Mapping and Modeling Approximate Computing Techniques

Mark Wyse André Baixo Thierry Moreau Bill Zorn Adrian Sampson James Bornholt
Luis Ceze Mark Oskin

University of Washington

ABSTRACT

The efficiency–accuracy trade-off of approximate computing spans a diverse array of techniques at both the hardware and software levels. While this diversity is key to the success of approximation research, it also entails considerable complexity in developing and validating new approximation techniques. Researchers must relate their ideas to a vast body of work on approximate computing and invest significant software and hardware implementation effort to validate their hypotheses. These problems slow the pace of innovation and, when they prove too much to bear, risk misdirecting research away from promising ideas.

This paper makes two contributions to more flexible research in approximate computing. We present the first taxonomy of current research in approximate computing, identifying two key dimensions in the space of techniques: fine- vs. coarse-grain, and deterministic vs. nondeterministic. We discuss underpopulated regions of this classification and identify trends in current research. Then, informed by this taxonomy, we present REACT, an open-source modeling framework that lets researchers rapidly evaluate approximate-computing techniques and lets practitioners understand the potential impact of approximation in their code.

REACT combines an application profiler with an energy model and error injection framework to measure the effects of approximation on user-provided applications. We show how REACT can easily model approximation techniques from diverse regions of our taxonomy. To demonstrate REACT's utility, we use it to evaluate several benchmarks for their amenability to approximation and to individual techniques. Our exploration shows that the energy savings and error vary widely between techniques and applications, and that coarse-grain techniques dominate fine-grain ones.

1. INTRODUCTION

Approximate computing exploits applications' tolerance of quality degradation to improve performance and energy efficiency. Research in approximate spans the entire system stack, from devices to software and algorithms. Although recent research has demonstrated tremendous potential for diverse application domains, researchers and practitioners alike face the persistent challenge of understanding the quality–energy trade-offs associated with approximate computing.

For researchers, developing new approximation techniques requires substantial investment in software and hardware development to validate new opportunities. Recent work has extended the reach of approximate computing to many areas of computer architecture: floating-point units [1], caches [2], DRAM [3], and accelerators [4, 5, 6, 7]. Software techniques are similarly diverse, from transforming loops [8] to eliding synchronization [9]. The diversity and reach of these techniques creates a struggle for researchers to understand the quality–energy trade-offs of their techniques and how they relate to existing work. Evaluating hardware techniques requires either detailed and laborious implementation work, with the risk of disappointing results, or extensive modifications to existing power modeling tools, with the risk of incorrect or irrelevant modeling [10]. Software techniques pose similar challenges and often rely on cumbersome, manual processes to evaluate quality degradation. There exists no general, understandable model for researchers to rapidly evaluate the potential of new approximate-computing opportunities on real applications.

Practitioners looking to employ approximation in their systems face substantial barriers to adoption as well. Many approximation techniques require specialized hardware [11] and extensive engineering effort to implement and evaluate. Despite ongoing work on compiler infrastructure for approximation [12], software techniques still require extensive programmer intervention. This leaves practitioners looking to exploit approximate computing with no way to easily evaluate the potential of approximation in their applications.

This paper makes two contributions to mapping and modeling the state of the field of approximate computing. First, we present a comprehensive taxonomy of approximate-computing techniques. To the best of our knowledge, ours is the first classification of the nascent approximate-computing literature. We identify the key dimensions in the space of approximate-computing techniques: hardware and software, granularity, determin-

ism, and the computational resources affected. We analyze these dimensions and the benefits and properties they can expose to researchers and practitioners. Highlighting under-explored regions of the space, we identify promising avenues for future work in approximation.

Based on our taxonomy, we propose REACT, a framework for rapid evaluation of approximate-computing techniques. REACT integrates energy models for approximation techniques with dynamic profiling and error-injection information for a user-provided application. For programmers looking to apply approximation, REACT asks only for a small investment—annotating their application once using the ACCEPT language [12]—and provides a framework to explore many approximation opportunities at both the hardware and software levels. For researchers, REACT provides rapid design space exploration. Researchers specify energy and error models for their new technique, and then can explore its effectiveness on applications without extensive implementation effort. We use the taxonomy of approximation techniques to identify the important variables our models must expose to be useful and accurate in research and in practice.

The REACT framework evaluates the energy savings and quality degradation of approximation opportunities using a first-order energy model, coupled to a customizable dynamic error injector. The custom energy model captures the effects of system components most relevant to approximate computing. The energy model's input is a dynamic application profile collected using a custom Pin tool [13]. REACT uses this model to estimate energy consumption for both precise and approximate executions of a provided application. For approximate executions, the energy model incorporates savings characterizations from a library of approximation techniques (which researchers can extend with new techniques). Our energy model's baseline mode produces energy estimates within 1% of McPAT [14], a widely used power modeling tool.

REACT performs error injection to model quality degradation in applications. Programmers provide applications with language annotations to specify where approximations are allowed. We extend an existing approximate compiler framework to simulate errors in data and code deemed approximate. The compiler instrumentation injects errors at run time according to characterizations from a library of approximation techniques (which, again, can be extended with new techniques). The error injection framework operates at the instruction granularity and, by calling a user-defined hook, can support arbitrarily complex error models. REACT also supports error injection at the function level, allowing accelerators and other coarse-grained techniques to be applied to the program.

To demonstrate the utility of REACT for rapidly evaluating approximation techniques, we implement a selection of techniques from our taxonomy and apply them to a suite of approximate-computing benchmarks. We find that each benchmark is amenable to different approximation techniques, and that the overall effectiveness of approximation varies widely between benchmarks. At a high level, we find that the benefits of coarse-grain techniques dominate fine-grain ones. We characterize the low effort required to develop technique models and to annotate benchmark suites with ACCEPT.

	Nondeterministic	Deterministic
Fine Grained	DRAM Refresh Rate [3, 15] SRAM Soft Error Exposure [2, 16] Soft Fault Tolerance [17, 18, 19] Synchronization Elision [9, 20, 21] Voltage Overscaling (ALU) [15, 6]	Bit-Width Reduction [22] Float-to-Fixed Conversion [23] Fuzzy Memoization [24] Hierarchical FPU [1] Load Value Approximation [25, 26] Lossy Compression and Data Packing [27] Precision Scaling (ALU) [5] Reduced-Precision FPU [15, 28] Underdesigned Multiplier [29]
Coarse Grained	Error-Prone Processors [30, 31, 32] Neural Acceleration (Analog) [33]	Algorithm Selection [34, 35] Code Perforation [8] Interpolated Memoization [36] Neural Acceleration (ASIC) [11] Neural Acceleration (FPGA) [4] Neural Acceleration (GPU) [37] Parallel Pattern Replacement [38] Parameter Adjustment [39]

Figure 1: Classification of approximation techniques among the two key dimensions.

2. TAXONOMY OF APPROXIMATIONS

Approximate computing encompasses a broad spectrum of techniques that relax accuracy to improve efficiency. Although the term is new, the principle is not: floating-point numbers, for example, efficiently but approximately represent the real numbers in the digital domain. Efficiency–accuracy trade-offs are also commonplace in digital signal processing applications, where techniques such as quantization and decimation are crucial for tractable designs. Any lossy compression scheme trades off quality for space savings.

The diversity of approaches can complicate discussions on the topic and obscure common patterns. A single monolithic category, *approximate computing*, that spans ideas as disparate as floating-point numbers, voltage overscaling [40], and loop perforation [8] is too broad to establish the general principles of the field. The literature needs finer distinctions within the umbrella term *approximate computing* to help define better abstractions. Which techniques are appropriate for which kinds of applications? Which mechanisms are amenable to different strategies for establishing quality guarantees? How do applications communicate their intent to a cornucopia of approximate software and hardware?

This section outlines a taxonomy of current research in approximate computing. The goals are to analyze the different dimensions that define the space of techniques, to relate them to applications, and to identify potential for future research in unpopulated regions. The taxonomy also defines the breadth of the interface REACT needs to provide to model the full space of approximation techniques.

Software Technique	Granularity	Deterministic	Computational Resource(s)
Algorithm Selection [34, 35]	Coarse	Yes	Compute, Storage, Communication
Bit-Width Reduction [22]	Fine	Yes	Compute
Code Perforation [8]	Coarse	Yes	Compute, Communication
Float-to-Fixed Conversion [23]	Fine	Yes	Compute
Lossy Compression and Data Packing [27]	Fine	Yes	Communication
Neural Acceleration (GPU) [37]	Coarse	Yes	Compute
Parallel Pattern Replacement [38]	Coarse	Yes	Compute, Communication
Parameter Adjustment [39]	Coarse	Yes	Compute, Storage, Communication
Synchronization Elision [9, 20, 21]	Fine	No	Communication
Hardware Technique	**Granularity**	**Deterministic**	**Computational Resource(s)**
DRAM Refresh Rate [3, 15]	Fine	No	Storage
Error-Prone Processors [30, 31, 32]	Coarse	No	Compute
Fuzzy Memoization [24]	Fine	Yes	Compute
Hierarchical FPU [1]	Fine	Yes	Compute
Interpolated Memoization [36]	Coarse	Yes	Compute, Storage
Load Value Approximation [25, 26]	Fine	Yes	Storage, Communication
Neural Acceleration (ASIC) [11]	Coarse	Yes	Compute, Storage, Communication
Neural Acceleration (Analog) [33]	Coarse	No	Compute, Storage, Communication
Neural Acceleration (FPGA) [4]	Coarse	Yes	Compute, Storage, Communication
Precision Scaling (ALU) [5]	Fine	Yes	Compute
Reduced-Precision FPU [15, 28]	Fine	Yes	Compute
SRAM Soft Error Exposure [2, 16]	Fine	No	Storage
Soft Fault Tolerance [17, 18, 19]	Fine	No	Compute, Storage
Underdesigned Multiplier [29]	Fine	Yes	Compute
Voltage Overscaling (ALU) [15, 6]	Fine	No	Compute

Table 1: Taxonomy of approximation techniques from the literature.

2.1 Dimensions

Table 1 enumerates the techniques in the literature and the dimensions in our taxonomy, and Figure 1 shows the same set of techniques laid out along the two dimensions we considered most important: determinism and granularity. We detail each dimension below.

Hardware vs. Software

One basic distinction categorizes techniques by whether or not they require new hardware. Software techniques, such as reducing the width of numerical representations [22], can work on today's commodity hardware. Hardware techniques, such as those that exploit analog circuits [33], can unlock new opportunities for resource savings that are unavailable when the hardware–software interface is left unchanged. Hardware- and algorithm-level approximation are not mutually exclusive: some of the literature's most promising techniques combine algorithmic transformations with new hardware support [11].

Research on hardware techniques requires new ISA support. Crucially, work has demonstrated that hardware abstractions need to expose approximation as an *opt-in* property: imprecision should only be allowed when the programmer and compiler explicitly allow it. This requires that approximations be configurable at some granularity: instructions [25, 26], memory address ranges [3], or dynamic code regions [41], for example. (See the *Granularity* dimension below.) Making approximation optional is critical because even approximate applications tend to need some precise data and computation [15].

Determinism

Approximations can be either deterministic or nondeterministic. Floating-point and digital neural acceleration [11] are examples of deterministic approximations. Their output is always the same for a given input. Conversely, nondeterministic approximations such as voltage overscaling [40] can produce different errors for the same input.

Determinism plays an important role in the feasibility of reasoning about and debugging approximations. Deterministic approximations can make bugs more reproducible, but they do not eliminate all testing challenges: catastrophic cancellation in floating point numbers, for example, can be difficult to discover at testing time. Nondeterministic approximations that follow statistical distributions are suitable for program analyses that reason about probabilistic program behavior [42, 43].

The kinds of errors that arise from deterministic versus nondeterministic techniques can look very different. Many deterministic approximations, such as floating-point rounding errors, are small but frequent, while nondeterministic approaches, such as voltage overscaling, yield infrequent but arbitrarily large errors.

Determinism defines the kinds of accuracy guarantees that are possible. For a deterministic approximate program, it is possible to prove a hard bound: a statement of the form *for any input, the output can be at most ϵ from the correct output.* This kind of guarantee is important in the context of long-running computations, such as scientific computing applications, where errors can compound catastrophically. A nondeterministic program can only offer a statistical bound: *for any input, the probability that the output exceeds a bound is less than P.* This weaker kind of guarantee is appropriate in contexts that compute many independent outputs: pixels, video frames, or particle positions, for example.

Granularity

We classify a technique as *fine-grained* if it applies to individual memory locations, instructions, or data packets. A technique is *coarse-grained* if it affects entire blocks, functions, or data structures.

An approximation technique's granularity is a critical factor in its generality and efficiency gains. A fine-grained approximation, such as one that applies to individual arithmetic instructions [40], can be very general: it can potentially apply to any multiplication in a program. But the efficiency gains are fundamentally limited to *non-control* components, since control errors can disrupt execution arbitrarily. Even if an approximate multiplier unit can be very efficient, the same technique can never improve the efficiency of a branch, an address calculation, or even the scheduling of an approximate multiply instruction. Approximations that work at a coarser granularity can apply holistically to entire algorithms, so their potential gains are larger. But these techniques tend to apply more narrowly: neural acceleration [11], for example, can replace large subcomputations but only when they meet a list of restrictions.

Computational Resource

Approximate computing techniques generally offer benefits in resource efficiency. The resource they benefit can vary: storage techniques can provide better density, for example, and CPU optimizations can improve performance and energy. We classify techniques according to the three classical resources in computer systems: *compute*, *storage*, and *communication*.

A given technique may affect any or all of these resources. The boundaries between the resources are often fluid: an optimization that reduces the width of floating-point numbers reduces the storage footprint, for example, but the same change intrinsically reduces the bandwidth demands between the processor and memory.

2.2 Discussion

The taxonomy's categorization reveals several insights about the landscape of current approximate-computing research. Figure 1 highlights two important dimensions, *determinism* and *granularity*, to visualize the population of the technique space.

Many approximation techniques are both deterministic and fine grained, but we find that benefits of such techniques might be limited (Section 4). Instruction control takes a significant fraction of energy and is typically not approximable. This finding suggests further effort for making control approximate will be fruitful. For example, one important direction isolates the effects of a whole processor, which permits arbitrarily bad control signals while preventing them from compromising the overall system [31].

The coarse-grain, deterministic quadrant is populated by several techniques characterized by their *algorithmic* approximation. These techniques can be realized in either hardware or software (e.g., interpolated memoization, alternative algorithm selection, etc.). This points to opportunities in designing approximate accelerators, whose energy efficiency benefits come from both approximation as well as specialization. This has potential because approximation enables accelerator design to exploit a broader set execution models [11] and optimizations.

The coarse-grain, nondeterministic quadrant is underpopulated. This is where approximation is likely to lead to the most energy benefits, as the techniques can both reduce the cost of control and avert the digital abstraction tax. Analog NPUs [33] are an important example. However, this quadrant is also where the risk of approximation is highest—controlling quality is challenging and there is more technological uncertainty in being able to design effective mixed-signal systems.

3. FRAMEWORK

With the diversity of techniques for approximate computing, it is difficult to evaluate which approaches are the most beneficial for a particular application. It is doubly challenging to predict how multiple techniques will interact. For example, what do the probabilistic, analog errors of voltage overscaling have in common with deterministic errors of reduced precision computation? How do we compare loop perforation of a code region in software to offloading all or a portion of that region to a neural network accelerator? The challenges of understanding energy and error trade-offs of different approximation techniques has motivated us to develop REACT, a tool for researchers and practitioners to rapidly and accurately explore approximate computing techniques. In this section, we present the details of REACT and how it enables understanding of the taxonomy dimensions.

REACT consists of an application profiler, an energy model, and an error analysis tool. We implement a custom profiler and linear energy model, and extend ACCEPT [12], an approximate computing compiler framework, to inject errors and measure quality. Figure 2 provides a high level overview of REACT's workflow. We logically divide the flow into an energy path and a quality path.

The energy path uses a dynamic application profiler to record architectural events, such as instruction categories, and micro-architectural events, such as cache

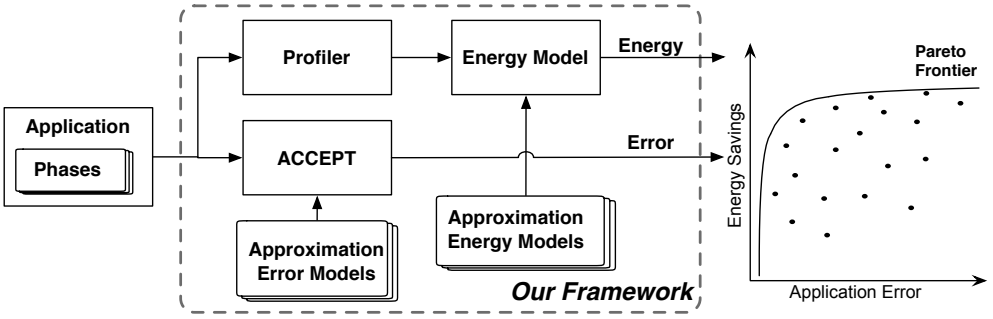


Figure 2: High level overview of REACT.

misses and branch mispredictions. We then feed this profile information into a linear energy model to estimate total energy consumption. The energy model incorporates the effects of each approximation technique to produce an estimate of energy savings. Sections 3.1 and 3.2 details the energy model.

The quality path uses a compiler pass to instrument the code and inject errors. The instrumentation makes calls to a library that implements an error-injection API, where the user can define arbitrary error behavior without modifying the compiler itself. Section 3.4 details this process.

Finally, REACT combines the energy and quality paths to search the efficiency–accuracy trade-off space for a given application. It produces a collection of *configurations* of the program and applies the modeled approximation techniques to different phases in the program. Section 3.5 describes the interaction between the two paths.

3.1 Application Profiling

The application profiler in REACT collects performance statistics, which are the inputs to the energy model. The profiler is implemented using Intel's Pin [13] dynamic binary instrumentation tool. It observes x86 instructions, from which we extract memory operations to produce a load/store instruction set. REACT's Pin tool executes programs with minimal overhead but collects only first-order architectural statistics—a trade-off with respect to full-system simulators that lets REACT quickly run full applications many times on large input sets.

The profiler groups the micro-operations into categories for the energy model. Table 2 enumerates these categories, which we selected while concurrently developing the energy model, error injection tool, and approximation techniques taxonomy to provide flexibility, simplicity, and accomodate for approximation strategies.

Using the instruction stream from the its front end, the profiler uses simple architectural models to collect micro-architectural statistics. It models a branch predictor and a memory hierarchy representative of a mid-range, low power processor, in the same class as an ARM Cortex-A9. We model a tournament-style branch predictor with a 4096-entry selection table, 12-bit Global History Register (GHR), 4096 entry Pattern History Table (PHT), and a 4096 entry Branch History Table (BHT) of local predictors. The data cache model mirrors the Cortex A9's parameters: a 4-way set associative, 32 KB L1 cache with 32 B blocks. We do not model an L2 cache. Together, these models provide the micro-architectural statistics—hits, misses, predictions, and mispredictions—that the energy model uses.

Category	Description
alusimple	Integer +, -, bitwise
alucomplex	Integer *, /, sqrt
fpusimple	Floating point +, -
fpucomplex	Floating point *, /, sqrt
branch.correct	Correct branch prediction
branch.mispredict	Branch mispredictions
l1d.hit	L1 D-cache access hit
l1d.miss	L1 D-cache access miss
other	All remaining instructions

Table 2: Application Profile Instruction Categories

3.2 Energy Model

REACT uses a simplified linear energy model to estimate the total energy consumption of an execution. Figure 3 provides the overview of our energy model. The model has three components for static power: core, L1 data cache, and DRAM memory. For dynamic power, we assign a fixed cost to micro-architectural and architectural events, each indicated by a gray shaded box in the figure. The linear energy model accommodates a variety of approximation strategies that scale its coefficients: for example, by reducing the dynamic power of multiply operations. REACT uses the model's outputs—one precise energy total, and one scaled approximate energy total—to estimate energy savings from executing a program approximately in comparison to the precise execution baseline.

Precise Baseline

We first describe the parameterized linear energy model. Section 3.3, below, describes our methodology for deriving the parameters from a more complex energy model.

The energy model computes the precise baseline cost

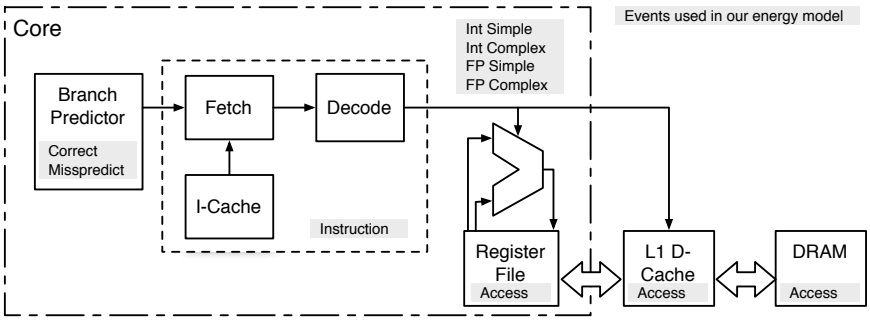


Figure 3: Overview of our energy model. The gray shaded boxes indicate events that are used by our model.

using the following linear equations.

$$Energy_{phase} = Static_{compute} + Dynamic_{compute} + Static_{memory} + Dynamic_{memory} \quad (1)$$

$$Static = Power \times CPI \times Instructions \quad (2)$$

$$Dynamic = Energy_{event} \times Count_{event} \quad (3)$$

In these equations, the *Power* and *Energy* terms are architecture-dependent parameters defining the cost of various operations or structures. *CPI* is the average cycles per instruction for the target architecture. The *Count* terms are properties of the program's dynamic profile and correspond to the events shown in Figure 3 (e.g., arithmetic operations and memory accesses).

Equation (1) computes the energy cost of a phase as the sum of the static and dynamic costs for compute and memory within the system. The total precise baseline cost is then the sum of Equation (1) over all phases of application execution.

Fine-Grained Approximations

Fine-grained approximation techniques operate at the event level, replacing individual operations with approximate versions. In REACT, the model for a fine-grained technique modifies one or more of the architectural parameters of the baseline system. For example, a model might reduce the energy consumption of integer arithmetic operations to model approximate arithmetic instructions. The energy cost of a fine-grained approximation is computed in the same way as the precise baseline cost, but using the appropriate modified architectural parameters. Multiple orthogonal fine-grained approximations can be aggregated into a single set of modified parameters.

Coarse-Grained Approximations

Coarse-grained approximation techniques replace entire application phases with approximate implementations. In REACT, we restrict such phases to functions. Our energy model for a coarse-grained technique modifies one or more of the terms on the right-hand side of Equation (1), and adds additional cost specific to the coarse-grained technique. To compute total energy of a coarse-grained phase, we first compute the precise baseline cost of the phase. Then, each term affected by the coarse-grained approximation is scaled by a user-specified factor (e.g., for placing the processor in a low-power mode). We also account for speedup (or slowdown) afforded by the technique, and prorate the static costs accordingly. Lastly, we introduce the technique specific dynamic invocation and static costs. The dynamic invocation cost is the number of invocations of the region multiplied by a per invocation cost. The static cost is computed in the same manner as Equation (2).

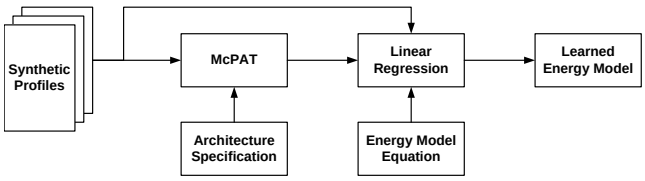


Figure 4: The coefficients in REACT's energy model are learned through linear regression using synthetically generated application profiles.

3.3 Energy Model Training & Validation

To make the abstract model above concrete, we need to calibrate its parameters. While the focus on first-order terms enables rapid exploration of energy savings and provides a simple interface to the tool, it potentially sacrifices the accuracy attainable using previous energy and power modeling research. To minimize modeling inaccuracies, we ground our model using McPAT [14], a widely used power modeling framework.

Figure 4 shows the workflow employed to learn the coefficients of our linear model. We first analyzed program traces to formulate a generative model that produces realistic synthetic program profiles. These synthetic profiles contain the same high-level micro-architectural events as detected by our profiler. In order to derive the micro-architectural statistics (e.g., BTB reads and writes, RAT and CDB access, etc.) required by McPAT from these synthetic profiles, we use conversion paramters found in popular performance simulators such as Gem5 [44] and Sniper [45]. The micro-architectural statistics complemented with a single-core ARM Cortex-A9 architecture description are passed into McPAT to

Energy Model Parameter	Cost
Dynamic Cost Per Event	
Instruction (fetch, decode, ...)	60.33 pJ
Simple ALU access	19.48 pJ
Complex ALU access	29.14 pJ
Simple FPU access	34.27 pJ
Complex FPU access	34.27 pJ
Branch (correctly predicted)	14.80 pJ
Branch (mispredict)	17.15 pJ
Register file access	3.87 pJ
DRAM memory access	171.70 pJ
L1 D-cache access	34.34 pJ
Static Cost	
Core (excluding register file)	44.43 mW
L1 D-cache	4.86 mW
DRAM memory	44.01 mW

Table 3: Learned Architectural Parameters for the ARM Cortex-A9 Class Processor.

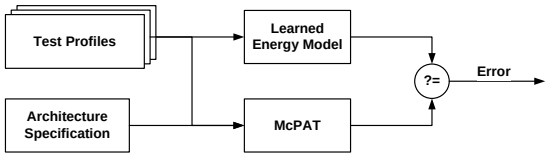


Figure 5: Energy model validation flow.

produce power results. McPAT's Cortex-A9 processor description itself has been validated against published data [46].

We derive an energy cost from the McPAT power output obtained from each program trace, the CPI and frequency, both specific to the ARM Cortex-A9 processor. In order to derive the coefficients for the dynamic elements of our energy model, we run linear regression on the synthetic profile data and the energy output produced by McPAT. Collecting the static costs is simpler since those are program-independent and directly produced by McPAT.

The resulting model provides the *Energy* and *Power* terms used in Equations (2)–(3) for the processor and cache, but not for DRAM. We assume a DRAM access costs 5 pJ per bit for a 64-bit access [47], plus and additional penalty of one hundred stalled cycles on the CPU. We assume DRAM static power to be on the same order of magnitude of processor static power, informed by previous results [48].

Table 3 lists the final learned parameters. We validate the calibrated model against McPAT on real program traces. We collect traces from the PERFECT [49] and PARSEC-3.0 [50] benchmark suites, generate energy values using McPAT and REACT, then compare the results. Overall, we observe an average total energy error of 0.87%, ranging from 0.31% to 1.38%, where our model consistently calculates a slightly greater energy value that McPAT. These results lead us to conclude that our energy model, which focuses on only first-order concerns, is a reasonable simplification of the reference model.

3.4 Quality Modeling

REACT also needs to model the quality impacts of approximation. To do so, it instruments programs to inject errors at execution time and provides an API for defining the errors to inject. REACT's instrumentation extends ACCEPT [12], an approximate-computing compiler framework, to inject errors at two granularities: at a fine granularity after each instruction, and at a coarse granularity on the output of functions. Users can implement REACT's error API to supply error models without modifying the compiler instrumentation.

This section details each error injection style.

Fine-Grained Error Injection

Fine-grained error injection occurs at the instruction level in REACT. As with EnerJ [15], values are marked as either *approximate* or *precise*, with precise being the implicit default. For instance, the program snippet:

```
APPROX int a;
int b;
```

defines an approximate integer variable `a` and a precise integer variable `b`. When compiled with ACCEPT, all instructions manipulating or depending solely on approximate values (e.g., a store to `a`) are marked as being *approximate* and intercepted with a hook to a user-defined error injection routine. At runtime, that hook receives information about the dynamic execution of the instruction, including opcode and operands. The hook invokes the user-defined error routine to modify the instruction and inject error appropriately. All approximate instructions within a given function are subjected to the same error injection routine, but the particular routine used may differ across functions.

Much like a conventional architectural simulator, the error injection models can be arbitrarily complex, and rely on an arbitrarily large amount of bookkeeping, since they are only intended to *model* the approximation techniques. For example, we implement Load Value Approximation [25], a complex technique that relies on logging the history of loads following a cache miss to avoid fetching data from memory by predicting the next load data value. REACT's expressiveness allows simulating error in both ALU operations and memory.

Coarse-Grained Error Injection

ACCEPT has also been modified to allow coarse-grained error injection at the function level. In this approach, an injection routine is specified to modify the outputs of a function, i.e. the live-outs of that function. The user-specified routine should be representative of the particular approximation technique it represents, and ACCEPT imposes no restrictions on the type of error introduced. For instance, when implementing neural acceleration, a user may choose to evaluate a neural network, or instead sample a probability distribution that

produces a similar whole-application quality degradation.

3.5 Linking Energy & Quality

REACT coordinates the energy and quality paths by applying corresponding sets of energy and error models to each phase in the program. In the current implementation, "phases" are defined by function boundaries, so all categories of models—coarse grain and fine grain, energy and error—apply to an entire function at a time.

For fine-grained approximations, REACT makes the simplifying assumption that energy scaling applies uniformly to the statistics for an entire phase. In practice, this means that some precise instructions, where error is not injected, are nonetheless counted as approximate in the energy model. We accept this simplification in the energy results in favor of an easily understandable and usable energy model.

4. EVALUATION

In this section, we evaluate REACT's usefulness as a tool for exploring approximate-computing techniques. We implement a selection of techniques from the taxonomy presented in Section 2 within REACT's infrastructure and apply them to a collection of benchmarks. The goals of this evaluation are to demonstrate the application of a broad range of techniques within REACT, and to compare the potential benefits of different approaches.

4.1 Techniques and Benchmarks

Table 4 lists the approximation techniques we model, which include a variety of fine-grained and coarse-grained hardware mechanisms. We list the components of the energy model affected and the type of error model required for each technique. The fine-grained techniques we consider are reduced-precision floating-point computation [28], voltage overscaled ALU units [51], and unreliable DRAM [3] and SRAM [2] arrays. We implement models for neural accelerators [11, 33], representative of coarse-grained techniques from literature. In addition, we model special-purpose, spatially laid-out accelerators for three benchmarks (*blackscholes*, *jpeg*, and *sobel*). We model both precise and approximate versions of these spatial accelerators to contrast the benefits of specialization and approximation.

We evaluate benchmarks written in C and C++ drawn from two benchmark suites: three applications from the ACCEPT approximate compiler [12] (*sobel*, *blackscholes*, and *jpeg*) and two from the PERFECT suite [49] (*histogram-equalization* and *fft-1d*). Both suites include output quality metrics; the accuracy measurements in our evaluation use these specifications. Each benchmark uses lightweight type annotations in the style of ACCEPT [12] and EnerJ [15] to distinguish approximate code from critical, non-approximatable code; Section 4.5 reports on the effort required to add these annotations.

Specialized Accelerator Models

As described in Section 3.2, coarse-grained accelerators incur both a dynamic per invocation and static cost. To calculate these costs for our models of custom fixed-function accelerators, we first count the number of arithmetic operations, cache accesses, and register accesses in the accelerated region of code. The dynamic cost is the sum of these counts multiplied by a per-event cost. Precise accelerators use the same per-operation cost as the baseline precise energy model, scaled by the width of the operation. Approximate accelerators incorporate approximate hardware components by replacing precise ALUs with voltage-overscaled ALUs, and floating-point hardware with reduced-precision floating-point units. In addition we assume the use of low-refresh rate DRAM, while cache and local register accesses are precise. Similarly, the static power cost is the sum of the arithmetic unit count multiplied by a per-unit power cost, plus the cost of a register file. We derive the power costs of each structure from CACTI [52].

4.2 Overview of Results

Figure 6 displays REACT's output for each benchmark. The plots show error–savings trade-off spaces. Each point represents a *configuration* of the application, which composes REACT's simulated techniques in different ways and at different levels of aggressiveness. In these configurations, each technique applies globally at a uniform level to all approximate code in the benchmark. The energy axis reflects our model for the approximate region of interest for each program, and the error axis shows the application-specific output quality metric.

One of REACT's goals is to enable researchers and practitioners to produce this type of trade-off plot. From these plots, we can examine the optimal configurations that are on the Pareto frontier. Across all benchmarks, we observe energy savings from none to nearly 100% and quality degradation on the application output from none to unacceptable. For example, Figure 6d, shows how *fft-1d* is not amenable to most of the approximation techniques we explored. This result does not mean approximation should not be applied to FFT routines, but rather suggests that these *particular* techniques are not a good choice. These widely varying results allow us to examine the benefits from approximate computing along many dimensions, including granularity and specialization. We discuss these observations in the following sections.

Figure 7 shows the breakdown of energy consumption by component for precise, fine-grained, and coarse-grained approximations. As with the results in Figure 6, we examine the energy consumed during the approximation amenable regions of each benchmark. For the fine- and coarse-grained bars, we select the approximate techniques exhibiting the greatest energy savings with at most 10% error. We omit coarse-grained techniques for *fft-1d* and *histeq*, as we did not implement coarse-grained models for these techniques. Benchmarks from the same suite exhibit similar energy breakdown by

Technique	Energy Terms Affected	Error Model Description
Reduced-Precision FPU [28]	FP Arithmetic Instructions	Reduced Mantissa Floating Point
Voltage Overscaling (ALU) [6]	Int Arithmetic Instructions	Random Bit Flips
DRAM Refresh Rate [51, 3]	$\text{Memory}_{\text{St}}$	Last-Access Dependent Bit Flip
Load Value Approximation [25]	Memory Access Energy	Load Value Predictor Model
Neural Acceleration [11, 33]	$\text{Compute}_{\text{Dyn/St}}$ & $\text{Memory}_{\text{Dyn}}$	Per-Invocation Random Error
Spatial Sobel Kernel	$\text{Compute}_{\text{Dyn/St}}$ & $\text{Memory}_{\text{Dyn}}$	Reduced Mantissa Floating Point
Spatial JPEG Kernel	$\text{Compute}_{\text{Dyn/St}}$ & $\text{Memory}_{\text{Dyn}}$	Reduced Mantissa Floating Point
Spatial Blackscholes Kernel	$\text{Compute}_{\text{Dyn/St}}$ & $\text{Memory}_{\text{Dyn}}$	Reduced Mantissa Floating Point

Table 4: Selected approximation techniques implemented in REACT. "Dyn" indicates that the dynamic energy of the component is affected, while "St" indicates an effect on the static energy.

component: the benchmarks from the PERFECT suite tend to be more memory intensive. We suspect this is due to the selection of approximate regions in each benchmark.

4.3 Fine- vs. Coarse-Grained Approximations

REACT's results let us examine the potential of approximation techniques along each dimension in our taxonomy. The *granularity* dimension yields a stark contrast in efficiency. Previous work [4, 11, 33] has observed that coarse-grained accelerators benefit from their ability to reduce the costly control overhead of general purpose processors.

Our results confirm this trend. Figures 6a–6c show the results for the three benchmarks where we applied coarse-grained techniques. In each plot, we observe a group of configurations demonstrating energy savings below around 20%, and a small number above 50%. All of the fine-grained techniques are in the former group, and all of the coarse-grained techniques are in the latter group. On average, the coarse-grained techniques achieve energy savings of 74%, while the average for the fine-grained techniques is only 7%.

4.4 Specialization and Acceleration

Our results also reveal conclusions surrounding the coupling of approximation and specialization. REACT includes models for approximate and precise versions of custom accelerators for *blackscholes*, *jpeg*, and *sobel*. On average, our precise spatial accelerators achieve energy gains of 69% while the approximate versions demonstrate gains of 80%, or 35% savings over the precise spatial accelerator. These results suggest that most of the benefit of these spatially laid-out accelerators comes from specialization, not approximation. That said, applying approximation on top of specialization can enable significant additional energy efficiency, ranging from 30% to 36%, at the cost of an average 12% application error. Comparatively, the same approximation strategy applied to traditional CPU designs yields an average 13% energy savings on the same benchmarks. This indicates that approximation techniques are more effective on fixed-function accelerators compared to CPU designs. This is due to the large control overheads traditionally found in CPUs that are incompressible when applying approximation techniques.

Benchmark	APPROX	ENDORSE
sobel	6	2
blackscholes	50	10
jpeg	3	9
fft	12	5
histogram-equalization	17	3

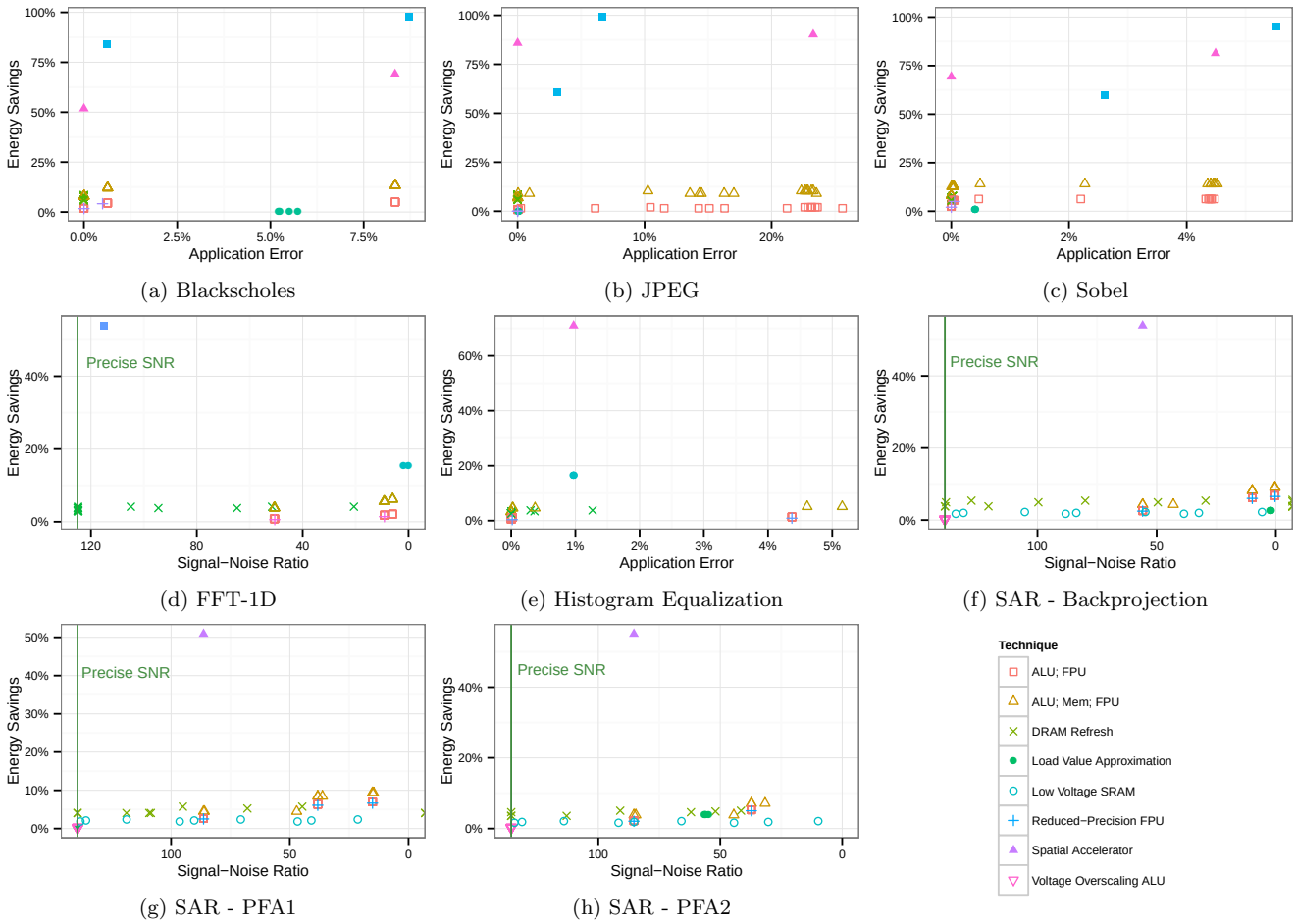
Table 5: Annotation counts for evaluated benchmarks.

In addition to fixed-function accelerators, we evaluate digital and analog variants of neural acceleration. The data in Figures 6a–6c demonstrate the power of neural-network–based approximate accelerators. Both the analog and digital neural network models demonstrate greater energy savings and lower quality degradation than the approximate spatial accelerators. The analog neural networks achieve energy savings of 97% on average with only 7% average error and the digital neural networks have average energy savings of 68% with an average error of only 2%. The results strongly suggest that coarse-grained techniques carry great potential when they are coupled with algorithmic transformations.

4.5 Programmer Effort

Programmer effort is an important factor in the feasibility of approximate computing. To measure effort, we quantify the annotation overhead for REACT's annotations, which are based on the type qualifier system of EnerJ [15] and ACCEPT [12]. Table 5 lists the number of source-code annotations for each benchmark. In the type system, *APPROX* mark data that is safe to approximate and *ENDORSE* denotes points where data transitions from approximate to precise. Overall, a small number of annotations are required to make REACT aware of approximation opportunities in each benchmark. *Blackscholes* requires the greatest number of modifications: 50 *APPROX* qualifiers and 10 *ENDORSE* markers.

The framework was useful in debugging the approximation annotations. REACT reveals unexpected behavior from programs, e.g. segmentation faults. These reports led us to fix the annotations for *sobel* and *histo-*



(a) Blackscholes (b) JPEG (c) Sobel

(d) FFT-1D (e) Histogram Equalization (f) SAR - Backprojection

(g) SAR - PFA1 (h) SAR - PFA2

Figure 6: Efficiency–accuracy trade-offs for eight benchmarks as explored by REACT using the techniques in Table 4.

gram-equalization so that they ran with gradual quality degradation but without catastrophic failure.

4.6 PERFECT Benchmarks

To further explore the benefits of approximation, we discuss the application of approximate computing to the PERFECT benchmark suite. Figures 6d–6h display the results of applying approximation to a selection of kernels from the PERFECT suite. PERFECT is composed of a set of required kernels and four sets of domain specific kernels. One of the domains involves image processing while the other three are focused on radar data processing. The *fft-1d* kernel covers the required domain, *histogram-equalization* represents the image processing domain, and we use the *Synthetic Aperture Radar (SAR)* kernels to represent the radar data processing domains.

The PERFECT suite focuses on image and signal processing applications common on embedded computing platforms. These classes of applications are generally considered amenable to approximation, which is confirmed by our results. We observe energy savings greater than 50% across the selected PERFECT kernels when applying coarse-grained approximation techniques.

Key Takeaways

In studying the PERFECT kernels, we formulate a few key takeaways regarding approximate computing.

First, coarse-grained approximation techniques vastly outperform fine-grained techniques, offering significant energy-efficiency gains at similar error behavior to fine-grained techinques. As noted in our analysis of other benchmarks, coarse-grained techniques benefit from both specialization and approximation. The combination of these approaches eliminate inefficiences such as control overhead found in traditional, general-purpose processors and exploit the relaxed accuracy and precision requirements of signal processing applications.

Second, applying approximation requires a deep understanding of the algorithm in question. The *fft-1d* kernel, shown in Figure 6d, exhibits what can happen when approximation is not applied properly. Most of the techniques applied to this application resulted in unacceptable output quality, well below the acceptable SNR of 100 suggested by the documentation. This behavior potentially implies the application is not amenable to approximation, the approximations applied were the wrong type of approximation for this application, or the approximations were applied at the wrong locations

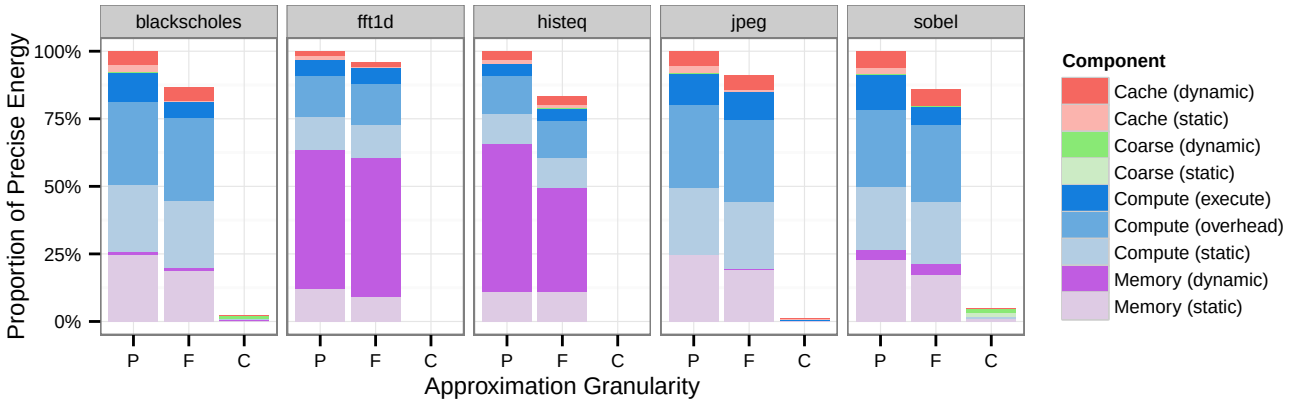


Figure 7: Component-wise energy breakdown for the Precise and best Fine-grained and Coarse-grained executions. We define best as the approximation technique exhibiting the greatest energy savings with at most 10% error.

within the application. In the case of *fft-1d*, we know from previous work that coarse-grained approximation via neural acceleration is profitable, providing energy savings of around 2x [4] at negligible quality degradation. The *SAR* kernels also exhibit potentially unacceptable error behavior, but show a moderately more graceful degradation of quality. We suspect this behavior is a result of the provided implemenations already being finely tuned for accuracy and precision requirements. Algorithms that have already been tuned for precision may be more sensitive to approximate computing techniques. A comprehensive understanding of the application can unlock large energy savings while lack of algorithmic knowledge may imply a false barrier for approximation.

Third, we expect the fraction of energy expended on memory and communication of data to increase in the future as applications continue to focus on data intensive algorithms. Examining Figure 7, we see energy consumption from memory associated costs, which includes communication in our model, accounting for more than half of the total energy consumed. This data implies that memory and communication costs are prime targets for approximation through techniques, such as low refresh rate DRAM, low voltage SRAM, or lossy compression, and significant energy-efficiency gains are possible.

5. RELATED WORK

This paper builds on the expansive body of work on techniques for approximate computing. Rather than reiterate the individual techniques here, we refer the reader to Section 2 for the exhaustive listing in our taxonomy.

Other recent works have focused on tools for exploring and prototyping approximate-computing techniques. The *quality-of-service profiler* by Misailovic et al. [53] automatically explores programs for code-perforation opportunities, and Ringenburg et al. [54] describe a similar system for nondeterministic hardware approximation. The goal in both systems is to mine recommendations for approximation and report these to the programmer. Intel's iACT [36] is compiler and runtime toolbox for exploring simulated approximations. Chippa et al. [55] describe a framework for measuring applications' "inherent" resilience and, thereby, their potential for approximation.

The REACT framework's trade-off space exploration resembles other systems for auto-tuning approximate programs, including ACCEPT [12], ExpAX [56], and Green [34]. Those systems' goal is to produce an optimized version of the program—in contrast to REACT, which is designed for prototyping and exploration.

REACT is related to other work that projects the potential benefit of new hardware before the design phase. The Aladdin framework [57], for example, models the power–performance design space for hardware accelerators derived directly from C code.

REACT uses a machine-learning approach to derive an analytical, linear model for system energy consumption. This approach builds on two bodies of work that models power in real systems using performance counters as a proxy [58, 59, 60], and recent micro-architecture independent analytical modeling [61].

6. CONCLUSION

While approximate computing promises to open new doors for advancing computational efficiency, its potential benefits and impact on applications are difficult to measure. Researchers and practitioners alike need tools to prototype new approximate-computing ideas and to explore their interaction with software.

This work details REACT, a framework for the rapid exploration of approximate-computing techniques. To inform REACT's design, we present a comprehensive taxonomy of current approximate computing research. We develop an energy model that can be quickly adapted to reflect arbitrary approximation techniques. We provide a straightforward API for injecting error into and assessing application quality. Together, REACT exposes the right set of knobs for prototyping the energy and error characteristics of new approaches to approxi-

mation before proceeding to a full hardware or software implementation.

Our taxonomy and our quantitative findings suggest promising areas for future approximate computing research. Coarse-grained techniques lead to far more efficient designs. Non-deterministic, coarse-grained techniques hold great potential but are less explored in the literature. We plan to release the REACT infrastructure as open-source software to let researchers draw new conclusions about novel techniques as the field progresses.

Acknowledgements

This work is supported in part by a DARPA seed grant and the Torode Family Professorship. We thank the Sampa group at UW-CSE for feedback and especially Joe Cross for providing detailed input during the execution of the study.

7. REFERENCES

[1] T. Y. Yeh, P. Faloutsos, M. Ercegovac, S. J. Patel, and G. Reinman, "The art of deception: Adaptive precision reduction for area efficient physics acceleration," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[2] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power," in *International Symposium on Computer Architecture (ISCA)*, 2002.

[3] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving refresh-power in mobile devices through critical data partitioning," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[4] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "SNNAP: Approximate computing on programmable SoCs via neural acceleration," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2015.

[5] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[6] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," in *Design, Automation and Test in Europe (DATE)*, 2010.

[7] B. Grigorian, N. Farahpour, and G. Reinman, "BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2015.

[8] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2011.

[9] L. Renganarayana, S. Vijayalakshmi, R. Nair, and D. Prener, "Programming with relaxed synchronization," in *ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012.

[10] T. Nowatzki, J. Menon, C.-H. Ho, and K. Sankaralingam, "gem5, GPGPUSim, McPAT, GPUWattch, ⟨your favorite simulator here⟩ considered harmful," in *Workshop on Duplicating, Deconstructing and Debunking (WDDD)*, 2014.

[11] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

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

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[14] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[15] 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 *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[16] I. J. Chang, D. Mohapatra, and K. Roy, "A priority-based 6T/8T hybrid SRAM architecture for aggressive voltage scaling in video applications," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 21, no. 2, pp. 101–112, 2011.

[17] A. Sundaram, A. Aakel, D. Lockhart, D. Thaker, and D. Franklin, "Efficient fault tolerance in multi-media applications through selective instruction replication," in *Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies*, 2008.

[18] D. Palframan, N. S. Kim, and M. Lipasti, "Precision-aware soft error protection for GPUs," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2014.

[19] D. S. Khudia and S. Mahlke, "Harnessing soft computations for low-budget fault tolerance," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[20] M. Rinard, "Parallel synchronization-free approximate data structure construction," in *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2013.

[21] S. Misailovic, S. Sidiroglou, and M. C. Rinard, "Dancing with uncertainty," in *ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012.

[22] C. Rubio-Gonzalez, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013.

[23] T. M. Aamodt and P. Chow, "Compile-time and instruction-set methods for improving floating- to fixed-point conversion accuracy," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, 2008.

[24] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions on Computers*, vol. 54, pp. 922 – 927, July 2005.

[25] J. S. Miguel, M. Badr, and N. Enright Jerger, "Load value approximation," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[26] B. Thwaites, G. Pekhimenko, A. Yazdanbakhsh, J. Park, G. Mururu, H. Esmaeilzadeh, O. Mutlu, and T. Mowry, "Rollback-free value prediction with approximate loads," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[27] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics

engines," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[28] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration Systems (VLSI)*, vol. 8, no. 3, 2000.

[29] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *VLSI Design*, 2011.

[30] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *Design, Automation and Test in Europe (DATE)*, 2010.

[31] Y. Yetim, M. Martonosi, and S. Malik, "Extracting useful computation from error-prone processors for streaming applications," in *Design, Automation and Test in Europe (DATE)*, 2013.

[32] Y. Yetim, S. Malik, and M. Martonosi, "CommGuard: Mitigating communication errors in error-prone parallel execution," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

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

[34] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[35] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[36] A. K. Mishra, R. Barik, and S. Paul, "iACT: A software-hardware framework for understanding the scope of approximate computing," in *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[37] B. Grigorian and G. Reinman, "Accelerating divergent applications on simd architectures using neural networks," in *International Conference on Computer Design (ICCD)*, 2014.

[38] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[39] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. C. Rinard, "Dynamic knobs for responsive power-aware computing," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

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

[41] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: an architectural framework for software recovery of hardware faults," in *International Symposium on Computer Architecture (ISCA)*, 2010.

[42] M. Carbin, S. Misailovic, and M. Rinard, "Verifying quantitative reliability of programs that execute on unreliable hardware," in *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[43] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze, "Expressing and verifying probabilistic assertions," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[44] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, 2011.

[45] T. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, 2011.

[46] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing," *ACM Trans. Archit. Code Optim.*, vol. 10, pp. 5:1–5:29, Apr. 2013.

[47] "LPDDR4 moves mobile." Presented by: Skinner, Daniel. Micron Technology, Inc. Mobile Forum 2013.

[48] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *USENIX Annual Technical Conference*, 2010.

[49] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and A. Tumeo, *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013. `http://hpc.pnnl.gov/projects/PERFECT/`.

[50] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[51] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: a low-power pipeline based on circuit-level timing speculation," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

[52] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi, "A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies," in *International Symposium on Computer Architecture (ISCA)*, 2008.

[53] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *International Conference on Software Engineering (ICSE)*, 2010.

[54] M. F. Ringenburg, A. Sampson, L. Ceze, and D. Grossman, "Quality of service profiling and autotuning for energy-aware approximate programming," Tech. Rep. UW-CSE-12-07-02, University of Washington, 2012.

[55] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *DAC*, 2013.

[56] H. Esmaeilzadeh, K. Ni, and M. Naik, "Expectation-oriented framework for automating approximate programming," Tech. Rep. GT-CS-13-07, Georgia Institute of Technology, 2013.

[57] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *International Symposium on Computer Architecture (ISCA)*, 2014.

[58] G. Contreras and M. Martonosi, "Power prediction for Intel XScale processors using performance monitoring unit events," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2005.

[59] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

[60] S. Rivoire, P. Ranganathan, and C. Kozyrakis, "A comparison of high-level full-system power models," in *USENIX Workshop on Power-Aware Computing and Systems (HotPower)*, 2008.

[61] S. Van den Steen, S. De Pestel, M. Mechri, S. Eyerman, T. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout, "Micro-architecture independent analytical processor performance and power modeling," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.