A Taxonomy of Approximate Computing Techniques

Thierry Moreau[†], Joshua San Miguel[‡], Mark Wyse[†], James Bornholt[†],
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 their most salient features: compute vs. data, deterministic vs. nondeterministic and coarse- vs. fine-grained. These axes allow us to address questions about the visibility, testability and flexibility of different techniques. We use this taxonomy to inform future research in approximate architectures, compilers and applications that will catalyze mainstream adoption of approximate computing.**

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 architectures, to circuit design. Cross-cutting concerns about energy efficiency, and the future of CMOS scaling, have created a boom in approximate computing research in recent years. 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 [11], tweaking floating-point precision [27] and code perforation [35], is too broad to identify the foundations of the field.

This paper presents a taxonomy of approximate computing research. We classify techniques along three axes: visibility of the approximate results, testability of the system, and flexibility of the efficiency–accuracy trade-off. These axes help to focus future research efforts.

II. Motivation

Our taxonomy characterizes approximation techniques according to three practical questions:

1) **Visibility:** How easily can the effects of an approximation technique be observed?
2) **Testability:** How well can an approximation technique be tested and debugged?
3) **Flexibility:** How aggressively can a technique trade-off error for efficiency?

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

A. Visibility

Visibility is the extent to which a technique exposes its approximation errors. The degree of visibility varies widely between techniques. For example, consider two apparently-similar techniques: (1) low supply voltage SRAM [13], 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. Both techniques allow bit-flips in data values. But seen through the lens of visibility, these techniques are quite different. To wit, if an erroneous bit-flip occurs, how can it be detected and managed? For low supply voltage SRAM, errors are introduced only upon reading and writing data. A precise check can thus be invoked on each approximate load and store instruction. On the other hand, for low refresh DRAM, error can be introduced at any point in the lifetime of the data. This uncertainty makes error management more costly and less prompt. The difference in visibility is due to the fact that low supply voltage SRAM only approximates the action of accessing data, whereas low refresh DRAM approximates the way data is stored. Our taxonomy distinguishes these two approaches (Section III-A): the former is a *compute* technique; the latter, a *data* technique.

B. Testability

Testability 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 [35] is an approximation technique that omits instructions during execution. In general,

Software Technique	Visibility	Testability	Flexibility
Approximate CUDA Kernels [29]	Compute	Det	Coarse
Approximate Synthesis [7]	Compute	Det	Coarse
Algorithm Selection [4], [5]	Compute	Det	Coarse
Code Perforation [35]	Compute	Det	Coarse
Parallel Pattern Replacement [28]	Compute	Det	Coarse
Bit-Width Reduction [23], [27]	Compute	Det	Fine
Float-to-Fixed Conversion [1]	Compute	Det	Fine
Approximate Parallelization [6]	Compute	Nondet	Coarse
Statistical Query [2]	Compute	Nondet	Coarse
Synchronization Elision [19], [24], [25]	Compute	Nondet	Coarse
Lossy Compression / Packing [29]	Data	Det	Coarse
Hardware Technique	**Visibility**	**Testability**	**Flexibility**
Digital Neural Acceleration [12], [14], [21]	Compute	Det	Coarse
Interpolated Memoization [20]	Compute	Det	Coarse
Load Value Approximation [34], [38], [39]	Compute	Det	Fine
Instruction Memoization [3]	Compute	Det	Fine
Precision Scaling [41], [43]	Compute	Det	Fine
Logical Simplifications [16], [42]	Compute	Det	Fine
Reduced-Precision FPU [40]	Compute	Det	Fine
Analog Neural Acceleration [37]	Compute	Nondet	Coarse
Approx. Processors [8], [15], [17], [44], [45]	Compute	Nondet	Fine
Voltage Overscaling (ALU) [11], [22]	Compute	Nondet	Fine
Approx. PCM Multi-Level Cells [31]	Compute	Nondet	Fine
SRAM Soft Error Exposure [9], [13]	Compute	Nondet	Fine
Approximate Value Deduplication [33]	Data	Det	Coarse
Approx. PCM Failed Cells [31]	Data	Nondet	Fine
Low-Refresh DRAM [18]	Data	Nondet	Fine

TABLE I: Taxonomy of approximate computing techniques.

its impact on error is the same regardless of the underlying system on which it is executed, so its testability is straightforward. On the other hand, synchronization elision [19], [24], [25] 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 dramatically on a production system with more parallelism. Our taxonomy distinguishes testability between *deterministic* techniques like code perforation and *nondeterministic* techniques like synchronization elision (Section III-B).

C. Flexibility

Flexibility reflects how easily a technique can trade-off 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 [20] or via fuzzy floating-point instructions [40]. Both techniques seem similar, yet they offer very different error–efficiency trade-offs. Function memoization is highly flexible; it can elide code regions that are as small as one or two instructions or as large as entire functions. But its flexibility also means that it can induce nearly arbitrary errors. Fuzzy floating-point instructions, on the other hand, limit efficiency gains but also confine errors to the execution of individual instructions. To characterize flexibility, 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) visibility, (2) testability, (3) flexibility* — and list three orthogonal taxonomy axes that address them: *(1) compute vs. data, (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. Visibility: Compute vs. Data

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

Compute techniques approximate the instructions executed by a program, while data techniques approximate the storage or representation of data values. From a program's perspective, an instruction is a momentary event. Thus any error introduced by a compute technique can always be traced to a single moment in time. We say that compute techniques yield errors with high *visibility*. On the other hand, a piece of data is not a momentary event. Thus any error introduced by a data technique can occur at any arbitrary time during the lifetime of the data. We say that such errors are *invisible*.

Naturally, visible errors are simple to detect. Revisiting the examples in Section II-A, low supply voltage SRAM [13] is a compute technique. 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 a data technique and is less visible. 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 data techniques is that they are not on the critical path; thus their latency costs can be made invisible as well. Compute techniques directly affect program runtime since they approximate the instruction stream, whereas data approximations can be performed lazily and off the critical path. For example, the Doppelgänger cache [33] is a data technique; it generates approximate values silently without stalling memory requests.

This taxonomy axis informs trade-offs in visibility. Compute techniques benefit from the guarantee of visible errors, which are easier to detect and manage. On the other hand, data techniques benefit from the ability to generate approximations off the critical path of program execution.

B. Testability: Deterministic vs. Nondeterministic

Definition 2. *An approximation technique is* ***deterministic*** *if given the same initial state, for each and 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 testability. Such approximations are possible via exposing analog noise, asynchrony and race conditions to the program. Revisiting the examples in Section II-B, synchronization elision [19], [24], [25] is a nondeterministic technique while code perforation [35] 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 testability. 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 [11], [22] generate approximations by risking exposure to the analog domain. This has low testability; error cannot be concretely evaluated and must be empirically measured. In comparison, precision-scaled ALUs [41] 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 testability. 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. Flexibility: Coarse-Grained vs. Fine-Grained

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

Flexibility depends on the *granularity* in which an approximation technique is employed. Fine-grained techniques lower the cost of executing an instruction or storing a bit. Coarse-grained techniques replace a set of instructions or bits with a more efficient or compact representation.

Coarse-grained techniques are highly flexible; they offer more opportunity for error–efficiency trade-offs. Revisiting the examples in Section II-C, fuzzy floating-point instructions [40] are fine-grained while fuzzy function memoization [20] 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 offers more flexibility; in the most extreme case, we can memoize the entire program for the highest efficiency. In terms of data, fine-grained techniques, such as low refresh DRAM [18], generate approximations in individual bits. Coarse-grained techniques, such as approximate deduplication [33], 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. Similarly, deduplicating the entire data footprint yields much higher error than deduplicating a single data block.

This taxonomy axis informs trade-offs in flexibility. 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 tighter.

IV. Discussion

We highlight the applicability of our proposed taxonomy by suggesting how it can inform future research in the field of 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 the 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 [41] 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 (possibly in a log scale) as well as confidence bounds (unless implicitly assumed). It is worth mentioning that coarse-grained techniques are built above the ISA layer in the system stack, thus they don't require A-ISAs support.

B. How Can It Inform Compilers/Runtimes Research?

Research on new 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.

Visibility is relevant to frameworks that focus on detecting and recovering from hardware faults. Relax [10] for instance can only work on top of compute techniques since errors have to be locally correctable [36]. Online monitoring proposals [26] that rely on precise replay are also only applicable to compute techniques.

Testability and flexibility are relevant to formulating statically-derived or empirically-observed application-level error bounds. Nondeterministic techniques require statistical methods like probabilistic assertions [32], while deterministic techniques can rely on hard assertions. Fine-grained techniques can inherit from the wealth of tools developed in numerical analysis research [27]. 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]–[6], [28], [30] 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.

Flexibility determines 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 [30].

Flexibility and testability 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. Neural networks are known not to produce errors below 1% for regression problems without overfitting. Thus they may not be preferred in financial applications where a 1% would result in large financial losses, but are acceptable to use on multimedia or consumer virtual reality applications.

V. Conclusion

A wealth of approximate computing techniques has been proposed in architecture, circuits, languages and compilers research. Distilling this cornucopia of proposals requires a well defined categorization that discriminates techniques based on their most salient properties. We present a taxonomy that categorizes approximate computing techniques based on visibility, testability and flexibility. Our proposed taxonomy can better inform cross-stack research in architecture, compilers/runtimes, and applications to catalyze the mainstream adoption of approximate computing.

References

[1] T. M. Aamodt and P. Chow, "Compile-time and instruction-set methods for improving floating- to fixed-point conversion accuracy," vol. 7, no. 3, 2008.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: Queries with bounded errors and bounded response times on very large data," in *EuroSys*, 2013.

[3] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *ToC*, vol. 54, no. 7, pp. 922 – 927, July 2005.

[4] 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 *PLDI*, 2009.

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

[6] B. Belhadj, A. Joubert, Z. Li, R. Héliot, and O. Temam, "Continuous real-world inputs can open up alternative accelerator designs." in *ISCA*, 2013.

[7] J. Bornholt, E. Torlak, D. Grossman, and L. Ceze, "Optimizing synthesis with metasketches." in *POPL*, 2016.

[8] L. N. Chakrapani, P. Korkmaz, B. E. S. Akgul, and K. V. Palem, "Probabilistic system-on-a-chip architectures," in *TDAES*, 2007.

[9] 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.

[10] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: an architectural framework for software recovery of hardware faults," in *ISCA*, 2010.

[11] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012.

[12] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.

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

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

[15] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. C. Rinard, "Dynamic knobs for responsive power-aware computing," in *ASPLOS*, 2011.

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

[17] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *DATE*, 2010.

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

[19] S. Misailovic, S. Sidiroglou, and M. C. Rinard, "Dancing with uncertainty," in *RACES*, 2012.

[20] A. K. Mishra, R. Barik, and S. Paul, "iACT: A software-hardware framework for understanding the scope of approximate computing," in *WACAS*, 2014.

[21] 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.

[22] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, “Scalable stochastic processors,” in *DATE*, 2010.

[23] E. Ozre, A. P. Nisbet, and D. Gregg, “A stochastic bitwidth estimation technique for compact and low-power custom processors,” 2008.

[24] L. Renganarayana, S. Vijayalakshmi, R. Nair, and D. Prener, “Programming with relaxed synchronization,” in *RACES*, 2012.

[25] M. Rinard, “Parallel synchronization-free approximate data structure construction,” in *HotPar*, 2013.

[26] M. Ringenberg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman, “Monitoring and debugging the quality of results in approximate programs.” in *OOPSLA*, 2011.

[27] 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 *SC*, 2013.

[28] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, “Paraprox: Pattern-based approximation for data parallel applications,” in *ASPLOS*, 2014.

[29] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, “Sage: Self-tuning approximation for graphics engines,” in *MICRO*, 2013.

[30] 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,” U. Washington, Tech. Rep. UW-CSE-15-01-01, 2015.

[31] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, “Approximate storage in solid-state memories,” in *MICRO*, 2013.

[32] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze, “Expressing and verifying probabilistic assertions,” in *PLDI*, 2014.

[33] J. San Miguel, J. Albericio, A. Moshovos, and N. Enright Jerger, “Doppelganger: A cache for approximate computing.” in *MICRO*, 2015.

[34] J. San Miguel, M. Badr, and N. Enright Jerger, “Load value approximation,” in *MICRO*, 2014.

[35] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard, “Managing performance vs. accuracy trade-offs with loop perforation,” in *FSE*, 2011.

[36] V. Sridharan, D. A. Liberty, and D. R. Kaeli, “A taxonomy to enable error recovery and correction in software.” in *Work. on Quality-Aware Design.*, 2008.

[37] 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 *ISCA*, 2014.

[38] M. Sutherland, J. San Miguel, and E. Jerger, “Texture cache approximation on GPUs,” in *WAX*, 2015.

[39] 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 *PACT*, 2014.

[40] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, “Reducing power by optimizing the necessary precision/range of floating-point arithmetic,” *VLSI*, 2000.

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

[42] S. Venkataramani, K. Roy, and A. Raghunathan, “Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits,” in *DATE*, 2014.

[43] 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 *MICRO*, 2007.

[44] Y. Yetim, S. Malik, and M. Martonosi, “CommGuard: Mitigating communication errors in error-prone parallel execution,” in *ASPLOS*, 2015.

[45] Y. Yetim, M. Martonosi, and S. Malik, “Extracting useful computation from error-prone processors for streaming applications,” in *DATE*, 2013.