My research focuses on the layered abstractions that make up computer systems, from programming languages to computer architecture. My dissertation work was on *approximate computing*, the idea that computer systems waste time and energy providing perfect correctness to many applications that do not require it. Approximate systems can be far more efficient, but they need cooperation from the entire computational stack to make inexact execution safe. I developed new abstractions that incorporate unreliability and imprecision and explored their implications through research on architecture, compilers, programming languages, and programmer tools. I plan to apply this interdisciplinary approach to new challenges that cross the boundaries between the layers of the computing stack.

Approximate Computing

My dissertation work developed some of the first architecture and programming language support for disciplined approximate computing. Approximate computing gives programmers principled ways to control efficiency–accuracy trade-offs. Some of today's most important application domains—climate simulation, augmented reality, and data mining, for example—use "soft" computations that do not require the same accuracy as a drug delivery system or an aerospace controller. This fundamental disconnect costs us performance, energy efficiency, and complexity: if systems can safely expose occasional bit flips or rounding errors, they can recover the efficiency cost of indiscriminate accuracy.

When I published my first paper on the topic in PLDI 2011, approximate computing was an unfamiliar term. Since then, the idea has gained momentum. An "approximate computing" session has begun to appear on the programs of major programming languages and architecture conferences, and 2014 marked the first two workshops on the topic at ASPLOS and PLDI. The *IEEE Spectrum* magazine invited me to write a general-audience feature article on approximate computing [18]. Industry has also taken notice: projects from Intel and Qualcomm have embraced the idea of soft, efficient computation [8, 11], and I have held three industry fellowships, from Facebook, Qualcomm, and Google, to pursue approximation research.

Approximate programming languages. I have developed programming languages that help developers write reliable software for unreliable, approximate machines. The EnerJ language [19] uses a system of type qualifiers to let programmers dictate where approximation is allowed—a neural network's weights, for example, or pixel data in a vision application. EnerJ comes with a proof of *non-interference* that guarantees at compile time that errors from approximate execution cannot corrupt critical state. EnerJ's dichotomy between critical and approximate data has influenced both language research [2,11] and hardware projects [10,16], and other groups have reused the simulation infrastructure I built for the project [3,10].

More recently, I mentored an undergraduate researcher to extend EnerJ with probabilistic control over computation quality. The new extension lets programmers constrain the required accuracy at important points in a program. The compiler then uses a Satisfiability Modulo Theories (SMT) solver to infer how accurate the bulk of the program must be to meet the programmer's demands. This programming model gives programmers strong statistical guarantees with minimal development effort.

Approximate architectures. To implement approximate computing, I have developed a range of architectural techniques that expose accuracy–efficiency trade-offs. I worked on a processor architecture that can switch between a high voltage and a low voltage for individual instructions to support software that needs tightly interleaved approximation [4] and an accelerator based on neural networks that exploits function-level approximation to reduce energy consumption threefold [5,12].

I have also explored approximation beyond the CPU in my work on *approximate storage* [20]. That work embraces the reliability quirks of emerging memory technologies such as phase-change memory (PCM) that otherwise threaten their viability. One technique exploits the analog properties of crystallization in PCM's underlying material to issue faster writes for approximate data; and a second adapts memories' existing error-correcting codes to maximize accuracy when complete correction is impossible. Both techniques use a new OS–hardware interface to set the reliability level for each block of data.

Compiler-driven approximation. Recently, I have focused on building infrastructure to support practical experimentation with approximate computing. I developed an end-to-end compiler toolchain that can approximate C and C++ programs on traditional CPUs and with an FPGA accelerator. The open-source project, called ACCEPT [1], is both a research tool and a workflow for practitioners to explore the potential of approximate computing. I worked with a team of undergraduates to make the framework accessible even to novice programmers.

Programmer tools. Programmers need debugging tools—the analogs of traditional testing and verification—for the approximate context. I worked on a set of tools for debugging quality problems in approximate programs and monitoring accuracy degradation at run time [15]. Recently, I have mentored an undergraduate researcher to explore crowdsourcing as a tool to gain better insight into computation quality when an application's notion of "accuracy" is subjective.

With collaborators from Microsoft Research, I proposed *probabilistic assertions*, a new language construct that lets programmers express correctness statistically [21]. Probabilistic assertions encode the likelihood that a condition must hold in a program: for example, in an image renderer, that each approximated output pixel should be within 10% of its precise equivalent at least 99% of the time. My probabilistic-assertion verifier uses symbolic execution to extract a Bayesian network from the program, where statistical techniques make it efficient to check the assertion.

Aside from approximate computing, I have worked on languages and architectures for safe parallel programming [6,22] and a machine-learning tool that automatically discovers energy and performance pitfalls in mobile Web browsers [17].

Next Steps

My work on approximate computing has underscored the potential of "full-stack" computer systems research. By rethinking the abstractions at the intersection of architecture, languages, and compilers, researchers can create new avenues for research and industrial development. I plan to continue this kind of interdisciplinary systems research through work on unsolved problems in approximate computing, new abstractions for statistical system behavior, and combating the abstraction tax in hardware and software.

Open challenges in approximation. In the near term, I plan to build on my work in approximate computing to address the area's next set of major challenges.

First, I plan to work on a technique for automatically synthesizing code to limit the quality impact of approximate computing. Checking programs off-line, as in my own work [21], is necessary but not sufficient: programmers need strong assurances that output degradation cannot run wild in deployment. To avoid requiring programmers to manually write on-line integrity checks for each approximate computation, I plan to build a tool that automatically derives checking code. The tool will synthesize probability distributions to reflect inputs that are likely to result in good accuracy, and then, at run time, check how well real behavior is matching the validated distributions. The end result is an efficient dynamic checking scheme that can offer programmers strong statistical guarantees for arbitrary code and approximation strategies.

I will also develop new hardware accelerators to bring us closer to the orders-of-magnitude efficiency potential that approximate computing promises. My previous work has suggested that the potential for approximation in traditional von Neumann machines is limited [4]; the greatest gains will come from specialized approximate accelerators [5]. I plan to work on a new configurable accelerator design tailored as a compilation target for approximate code. The design will extend spatial architectures [14] with accuracy–efficiency trade-offs that exploit the design, including probabilistic timing speculation, lossy data compression, bit width reduction, and sub-threshold SRAM cells. A co-designed compiler workflow will extend constraint-based scheduling approaches [13,14] with relaxed correctness. The low-level control that spatial architectures offer makes them a perfect match for approximate computing: control overhead is minimal, and an accompanying compiler can search for optimal mappings onto approximate hardware resources.

Coping with statistical behavior and performance. Beyond approximate computing, I see many opportunities to use my experience with statistical program behavior to address other ways that programmers contend with probabilities. For example, cloud services and distributed systems often have unpredictable performance, leading to excessive tail latency [9]. Traditional debuggers and diagnosis tools are poorly suited to this kind of statistical property: they work well when a bug causes something that should never happen, but not when something happens *more often* than it should. It may even be impossible to require that latency *never* exceeds n seconds, or that a model should never misclassify an image.

I plan to build on my work on probabilistic assertions [21] to design language abstractions and compiler techniques that help programmers control the latent probabilities in their programs. To help control unpredictable performance, I will build a language abstraction that explicitly represents potential nondeterminism: for example, writing into an OS buffer that might be full. A tool can then analyze the way that these individual pitfalls compose to impact a program's overall performance profile—and suggest tuning strategies to reign in tail latency. I will also extend these techniques to other statistical systems:

privacy-preserving services with obfuscation and machine-learning applications. This research direction will require language features co-designed with compiler techniques to let ordinary programmers reason about and control statistical program behavior.

Bridging semantic gaps. My long-term research agenda will focus on bridging the semantic gaps between layers of the system stack. Information is "lost in translation" from the programmer's intention to the programming language, to executable code, and to a processor's internal state. These layered abstractions make programming tractable, but each semantic gap incurs costs.

I plan to design a hybrid compiler–architecture system that can shift the cost of execution safety between run time and compile time. Traditional microarchitectural protections—precise exceptions, virtual memory protection, or even pipeline hazards—are critical when running untrusted code, but they are redundant when a compiler can certify safety ahead of time. Moreover, traditional hardware safety mechanisms do not suffice to enforce important higher-level security properties, such as a Web browser's same-origin policy. Memory-safe languages and managed runtimes already offer guarantees that today's hardware ignores [7], and new language constructs could help convey richer safety properties to the hardware. I will evaluate an architecture that exposes a spectrum of flexible safety checks along with a co-designed compiler and language. Together, the system will avoid run-time costs for properties that they can prove at compile time and offload expensive software policy enforcement to hardware.

I also want to build abstractions that give programmers control over energy efficiency. Energy is often as important as performance, but current systems offer limited insight into energy costs: energy monitors are coarse-grained, and direct energy controls are typically nonexistent. I plan to explore hardware-enforced *energy protection*, which will let the OS and applications express energy limits for individual microarchitectural components. A language interface will let software handle energy violations to kill runaway tasks or even enable approximation to keep power low. By involving the programmer, an energy protection abstraction will let systems harness higher-level domain knowledge to optimize for energy efficiency.

Through this research direction, I hope to blur the line between hardware and software design. In an era where efficiency benefits from hardware alone are faltering, programmers should control the entire system to tailor it for an application. I plan to apply the same style of research I used for approximate computing to these inherently interdisciplinary challenges: I will use research at every layer in the computing stack to build new fundamental abstractions.

References

[1] ACCEPT, an approximate compiler: Documentation. `https://sampa.cs.washington.edu/accept/`.

[2] Michael Carbin, Sasa Misailovic, and Martin Rinard. Verifying quantitative reliability of programs that execute on unreliable hardware. In *OOPSLA*, 2013.

[3] Hadi Esmaeilzadeh, Kangqi Ni, and Mayur Naik. Expectation-oriented framework for automating approximate programming. Technical Report GT-CS-13-07, Georgia Institute of Technology, 2013.

[4] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, 2012.

[5] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.

[6] Emily Fortuna, Brandon Lucia, Adrian Sampson, Benjamin Wood, and Luis Ceze. Greedy coherence. In *Workshop on Hardware Support for Parallel Program Correctness*, 2011.

[7] Galen Hunt and James Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2), April 2007.

[8] Samir Kumar. Introducing Qualcomm Zeroth processors: Brain-inspired computing. `https://www.qualcomm.com/news/onq/2013/10/10/introducing-qualcomm-zeroth-processors-brain-inspired-computing`.

[9] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *SoCC*, 2014.

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

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

[12] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *HPCA*, 2015.

[13] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *PLDI*, 2013.

[14] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *PLDI*, 2014.

[15] Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. Debugging approximate programs via dynamic analysis. In *ASPLOS*, 2015.

[16] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. SAGE: Self-tuning approximation for graphics engines. In *MICRO*, 2013.

[17] Adrian Sampson, Calin Cascaval, Luis Ceze, Pablo Montesinos, and Dario Suarez Gracia. Automatic discovery of performance and energy pitfalls in HTML and CSS. In *IISWC*, 2012.

[18] Adrian Sampson, Luis Ceze, and Dan Grossman. EnerJ, the language of good-enough computing. *IEEE Spectrum*, 10 2013.

[19] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[20] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. In *MICRO*, 2013.

[21] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. In *PLDI*, 2014.

[22] Benjamin Wood, Adrian Sampson, Luis Ceze, and Dan Grossman. Composable specifications for structured shared-memory communication. In *OOPSLA*, 2010.