The Aladdin Approach to Accelerator Design and Modeling

The authors developed Aladdin, a pre-RTL, power-performance accelerator modeling framework and demonstrated its application to system-on-chip (SoC) simulation. Aladdin estimates performance, power, and area of accelerators within 0.9, 4.9, and 6.6 percent with respect to register-transfer level (RTL) implementations. Integrated with architecture-level general-purpose core and memory hierarchy simulators, Aladdin provides a fast but accurate way to model accelerators' power and performance in an SoC environment.

Yakun Sophia Shao
Brandon Reagen
Gu-Yeon Wei
David Brooks
Harvard University

As we near the end of Dennard scaling, traditional performance and power scaling benefits based on technology improvements no longer exist. At the same time, transistor density improvements continue, resulting in the dark silicon problem, wherein chips have more transistors than a system can fully power at any point in time.[1] To overcome these challenges, application-specific hardware acceleration has surfaced as a promising approach because it delivers orders of magnitude performance and energy benefits compared to general-purpose solutions. Analysis of die photos (http://vlsiarch.eecs.harvard.edu/accelerators/die-photo-analysis) from Apple's A6 (iPhone 5), A7 (iPhone 5s), and A8 (iPhone 6) systems on chips (SoCs) shows that more than half of the die area is dedicated to blocks that are neither CPUs nor GPUs, but rather specialized IP blocks (see Figure 1). We also observe a consistent trend of an increasing number of specialized IP blocks across generations of Apple's SoCs (see Figure 2). The natural evolution of this trend leads to a growing volume and diversity of customized accelerators in future heterogeneous architectures ranging from mobile SoCs to high-performance servers (Figure 3). To thoroughly evaluate such architectures, designers must perform large design space exploration to understand the tradeoffs across the entire system, which is currently infeasible due to the lack of a fast simulation infrastructure for accelerator-centric systems.

Computer architects have long been developing and using high-level power and performance simulation tools for general-purpose cores and GPUs. In contrast, current accelerator-related research relies primarily on register-transfer level (RTL) implementations, a tedious and time-consuming process. It takes hours, if not days, to generate and simulate RTL and then synthesize it into logic to estimate the power and performance of a single accelerator design. This low-level, RTL-centric infrastructure cannot support

Published by the IEEE Computer Society

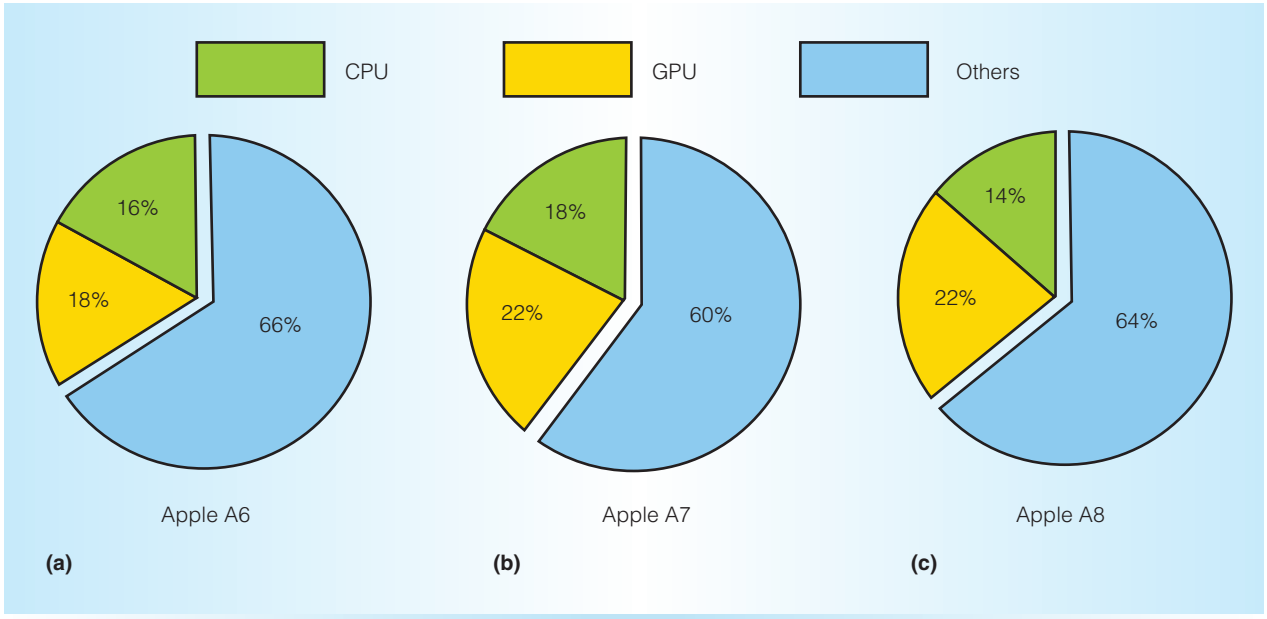

Figure 1. Die area breakdown of Apple's systems on chips (SoCs). (a) A6 (iPhone 5), (b) A7 (iPhone 5s), and (c) A8 (iPhone 6). More than half of the die area is dedicated to specialized IP blocks.

large architecture-level design space exploration, which is required to navigate the vast parameter space governing the interactions between general-purpose cores, accelerators, and shared resources, including cache hierarchies and on-chip networks. Hence, there is a clear need for a fast, high-level design tool to enable broad design space exploration for next-generation customized architectures.

In this article, we present Aladdin, a pre-RTL, power-performance simulator designed to enable rapid design space exploration for accelerator-centric systems. Aladdin takes high-level language descriptions of algorithms as inputs and uses dynamic data dependence graphs (DDDG) as a representation of an accelerator without having to generate RTL. Starting with an unconstrained program DDDG, which corresponds to an initial representation of accelerator hardware, Aladdin applies optimizations as well as constraints to the graph to create a realistic model of accelerator activity. We rigorously validated Aladdin against RTL implementations of accelerators from both handwritten Verilog and a commercial high-level synthesis (HLS) tool for various applications, including accelerators in Memcached,[2] HARP,[3] NPU,[4] and a commonly used throughput-oriented benchmark suite called SHOC.[5] Our results show that Aladdin can model performance within 0.9 percent, power within 4.9 percent, and area within 6.6 percent compared to accelerator designs generated by traditional RTL flows. In

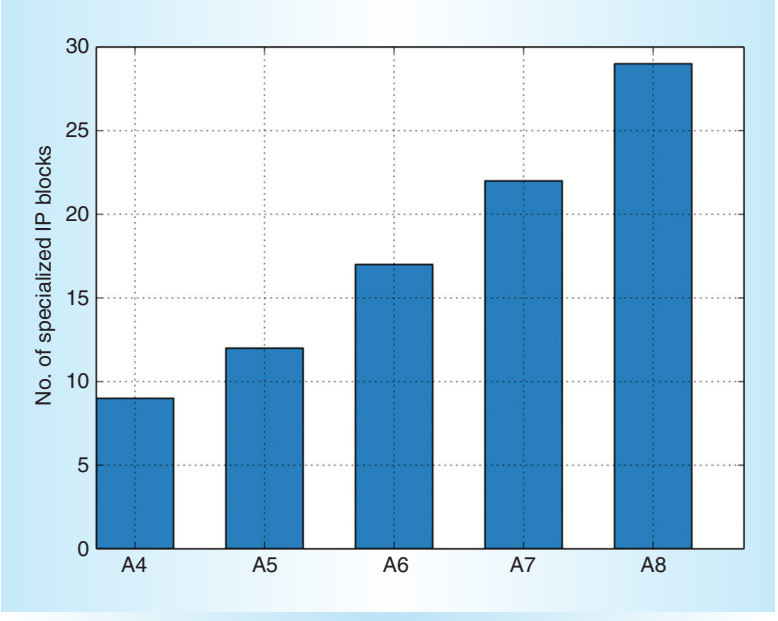

Figure 2. Number of specialized IP blocks across generations of Apple's SoCs. We see a consistent increase in the number of blocks.

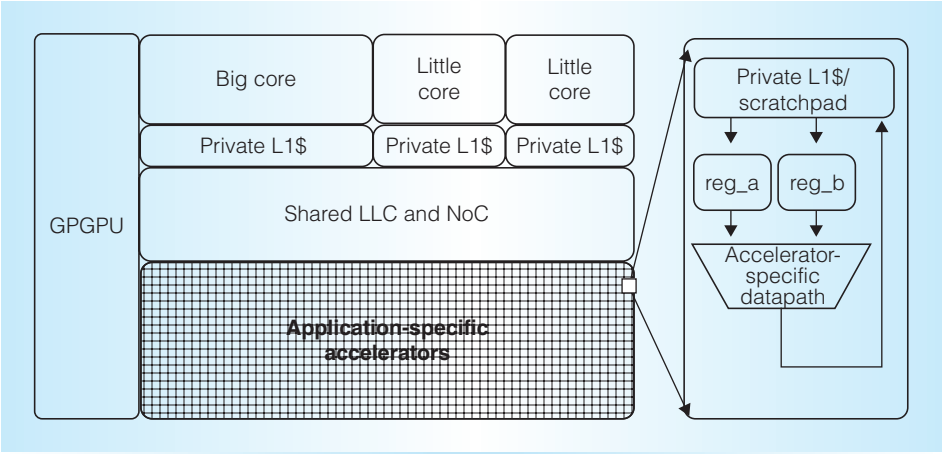

Figure 3. Future heterogeneous architecture will include a large number of customized accelerators. Large design space exploration is needed to design such architecture.

addition, Aladdin provides these estimates more than 100 times faster.

Aladdin captures accelerator design tradeoffs, enabling new architectural research directions in heterogeneous systems comprising general-purpose cores, accelerators, and a shared memory hierarchy. We demonstrated this capability by integrating Aladdin with a full memory hierarchy model. Such infrastructure lets users explore customized and shared memory hierarchies for accelerators in a heterogeneous environment. In a case study with the GEMM benchmark, Aladdin uncovers significant high-level design tradeoffs by evaluating a broader design space of the entire system. This analysis results in more than 3× performance improvements compared to the conventional approach of designing accelerators in isolation.

Background

Hardware acceleration exists in many forms, ranging from analog accelerators, fixed-function accelerators, and programmable accelerators, such as GPUs and digital signal processors. In this work, we focus on fixed-function accelerators. We discuss the design flow, design space, and state-of-the-art research infrastructure for fixed-function accelerators in order to illustrate the challenges associated with current accelerator research and discuss why a tool like Aladdin opens up new research opportunities for architects.

Accelerator design flow

The current accelerator design flow requires multiple CAD tools, which is inherently tedious and time-consuming. The process starts with a high-level description of an algorithm; then, designers either manually implement the algorithm in RTL or use HLS tools, such as Xilinx's Vivado HLS, to compile the high-level implementation (for example, C/C++) to RTL.

Writing RTL manually takes significant effort, and the quality highly depends on designers' expertise. Although HLS tools offer opportunities to automatically generate the RTL implementation, extensively tuning C code is still necessary to meet design requirements. After generating RTL, designers must use commercial CAD tools, such as Synopsys's Design Compiler and Mentor Graphics' ModelSim, to estimate power and cycle counts.

In contrast, Aladdin takes unmodified, high-level language descriptions of algorithms to generate a DDDG representation of accelerators, which accurately models the cycle-level power, performance, and area of realistic accelerator designs. As a pre-RTL simulator, Aladdin is orders of magnitude faster than existing CAD flows.

Accelerator design space

Despite the application-specific nature of accelerators, the accelerator design space is quite large, given a range of architecture- and

circuit-level alternatives. Figure 4 illustrates a power-performance design space of accelerator designs for the GEMM workload from the SHOC benchmark suite. The square points were generated from a commercial HLS flow sweeping datapath parameters, including loop-iteration parallelism, pipelining, array partitioning, and clock frequency. However, HLS flows generally provision a fixed latency for all memory accesses, implicitly assuming local scratchpad memory fed by direct memory access (DMA) controllers.

Such simple designs are not well suited for capturing data locality or interactions with complex memory hierarchies. The circle points in Figure 4 were generated by Aladdin integrated with a full cache hierarchy and memory model, sweeping not only datapath parameters but also memory parameters. By doing so, Aladdin exposes a rich design space that incorporates the realistic memory penalties in terms of time and power, which is impractical with existing HLS tools alone.

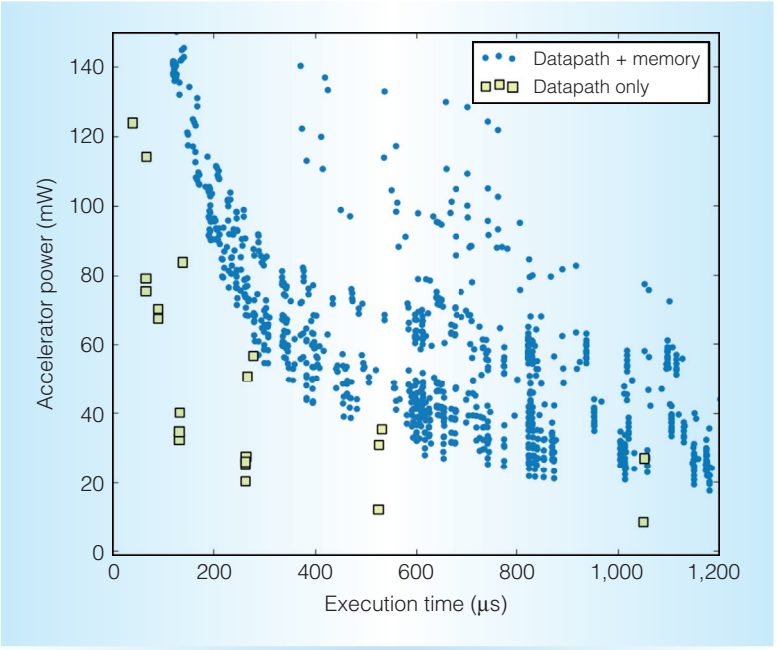

Figure 4. GEMM design space. A large design space is overlooked if we ignore the memory parameters.

State-of-the-art accelerator research infrastructure

The *International Technology Roadmap for Semiconductors* (*ITRS*) predicts hundreds to thousands of customized accelerators by 2022.[6] However, state-of-the-art accelerator research projects still contain only a handful of accelerators because of the cumbersome design flow that inhibits computer architects from evaluating large accelerator-centric systems. Table 1 categorizes accelerator-related research projects in the computer architecture community over the past five years based on the means of implementation (handwritten RTL versus HLS tools) and the scope of possible design exploration.

Researchers have been able to propose novel implementations of accelerators for a wide range of applications, by either writing RTL directly or using HLS tools despite the time-consuming process. With the help of the HLS flow, we have begun to see studies evaluating design tradeoffs in accelerator datapaths, which is otherwise impractical using handwritten RTL. However, as discussed earlier, HLS tools cannot easily navigate large design spaces of heterogeneous architectures. This inadequacy in infrastructure has confined the exploratory scope of accelerator research.

The Aladdin framework

Aladdin takes high-level language descriptions of algorithms and accelerator design parameters as inputs and then outputs power, performance, area, and cycle-level activities of accelerator implementations, including memory activity that can be connected with shared memory and interconnect models. Aladdin can be first used as an accelerator simulator that models an accelerator's behavior in an accelerator-rich SoC to facilitate the evaluation of the interaction between accelerators and shared resources like memory. On the other hand, Aladdin can also be used as an early-stage accelerator design assistant that allows SoC designers to quickly navigate the large design space of accelerators before they start implementing RTL, thereby greatly reducing design iterations.

Modeling methodology

The foundation of the Aladdin infrastructure is the use of DDDG to represent accelerators. A DDDG is a directed acyclic graph

Table 1. Accelerator research infrastructure.

Means of implementation	Novel accelerator design	Accelerator datapath tradeoffs	Heterogeneous SoC tradeoffs
Hand-coded RTL	Buffer-integrated cache,[7] Memcached,[2,8] Sonic Millip3De,[9] HARP[3]	Inadequate	Inadequate
High-level synthesis (HLS)	LINQits,[10] Convolution Engine,[11] Conservation Cores[12]	Cong,[13] Liu,[14] Reagen[15]	Inadequate

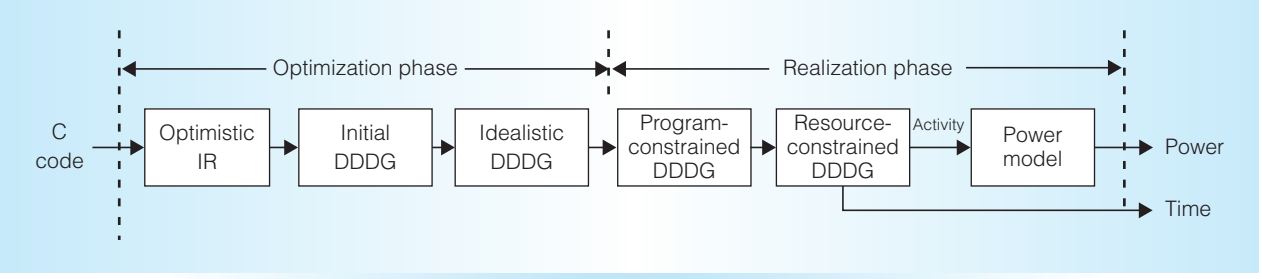

Figure 5. Overview of the Aladdin framework. Aladdin takes C code and passes it through an optimization phase and a realization phase to estimate an accelerator's power and performance.

wherein nodes represent computation and edges represent dynamic data dependences between nodes. The dataflow nature of hardware accelerators makes the DDDG a good candidate to model their behavior. Figure 5 illustrates Aladdin's overall structure, starting from a C description of an algorithm and passing through an optimization phase, where the DDDG is constructed and optimized to derive an idealized representation of the algorithm. The idealized DDDG then passes to a realization phase that restricts the DDDG by applying realistic program dependences and resource constraints. User-defined configurations allow wide design space exploration of accelerator implementations. The outcome of these two phases is a pre-RTL, power-performance model of accelerators.

Aladdin uses a DDDG to represent program behaviors so that it can take arbitrary C code descriptions of an algorithm—without any modifications—to expose algorithmic parallelism. This fundamental feature lets users rapidly investigate different algorithms and accelerator implementations. Owing to its optimistic nature, dynamic analysis has been previously deployed in parallelism research exploring the limits of instruction level parallelism (ILP)[16] and recent modeling frameworks for multicore processors.[17] These studies sought to quickly measure the upper bound of performance achievable on an ideal parallel machine. Our work has two main distinctions from these efforts. First, previous efforts model traditional Von Neumann machines where instructions are fetched, decoded, and executed on a fixed, but programmable, architecture. In contrast, Aladdin models a vast palette of different accelerator implementation alternatives for the DDDG; the optimization phase incorporates typical hardware optimizations, such as removing memory operations via customized storage inside the datapath and reducing the bit width of functional units. The second distinction is that Aladdin provides a realistic power-performance model of accelerators across a range of design alternatives during its realization phase, unlike previous work that offered an upper-bound performance estimate.

In contrast to dynamic approaches, parallelizing compilers and HLS tools use program dependence graphs (PDG) that statically capture both control and data dependences. Static analysis is inherently conservative in its dependence analysis, because it is used to generate code and hardware that works in all

circumstances and is built without runtime information. A classic example of this conservatism is the enforcement of false dependences that restrict algorithmic parallelism. For instance, programmers often use pointers to navigate arrays, and disambiguating these memory references is a challenge for HLS tools. Such situations frequently lead to designs that are more sequential compared to what a human RTL programmer would develop. Therefore, although HLS tools offer the opportunity to automatically generate RTL, designers still need to extensively tune their C code to expose parallelism explicitly. Thus, Aladdin differs from HLS tools; it is a pre-RTL simulator to model the accelerator's behavior, whereas HLS is burdened with generating actual, correct hardware.

Optimization phase

The optimization phase forms an idealized DDDG that represents only the fundamental dependences of the algorithm. An idealized DDDG for accelerators must satisfy three requirements:

- express only necessary computation and memory accesses,
- capture only true read-after-write dependences, and
- remove unnecessary dependences in the context of customized accelerators.

Here, we describe how Aladdin's optimization phase addresses these requirements.

Optimistic intermediate representation. Aladdin builds the DDDG from a dynamic instruction trace, where the choice of the instruction set architecture (ISA) significantly impacts the complexity and granularity of the nodes in the graph. In fact, a trace using a machine-specific ISA contains instructions that are not part of the program but produced due to the artifacts of the ISA (for example, register spills).[18] To avoid such artifacts, Aladdin uses a high-level, machine-independent intermediate representation (IR) provided by the LLVM compiler. The LLVM IR is optimistic because it allows an unlimited number of registers, eliminating additional instructions associated with register spilling. We use a customized LLVM pass to emit fully optimized IR instructions in a trace file. The trace includes dynamic instruction information such as operation codes, register IDs, parameter data values, and dynamic addresses of memory operations.

Initial DDDG. Aladdin analyzes both register and memory dependences based on the IR trace. Only true read-after-write data dependences are respected in the initial DDDG construction. This DDDG is optimistic enough for the purpose of ILP limit studies but is missing several characteristics of hardware accelerators.

Idealized DDDG. Hardware accelerators can take on application-specific features not modeled in the initial DDDG. Aladdin generates an idealized DDDG representation of the original algorithm by employing commonly used hardware customization strategies such as bit-width tuning, strength reduction, induction-variable removal, and memory-to-register conversion.[19]

Realization phase

The realization phase uses program and resource parameters, defined by users, to constrain the idealized DDDG generated in the optimization phase.

Program-constrained DDDG. The idealized DDDG assumes that hardware designers can eliminate all control and false data dependences at design time, which is too optimistic. To model more realistic hardware designs, program-constrained DDDG brings back the control and data dependences that are hard to resolve in static time. For example, the idealized DDDG optimistically removes all false memory dependences between dynamic instructions, keeping true read-after-write dependences. This is realistic for memory accesses with addresses that can be resolved at design time. However, some algorithms have input-dependent memory accesses (for example, histogram), wherein different inputs result in different dynamic dependences. Without runtime memory disambiguation support, designers must make conservative assumptions about memory dependences to ensure correctness. To model realistic memory dependences, the realization

Table 2. Realization phase user-defined parameters. *i*::*j*::*k* denotes a set of values from *i* to *k* by a stepping factor *j*.

Parameters	Example range
Loop rolling factor	[1::2::trip count]
Clock period (ns)	[1::2::16]
Functional unit latency	Single cycle, pipelined
Memory ports	[1::2::64]

phase includes memory ambiguation that constrains the input-dependent memory accesses by adding dependences between all dynamic instances of a load-store pair, as long as a true memory dependence is observed for any pair.

Resource-constrained DDDG. Finally, Aladdin accounts for user-specified hardware resource constraints, a subset of which is shown in Table 2. Users specify the type and size of hardware resources in an input configuration file. Aladdin then binds the program-constrained DDDG onto the hardware resources, leading to the resource-constrained DDDG. Aladdin can easily sweep resource parameters to explore an algorithm's design space.

An example. Figure 6 illustrates different phases of Aladdin transformations using a microbenchmark as an example. After the IR trace of the C code has been produced, the optimization and realization phases generate the resource-constrained DDDG that models accelerator behavior. In this example, we assume the user wants an accelerator with a factor-of-2 loop-iteration parallelism and without loop pipelining. The solid arrows in the DDDG are true data dependences, and the dashed arrows represent resource constraints, such as loop rolling and turning off loop pipelining. The horizontal dashed lines represent clock cycle boundaries. The corresponding resource activities are shown to the right of the DDDG example. We see that the DDDG reflects the dataflow nature of the accelerator. Figure 7 shows a realistic cycle-level resource activity for one implementation of the fast Fourier transform (FFT) benchmark, wherein Aladdin accurately captures the distinct phases of FFT.

Power and area models. We can construct detailed power and area models by synthesizing microbenchmarks that exercise the functional units. Our microbenchmarks cover all of the computational instructions in IR so that there is a one-to-one mapping between nodes in the DDDG and functional units in the power model. We synthesized these microbenchmarks using Synopsys's Design Compiler in conjunction with a commercial 40-nm standard cell library to characterize the switching, internal, and leakage power of each functional unit. This characterization is fully automated to easily migrate to new technologies. The memory power model is based on a commercial register file and static RAM (SRAM) memory compiler that accompanies our standard cell library. We compared this memory power model to CACTI and found consistent trends, but we retained the memory compiler model for consistency with the standard cell library.

Limitations

As a dynamic modeling approach, Aladdin has its limitations.

Algorithm choices. Aladdin does not automatically sweep different algorithms. Rather, it provides a framework to quickly explore various hardware designs of a particular algorithm. This means designers can use Aladdin to quickly and quantitatively compare the design spaces of multiple algorithms for the same application to find the most suitable algorithm choice.

Input dependent. Aladdin models only those operations that appear in the dynamic trace, which means it does not instantiate hardware for code not executed with a specific input. For Aladdin to completely model the hardware cost of a program, users must provide inputs that exercise all paths of the code.

Input C code. Aladdin can create a DDDG for any C code. However, in terms of modeling accelerators, C constructs that require resources outside the accelerator, such as system calls and dynamic memory allocation, are not modeled. In fact, understanding how to handle such events is a research direction that Aladdin facilitates.

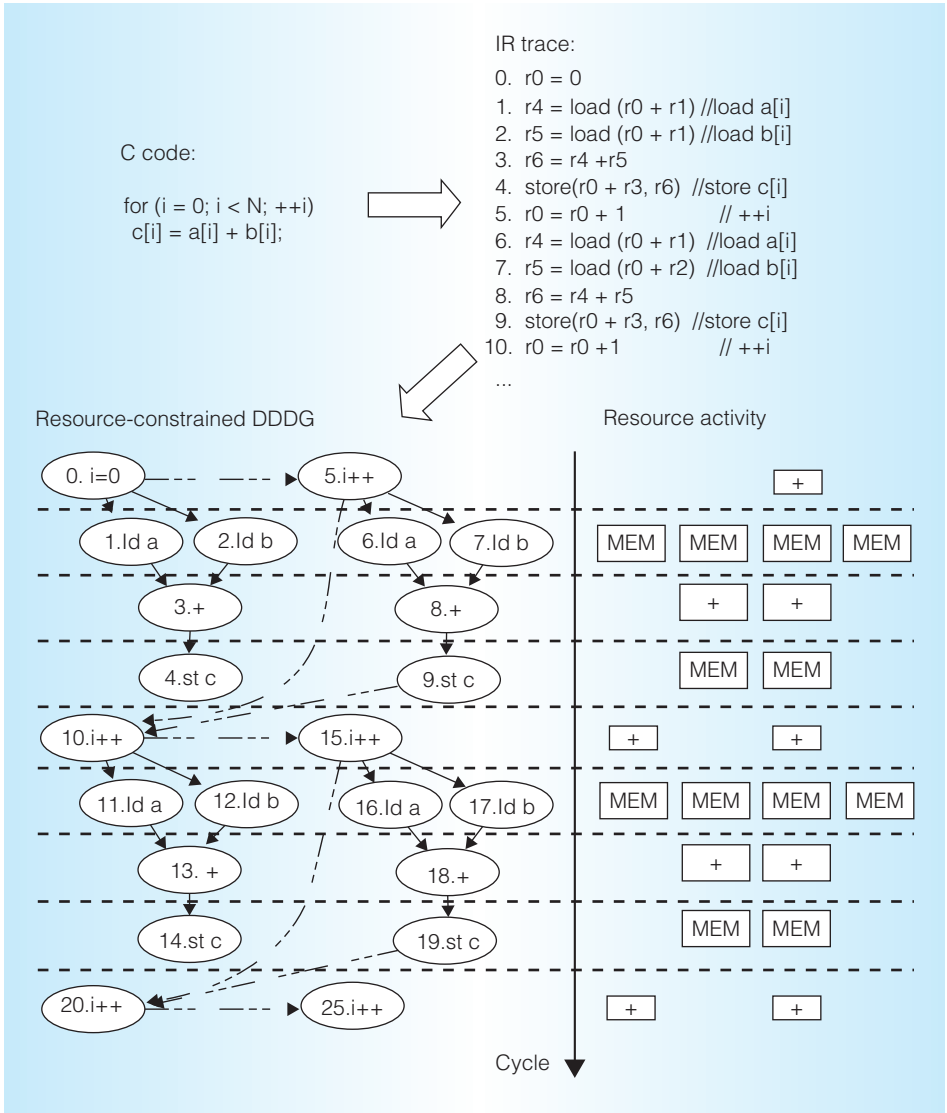

Figure 6. Different phases of Aladdin transformation illustrated with a microbenchmark: starting from a C code, to an intermediate representation (IR) trace, then a resource-constrained dynamic data dependence graph (DDDG), and finally a dynamic activity profile.

Aladdin validation

Figure 8 outlines the methodology used to validate Aladdin. To generate Verilog, we either hand-code RTL or use Xilinx's Vivado HLS tool. The RTL design flow is an iterative process and requires extensive tuning of both RTL and C code. The power and area estimates from Aladdin are compared against Verilog synthesized by Design Compiler using commercial 40-nm standard cells. Aladdin's performance model is validated against ModelSim Verilog simulations. The Switching Activity Interchange Format (SAIF) activity file generated from ModelSim is fed to Design Compiler to capture the switching activity at the gate level.

Figure 9 shows that Aladdin accurately models performance, power, and area compared against RTL implementations across all of the presented benchmarks with average errors of 0.9, 4.9, and 6.5 percent, respectively. For each SHOC workload, we validated six points on the power-performance Pareto frontier. The SHOC validation results show that Aladdin accurately models entire design spaces, whereas for single accelerator

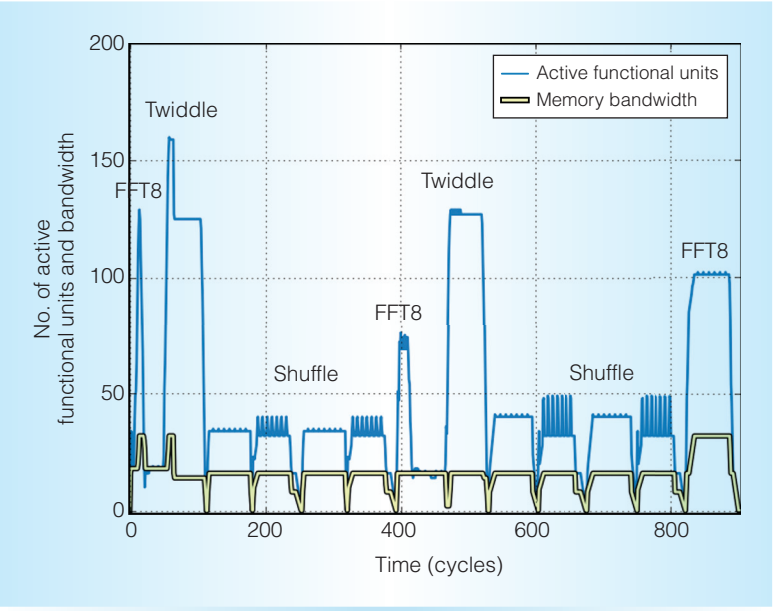

Figure 7. Realistic cycle-level functional units and memory activity of a fast Fourier transform (FFT) accelerator. This cycle-level activity is also an input to the power model to represent the dynamic activity of accelerators.

designs, Aladdin is not subject to HLS shortcomings and can accurately model different customization strategies.

Aladdin also enables rapid design space exploration of accelerators. Table 3 quantifies the differences in algorithm-to-solution time to explore a design space of the FFT benchmark with 36 points. Compared to traditional RTL flows, Aladdin skips the time-consuming RTL generation, synthesis, and simulation process. On average, it takes 87 minutes to generate a single design using the RTL flow but only 1 minute for Aladdin, including both Aladdin's optimization phase (50 seconds) and realization phase (12 seconds). However, because Aladdin needs to perform the optimization phase only once for each algorithm, this optimization time can be amortized across large design spaces. Consequently, it takes only 7 minutes to enumerate the full design space of FFT with Aladdin, compared to 52 hours with the RTL flow.

Embracing the era of specialization

Customized architectures comprising CPUs, GPUs, and accelerators are already seen in mobile systems and are beginning to emerge in servers and desktops. However, current SoC designs that pull together hard IP blocks into a single integrated substrate—simply continuing the multidecade trend of SoC integration—cannot leverage higher-level coordination and optimization between traditional general-purpose cores, accelerators, and shared resources such as cache hierarchies. As the industry dives further into this era of specialization, there is a clear need for a design methodology like Aladdin that facilitates broad design space exploration of next-generation heterogeneous architectures.

To illustrate, we consider a heterogeneous system comprising a general-purpose core, a GEMM accelerator, and a shared L2 cache that can be accessed by both the core and the accelerator. Figure 10a shows the accelerator design space without memory contention from the general-purpose core. We modulate the algorithmic blocking factor and find that a blocking factor of 16 is always better than 32 with respect to both power and performance.

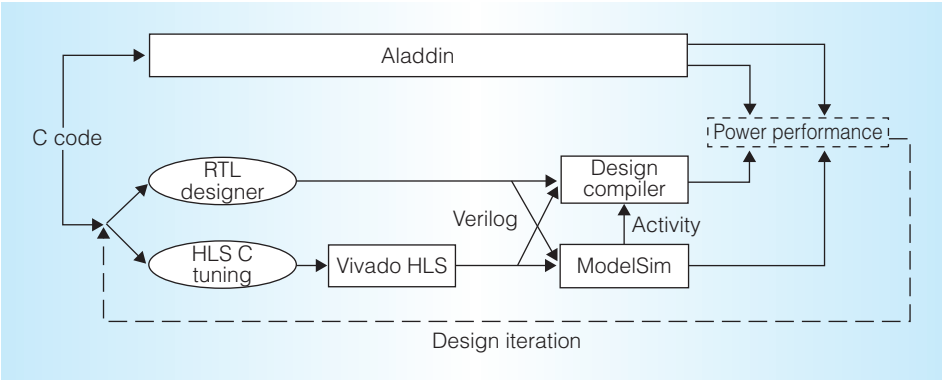

Figure 8. Aladdin validation flow. Aladdin's estimates are compared against synthesized accelerators using commercial CAD tools.

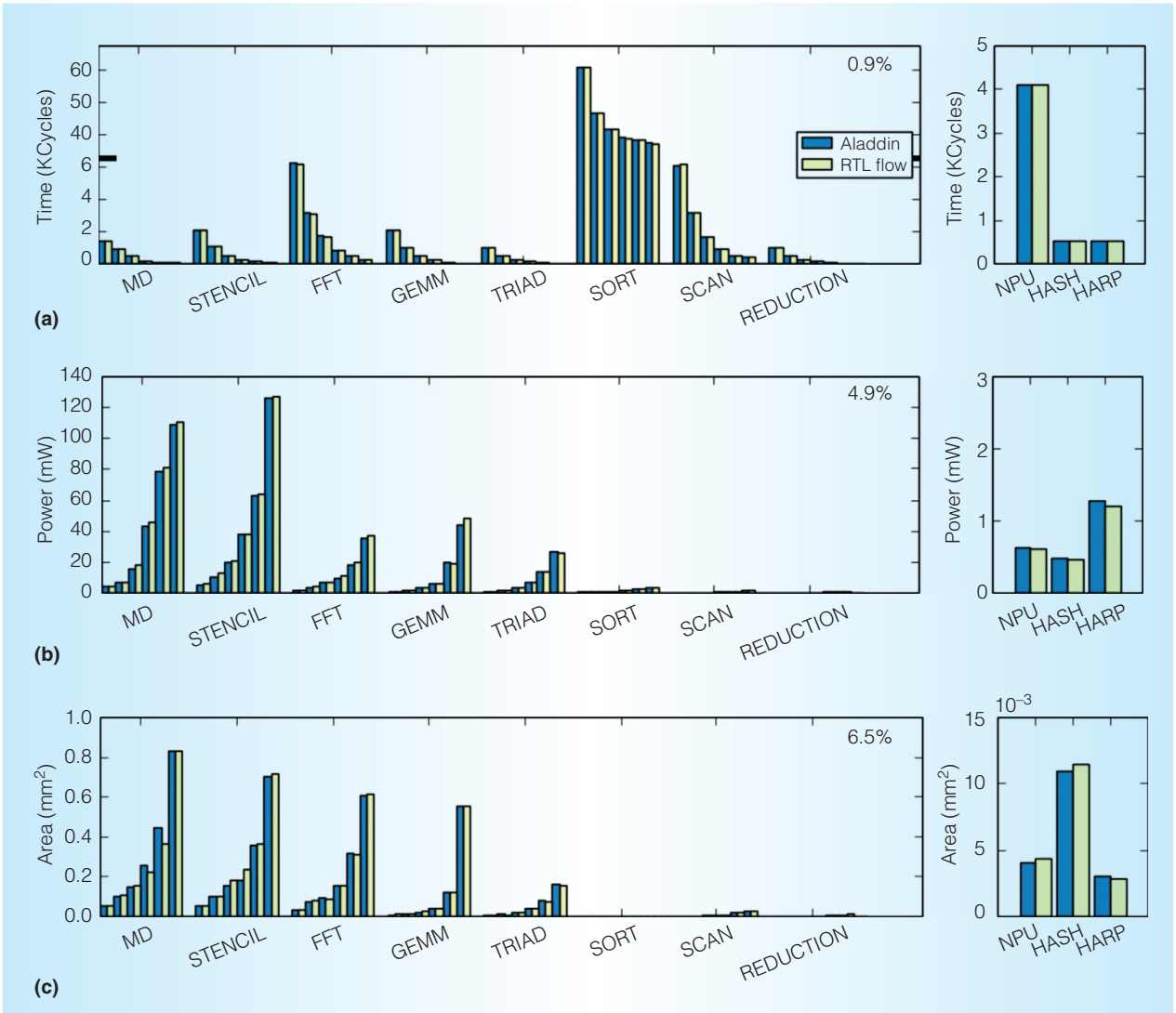

Figure 9. Aladdin validation results against RTL implementations. (a) Performance, b) power, and (c) area validation. Aladdin accurately models performance, power, and area across various benchmarks.

Table 3. Algorithm-to-solution time per design.

Design stage	Hand-coded RTL	HLS	Aladdin
Programming effort	High	Medium	N/A
RTL generation	Designer dependent	37 minutes	N/A
RTL simulation time	5 minutes		N/A
RTL synthesis time	45 minutes		N/A
Time to solution per design	87 minutes		1 minute
Time to solution (36 designs)	52 hours		7 minutes

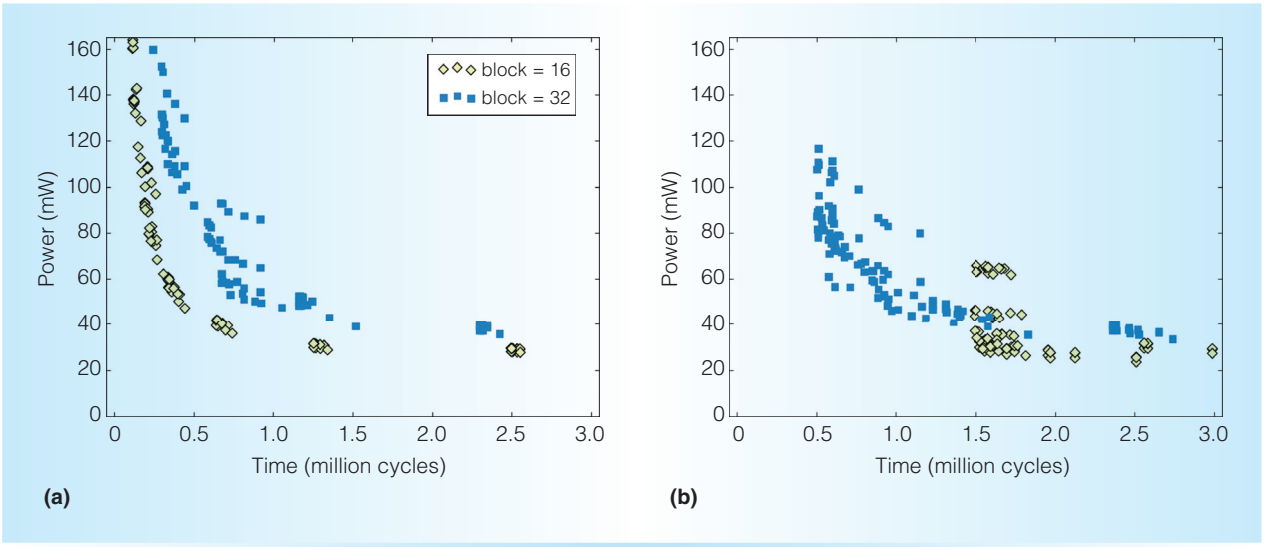

Figure 10. GEMM design space: (a) without contention and (b) with contention. A blocking factor of 16 outperforms a blocking factor of 32 when no memory contention presents, but the opposite occurs when there is memory contention.

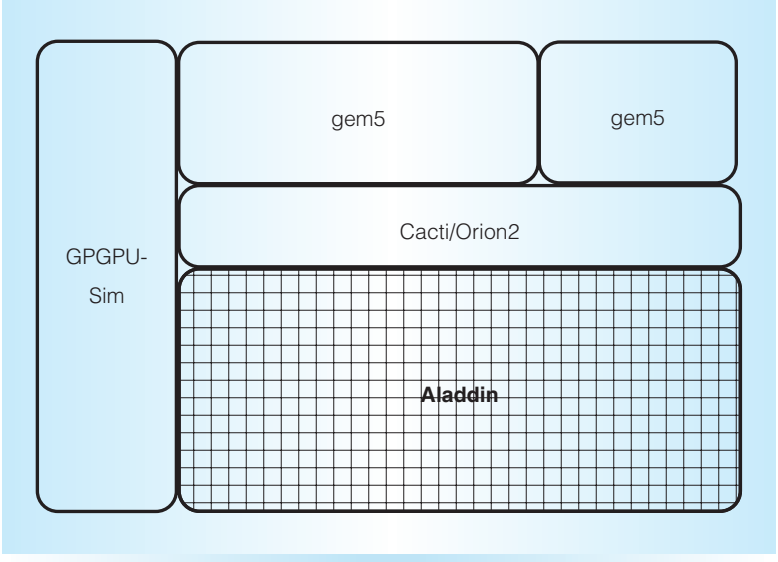

Figure 11. SoC simulation infrastructure with Aladdin. Aladdin can be integrated with other architectural simulators to model future heterogeneous SoCs.

However, when both the accelerator and the core actively access the shared cache, shown in Figure 10b, we see that blocking factor 32 offers better power-performance tradeoffs than blocking factor 16. With a larger blocking factor, the accelerator requires fewer references to the matrices in total and, thus, fewer data requests from the L2 cache. If designers had picked a blocking factor of 16 instead of 32 without considering the memory contention, the accelerator could suffer up to 3× performance degradation in a contended system. Aladdin helps designers find and avoid such design decisions by evaluating system-wide accelerator design tradeoffs, which is not possible with conventional RTL-based design flows.

Moving forward, the era of specialization poses unique opportunities and challenges for computer architects. Despite the energy efficiency benefit, accelerators lack flexibility and programmability, especially compared with general-purpose processors. For example, virtual memory and cache coherency are well-known mechanisms to ease programmability in traditional systems but are not commonly applied in today's accelerators. Instead, accelerators rely heavily on software-managed scratchpad memory to provide fixed-latency memory access and DMA to communicate data back and forth, which leads to significant chip resource and software engineering cost. To redesign today's SoCs to support programmability, architects can use Aladdin to evaluate the cost and benefit by applying different mechanisms to accelerators without worrying about low-level implementation details.

Aladdin is publicly available, and we have integrated Aladdin with the gem5 simulator,[20] one of the most widely used architectural

simulators in the community. Aladdin, integrated with gem5, lets researchers model an SoC-like environment (for example, Figure 11). Tools like Aladdin will transform the process of architectural design for heterogeneous systems, unlocking new opportunities for wide adoption of accelerator-rich architectures. MICRO

Acknowledgments

This work was partially supported by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, the National Science Foundation (NSF) Expeditions in Computing Award no. CCF-0926148, DARPA under contract no. HR0011-13-C-0022, and a Google Faculty Research Award. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

1. H. Esmaeilzadeh et al., "Dark Silicon and the End of Multicore Scaling," *Proc. 38th Ann. Int'l Symp. Computer Architecture*, 2011, pp. 365–376.
2. K.T. Lim et al., "Thin Servers with Smart Pipes: Designing SOC Accelerators for Memcached," *Proc. 40th Ann. Int'l Symp. Computer Architecture*, 2013, pp. 36–47.
3. L. Wu et al., "Navigating Big Data with High-Throughput, Energy-Efficient Data Partitioning," *Proc. 40th Ann. Int'l Symp. Computer Architecture*, 2013, pp. 249–260.
4. H. Esmaeilzadeh et al., "Neural Acceleration for General-Purpose Approximate Programs," *Proc. 45th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2012, pp. 449–460.
5. A. Danalis et al., "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," *Proc. 3rd Workshop General-Purpose Computation on Graphics Processing Units*, 2010, pp. 63–74.
6. *System Drivers*, ITRS, 2013; www.itrs.net/ITRS%201999-2014%20Mtgs,%20Presentations%20&%20Links/2013ITRS/2013Chapters/2013SysDrivers_Summary.pdf.
7. C.F. Fajardo et al., "Buffer-Integrated-Cache: A Cost-Effective SRAM Architecture for Handheld and Embedded Platforms," *Proc. 48th Design Automation Conf.*, 2011, pp. 966–971.
8. M. Lavasani, H. Angepat, and D. Chiou, "An FPGA-Based In-line Accelerator for Memcached," *IEEE Computer Architecture Letters*, 2013, pp. 57–60.
9. R. Sampson et al., "Sonic Millip3de: A Massively Parallel 3D-Stacked Accelerator for 3D Ultrasound," *Proc. IEEE 19th Int'l Symp. High Performance Computer Architecture*, 2013, pp. 318–329.
10. E.S. Chung, J.D. Davis, and J. Lee, "Linqits: Big Data on Little Clients," *Proc. 40th Int'l Symp. Computer Architecture*, 2013, pp. 261–272.
11. W. Qadeer et al., "Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing," *Proc. 40th Ann. Int'l Symp. Computer Architecture*, 2013, pp. 24–35.
12. G. Venkatesh et al., "Conservation Cores: Reducing the Energy of Mature Computations," *Proc. 15th Conf. Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 205–218.
13. J. Cong, K. Gururaj, and G. Han, "Synthesis of Reconfigurable High-Performance Multicore Systems," *Proc. ACM/SIGDA Int'l Symp. Field Programmable Gate Arrays*, 2009, pp. 201–208.
14. H.-Y. Liu, M. Petracca, and L.P. Carloni, "Compositional System-Level Design Exploration with Planning of High-Level Synthesis," *Proc. Conf. Design, Automation and Test in Europe*, 2012, pp. 641–646.
15. B. Reagen et al., "Quantifying Acceleration: Power/Performance Trade-offs of Application Kernels in Hardware," *Proc. Int'l Symp. Low Power Electronics and Design*, 2013, pp. 395–400.
16. D.W. Wall, "Limits of Instruction-Level Parallelism," *Proc. 4th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 176–188.
17. D. Jeon et al., "Kismet: Parallel Speedup Estimates for Serial Programs," *Proc. ACM Int'l Conf. Object Oriented Programming*

Systems Languages and Applications, 2011, pp. 519–536.

18. Y.S. Shao and D. Brooks, "ISA-Independent Workload Characterization and Its Implications for Specialized Architectures," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software*, 2013, pp. 245–255.
19. Y.S. Shao et al., "Aladdin: A Pre-RTL, Power Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures," *Proc. ACM/IEEE 41st Int'l Symp. Computer Architecture*, 2014, pp. 97–108.
20. N.L. Binkert et al., "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011, pp. 1–7.

Yakun Sophia Shao is a PhD candidate in computer science at Harvard University. Her research focuses on computer architecture, particularly on modeling and design for specialized architectures. Shao has an MS in computer science from Harvard University. Contact her at shao@eecs.harvard.edu.

Brandon Reagen is a PhD student in computer science at Harvard University. His research interests include methods for low-power, high-performance design leveraging specialized hardware, accelerator-centric systems, and computer architecture/VLSI codesign. Reagen has an MS in computer science from Harvard University. Contact him at reagen@fas.harvard.edu.

Gu-Yeon Wei is a Gordon McKay Professor of Electrical Engineering and Computer Science at Harvard University. His research interests include mixed-signal integrated circuits, computer architecture, and runtime software, looking for cross-layer opportunities to develop energy-efficient systems. Wei has a PhD in electrical engineering from Stanford University. Contact him at guyeon@eecs.harvard.edu.

David Brooks is the Haley Family Professor of Computer Science at Harvard University. His research interests include architectural and software approaches to address power, thermal, and reliability issues for embedded and high-performance computing systems. Brooks has a PhD in electrical engineering from Princeton University. Contact him at dbrooks@eecs.harvard.edu.