Hardware–Software Co-Design: Not Just a Cliché

Adrian Sampson, James Bornholt, and Luis Ceze

University of Washington
`{asampson, bornholt, luisceze}@cs.washington.edu`

Abstract

The age of the air-tight hardware abstraction is over. As the computing ecosystem moves beyond the predictable yearly advances of Moore's Law, appeals to familiarity and backwards compatibility will become less convincing: fundamental shifts in abstraction and design will look more enticing. It is time to embrace *hardware–software co-design* in earnest, to cooperate between programming languages and architecture to upend legacy constraints on computing. We describe our work on *approximate computing*, a new avenue spanning the system stack from applications and languages to microarchitectures. We reflect on the challenges and successes of approximation research and, with these lessons in mind, distill opportunities for future hardware–software co-design efforts.

1998 ACM Subject Classification C.5 Computer System Implementation

Keywords and phrases approximation, co-design, architecture, verification

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

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 But growth in computing power has protected outdated abstractions and encouraged layering even more abstractions, whatever the cost.

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.

Hardware–software co-design presents significant performance and efficiency opportunities that are unavailable without crossing the abstraction gap. For example, embedded systems depended on co-design from their inception. The embedded domain uses software for flexibility while specialized hardware delivers performance. General-purpose computing features many hardware extensions employed to better serve software—virtual memory, ISA extensions for security, and so on. While these mechanisms have been successful, they are ad hoc responses to trends in software design.

Hybrid hardware–software research cannot just be a cliché: now more than ever, true cooperation is crucial to improving the performance and efficiency of future computing systems. Over the past five years, our research group has been exploring *approximate computing*, a classic example of a hardware–software co-design problem. Approximate computing trades accuracy for performance or efficiency, exploiting the fact that many applications are robust to some level of imprecision. Our experience has shown that neither software nor hardware alone can unlock the full potential of approximation; optimal solutions require co-design between programming models and hardware accelerators or ISA extensions. We discuss our experience with approximate computing and highlight lessons for future co-design efforts.

We believe we can go much further by truly designing hardware and software in unison. We survey the opportunities for hardware software co-design in four broad categories. The first is effective hardware acceleration, where the best algorithm for general-purpose hardware and the best custom hardware for generic algorithms both fall short. For example, the Anton [12] special machine for molecular dynamics, co-designed a simulation algorithm with the hardware. The second category is reducing redundancy and reassigning responsibilities. For example, if all programs were guaranteed to be free of memory bugs, can we drop support for memory protection? Or if programs were free of data-races by construction, can we rethink cache coherence? The third category is hardware support for domain-specific languages, whose booming popularity democratizes programming by providing higher-level semantics and tooling, but today must still compile to low-level general-purpose hardware. The final category is research beyond the CPU: unexplored opportunities abound for hardware–software cooperation in networks, memory, storage, and less glamorous parts of computer hardware such as power supplies.

2 Approximate Computing

For the last five years, our group has explored one instance of a hardware–software co-design opportunity: *approximate computing*. 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, 7, 10, 11, 25, 26, 28, 30, 35].

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.

Our research direction spans languages [38], runtime systems, programmer tools including debuggers [36], compilers [41], synthesis tools for both software and hardware, microarchitecture [13], hardware techniques [43, 29], data stores [40], and communication services.

Safety and Quality-of-Results

A core requirement in writing programs with approximate components is *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 [38]. Other work from MIT consists of a proof system for deriving safety guarantees in the face of unreliable components [5]. These crucial safety guarantees allow systems to prove at compile time that approximation cannot introduce catastrophic failures into otherwise-good programs.

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 [41, 6] or by relying on dynamic checks [36, 17].

Approximation Techniques

The purpose of allowing approximation is to trade accuracy for energy savings. At the highest level, there are three categories of approximation techniques: algorithmic, compiler/runtime, and hardware. Algorithmic approximation can be achieved by the programmer providing multiple implementations of a given computation and the compiler/runtime choosing among them based on QoR needs. A compiler can generate code for approximate execution by eliding some computations [28] or reducing value precision whenever allowed by the approximation specification. Approximation research has explored several approximate execution techniques with hardware support, among them: compiler-controlled voltage overscaling [13]; using learning-based techniques such as neural networks or synthesis to approximate kernels of imperative programs in a coarse-grain way [14]; adopting analog components for computation [43]; designing efficient storage systems that can lose bits [40]; and extending architecture mechanisms with imprecise modes [27].

While approximation only at the algorithm level together with compiler/runtime support applies to off-the-shelf hardware (and we intend to further explore that space too), our experience has shown that the greatest energy benefit comes from hardware-supported approximation with language/architecture co-design [29].

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 [36].

2.1 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 [38, 5] but more nuanced quality properties have proven more challenging. We have initial ways to prove and reason about limited probabilistic quality properties [6, 4, 41], 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 [17, 36]. 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 [13, 27] and new, discrete accelerators [47, 14]. 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.

2.2 Lessons from Approximation

Our group's experience with approximate computing as a cross-cutting concern has had both successes and failures. The path through this research has yielded lessons both for approximation research specifically and hardware–software co-design generally.

The von Neumann Curse

When doing approximation at the instruction granularity in a typical von Neumann machine, the data path can be approximate but the control circuitry likely can't. Given that control accounts for about 50% of hardware resources, gains are fundamentally limited to $2\times$, which is hardly enough to justify the trouble. We therefore have more hopes for coarse-grain approximation techniques than fine-grain.

And that was just an example. Many other promising avenues of our work have fallen afoul of the conflation of program control flow and data flow. For example, if we want to approximate the contents of a register, we need to know whether it represents pixel data amenable to approximation or a pointer, which may be disastrous to approximate.

Separation problems are not unique to approximation. Secure systems, for example, could profit from a guarantee that code pointers are never manipulated as raw data. Future explorations of hardware–software co-design would benefit from architectural support for separating control flow from data flow.

The High Cost of Hardware Design

In our work on approximation with neural networks, we achieve the best energy efficiency when we use a custom hardware neural processing unit, or NPU [14, 43]. Our first evaluations of the work used cycle-based architectural simulation, which predicted an average 3× energy savings. Later, we implemented the NPU on an FPGA [29]. On this hardware, we measured an average energy savings of 1.7×. The difference is due partially to the FPGA's overhead and clock speed and partially due to the disconnect between simulation and reality. Determining the exact balance would require a costly ASIC design process. Even building the FPGA implementation took almost two years.

Hardware–software co-design involves an implied imbalance: software is much faster to iterate on than hardware. Future explorations of hardware–software co-design opportunities would benefit from more evolution of hardware description languages and simulation tools. We should not have to implement hardware three times—first in simulation, second in an HDL for an efficient FPGA implementation, and again for a high-performance ASIC.

Trust the Compiler

Hybrid hardware–software research constantly needs to divide work between the programmer, the compiler, and the hardware. In our experience, a hybrid design should delegate as much as possible to the compiler. For example, the Truffle CPU [13] has dual-voltage SRAM arrays that require every load to match the precision level of its corresponding store. This invariant would be expensive to enforce with per-register and per-cache-line metadata, and it would be unreasonable for programmers to specify manually. Our final design leaves all the responsibility to the compiler, where enforcement is trivial.

Relegating intelligence to the compiler comes at a cost in safety: programming to the Truffle ISA directly is dangerous and error-prone. Fortunately, modern programmers rarely write directly to the CPU's hardware interface. Researchers should treat the ISA as a serialization channel between the compiler and architecture—not a human interface. Eschewing direct human–hardware interaction can pay off in fundamental efficiency gains.

3 Opportunities for Co-Design

3.1 Programming Hardware

The vast majority of programming languages research is on languages that target a very traditional notation of "programming." Programs must eventually be emitted as a sequence of instructions, meant to be interpreted by processors that will load them from memory and execute them in order. The programming languages and architecture communities should not remain satisfied with this traditional division of labor. The instruction set architecture abstraction imposes limits on the control that programmers can have over how hardware works, and compatibility concerns limit the creativity of architecture designs. Hybrid hardware–software research projects should design new hardware abstractions that violate the constraints of traditional ISAs.

Some recent work revived interest in languages for designing hardware and FPGA configurations [3, 31] or applied language techniques to special-purpose hardware like networking equipment [15] and embedded processor arrays [33]. But we can do more. A new story for programmable hardware will require radically new architecture designs, hardware interfaces that expose the right costs and capabilities, programming models that can abstract these interfaces' complexity, and careful consideration of the application domains.

Rethinking Architectures from the Ground Up

The central challenge in designing programmable hardware is finding the right architectural trade-off between reconfigurability and efficiency. Co-design is only possible if we expose more than what current CPUs do, but providing too much flexibility can limit efficiency. Current field-programmable gate arrays (FPGAs) are a prime example that misses the mark today: FPGAs strive for bit-level reconfigurability, and they pay for it in both performance and untenable tool complexity. New architectures will need to carefully choose the granularity of reconfiguration to balance these opposing constraints.

Exposing Communication Costs

The von Neumann abstraction is computation-centric: the fundamental unit of work is computation. But the costs in modern computers, especially in terms of energy, are increasingly consumed by *communication* more than computation. New programmable hardware will need to expose abstractions that reflect this inversion in the cost model. Unlike earlier attempts to design processors with exposed communication costs [16], these new abstractions need not be directly comprehensible for humans: we do not need to expect programmers to write this “assembly language” directly. Instead, we should design abstractions with their software toolchains in mind from the beginning.

Managing Complexity

New programmable hardware will need to work in concert with programming languages and compiler workflows that can manage their inherent complexity. Crucially, the toolchain will need to balance convenient automation with programmer control. It can be extremely powerful to unleash programmers on the problem of extracting efficiency from hardware, but history has also shown that overly complex programming models do not go mainstream. New toolchains should combine both language ideas, which can safely expose complexity, compiler techniques like program synthesis, which can hide the complexity.

General vs. Domain-Specific Architectures

There is a well-known trade-off in architecture between generality and efficiency [18]. The best reconfigurable hardware may never be fully general purpose: domain-specific languages, tools, and chips can have critical advantages for their sets of applications. At every level in the stack, new designs should provide tools for building domain-specific components.

3.2 Simpler Hardware With Software Verification

Modern computer architectures spend considerable resources protecting software from itself, with kernel/user mode, process isolation through virtual memory, sophisticated cache coherency and memory consistency models to extract maximum performance, and others. These protections add complexity to hardware design and implementation, and cost both time and energy during execution.

Rapid improvement in software verification has greatly expanded both the scope and the strength of the safety properties we can (automatically or manually) prove about programs. In particular, a system that is verified from the microkernel up to the application appears within reach [24]. The power of verification presents an opportunity to design new hardware architectures specifically for verified software. These architectures would avoid the energy and time overhead of dynamic protection mechanisms and simplify future scaling efforts.

Virtual Memory

Major architectures provide virtual memory to help operating systems provide process isolation. Each process sees its own address space and has no way of accessing the physical memory allocated to other processes. While this abstraction is convenient for operating system and application programmers, virtual memory protection entails considerable hardware complexity and cost.

Aiken et al. showed in 2006 that the cycle overhead of hardware virtual memory protection is up to 38% on the SPECweb99 benchmark [2]. Half this cost (19 pp) is due to switching between multiple address spaces (kernel and application) on control transfers, and the pressure these address spaces place on the TLB. An additional 12 pp is due to switching protection modes between ring 0 (kernel) and ring 3 (application) on control transfers. Working sets are growing much faster than the size of TLBs (combining L1 and L2, TLB entries have grown from 560 pages or 2,240 kB on Nehalem (2008) to 1,088 pages or 4,352 kB on Haswell (2014), and so TLB pressure and virtual memory overhead will only continue to increase.

The Singularity project explored the opportunity to abandon virtual memory in favor of *software-isolated processes (SIPs)* [22]. Instead of asking virtual memory hardware to provide process isolation, Singularity guarantees isolation between SIPs through static verification and some runtime checks. The verification is made feasible by Singularity's Sing# language for applications, which is type- and memory-safe. The software runtime checks are very low overhead; for example, a ping-pong message between two processes on Singularity is 7× faster than Linux and 8× faster than Windows [21].

Concurrency

A substantial portion of multiprocessor hardware is dedicated to cache coherence and memory consistency. These mechanisms work with the programming model, compilers and operating systems to preserve a simple memory model for programmers. But coherence protocols and consistency enforcement are neither simple nor cheap.

Memory models are complex to understand, and even more complex to implement correctly. Verification efforts have uncovered bugs in both hardware implementations of relaxed consistency models [19] and in the software memory models that attempt to abstract over them [46]. Given these difficulties, some have advocated for abandoning shared-memory hardware altogether (for example, Intel's Single-Chip Cloud Computer (SCC) has 48 cores without coherence), but this seems extreme.

Coherence and consistency come with significant performance and hardware overheads. DeNovo is a coherence protocol that works together with a disciplined shared-memory programming model to reduce coherence overheads [8]. Compared to MESI, DeNovo executes benchmarks up to 60% faster due to reduced network traffic, read misses and stall times.

Verification of data race freedom would enable simpler hardware designs and therefore increase the potential for future scaling. Traditional work in verifying race freedom (including the DeNovo programming model) has required explicit program annotations [23]. But recent work in formalizing hardware memory models [42] suggests that we should work towards *automatically* verifying race freedom. A hardware architecture for verified software would take advantage of provable race freedom to reduce coherence traffic and state overhead, and so save energy and time.

3.3 Hardware-backed DSLs

Today's programming languages are too complex for many of the tasks today's programmers want to complete. We ask programmers to reason about subtle issues like memory management, concurrency and race conditions, and large standard libraries. Domain-specific languages (DSLs) are a way to democratize programming by providing languages with higher-level semantics and simpler tooling, and have seen considerable success [45]. But traditional DSLs are often implemented as slow, interpreted languages, reducing their usefulness for many programmers.

Delite is a compiler framework for implementing high-performance DSLs [44]. The framework provides components common to high-performance implementations, such as parallel versions of higher-order functions, compiler optimizations, and the ability for DSL implementers to provide domain-specific optimizations for the code generator. Delite compiles DSLs to an intermediate representation which can then be compiled to multiple targets such as C++ and OpenCL.

While Delite makes for high-performance DSLs, they ultimately execute on general-purpose hardware. There is the opportunity to build hardware accelerators specialized to particular DSLs, or even to individual programs written in a given DSL. For example, the Halide DSL for image processing compiles to SIMD or CPU+GPU code and sees considerable performance gains [34]. But Darkroom, another DSL targeting the same domain, compiles directly to pipelined ASIC implementations for even higher performance [20].

Previous work on compiling DSLs to hardware has required custom design and implementation work for each DSL compiler. Future work should target a more general compiler from DSL implementation to hardware. We should take motivation from recent work on lightweight modular staging [37], which builds high-performance compilers by staging interpreter code. Existing high-level synthesis (HLS) tools for synthesizing circuits from higher-level programming languages are difficult to use and even more difficult to debug, because they are required to implement the rich and unsafe semantics of C. We should aim to build DSL frameworks with hardware synthesis in mind to unlock the potential of ASIC performance.

3.4 Beyond the CPU

Researchers is both architecture and programming languages tend to focus on one component above all others: the CPU. While the heart of a computer has special importance, it is not the only piece that deserves attention. When the goal is energy efficiency, for example, it is crucial to bear in mind that most of a smartphone's battery capacity goes to powering the screen and radio—the CPU is a comparatively small drain [39]. And in the datacenter, network and storage performance can outweigh the importance of raw compute power.

Hardware–software research is in a good position to seek out opportunities beyond the CPU: in memory, storage, networking, and even more mundane systems such as displays, sensors, and cooling infrastructure.

Non-volatile main memory is an example of an emerging architectural trend in desperate need of attention from a programming-model perspective. Initial work has shown that traditional abstractions are insufficient [32] and has proposed new approaches for managing non-volatile state [9, 48].

Language research should also seek out opportunities in networking. Recent software-defined networking work [15] has shown how to build better abstraction for networking equipment, but that work focuses on traditional roles for the network. We should also consider new ways to program the network. For instance, in the data center, what computation would be best offloaded from servers into the network? How can we productively program mobile device radio modems while preventing disastrous network attacks?

More experimental research should consider collaboration with the less glamorous parts of computer hardware. Power supply systems are one such example that are typically hidden at all costs from the end programmer. Researchers should question this separation. For example, can datacenter applications extract more energy efficiency by programming their power distribution systems? How can we re-design the battery and charging systems in a mobile phone to specialize them to a particular use patterns? Can applications and operating systems collaborate with hardware to fairly divide energy resources? Peripheral, traditionally non-architectural components like power systems are ripe for rethinking, but only in the context of co-design with a new programming model.

4 Conclusion

The air-tight hardware abstraction continues to serve innovation in programming languages well. But as physical challenges begin to threaten Moore's law, the biggest leaps in performance and efficiency will come from true hardware–software co-design. We believe our community can go far beyond the current primitive cooperation between hardware and software by designing them *together*. We can take inspiration from successes in embedded system design and our experiences with approximate computing. Exciting opportunities abound for hardware accelerators co-designed with software, reduced redundancy between hardware and software mechanisms, and entirely new classes of optimizations. We hope that our experiences and successes with hardware–software co-design will encourage researchers to cross the abstraction boundary to design more integrated and thoughtful systems.

References

1 Anant Agarwal, Martin Rinard, Stelios Sidiroglou, Sasa Misailovic, and Henry Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical report, MIT, 2009.

2 Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Workshop on Memory System Performance and Correctness (MSPC)*, 2006.

3 J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing hardware in a Scala embedded language. In *DAC*, June 2012.

4 James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. Uncertain<T>: A First-Order Type for Uncertain Data. In *ASPLOS*, 2014.

5 Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, 2012.

6 Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA*, 2013.

7 Lakshmi N. Chakrapani, Bilge E. S. Akgul, Suresh Cheemalavagu, Pinar Korkmaz, Krishna V. Palem, and Balasubramanian Seshasayee. Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology. In *DATE*, 2006.

8 Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarite V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

9 Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.

10 M. de Kruijf and K. Sankaralingam. Exploring the synergy of emerging workloads and silicon reliability trends. In *SELSE*, 2009.

11 Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *ISCA*, 2010.

12 Martin M. Deneroff, David E. Shaw, Ron O. Dror, Jeffrey S. Kuskin, Richard H. Larson, John K. Salmon, and Cliff Young. Anton: A specialized ASIC for molecular dynamics. In *Hot Chips*, 2008.

13 Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

14 Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural Acceleration for General-Purpose Approximate Programs. In *International Symposium on Microarchitecture (MICRO)*, 12 2012.

15 N. Foster, A. Guha, M. Reitblatt, A. Story, M.J. Freedman, N.P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. Languages for software-defined networks. *Communications Magazine, IEEE*, 51(2):128–134, February 2013.

16 Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS*, 2002.

17 Beayna Grigorian and Glenn 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.

18 Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, 2010.

19 Sudheendra Hangal, Durgam Vhia, Chaiyasit Manovit, Jiun-Weu Joseph Lu, and Sridhar Narayanan. TSOtool: A program for verifying memory systems using the memory consistency model. In *31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

20 James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4), 2014.

21 Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawbliztel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing OS processes to improve dependability and safety. In *EuroSys*, 2007.

22 Galen Hunt and James Larus. Singularity: Rethinking the software stack. In *21st ACM Symposium on Operating System Principles (SOSP)*, 2007.

23 Rajesh K. Karmani, P. Madhusudan, and Brandon M. Moore. Thread contracts for safe parallelism. In *16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

24 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

25 Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A. Jacobson, and Subhasish Mitra. ERSA: error resilient system architecture for probabilistic applications. In *DATE*, 2010.

26 Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: Saving refresh-power in mobile devices through critical data partitioning. In *ASPLOS*, 2011.

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

28 Sasa Misailovic, Stelios Sidiroglou, Hank Hoffman, and Martin Rinard. Quality of service profiling. In *ICSE*, 2010.

29 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.

30 Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L. Jones. Scalable stochastic processors. In *DATE*, 2010.

31 Rishiyur S. Nikhil and Arvind. What is bluespec? *SIGDA Newsl.*, 39(1):1–1, January 2009.

32 Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *ISCA*, 2014.

33 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.

34 Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, 2013.

35 Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Onward!*, 2010.

36 Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. Monitoring and debugging the quality of results in approximate programs. In *ASPLOS*, 2015.

37 Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, 2012.

38 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.

39 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.

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

41 Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn McKinley, Dan Grossman, and Luis Ceze. Expressing and Verifying Probabilistic Assertions. In *PLDI*, 2014.

42 Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009.

43 Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. General-purpose code acceleration with limited-precision analog computation. In *ISCA*, 2014.

44 Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s), 2014.

45 Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In *Onward!*, 2011.

46 Emina Torlak, Mandana Vaziri, and Julian Dolby. MemSAT: Checking axiomatic specifications of memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

47 Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Quality programmable vector processors for approximate computing. In *MICRO*, 2013.

48 Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.