REACT: A Framework for Rapid Exploration of Approximate Computing Techniques

Mark Wyse Andre Baixo Thierry Moreau Bill Zorn
James Bornholt Adrian Sampson Luis Ceze Mark Oskin
University of Washington
{wysem, andreolb, moreau, billzorn, bornholt, asampson, luisceze, oskin}@cs.washington.edu

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. To overcome this complexity and foster innovation in research, we propose REACT, a modeling framework that lets researchers rapidly evaluate approximate-computing techniques and captures the efficiency–accuracy tradeoff created by approximation. We describe the components of REACT and explain how our framework models approximation techniques from the diverse taxonomy of existing research.

1. Motivation

Approximate computing promises significant energy efficiency gains for applications tolerant of relaxed quality guarantees, but accurately estimating these quality-efficiency trade-offs requires significant effort for both researchers and practitioners. Rather than incur the expense of building hardware, researchers exploring new approximation techniques often spend months modifying detailed power modeling tools [2] to gain insights into power and energy savings. Developers looking to exploit approximation opportunities in their programs are often forced to rewrite their program or manually insert error injection code to gain insight into program behavior under relaxed quality guarantees. These burdens constrain the ability of researchers to validate new approximate computing techniques in the design stage, and prevent practitioners from easily adopting approximation in their applications.

We propose REACT, a modeling framework allowing quick and accurate exploration of the efficiency–accuracy trade-off created by approximate computing. In this paper, we detail the components of REACT, discuss validation of those components, and present an abbreviated taxonomy of existing approximation techniques.

2. Modeling Framework

REACT consists of an application profiler, an energy model, and a quality model. We implement a custom profiler and linear energy model, and extend ACCEPT [12], an approximate computing compiler framework, to perform quality modeling via user-directed error injection. Profiling and modeling results are aggregated at the function level, allowing different approximation techniques to be applied to distinct phases of a program.

2.1 Application Profiling

The application profiler in REACT is implemented as a tool within Intel's Pin [7] framework. The profiler observes an x86 instruction stream, from which we extract memory operations and produce a load/store instruction set. The profiler groups instructions into categories for the energy model (see Table 2). Using the instruction stream, the profiler simulates a tournament-style branch predictor and multi-level data cache hierarchy. We aggregate profiling data at the function level, and consider functions to be execution phase boundaries.

2.2 Energy Modeling

REACT uses a simplified linear energy model to estimate the total energy consumption of an application's execution. This model captures the dynamic cost of architectural and micro-architectural events and the static cost of architectural structures. The model's inputs are the application profile, energy models for approximation techniques, and the architectural parameters of the system. REACT uses the model's outputs–one precise energy total, and one approximate energy total–to estimate energy savings under approximation.

Precise Baseline The energy model computes the precise baseline cost using the following linear equations.

$$\begin{aligned} Energy_{phase} = &\\ & Static_{compute} + Dynamic_{compute} \\ & + Static_{memory} + Dynamic_{memory} \end{aligned} \tag{1}$$

$$Static = Power \times CPI \times Instructions \tag{2}$$

$$Dynamic = Energy_{event} \times Count_{event} \tag{3}$$

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 (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.

Technique	Energy Model Category	Error Model Description
DRAM Refresh Rate [1, 6]	$\text{Memory}_{\text{St}}$	Last-Access Dependent Bit Flip
Drowsy Cache [3]	$\text{Memory}_{\text{St}}$	Read Upset/Write Failure Bit Error
Load Value Approximation [8]	Memory Access Energy	Load Value Predictor Model
Neural Acceleration [9]	$\text{Compute}_{\text{Dyn/St}}$ & $\text{Memory}_{\text{Dyn}}$	Per-Invocation Random Error
Reduced-Precision FPU [13]	FP Arithmetic Instructions	Narrow Mantissa Floating Point
Underdesigned Multiplier [4]	Int Multiply Instructions	Per-Invocation Random Error
Voltage Overscaling (ALU) [10]	Int Arithmetic Instructions	Random Bit Flips
Precision Scaling (ALU) [14]	Int Arithmetic Instructions	LSB Zeroing

Table 1. 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.

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

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).

2.2.1 Energy Model Validation

REACT employs a custom linear energy model to estimate the energy savings available through approximation. While this focus enables rapid exploration and provides a simple interface, it potentially sacrifices the accuracy attainable using existing energy and power modeling tools. To minimize inaccuracies, we ground our model using McPAT [5], a widely used power modeling framework. We apply linear regression to learn each of the *Energy* and *Power* terms using in Equations (2)–(3). Overall, we observe an average total energy error of 0.87%, ranging from 0.31% to 1.38%. These results lead us to conclude that our energy model, which focuses on first-order concerns, is a reasonable simplification of the reference model.

2.3 Quality Modeling

REACT performs quality modeling via user-directed error injection. We extend 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.

Fine-Grained Error Injection Fine-grained error injection occurs at the instruction level in REACT. As with EnerJ [11], 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 invokes a user defined error routine. All approximate instructions in a given function are subjected to the same error injection routine, but the particular routine used may differ across functions.

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. Taxonomy of Approximations

Table 1 presents a taxonomy of selected approximation techniques, shows how REACT's energy model captures their effect, and provides a high-level description of the error model. For space, we select only a few techniques found in literature, although our current taxonomy is much larger and continues to grow. Our energy and quality models are designed to capture the effect of a wide range of approximation techniques, including new techniques we may not have anticipated, to make REACT a useful tool for both researchers and practitioners. Not shown are different dimensions that the taxonomy can be examined from, including determinism, hardware versus software, computational resource affected, and granularity.

4. Conclusion & Future Work

REACT is a framework enabling accurate and rapid exploration of the energy-quality trade-off space of approximate computing. We have implemented REACT, constructed a full taxonomy, and evaluated a few of the techniques on a small number of benchmarks. Our future work is centered around implementing more approximation models and evaluating approximation on more benchmarks.

References

[1] 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 *MICRO*, 2003.

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

[3] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *ISCA*, 2002.

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

[5] 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 *MICRO*, 2009.

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

[7] 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 *PLDI*, 2005.

[8] J. S. Miguel, M. Badr, and N. E. Jerger. Load value approximation. In *MICRO*, 2014.

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

[10] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones. Scalable stochastic processors. In *DATE*, 2010. URL `http://portal.acm.org/citation.cfm?id=1871008`.

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

[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. Technical Report UW-CSE-15-01-01, University of Washington, 2015.

[13] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. 8(3), 2000.

[14] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. In *MICRO*, 2013.