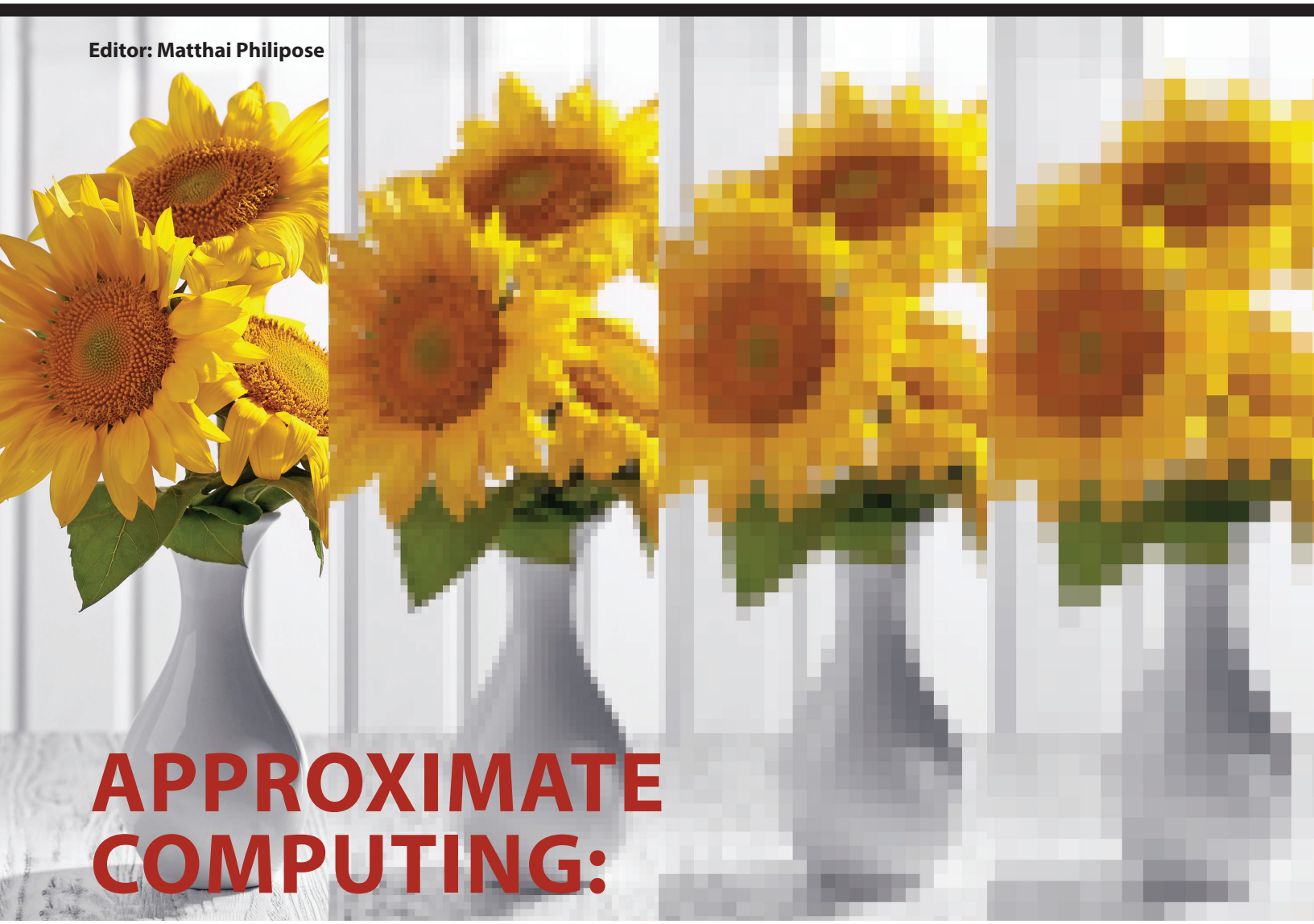
Luis Ceze *University of Washington, Seattle, WA*
Adrian Sampson *Cornell University, Ithaca, NY*

Editor: Matthai Philipose



APPROXIMATE COMPUTING:

Unlocking Efficiency with Hardware–Software Co-Design

Generations of computer scientists and practitioners have worked under the assumption that computers will keep improving themselves: just wait a few years and Moore's Law will solve your scaling problems. This reliable march of electrical-engineering progress has sparked revolutions in the ways humans use computers and interact with the world and each other. But growth in computing power has protected outdated abstractions and encouraged layering even more abstractions, whatever the cost.

Photo, bigstockphoto.com

The free lunch seems to be over: single-thread performance has stagnated, Dennard scaling has broken down, and Moore's Law threatens to do the same. The shift to multi-core designs worked as a stopgap in the final years of frequency advancements, but physical limits have dashed hopes of long-term exponential gains through parallelism.

The next phase of computing advances will need to leave old abstraction boundaries behind. Hardware designs will need to react to application needs, and software will need to cope with the limitations of hardware.

The general idea of accuracy trade-offs has been around for a long time: for example, floating point numbers and lossy image compression are ubiquitous. Over the past six years, the architecture, programming languages and systems communities have actively explored approximate computing, a new direction in hardware–software co-design research that exploits the same kinds of trade-offs in the way we design systems. The idea is that current abstractions in computer systems fail to incorporate an important dimension of the application design space: not every application needs the same degree of accuracy all the time. These applications span a wide range of domains including big-data analytics, web search, machine learning, cyber-physical systems, speech and pattern recognition, augmented reality, and many more. These kinds of programs can tolerate unreliable and inaccurate computation, and approximate computing research shows how to exploit this tolerance for gains in performance and efficiency [1], [5], [6], [7], [11], [12], [14], [16], [17].

Approximate computing is a classic cross-cutting concern: its full potential is not reachable through software or hardware alone, but only through changing the abstractions and contracts between hardware and software. Advances in approximation require co-design between architectures that expose accuracy – efficiency trade-offs and the programming systems that make those trade-offs useful for programmers. We have explored projects across the entire system stack – from programming languages and tools down through the hardware – that enable computer systems to trade off accuracy of computation, communication, and storage for gains in efficiency and performance.

This article highlights two main challenges in approximate computing: we need hardware technologies that can smoothly trade off energy for accuracy, and we need ways to safely program these approximate computers. We then enumerate some remaining open challenges in approximation.

EXPLOITING APPROXIMATION FOR EFFICIENCY

The first ingredient in approximate computing is a system that can trade off accuracy for efficiency in some resource. The efficiency can come in many flavors: performance, energy, storage density, network capacity, or any other constraint. We categorize some recent advantages into approximation for computation and for storage.

Computation

Approximation research has explored several approximate execution techniques with hardware support, among them: compiler-controlled voltage overscaling [8] and fine-grain control of bit-width via power gating [24]. The techniques with the largest efficiency gains work by replacing entire sections of precise code with approximate alternatives, such as hardware neural network accelerators [9] or even analog components [23]. Analog neural acceleration can make approximate applications use 6.3× less energy [23].

Storage

A lot of hardware resources are devoted to storage, from DRAM to flash to spinning media. Approximation can also enable more effective use of storage resources. Some notable examples are Flikker [12], which reduces DRAM refresh rates to reduce energy consumption at the cost of potential data corruption of approximate date; approximate storage [21] in solid state memories to increase storage density and write performance by reusing worn blocks or more densely exploiting the available dynamic range; and extending the microarchitecture's memory system with imprecise modes [13]. Approximating memory writes can even extend the usable lifetime of memories that wear out by 23% [21].

A Case Study: Neural Acceleration with On-Chip FPGAs

A powerful approach to approximate computing, neural acceleration, works by substituting entire regions of code in a program with machine learning models [9]. Neural acceleration trains neural networks to mimic and replace regions of approximate imperative code. Once the neural network is trained, the system no longer executes the original code and instead invokes the neural network model on a neural processing unit (NPU) accelerator. Neural networks have efficient hardware implementations, so this workflow can offer significant energy savings over traditional execution.

Neural acceleration consists of three phases: programming, compilation, and execution.

Programming: To use neural acceleration in ACCEPT, the programmer uses profiling information and type annotations to mark code that is amenable to approximation. For many applications, it is easy to identify the "core" approximate data, such as the pixel array in an image filter algorithm, that dominates the program's execution. The programmer also provides a quality metric that measures the accuracy of the program's overall output.

Compilation: The compiler implements neural acceleration in four phases: region selection, execution observation, training, and code generation. ACCEPT first identifies large regions of code that are safe to approximate and nominates them as candidates for neural acceleration. Next, it executes the program with test cases and

records the inputs and outputs to each target code region. It then uses this input–output data to train a neural network that mimics the original code. Training can use standard techniques for neural networks: we use the standard *backpropagation* algorithm. Finally, the compiler generates an executable that replaces the original code with invocations of a special accelerator, the neural processing unit (NPU), that implements the trained neural network.

Execution: During deployment, the transformed program begins execution on the main core and configures the NPU. Throughout execution, the program invokes the NPU to perform a neural network evaluation in lieu of executing the code region it replaced. Invoking the NPU is faster and more energy-efficient than executing the original code region on the CPU, so the program as a whole runs faster.

Our NPU implementation, SNNAP, runs on off-the-shelf *field-programmable gate arrays* (FPGAs). Using existing, affordable hardware means that SNNAP can provide benefit today, without waiting for new silicon. SNNAP uses an emerging class of heterogeneous computing devices called Programmable System-on-Chips (PSoCs). These devices combine a set of hard processor cores with programmable logic on the same die. Compared to conventional FPGAs, this integration provides a higher-bandwidth and lower-latency interface between the main CPU and the programmable logic.

Implementation on the Zynq: We have implemented SNNAP on a commercially available PSoC: the Xilinx Zynq-7020 on the ZC702 evaluation platform [25]. The Zynq includes a Dual Core ARM Cortex-A9 and an FPGA fabric. The CPU–NPU interface composes three communication mechanisms on the Zynq PSoC [26] for high bandwidth and low latency. First, when the program starts, it configures SNNAP using the medium-throughput General Purpose I/Os (GPIOs) interface. Then, to use SNNAP during execution, the program sends inputs using the high-throughput ARM Accelerator Coherency Port (ACP). The processor then uses the ARMv7 SEV/WFE signaling instructions to invoke SNNAP and enter sleep mode. Finally, the accelerator writes outputs back to the processor's cache via the ACP interface and, when finished, signals the processor to wake up.

Microarchitecture: Our design, shown in Figure 1, consists of a cluster of *Processing Units* (PUs) connected through a bus. Each PU is composed of a control block, a chain of *Processing Elements* (PEs), and a sigmoid unit, denoted by the SIG block. The PEs form a one-dimensional systolic array that feeds into the sigmoid unit. Systolic arrays excel at exploiting the regular data-parallelism found in neural networks, and are amenable to efficient implementation on modern FPGAs. When evaluating a layer of a neural network, PEs read the neuron weights from a local scratchpad memory where temporary results can also be stored. The sigmoid unit implements a nonlinear neuron-activation function using a lookup table. The PU control block contains a configurable sequencer that orchestrates communication between the PEs and the sigmoid unit. The PUs can be programmed to operate independently, so different PUs can either be used to parallelize the invocations of a single neural network or used to evaluate different neural networks concurrently.

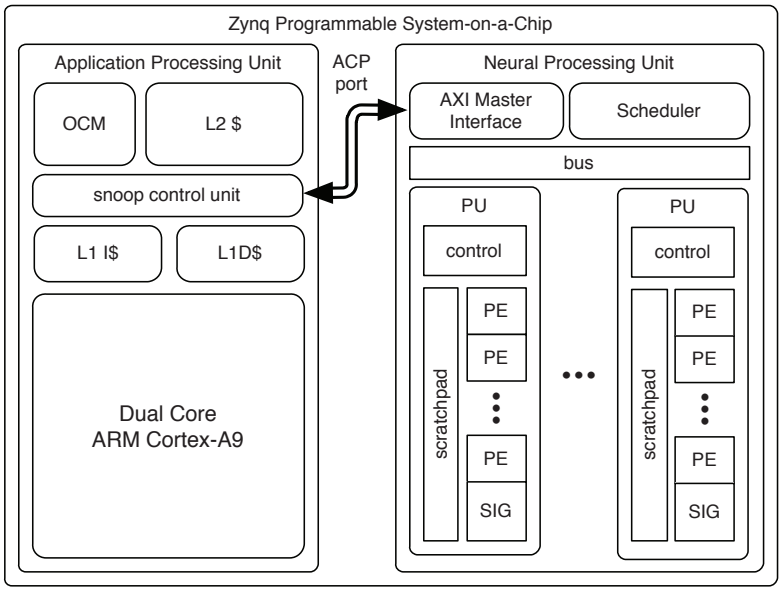


FIGURE 1. SNNAP system diagram: Each Processing Unit (PU) contains a chain of Processing Elements (PE) feeding into a sigmoid unit (SIG).

Results: Across seven approximate benchmarks, SNNAP offers an average speedup of 3.8× within acceptable quality bounds [15]. The approach can outperform traditional high-level synthesis tools that produce precise FPGA configurations.

SAFETY AND QUALITY OF RESULTS

It is not enough to build approximate hardware components: using approximation has deep implications for the way we write software. The second main thrust in approximate-computing research is on programming languages, compilers, and tools that make approximate programs safe accurate.

First, programming models need to enforce *safety*: approximate components must not compromise safe execution (e.g., no uncontrolled jumps or critical data corruption) and must interact with precise components only in well-defined ways allowed by the programmer. Our work met this need with language support in the form of type qualifiers for approximate data and type-based static information-flow tracking [19]. Other work from MIT consists of a proof system for deriving safety guarantees in the face of unreliable components [3]. These crucial safety guarantees allow systems to prove at compile time that approximation cannot introduce catastrophic failures into otherwise good programs.

THE NEXT PHASE OF COMPUTING ADVANCES WILL NEED TO LEAVE OLD ABSTRACTION BOUNDARIES BEHIND. HARDWARE DESIGNS WILL NEED TO REACT TO APPLICATION NEEDS, AND SOFTWARE WILL NEED TO COPE WITH THE LIMITATIONS OF HARDWARE

Beyond safety, another key requirement is ways to specify and ensure acceptable *quality of results* (QoR). Languages must enable programmers to declare the magnitude of acceptable approximation in multiple forms and granularities. For example, QoR can be set for a specific value (X should be at most Y% from its value in a fully precise execution), or one could attach QoR to a set of values (at most N values in a set can be in error). One can provide a QoR specification only for the final output of a program or for intermediate values. QoR specifications can then guide the compiler and runtime to choose and control the optimal approximate execution engine from a variety of software and hardware approximation mechanisms. While quality constraints are more general and therefore more difficult to enforce statically than safety requirements, initial tactics have seen success by limiting the kinds of approximation they can work with [22], [4] or by relying on dynamic checks [18], [10].

A Case Study: An Approximate Compiler

ACCEPT is compiler framework for approximate computing. It combines programmer annotations, code analysis, optimizations, and profiling feedback to make approximation safe and keep control in the hands of programmers.

ACCEPT's frontend, built atop the LLVM compiler infrastructure, extends the syntax of C and C++ to incorporate an APPROX keyword that programmers use to annotate data types. ACCEPT's analysis identifies code that can affect only variables marked as APPROX. Optimizations use these analysis results to avoid transforming the precise parts of the program. An autotuning component measures program executions and uses heuristics to identify program variants that maximize performance and output quality. The final output is a set of Pareto-optimal versions of the input program that reflect its efficiency–quality trade-off space.

Safety constraints and feedback: Because program relaxations can have outsize effects on program behavior, programmers need *visibility* into — and *control* over — the transformations the compiler applies. To give the programmer fine-grained control over relaxations, ACCEPT extends an existing lightweight annotation system for approximate computing based on type qualifiers [19]. ACCEPT gives programmers visibility into the relaxation process via feedback that identifies which transfor- mations can be applied and which annotations are constraining it. Through annotation and feedback, the programmer iterates toward an annotation set that unlocks new performance benefits while relying on an assurance that critical computations are unaffected.

Automatic program transformations: Based on programmer annotations, ACCEPT's compiler passes apply program transformations that involve only approximate data. To this end, ACCEPT provides a compiler analysis library that finds regions of code that are amenable to transformations. An ensemble of optimization strategies transform these regions. One critical optimization targets SNNAP, our neural accelerator, which we describe in more detail below.

Autotuning: While a set of annotations may permit many different safe program relaxations, not all of them are beneficial in the quality–performance trade-off they offer. A practical approximation mechanism must help programmers choose from among many candidate relaxations for a given program to strike an optimal balance between performance and quality. ACCEPT's autotuner heuristically explores the space of possible relaxed programs to identify Pareto-optimal variants.

Results: ACCEPT is an open-source project available at https://sampa.cs.washington.edu/accept/ By combining multiple approximation strategies, ACCEPT can speed up applications by 2.3× on commodity x86 hardware and by 4.8× on an off-the-shelf FPGA–CPU hybrid system [20].

NEXT STEPS IN APPROXIMATION

Controlling quality

The community has allocated more attention to assuring safety of approximate programs than to controlling quality. Decoupling safety from quality has been crucial to enabling progress on that half of the equation [19], [3] but more nuanced quality properties have proven more challenging. We have initial ways to prove and reason about limited probabilistic quality properties [4], [2], [22], but we still lack techniques that can cope with arbitrary approximation strategies and still produce useful guarantees.

We also need ways to measure quality at run time. If approximate programs could measure how accurate they are without too much overhead, they could offer better guarantees to programmers while simultaneously exploiting more aggressive optimizations [10], [18]. But there is not yet a general way to derive a cheap, dynamic quality check for an arbitrary program and arbitrary quality criterion. Even limited solutions to the dynamic-check problem will amplify the benefits of approximation.

Defining quality

Any application of approximate computing rests on a *quality metric*. Even evaluations for papers on approximation need to measure their effectiveness with some accuracy criterion. Unlike traditional criteria — energy or performance, for example — the right metric for quality is not obvious. It varies per program, per deployment, and even per user. The community does not have a satisfactory way to decide on the right metric for a given scenario: we are so far stuck with guesses.

A next step in approximation research should help build confidence that we are using the right quality metrics. We should adopt techniques from software engineering, human-computer interaction, and application domains like graphics to help gather evidence for good quality metrics. Ultimately, programmers need a sound methodology for designing and evaluating quality metrics for new scenarios.

The right accelerator

Hardware approximation research has fallen into two categories: extensions to traditional architectures [8], [13] and new, discrete accelerators [24], [9]. The former category has yielded simpler programming models, but the fine-grained nature of the model means that efficiency gains have been limited. Coarser-grained, accelerator-oriented approaches have yielded the best results to date. There are still opportunities for co-designing accelerators with programming models that capture the best of both approaches. The next generation of approximate hardware research should co-design an accelerator design with a software interface and compiler workflow that together attack the programmability challenges in approximation: safety and quality. By decoupling approximation from traditional processors, new accelerators could unlock new levels of efficiency while finally making approximate computing palatable hardware vendors.

Tools

A final key component for making approximate programming practical is software-development tools. We need tools to help programmers identify approximation opportunities, understand the effect of approximation at the application level, assist with specifying QoR requirements, and help test and debug applications with approximate components. Our first steps in this direction are a debugger and a post-deployment monitoring framework for approximate programs [18].

CONCLUSION

Many applications can tolerate imprecision, and many systems can be far more efficient when we let them operate imprecisely. Approximate computing is a new hardware – software research direction that exploits these trends. ACCEPT and SNNAP are two recent, open-source efforts to bring early-stage approximation research into the mainstream. Together, they can offer significant speedups on approximate applications while guaranteeing safety and enforcing high-quality output. ■

Luis Ceze is the Torode Family Associate Professor in the Computer Science and Engineering Department at the University of Washington. His research focuses on the intersection between computer architecture, programming languages and biology. He believes nature is a fantastic source of inspiration and parts for future computer systems.

Adrian Sampson is an assistant professor at Cornell. His research is on building better abstractions for computer systems, especially at the boundary between hardware and software. He believes approximate computing is an inevitable consequence of concurrent trends in technology and applications.

REFERENCES

[1] A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann, "Using code perforation to improve performance, reduce energy consumption, and respond to failures," MIT, Tech. Rep., 2009.

[2] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain<T>: A First-Order Type for Uncertain Data," in *ASPLOS*, 2014.

[3] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *PLDI*, 2012.

[4] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *OOPSLA*, 2013.

[5] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee, "Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology," in *DATE*, 2006. [Online]. Available at http://portal.acm.org/citation.cfm?id=1131790

[6] M. de Kruijf and K. Sankaralingam, "Exploring the synergy of emerging workloads and silicon reliability trends," in SELSE, 2009.

[7] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: an architectural framework for software recovery of hardware faults," in ISCA, 2010.

[8] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[9] ——, "Neural Acceleration for General-Purpose Approximate Programs," in *International Symposium on Microarchitecture (MICRO)*, 12 2012.

[10] B. Grigorian and G. Reinman, "Improving coverage and reliability in approximate computing using application-specific, light-weight checks," in *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[11] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: error resilient system architecture for probabilistic applications," in DATE, 2010.

[12] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving refresh-power in mobile devices through critical data partitioning," in *ASPLOS*, 2011.

[13] J. S. Miguel, M. Badr, and N. E. Jerger, "Load value approximation," in *MICRO*, 2014.

[14] S. Misailovic, S. Sidiroglou, H. Hoffman, and M. Rinard, "Quality of service profiling," in *ICSE*, 2010.

[15] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "SNNAP: Approximate computing on programmable SoCs via neural acceleration," in *HPCA*, 2015.

[16] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," in *DATE*, 2010. [Online]. Available at http://portal.acm.org/citation.cfm?id=1871008

[17] M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou, "Patterns and statistical analysis for understanding reduced resource computing," in *Onward!*, 2010.

[18] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman, "Monitoring and debugging the quality of results in approximate programs," in *ASPLOS*, 2015.

[19] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *PLDI*, 2011.

[20] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "ACCEPT: A programmer-guided compiler framework for practical approximate computing," U. Washington, Tech. Rep. UW-CSE- 15-01-01, 2015.

[21] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in MICRO, 2013.

[22] A. Sampson, P. Panchekha, T. Mytkowicz, K. McKinley, D. Grossman, and L. Ceze, "Expressing and Verifying Probabilistic Assertions," in PLDI, 2014.

[23] R. St. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," in *ISCA*, 2014.

[24] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghu- nathan, "Quality programmable vector processors for approximate computing," in *MICRO*, 2013.

[25] Xilinx, Inc., "All programmable SoC." [Online]. Available: http://www.xilinx.com/products/silicon- devices/soc/

[26] ——, "Zynq UG585 technical reference manual." [Online]. Available at http://www.xilinx.com/support/documentation/user guides/