A Taxonomy of General Purpose Approximate Computing Techniques

Thierry Moreau[†], Joshua San Miguel[‡], Mark Wyse[†], James Bornholt[†], Armin Alaghi[†], Luis Ceze[†], Natalie Enright Jerger[‡] and Adrian Sampson[§]

[†]University of Washington — [‡]University of Toronto — [§]Cornell University

***Abstract*—Approximate computing is the idea that systems can gain performance and energy efficiency if they expend less effort on producing a "perfect" answer. Approximate computing techniques propose various ways of exposing and exploiting accuracy–efficiency trade-offs. We present a taxonomy that classifies approximate computing techniques according to salient features: visibility, determinism, and coarseness. These axes allow us to address questions about the correctability, reproducibility, and control over accuracy–efficiency tradeoffs of different techniques. We use this taxonomy to inform research challenges in approximate architectures, compilers, and applications.**

I. Introduction

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, where techniques such as quantization and decimation are crucial for tractable designs.

Opportunities abound for exploiting efficiency–accuracy trade-offs at every layer of the system stack, from compilers to circuit design. Cross-cutting concerns about energy efficiency and the future of CMOS scaling have created a boom in approximate computing research. While exciting, the multitude of approaches complicates discussions and obscures common patterns. A single monolithic "approximate computing" label, spanning ideas as disparate as voltage over-scaling [9], tweaking floating-point precision [24], and code perforation [33], is too broad to identify the foundations of the field.

This paper presents a taxonomy of general-purpose approximate computing techniques. An approximate computing technique is deemed general if it is not specific to a given algorithm or application domain. We classify techniques along three axes: correctability of the approximation effects, reproducibility of the approximate results, and control over the efficiency–accuracy trade-offs.

II. Motivation

Our taxonomy characterizes approximation techniques around three practical concerns:

1) **Correctability:** How can the effects of an approximation technique be detected and corrected?
2) **Reproducibility:** How easily can the results of an approximation technique be reproduced for testing?
3) **Control:** How much confidence over the error magnitude does an approximation technique provide?

In this section, we present examples to highlight the importance of these questions and demonstrate how they distinguish techniques that may seem similar at first glance.

A. Correctability of the Approximation Effects

Correctability reflects the cost and complexity of detecting and compensating for approximation errors. The degree of correctability varies widely between techniques. For example, consider two seemingly similar techniques: (1) low supply voltage SRAM [9], which allows for soft errors when accessing data in SRAM; and (2) low refresh DRAM [18], which allows for soft errors in DRAM data cells. For low supply voltage SRAM, errors are introduced when an instruction reads or writes the data. A precise check can thus be invoked on each approximate load and store instruction in order to recover from a faulty operation. On the other hand, for low refresh DRAM, the error can be introduced at any point in the lifetime of the data independent of any instruction's execution. This uncertainty makes error management more costly and less prompt. Our taxonomy distinguishes these two approaches (Section III-A) in terms of their *architectural visibility*.

B. Reproducibility of the Approximate Results

Reproducibility is the degree to which error can be measured during development and generalized to production. It can be difficult to reason about the error introduced by an approximation technique. We often rely on measurements from test systems to decide whether or not the error is within an acceptable range. For example, code perforation [33] is an approximation technique that omits instructions during execution. In general, its impact on error is the same regardless of the underlying system on which it is executed, so its reproducibility is straightforward. On the other hand, synchronization elision [7] omits calls to synchronization primitives like locks. We can measure the error of synchronization elision on a test system and deem it satisfactory, but we may find that error increases on a different production system. Our taxonomy distinguishes reproducibility between *deterministic* techniques like code perforation and *nondeterministic* techniques like synchronization elision (Section III-B).

Software Technique	Visible	Deterministic	Coarse
Approximate GPU Kernels [17], [26]	Y	Y	Y
Approximate Synthesis [6], [20]	Y	Y	Y
Algorithm Selection [4], [5]	Y	Y	Y
Code Perforation [33]	Y	Y	Y
Lossy Compression / Packing [26]	Y	Y	Y
Parallel Pattern Replacement [25]	Y	Y	Y
Bit-Width Reduction [24]	Y	Y	N
Float-to-Fixed Conversion [1]	Y	Y	N
Approximate Parallelization [7]	Y	N	Y
Statistical Query [2]	Y	N	Y
Synchronization Elision [7]	Y	N	Y
Hardware Technique	**Visible**	**Deterministic**	**Coarse**
Digital Neural Acceleration [10]	Y	Y	Y
Interpolated Memoization [21]	Y	Y	Y
Approximate Warp Deduplication [39]	Y	Y	Y
Bit-Width Reduction [15]	Y	Y	N
Clock Overgating [14]	Y	Y	N
Load Value Approximation [32]	Y	Y	N
Approximate Cache Coherence [22]	Y	Y	N
Instruction Memoization [3]	Y	Y	N
Precision Scaling [12], [13], [37]	Y	Y	N
Logical Simplifications [38]	Y	Y	N
Reduced-Precision FPU [36]	Y	Y	N
Analog Neural Acceleration [35]	Y	N	Y
Approx. Processors [16], [40]	Y	N	N
Voltage Overscaling [9], [15]	Y	N	N
Stochastic Logic [11]	Y	N	N
Approx. PCM Multi-Level Cells [28]	Y	N	N
SRAM Soft Error Exposure [9]	Y	N	N
Approximate Value Dedup. [30], [31]	N	Y	Y
Approx. PCM Failed Cells [28]	N	N	N
Low-Refresh DRAM [18]	N	N	N

TABLE I: Taxonomy of approximate computing techniques.

C. Control over the Accuracy–Efficiency Tradeoffs

Control reflects how easily a technique can trade accuracy for efficiency gains. All approximate computing techniques enable such a trade-off. However, they fall all along the accuracy–efficiency curve; some favor efficiency while others favor accuracy. Consider a program that performs many floating-point computations. We can approximate this program either via fuzzy function memoization [21] or via fuzzy floating-point instructions [3]. Both techniques seem similar, yet they offer very different error–efficiency trade-offs. Function memoization can elide code regions that are as small as one or two instructions or as large as entire functions, which can lead to arbitrary errors if not tested exhaustively. Fuzzy floating-point instructions, on the other hand, limit efficiency gains due to control overheads but also confine errors to the execution of individual instructions, meaning that traditional techniques such as interval analysis can be used to guarantee control over the error introduced by the technique. To characterize control over errors, our taxonomy distinguishes between techniques based on their *granularity* (Section III-C).

III. Taxonomy

We guide our taxonomy with the motivation questions detailed in Section II—*(1) correctability, (2) reproducibility, (3) control*—and list three orthogonal taxonomy axes that address them: *(1) architectural visibility vs. invisibility, (2) deterministic vs. nondeterministic, (3) coarse-grained vs. fine-grained*. For each taxonomy dimension, we provide a formal definition, examples and discuss practical implications. Table I lists a set of recent approximation techniques we surveyed and classified along these three dimensions. In this table, note that we classify techniques as software or hardware; we do not elaborate on this as a taxonomy axis since it does not inform any interesting new insights or properties.

A. Correctability: Architecturally Visible vs. Invisible

Definition 1. *Consider a program as a sequence of instructions that operate on data. An approximation technique is* ***architecturally invisible*** *if it can introduce error even when the sequence of instructions is null. Otherwise it is an* ***architecturally visible*** *technique.*

Architecturally visible techniques introduce errors during the execution of a specific instruction, and architecturally invisible techniques introduce errors silently. Naturally, visible errors are simple to detect: they can be traced to a specific moment in time. On the other hand, invisible errors are attributed to a phenomenon that occurs below the architectural stack, e.g., a micro-architectural event, or a physical event occurring at the circuit level. Consequently, architecturally invisible techniques can require expensive error detection and correction mechanisms and are harder to monitor dynamically.

Revisiting the examples in Section II-A, low supply voltage SRAM [9] is architecturally visible. It approximates (via bit upsets) only upon memory operations; thus, detecting and managing error is straightforward. For write upsets, for example, adding a precise check after a write operation can immediately catch (and roll back) any erroneous approximations. On the other hand, low refresh DRAM [18] is architecturally invisible: since it yields bit flips at arbitrary times, a precise check after a write operation cannot draw any conclusions about error. Even if the precise check passes, an erroneous bit-flip can still occur some time later.

Though errors are invisible, an advantage of architecturally invisible techniques is that they are not on the critical path; thus their latency costs can be made invisible as well. Architecturally visible techniques can introduce run time overheads, whereas invisible approximations can be performed in the background. For example, the Doppelgänger cache [31] is an architecturally invisible technique; it generates approximate values silently upon a microarchitectural event without stalling memory requests.

This taxonomy axis informs trade-offs in error correctability. Architecturally visible techniques benefit from errors which are easier to detect and correct. On the other hand, architecturally invisible techniques benefit from generating approximations off the critical path of program execution.

B. Reproducibility: Deterministic vs. Nondeterministic

Definition 2. *An approximation technique is* ***deterministic*** *if, given the same initial state, for every input I_j, it yields constant error E_j. An approximation technique is* ***nondeterministic*** *if, given the same initial state, there exists some input I_j for which it yields more than one error value $E_{j0}, ..., E_{jn}$.*

Nondeterministic techniques can pose a challenge for testing and debugging. When developing techniques, the conventional approach is to evaluate error and efficiency on a test system and extrapolate to production systems. This is effective for

deterministic techniques since they produce the same approximations regardless of the underlying system; errors are *reproducible*. It is possible for a user to declare any error threshold ϵ and concretely evaluate whether or not it is always satisfied for a given input. However, this is not true for nondeterministic techniques. For a given input, error can only be probabilistically evaluated; ϵ must be accompanied by some probability and confidence.

Nondeterministic techniques have limited reproducibility. Such approximations are possible via exposing analog noise, asynchrony and race conditions to the program. Revisiting the examples in Section II-B, synchronization elision [7] is a nondeterministic technique while code perforation [33] is deterministic. Whereas perforating computations yields the same output on any system, eliding synchronization primitives exposes race conditions. This increases the number of possible outputs and limits reproducibility. The amount of error via synchronization elision can vary greatly across systems depending on the amount of thread-level parallelism. Nondeterministic techniques can also expose analog noise. For example, voltage-overscaled ALUs [9] generate approximations by risking exposure to the analog domain. This has low reproducibility; error cannot be concretely evaluated and must be empirically measured. In comparison, precision-scaled ALUs [37] are deterministic. Scaling precision in the digital representation of data yields the same output on any system.

As a trade-off, nondeterministic techniques can generally offer more opportunity for efficiency gains. By exposing the stochastic nature of the physical world, they avoid the expensive digital abstraction tax. For example, voltage-overscaled ALUs significantly improve efficiency by relaxing the safety margins enforced by digital circuitry.

This taxonomy axis informs trade-offs in reproducibility. Deterministic techniques benefit from high reproducibility, simplifying testing and debugging. On the other hand, nondeterministic techniques benefit from more opportunities for approximation that only exist outside the digital domain.

C. Error Control: Coarse-Grained vs. Fine-Grained

Definition 3. *An approximation technique is **coarse-grained** if it reduces the data footprint or the number of dynamic instructions in a program. Otherwise, it is **fine-grained**.*

Control over the error introduced by a technique depends on the *granularity* at which an approximation technique is employed. Fine-grained techniques lower the cost of executing an instruction or storing a word of data. Coarse-grained techniques replace a set of instructions or a block of data with a more efficient or compact representation.

Coarse-grained techniques offer more opportunity for error–efficiency trade-offs. Revisiting the examples in Section II-C, fuzzy floating-point instructions [36] are fine-grained while fuzzy function memoization [21] is coarse-grained. Whereas the former improves the efficiency of individual instructions, the latter can improve the efficiency of an entire block or function. The latter, in the most extreme case, can memoize the entire program for the highest efficiency. In terms of storage, fine-grained techniques, such as low refresh DRAM [18], generate approximations in individual bits. Coarse-grained techniques, such as approximate deduplication [31], reduce data footprint. The latter can be more aggressively tuned for efficiency gains, to the point where the entire data footprint is deduplicated into a single data block.

Naturally, the coarser the granularity of a technique, the higher the risk of error. Fine-grained techniques do not remove any data nor instructions. Conversely, coarse-grained techniques risk information loss as more data and more instructions are omitted. In the previous examples, though memoizing an entire program yields highest efficiency, it also yields highest error. Holistically approximating regions of code can disregard rarely-used control-flow paths when not exercised. Neural approximation [10] is an example of a coarse-grained technique that can subsume entire functions, including potentially complex control flow. This coarseness makes testing and analysis challenging.

This taxonomy axis informs trade-offs in error control. Coarse-grained techniques benefit from greater opportunities for aggressive efficiency gains. On the other hand, fine-grained techniques can limit error and are generally better suited for programs where quality constraints are conservative.

IV. Discussion

We highlight the applicability of our proposed taxonomy by suggesting how it can inform future research in approximate computing. We formulate a three-pronged answer that address the questions across layers of the compute stack: *(1) architecture, (2) compilers and runtimes and (3) applications*.

A. How Can It Inform Architecture Research?

Research on new approximation techniques motivates the need for approximation-aware ISAs (A-ISA). Since the days of the IBM System/360, architects have distinguished between architecture and implementation to guarantee the *forward-compatibility* of their hardware. An A-ISA can express instruction-level error bounds that need to be respected when deployed on current or future hardware. Such an abstraction layer would allow hardware designers to modify the implementation of approximations down the road in a way that remains invisible to the software. We make a distinction between two types of A-ISAs: strict A-ISAs and statistical A-ISAs. Strict A-ISAs are applicable to deterministic fine-grained techniques and provide strict error bounds on the execution of an instruction. Examples of A-ISAs include the Quality-Programmable ISA [37], which provides strict error bounds relative to the maximum output value of the instruction. Statistical A-ISAs, on the other hand, are applicable to nondeterministic fine-grained techniques and provide statistical failure guarantees. Such an ISA would have to include probability bounds as well as confidence bounds.

B. How Can It Inform Compilers/Runtimes Research?

Research on approximation techniques motivates the development of frameworks to make approximations *safe* to use. Such frameworks include new languages, compilers and runtimes. We discuss how each taxonomy can inform the applicability of framework proposals.

Architectural visibility is relevant to frameworks that focus on detecting and recovering from hardware faults. Relax [8], for instance, can only work on top of architecturally visible techniques because errors must be locally correctable [34]. Online monitoring proposals [23] that rely on precise replay are also only applicable to architecturally visible techniques.

Determinism and coarseness are relevant to formulating statically-derived or empirically-observed application-level error bounds. Nondeterministic techniques require statistical methods like probabilistic assertions [29], while deterministic techniques can rely on hard assertions. Fine-grained techniques can inherit from the wealth of tools developed in numerical analysis research [24]. More specifically, deterministic fine-grained techniques have the advantage of providing strict error bounds at an instruction granularity. Thus, they can provide hard worst-case error bounds for many algorithmic patterns, as opposed to empirically derived average-case error bounds. Coarse-grained techniques have seen a wealth of frameworks [4], [5], [19], [25], [27] that generally rely on empirical error measurements to provide varying levels of error guarantees via quality autotuning.

C. How Can It Inform Applications Research?

Research on new approximation techniques motivates better understanding on the *applicability* of such techniques. Application designers care about (1) whether a technique can be applied to their algorithms, and (2) whether a technique can meet the quality guarantees they wish to enforce.

Coarseness correlates to how general a technique is to algorithmic patterns. Fine-grained techniques are broadly generalizable: any approximate floating-point algorithm can make use of reduced-precision FPUs. Coarse-grained techniques, on the other hand, have to adhere to specific code patterns: neural acceleration only applies to precise-pure regions of code, while loop-perforation applies to loops free of early exits [27].

Determinism and coarseness will both determine the error behavior that the application will see. Nondeterministic techniques generally yield large rarely-occurring errors while deterministic techniques yield small frequently-occurring errors. Nondeterministic techniques would generally not be used in mission-critical systems. The magnitude of an error is generally better controlled on deterministic fined-grained techniques as opposed to deterministic coarse-grained techniques.

V. Conclusion

A wealth of approximate computing techniques has been proposed in architecture, circuits, languages, and compilers research. We present a taxonomy that categorizes approximate computing techniques based their most salient properties: visibility, determinism, and coarseness, to better inform cross-stack research in architecture, tools, and applications.

References

[1] T. M. Aamodt and P. Chow, "Compile-time and instruction-set methods for improving floating- to fixed-point conversion accuracy," *TECS*, 2008.

[2] S. Agarwal *et al.*, "BlinkDB: Queries with bounded errors and bounded response times on very large data," in *EuroSys*, 2013.

[3] C. Alvarez *et al.*, "Fuzzy memoization for floating-point multimedia applications," *ToC*, 2005.

[4] J. Ansel *et al.*, "PetaBricks: A language and compiler for algorithmic choice," in *PLDI*, 2009.

[5] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *PLDI*, 2010.

[6] J. Bornholt *et al.*, "Optimizing synthesis with metasketches." in *POPL*, 2016.

[7] S. Campanoni *et al.*, "HELIX-UP: relaxing program semantics to unleash parallelization." 2015.

[8] M. de Kruijf *et al.*, "Relax: an architectural framework for software recovery of hardware faults," in *ISCA*, 2010.

[9] H. Esmaeilzadeh *et al.*, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012.

[10] H. Esmaeilzadeh *et al.*, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.

[11] B. R. Gaines, "Stochastic computing systems," in *Advances in information systems science*. Springer, 1969, pp. 37–172.

[12] A. Jain *et al.*, "Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation," in *MICRO*, 2016.

[13] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *DAC*, 2012.

[14] Y. Kim *et al.*, "Designing approximate circuits using clock overgating," in *DAC*, 2016.

[15] S. Lee *et al.*, "High-level synthesis of approximate hardware under joint precision and voltage scaling," in *DATE*, 2017.

[16] L. Leem *et al.*, "ERSA: Error resilient system architecture for probabilistic applications," in *DATE*, 2010.

[17] A. Li *et al.*, "SFU-driven transparent approximation acceleration on GPUs," in *ICS*, 2016.

[18] S. Liu *et al.*, "Flikker: Saving refresh-power in mobile devices through critical data partitioning," in *ASPLOS*, 2011.

[19] D. Mahajan *et al.*, "Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration," in *ISCA*, 2016.

[20] S. Misailovic *et al.*, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *OOPSLA*, 2014.

[21] A. K. Mishra *et al.*, "iACT: A software-hardware framework for understanding the scope of approximate computing," in *WACAS*, 2014.

[22] P. V. Rengasamy *et al.*, "Exploiting staleness for approximating loads on CMPs," in *PACT*, 2015.

[23] M. Ringenberg *et al.*, "Monitoring and debugging the quality of results in approximate programs." in *ASPLOS*, 2015.

[24] C. Rubio-Gonzalez *et al.*, "Precimonious: Tuning assistant for floating-point precision," in *SC*, 2013.

[25] M. Samadi *et al.*, "Paraprox: Pattern-based approximation for data parallel applications," in *ASPLOS*, 2014.

[26] M. Samadi *et al.*, "SAGE: Self-tuning approximation for graphics engines," in *MICRO*, 2013.

[27] A. Sampson *et al.*, "ACCEPT: A programmer-guided compiler framework for practical approximate computing," U. Washington, Tech. Rep.

[28] A. Sampson *et al.*, "Approximate storage in solid-state memories," in *MICRO*, 2013.

[29] A. Sampson *et al.*, "Expressing and verifying probabilistic assertions," in *PLDI*, 2014.

[30] J. San Miguel *et al.*, "The bunker cache for spatio-value approximation," in *MICRO*, 2016.

[31] J. San Miguel *et al.*, "Doppelganger: A cache for approximate computing." in *MICRO*, 2015.

[32] J. San Miguel *et al.*, "Load value approximation," in *MICRO*, 2014.

[33] S. Sidiroglou *et al.*, "Managing performance vs. accuracy trade-offs with loop perforation," in *FSE*, 2011.

[34] V. Sridharan *et al.*, "A taxonomy to enable error recovery and correction in software." in *Work. on Quality-Aware Design.*, 2008.

[35] R. St. Amant *et al.*, "General-purpose code acceleration with limited-precision analog computation," in *ISCA*, 2014.

[36] J. Y. F. Tong *et al.*, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *VLSI*, 2000.

[37] S. Venkataramani *et al.*, "Quality programmable vector processors for approximate computing," in *MICRO*, 2013.

[38] S. Venkataramani *et al.*, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *DATE*, 2014.

[39] D. Wong *et al.*, "Approximating warps with intra-warp operand value similarity," in *HPCA*, 2016.

[40] Y. Yetim *et al.*, "Extracting useful computation from error-prone processors for streaming applications," in *DATE*, 2013.