magnitude are considered in which a two-phase approach was adopted to tackle the problem [55]. In the first phase, the circuit is synthesized under error magnitude constraint only. Then, the obtained circuit is iteratively refined to meet the error rate constraint in the second phase. SALSA [56] encodes quality constraints into a virtual logic function and simplifies the circuit with "approximation don't cares" that arise from quality relaxation. By doing so, designers could specify more sophisticated quality constraints. Another advantage of SALSA is that it can reuse off-the-shelf logic synthesis tools for approximate circuit synthesis. The above techniques are mainly applicable for approximating combinational logic. ASLAN [57] is able to synthesize approximate sequential circuit, by extending SALSA with virtual sequential quality constraint circuits.

Recently, Yazdanbakhsh et al. [9] proposed a set of language annotations for Verilog, namely Axilog for approximate hardware design. Axilog enables designers to approximate certain parts of the design, while keeping the critical parts accurate. While it has great potential, how to effectively make use of such language extension during logic synthesis remains a largely unexplored problem.

Finally, there are also some recent attempts for the high-level synthesis of approximate circuits, in which operator transformation [58], bitwidth optimization, and approximation-aware list scheduling [59] are used for circuit simplification.

Neural accelerators

Neural networks, by their nature, are general-purpose approximate functions. With the emergence of the approximate computing paradigm, they are proposed to accelerate hot codes in error-resilient applications [14]–[16].

There is a large body of work on hardware implementations of neural networks, and they can be implemented with digital logic, analog-signal, or mixed-signal circuit. Generally speaking, digital implementations offer high precision and better reliability, while analog-signal and mixed-signal implementations are much more compact and energy efficient. Note that, however, it becomes increasingly harder for analog/mixed-signal circuits to be efficiently implemented and reap the key benefits of technology scaling than for digital circuits. Consequently, to determine whether an analog implementation of neural accelerators is superior to a digital implementation, a thorough study must be performed considering various practical issues associated with large-scale integration and manufacturability.

Recently, RRAM crossbar array formed by the emerging memristor devices has gained a lot of interest due to its ultraintegration density. As it can naturally approximate the matrix–vector multiplication without actual computation, we can use it for approximate computing. When compared to its digital implementation counterpart, the energy efficiency is shown to be an order of magnitude better [16]. While having great potential, RRAM-based systems face many challenges as well. First, the energy overhead of the interfacing circuits between the RRAM-based analog data processing unit and the rest of the system is quite high and may offset the efficiency gains from such approximation. Second, the nonideal factors of RRAM devices such as signal fluctuation and process variation make the quality control of approximation more challenging.

Challenges and future research directions

While approximate computing has gained significant traction in recent years, it is still in its infancy. Current solutions often focus on a single layer of the hardware/software stack, and most of them work only for a small set of applications that they are designed to handle. However, unlike days past where software and hardware could independently make progress with stable unifying interface (e.g., an ISA), if an application is unable to effectively communicate its accuracy requirements to the underlying hardware platform and monitor the computational quality at runtime, it is unrealistic to expect the system to be able to achieve a desired amount of energy-efficiency gain under quality constraints. There are some research attempts (e.g., [20]) to construct new hardware/software interface for approximate computing, but such general solutions usually leave a heavy burden on the programmers to ensure application quality and hence hinder their adoption.

Furthermore, software engineering—the construction of complex and reliable code—has been enabled by abstraction and composition, starting with Boolean algebra and working up. We do not yet have an equivalent way of specifying, characterizing, and reasoning unreliable hardware, or APIs to higher level library routines that may produce nondeterministic results. For example, many approximate computing works so far have shown output images and asked the reader to rate the quality as subjectively good enough. While this will serve for end-user perceptual studies, it does not enable formal reasoning about the system. In order to apply scientific and engineering methods to designing, optimizing, and manufacturing an approximate computing system (both hardware and software), we will need to refine the end-user perception metric into quantifiable, testable specifications for individual components along with solutions for the following questions. First, how does a manufacturer determine whether a part (e.g., approximate processor) on a production line passes qualifying tests for shipment (or guarantees the quality of approximate computing for a given part)? Second, how should hardware be characterized? Third, what architectural guarantees will software rely on? Fourth, how should library API functions be specified for introduction of errors, or the propagation of errors from inputs to outputs? Finally, what programming languages (syntax, type systems, etc.) and tools (program analysis, debuggers) should capture the programmer's intent and assist with meeting a specification?

To sum up, existing work in approximate computing has shown great promises for certain domains, but significant innovations and research efforts are needed to enable approximate computing as a practical mainstream computing paradigm. ■

■ References

[1] Y.-K. Chen et al., "Convergence of recognition, mining, synthesis workloads and its implications," *Proc. IEEE*, vol. 96, no. 5, pp. 790–807, May 2008.

[2] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *Proc. IEEE/ACM Design Autom. Conf.*, 2013, DOI: 10.1145/2463209.2488873.

[3] S. Borkar, T. Karnik, and V. De, "Design and reliability challenges in nanometer technologies," in *Proc. IEEE/ACM Design Autom. Conf.*, 2004, pp. 7–11.

[4] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *Proc. IEEE Eur. Test Symp.*, 2013, DOI: 10.1109/ETS.2013.6569370.

[5] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 25–34.

[6] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using code perforation to improve performance, reduce energy consumption, respond to failures," Massachusetts Inst. Technol. (MIT), Cambridge, MA, USA, Tech. Rep. MIT-CSAIL-TR-2009-042, 2009.

[7] A. Sampson et al., "EnerJ: Approximate data types for safe and general low-power computation," in *Proc. Int. Conf. Programm. Lang. Design Implement.*, 2011, pp. 164–174.

[8] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain ⟨T⟩: A first-order type for uncertain data," in *Proc. 19th Int. Conf. Architect. Support Programm. Lang. Oper. Syst.*, 2014, pp. 51–66.

[9] A. Yazdanbakhsh et al., "Axilog: Language support for approximate hardware design," in *Proc. IEEE/ACM Design Autom. Test Eur.*, 2015, pp. 812–817.

[10] Q. Zhang, F. Yuan, R. Ye, and Q. Xu, "ApproxIt: An approx-imate computing framework for iterative methods," in *Proc. ACM/IEEE Design Autom. Conf.*, 2014, DOI: 10.1109/DAC.2014.6881424.

[11] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning approximation for graphics engines," in *Proc. Int. Symp. Microarchitect.*, 2013, pp. 13–24.

[12] H. Jiang, J. Han, and F. Lombardi, "A comparative review and evaluation of approximate adders," in *Proc. Great Lakes Symp. VLSI*, 2015, pp. 343–348.

[13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *Proc. Int. Conf. Architect. Support Programm. Lang. Oper. Syst.*, 2012, pp. 301–312.

[14] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proc. Int. Symp. Microarchitect.*, 2012, pp. 449–460.

[15] R. Amant et al., "General-purpose code acceleration with limited-precision analog computation," in *Proc. Int. Symp. Comput. Architect.*, 2014, pp. 505–516.

[16] B. Li et al., "RRAM-based analog approximate computing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 12, pp. 1905–1917, Dec. 2015.

[17] C. Alvarez, J. Corbal, and M. Valero, “Fuzzy memoization for floating-point multimedia applications,” *IEEE Trans. Comput.*, vol. 54, no. 7, pp. 922–927, Jul. 2005.

[18] Y. Tian, Q. Zhang, T. Wang, F. Yuan, and Q. Xu, “ApproxMA: Approximate memory access for dynamic precision scaling,” in *Proc. Great Lakes Symp. VLSI*, 2015, pp. 337–342.

[19] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, “Approximate storage in solid-state memories,” in *Proc. Int. Symp. Microarchitect.*, 2013, pp. 25–36.

[20] S. Venkataramani, V. K. Chippa, and S. T. Chakradhar, “Quality programmable vector processors for approximate computing,” in *Proc. Int. Symp. Microarchitect.*, 2013, DOI: 10.1145/2540708.2540710.

[21] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman, “Monitoring and debugging the quality of results in approximate programs,” in *Proc. Int. Conf. Architect. Support Programm. Lang. Oper. Syst.*, 2015, pp. 399–411.

[22] W. Baek and T. M. Chilimbi, “Green: A framework for supporting energy-conscious programming using controlled approximation,” in *Proc. ACM SIGPLAN Conf. Programm. Lang. Design Implement.*, 2010, pp. 198–209.

[23] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, “Rumba: An online quality management system for approximate computing,” in *Proc. Int. Symp. Comput. Architect.*, 2015, pp. 554–566.

[24] J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. Van Gael, “Measure transformer semantics for bayesian machine learning,” in *Programming Languages and Systems*, vol. 6602, G. Barthe, Ed. Berlin, Germany: Springer-Verlag, 2011, pp. 77–96, ser. Lecture Notes in Computer Science.

[25] M. Carbin, S. Misailovic, and M. C. Rinard, “Verifying quantitative reliability for programs that execute on unreliable hardware,” in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Programm. Syst. Lang. Appl.*, 2013, pp. 33–52.

[26] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, “Proving programs robust,” in *Proc. 19th ACM SIGSOFT Symp./13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 102–112.

[27] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum, “Church: A language for generative models,” in *Proc. Uncertainty Artif. Intell.*, 2008, pp. 220–229.

[28] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, “Probabilistic programming,” in *Proc. Future Softw. Eng.*, 2014, pp. 167–181.

[29] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen, “The ins and outs of the probabilistic model checker MRMC,” *J. Performance Eval.*, vol. 68, pp. 90–104, Feb. 2011.

[30] D. Kozen, “Semantics of probabilistic programs,” *J. Comput. Syst. Sci.*, vol. 22, no. 3, pp. 328–350, 1981.

[31] M. Kwiatkowska, G. Norman, and D. Parker, “Probabilistic symbolic model checking with PRISM: A hybrid approach,” *Int. J. Softw. Tools Technol. Transfer*, vol. 6, no. 2, pp. 128–142, 2004.

[32] A. Legay and B. Delahaye, “Statistical model checking: An overview,” 2010. [Online]. Available: http://arxiv.org/abs/1005.1327

[33] T. Minka, J. M. Winn, J. P. Guiver, and D. A. Knowles, “Infer.NET 2.5,” Microsoft Research, Cambridge, MA, USA, 2012.

[34] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, “Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels,” in *Proc. ACM Int. Conf. Object Oriented Programm. Syst. Lang. Appl.*, 2014, pp. 309–328.

[35] J. Park, X. Zhang, K. Ni, H. Esmaeilzadeh, and M. Naik, “Expectation-oriented framework for automating approximate programming,” Georgia Inst. Technol., Atlanta, GA, USA, Tech. Rep. GT-CS-14-05, 2014.

[36] M. C. Rinard, “Probabilistic accuracy bounds for fault-tolerant computations that discard tasks,” in *Proc. 20th Annu. Int. Conf. Supercomput.*, 2006, pp. 324–334.

[37] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, “Paraprox: Pattern-based approximation for data parallel applications,” in *Proc. 19th Int. Conf. Architect. Support Programm. Lang. Oper. Syst.*, 2014, pp. 35–50.

[38] A. Sampson et al., “Expressing and verifying probabilistic assertions,” in *Proc. 35th ACM SIGPLAN Conf. Programm. Lang. Design Implement.*, 2014, pp. 112–122.

[39] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic optimization of floating-point programs with tunable precision,” in *Proc. 35th ACM SIGPLAN Conf. Programm. Lang. Design Implement.*, 2014, pp. 53–64.

[40] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic optimization of floating-point programs with tunable precision,” in *Proc. 35th ACM SIGPLAN Conf. Programm. Lang. Design Implement.*, 2014, pp. 53–64.

[41] J. Sorber et al., "Eon: A language and runtime system for perpetual systems," in *Proc. 5th Int. Conf. Embedded Netw. Sensor Syst.*, 2007, pp. 161–174.

[42] U. R. Karpuzcu, I. Akturk, and N. S. Kim, "Accordion: Toward soft near-threshold voltage computing," in *Proc. Int. Symp. High Performance Comput. Architect.*, 2014, pp. 72–83.

[43] S. Z. Gilani, N. S. Kim, and M. Schulte, "Scratchpad memory optimization for digital signal processing applications," in *Proc. IEEE/ACM Design Autom. Test Eur.*, 2011, DOI: 10.1109/DATE.2011.5763158.

[44] D. Palframan, N. S. Kim, and M. H. Lipasti, "Precision-aware soft error protection for GPUs," in *Proc. Int. Symp. High Performance Comput. Architect.*, 2014, pp. 49–59.

[45] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving refresh-power in mobile devices through critical data partitioning," in *Proc. Int. Conf. Architect. Support Programm. Lang. Oper. Syst.*, 2011, pp. 213–224.

[46] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *Proc. Int. Symp. Microarchitect.*, 2013, pp. 25–36.

[47] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads," in *Proc. Int. Conf. Parallel Architect. Compilat. Tech.*, 2008, pp. 325–334.

[48] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy, "Design of voltage-scalable meta-functions for approximate computing," in *Proc. IEEE/ACM Design Autom. Test Eur.*, 2011, pp. 950–955.

[49] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "IMPACT: Imprecise adders for low-power approximate computing," in *Proc. IEEE/ACM Int. Symp. Low-Power Electron. Design*, 2011, pp. 409–414.

[50] Y. Kim, Y. Zhang, and P. Li, "An energy efficient approximate adder with carry skip for error resilient neuromorphic VLSI systems," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 2013, pp. 130–137.

[51] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 2013, pp. 48–54.

[52] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *Proc. IEEE Int. Conf. VLSI Design*, 2011, pp. 346–351.

[53] C. Liu, J. Han, and F. Lombardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *Proc. IEEE/ACM Design Autom. Test Eur.*, 2015, Art. ID 95.

[54] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *Proc. IEEE/ACM Design Autom. Test Eur.*, 2010, pp. 957–960.

[55] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 2013, pp. 779–786.

[56] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: Systematic logic synthesis of approximate circuits," in *Proc. IEEE/ACM Design Autom. Conf.*, 2012, pp. 796–801.

[57] A. Ranjan et al., "ASLAN: Synthesis of approximate sequential circuits," in *Proc. IEEE/ACM Design Autom. Test Eur.*, 2014, DOI: 10.7873/DATE.2014.377.

[58] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "ABACUS: A technique for automated behavioral synthesis of approximate computing circuits," in *Proc. IEEE/ACM Design Autom. Test Eur.*, 2014, Art. ID 361.

[59] C. Li,W. Luo, S. S. Sapatnekar, and J. Hu, "Joint precision optimization and high level synthesis for approximate computing," in *Proc. IEEE/ACM Design Autom. Conf.*, 2015, Art. ID 104.

Qiang Xu is an Associate Professor in Computer Engineering at the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong. His current research interests include fault-tolerant computing and trusted computing. His research has received the IEEE/ACM Design Automation and Test in Europe (DATE) Best Paper Award in 2004. He is a Member of the IEEE.

Todd Mytkowicz is a Researcher at Microsoft Research, Redmond, WA, USA, working at the intersection of programming languages and systems, with a specific focus on abstractions for parallelism and performance. His research interests span program analysis, probabilistic programming, verification, and concurrency. His research received three SIGPLAN research highlights nominations, was chosen as an IEEE Micro Top Pick, and is used in production systems at Microsoft.

Nam Sung Kim is an Associate Professor at the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, USA. His current research interests span circuit, architecture, and system for energy-efficient computing. His research received the IEEE/ACM International Symposium on Microarchitecture (MICRO) Best Paper Award and was chosen as an IEEE Micro Top Pick. He is a Fellow of the IEEE.

■ Direct questions and comments about this article to Qiang Xu, Computer Science and Engineering Dept., The Chinese University of Hong Kong, Shatin, Hong Kong; qxu@cse.cuhk.edu.hk.