RC25600 (WAT1604-033) April 13, 2016
Computer Science

IBM Research Report

A Systolic Approach to Deriving Anytime Algorithms for Approximate Computing

Joshua San Miguel
University of Toronto
Canada

Ravi Nair, Vijayalakshmi Srinivasan, Daniel A. Prener
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

A Systolic Approach to Deriving Anytime Algorithms for Approximate Computing

Joshua San Miguel[†]
University of Toronto
joshua.sanmiguel@mail.utoronto.ca

Vijayalakshmi Srinivasan
IBM T. J. Watson Research Center
viji@us.ibm.com

Ravi Nair
IBM T. J. Watson Research Center
nair@us.ibm.com

Daniel A. Prener
IBM T. J. Watson Research Center
prener@us.ibm.com

ABSTRACT

Approximate computing is an emerging paradigm enabling tradeoffs between accuracy and efficiency. However, a fundamental challenge persists: state-of-the-art techniques lack the ability to enforce runtime guarantees on accuracy. The convention is to 1) employ training/rollback mechanisms which add complexity, or 2) present experimental results that demonstrate low error, though only empirically/statistically. We offer a solution that revisits concepts from anytime algorithms. Originally explored for real-time decision problems, anytime algorithms have the property of producing outputs with increasing accuracy over time. We propose the Anytime Automaton, a pipelined computation model that enables anytime approximations via sampling. The automaton executes applications such that 1) they can be interrupted while still delivering valid output, and 2) their accuracy increases over time and is guaranteed to eventually reach precise output. The automaton can be stopped at any point that the user is satisfied, expending just enough time and energy for an acceptable output. If the output is not acceptable, it is a simple matter of letting the application run longer.

1. INTRODUCTION

The rise of approximate computing has garnered much interest in the architecture community. This paradigm of trading off accuracy for performance and energy efficiency continues to inspire novel and creative new approximation techniques [4, 7, 9, 16, 17, 18]. However, despite the substantial benefits offered by approximate computing, it has not yet earned widespread acceptance to merit its adoption in real processors. This is due to the fundamental challenge of providing guarantees on error. State-of-the-art approximate computing techniques are unable to enforce runtime error bounds and guarantees on accuracy. The conventional approach is to either 1) employ training/rollback mechanisms to maintain accuracy, or 2) present experimental results showing that error is empirically/statistically low. These approaches can be complex and are still insufficient in guaranteeing acceptable accuracy at runtime.

To address this, we revisit concepts from anytime algorithms [3, 8, 10]. Originally proposed for planning and decision processes in artificial intelligence, anytime algorithms uphold two key properties: 1) they can be stopped at any time while still producing a valid output, and 2) they guarantee progressively increasing output quality over time. We believe that these properties offer a solution to the challenges of approximate computing.

We propose the **Anytime Automaton**, a computation model that enables anytime approximations. The model represents approximate applications as a pipeline of computation stages and employs sampling techniques in an anytime way. We show that this model is able to extract parallelism even out of sequential computations. Our anytime automaton model allows applications to execute such that they deliver intermediate approximate outputs with improving accuracy over time, guaranteeing that the precise output is eventually reached. In this way, the anytime automaton can be stopped once the output is deemed acceptably accurate by the user, expending just the right amount of computation time and energy. If the output is not yet acceptable, it is only a simple matter of letting the automaton run longer. We evaluate the model on PERFECT [2] benchmarks, demonstrating promising results with runtime-accuracy profiles.

2. BACKGROUND AND MOTIVATION

In this section, we provide background on approximate computing and anytime algorithms. We discuss the challenge of providing accuracy guarantees in approximate computing and motivate our solution of providing a model for anytime approximations.

2.1 Approximate Computing

Approximate computing introduces application output error/accuracy as an axis in architectural design, trading off for improved performance and energy efficiency. State-of-the-art approximate computing techniques have been proposed both in software and hardware. In software, eliding code/computation can yield acceptable approximations. Examples include loop perforation [19] and relaxed synchro-

[†]This work was done when this author was at IBM T. J. Watson Research Center.

nization [15] (i.e., approximation via lock elision). In hardware, many techniques exploit the physical characteristics of computation and storage elements. For example, the operating voltages of functional units can be scaled to trade-off accuracy for performance or power-efficiency [14]. SRAM caches [5], DRAM [11] and phase-change memories [17] are also amenable to such physical approximations. Though these techniques achieve substantial efficiency gains, it can be difficult to reason about error in the application output.

Some approximate computing techniques provide error control via informed approximations (i.e., based on previous precise outputs). Offline training phases inform the configuration of neural accelerators [4, 7, 13, 20]. Value-based approximations [18, 21] learn from the precise data values of previous memory accesses. Memoization-based approximations [1] exploit the recurrence of common computations. Precision-based approximations [22, 24] make informed approximations based on bit significance in data types.

Other techniques, such as Rumba [9] and SAGE [16], implement online error control. However, such mechanisms can introduce considerable overhead and complexity, requiring knowledge of error metrics as well as the source of error, to isolate troublesome computation/data and re-execute precisely. Furthermore, online error controllers rely on statically-defined error metrics, which may not always be applicable. For example, 10% average error may be visually acceptable for one set of images but not for others. Error control is best left to users, since the definition of what is acceptable can vary from one case to another.

2.2 Anytime Algorithms

An anytime algorithm is an algorithm that produces an output with progressively increasing accuracy over time. Anytime algorithms were first explored in terms of time-dependent planning and decision making [3, 8, 10]. They are generally studied in the context of artificial intelligence under real-time constraints, where suboptimal output quality can be more acceptable than exceeding time limits. Anytime algorithms can be characterized as either contract or interruptible algorithms [25]. Contract algorithms make online decisions to schedule their computations to meet a runtime deadline. Researchers have explored optimal scheduling policies for contract anytime algorithms [6, 23] as well as the error composition of anytime algorithms [25]. On the other hand, interruptible algorithms can deliver an output when stopped (or paused) at any moment. Our work focuses on interruptible anytime algorithms, which provide stronger guarantees for real-time and user-interactive applications.

Despite the wealth of research on anytime algorithms, there is little to no work on its implications to computer architecture. The most relevant work explores porting contract anytime algorithms to GPUs and providing CUDA-enabled online quality control [12]. Anytime algorithms are able to offer strong accuracy guarantees. However, such guarantees are derived at an algorithmic level; the anytime concept is typically regarded as a property built into algorithms as opposed to a general technique that can be employed on applications. In our work, we integrate the anytime concept to approximate computing, providing a general model for executing applications such that their output accuracy increases with time.

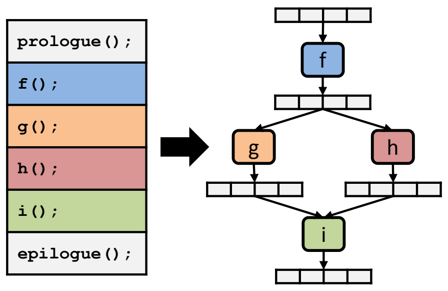


Figure 1: High-level overview of anytime automaton.

3. THE ANYTIME AUTOMATON

The **Anytime Automaton** is a computation model that represents approximate applications as anytime stages. Figure 1 shows a high-level overview. An approximate application is broken down into computation stages with input/output buffers, connected in a directed, acyclic graph. Sampling techniques (Section 3.1) can then be applied to stages (where applicable), allowing them to produce anytime outputs. This allows stages to execute in parallel as a pipeline (Section 3.2), since they can deliver intermediate approximate outputs as opposed to just the single precise output in the end. Data is streamed through the stages, and each stage produces approximate output that progressively increases in accuracy over time, eventually reaching the precise output. The anytime automaton can be stopped (or paused) once the application output is deemed acceptably accurate, expending just the right amount of computation time and energy. The decision of stopping can either be automated, user-specified (as in interactive computing) or enforced by time constraints (as in real-time computing). In all cases, the user can rely on the comfort of knowing that error eventually diminishes.

An example pipeline is shown in Figure 2. Each of the four stages f, g, h and i are anytime; in this case, their computations are broken into two parts (e.g., f_1 and f_2). As soon as f_1, g_1, h_1 and i_1 have executed, an approximate output O_{111} is available, and thus the application can already be stopped here. If the approximate output is not acceptable, the pipeline can simply continue executing, progressively improving the output accuracy until the final precise output O_{222}. In this way, the anytime automaton is able to extract parallelism out of sequential applications. Whereas g and h would have to wait for f to finish in the original application, our model enables f to produce an approximate (but still acceptable) output so that g and h can already start executing.

To ensure correctness in the model, the following properties must hold:

1. Each computation stage is idempotent under sequential composition (i.e., $f()$ applied twice on I yields the same end result).
2. Computation stages do not communicate via any global variables.
3. Each output buffer has exactly one writer stage.
4. Updates to output buffers are atomic.

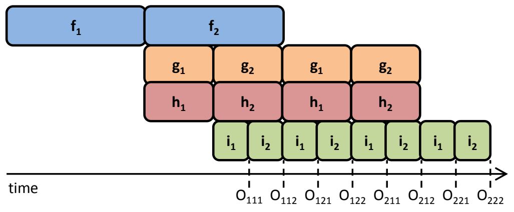


Figure 2: Parallel pipeline of anytime automaton.

3.1 Anytime Approximations via Sampling

In this section, we show how to sample the input and output data sets of a computation stage to generate anytime approximations. Specifically, instead of waiting to process all elements in a data set before delivering the final output, sampling recognizes that the intermediate output (of the elements processed so far) can serve as an acceptable approximation.

A computation stage f with input I and output O can be represented as:

$$f(I) \rightarrow O$$

We say that f is an **anytime** computation stage if it can be represented as a set of intermediate computations $f_1, ..., f_n$ executed sequentially, where each f_i operates on progressively larger samples of the input/output data:

$$f_1(I, O_0), ..., f_n(I, O_{n-1}) \rightarrow O$$

O_0 is the initial value in the output buffer. Since each f_i processes progressively larger samples, O_i generally improves on the accuracy of O_{i-1}, eventually reaching the precise output $O_n = O$. In the simple example in Figure 2, f, g, h and i are all anytime stages with $n = 2$. Section 3.1.1 discusses **input sampling**, which is well-suited to reductions on input I, and Section 3.1.2 discusses **output sampling**, which is well-suited to map operations on output O.

3.1.1 Input Sampling

Input sampling enables anytime approximations for reduction computations. Reductions process elements in the input set and accumulate values in the output buffer. Intuitively, performing the reduction on only a sample of the input set can yield acceptable approximations of the final accumulated output.

Reductions are most commonly performed using commutative operators. Examples include computing a sum, searching for an element or building a histogram. A computation stage f is **commutative** if it can be represented as:

$$\forall\, i, \quad f_i(I, O_{i-1}) = O_{i-1} \bigtriangleup x_i(I)$$

where $\bigtriangleup$ is some commutative operation and $x_1, ..., x_n$ are computations independent of O.

Figure 3 shows an example of anytime histogram construction using input sampling; $\bigtriangleup$ is the addition operator. As more input elements are processed over time, the approximate histogram approaches the precise output. In this example, sampling is performed with a pseudo-random permutation; we discuss various permutation functions and their suitability for different applications in Section 3.1.3).

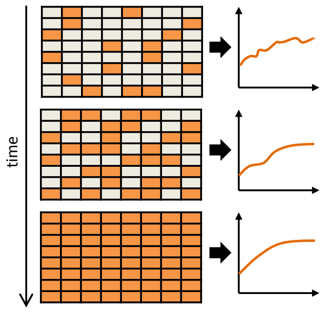


Figure 3: Example of input sampling with a pseudo-random permutation for anytime histogram construction.

3.1.2 Output Sampling

Whereas input sampling is applicable to reductions, output sampling is well-suited for map operations. We generalize map operations to computations that generate a set of distinct output elements, each of which are computed from some element(s) in the input set:

$$\forall\, i, \quad f_i(I, O_{i-1}) = O_{i-1},\ O_i[i] = x_{m(i)}(I)$$

where $m(i)$ is some mapping of input elements to the output element at index i. In this case, the commutative operation is a union of disjoint sets: $O_{i-1} \cup X_{m(i)}$. Output sampling is applicable to common map computations. Examples include generating pixels of an image, processing lists of independent items or simulating the movement of particles.

3.1.3 Sampling Permutations

For a commutative stage f, the final precise output can be computed from any sequential ordering of $x_1, ..., x_n$. Thus the order can be permuted to better suit the type of computation. In the histogram construction example (Figure 3), accessing the elements in their sequential memory order may result in biased approximate outputs (i.e., biased towards the first elements in memory order). To avoid such bias, a uniform random permutation is more suitable as shown in the figure.

Applying a permutation to f yields:

$$\forall\, i, \quad f_i(I, O_{i-1}) = O_{i-1} \bigtriangleup x_{p(i)}(I)$$

for input sampling and:

$$\forall\, i, \quad f_i(I, O_{i-1}) = O_{i-1},\ O_i[p(i)] = x_{m(p(i))}(I)$$

for output sampling, where $p(i)$ is the **permutation** function and is bijective (i.e., a one-to-one and onto mapping of i). As long as p is bijective, the precise output is guaranteed since all x_i computations are still performed exactly once.

Depending on the computation, some permutations may be more suitable than others. In general, we find that the three most common permutations are sequential (for priority-ordered data sets), tree (for ordered data sets without priority) and pseudo-random (for unordered data sets).

Sequential Permutation. The default permutation is sequential, where elements are accessed in memory order (i.e., ascending index i). This can be expressed simply as $p(i) = i$ or $p(i) = n + 1 - i$, for $i \in [1...n]$. Sequential sampling is

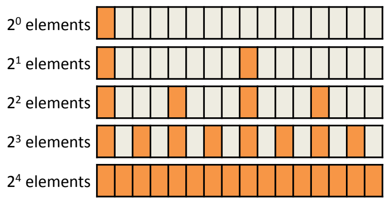


Figure 4: One-dimensional tree sampling permutation example. This shows which indices have been accessed after $2^0, \ldots, 2^4$ elements are processed.

well-suited for data sets that are ordered based on ascending/descending priority or significance to the final output. Examples include priority queues or bitwise operations.

Tree Permutation. For some computation stages, elements in data sets are not prioritized but are still ordered; the positions of elements are significant to the computation. Examples include image pixels or functions of time (e.g., audio wave signal, video frames). We find that an N-dimensional bit-reverse (or tree) permutation is well-suited for sampling these data sets. With a tree permutation, the data set is effectively accessed at progressively increasing resolutions. For example, sampling pixels in a tree permutation implies that after 4 pixels have been processed, a 2×2 image is sampled. After 16 pixels, a 4×4 image is sampled, and so on. This is visualized and discussed later in Figure 5.

The tree permutation accesses elements in bit-reverse order along each of N dimensions, interleaving between dimensions. Thus $p(i)$ is simply a permutation of the bits of index i. For example, the tree permutation for a one-dimensional set of 16 elements can be expressed as:

$$p: \quad b_3b_2b_1b_0 \rightarrow b_0b_1b_2b_3$$

where b_j is the jth bit of the set index i. This is shown in Figure 4. Elements are accessed in the form of a perfect 2^N-ary tree, where $N = 1$. This produces samples with progressively increasing resolution along one dimension. Note that since the tree permutation is a one-to-one correspondence of bits in the set index, p is a bijective function.

Figure 5 shows an example of the tree permutation on a two-dimensional data set (e.g., image pixels). For 8×8 elements, the permutation function p can be expressed as:

$$p: \quad b_5b_4b_3\,b_2b_1b_0 \rightarrow b_5b_3b_1\,b_4b_2b_0 \rightarrow b_1b_3b_5\,b_0b_2b_4$$

where $b_5b_4b_3$ is the original row index and $b_2b_1b_0$ is the original column index. First, the set index is deinterleaved to produce new row and column indices. Then the new row and column indices are each reversed. As before, elements are accessed in the form of a perfect 2^N-ary tree, where $N = 2$. This produces samples with progressively increasing two-dimensional resolution.

Pseudo-Random Permutation. When the data set is unordered, to avoid bias in the memory ordering of elements, we find that a pseudo-random permutation is most suitable. Examples include simulated annealing, k-means clustering or histogram construction (Figure 3). A true random permutation would be ideal; however, the permutation function p would not be bijective (i.e., we would not be able to guarantee that all elements are processed exactly once). For a pseudo-random permutation, p can be computed using any deterministic pseudo-random number generator. In our experiments, we use a linear-feedback shift register (LFSR), which is very simple to implement in hardware.

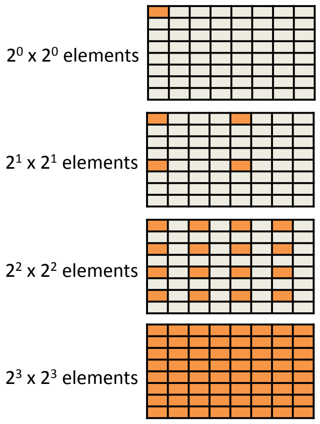


Figure 5: Two-dimensional tree sampling permutation example. This shows which indices have been accessed after $2^0, \ldots, 2^6$ elements are processed.

3.2 Anytime Pipeline

The anytime automaton is able to extract more parallelism out of applications. Consider the example in Figure 1. In the original application, computation i is dependent on g and h, which are both dependent on f. These dependences enforce that the computations execute sequentially, as is written in the example code. However, by building the pipeline, the automaton model allows all computations to run in parallel. Figure 2 takes a closer look. By recognizing that f can be broken down into f_1 and f_2, our model is able to provide an intermediate (but still acceptable) output of f. This allows g and h to begin executing without having to wait for all of f to finish.

This section discusses how to compose anytime (and non-anytime) stages into a pipeline. Without loss of generality, we limit the discussion to two computation stages:

$$f(I) \rightarrow F \qquad g(F) \rightarrow G$$

where g is dependent on f. If f is an anytime computation, then g can be computed on any or all intermediate F_i outputs such that:

$$g(F_1), \ldots, g(F_n) \rightarrow G$$

where $g(F_i) \rightarrow G_{Fi}$. At any point in time, g processes whichever output F_i happens to be in the buffer. Both stages can simply execute concurrently and independently of each other; no synchronization between them is necessary to ensure correctness. The only requirement is that g is eventually computed on $F_n = F$ to produce the precise output $G_{Fn} = G$. This implies that the precise output is always reachable.

These stages form a parallel pipeline since any f_i can execute in parallel to any $g(F_j)$ where $j < i$. An example is shown in Figure 6. The outputs of f flow through the pipeline, producing final outputs $O_1, \ldots, O_n$ with progressively increasing accuracy. At any point in time, g simply

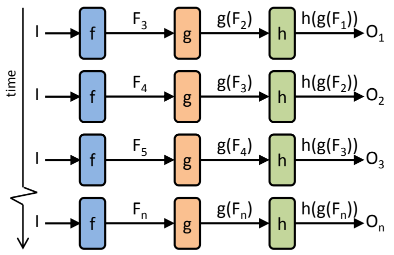


Figure 6: Anytime pipeline example.

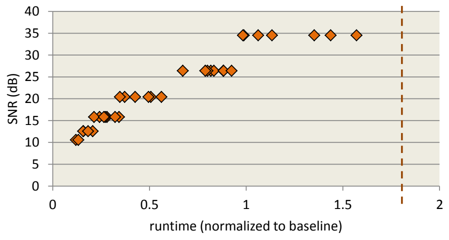


Figure 7: Runtime-accuracy of 2dconv anytime automaton. The vertical line indicates an SNR of infinity.

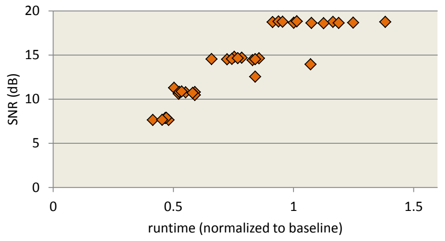


Figure 8: Runtime-accuracy of histeq anytime automaton.

processes the most recent available output of f. As a result, though f, g and h may be sequentially dependent in the original application, the automaton allows them to execute in parallel.

If g is also an anytime computation, then each $g(F_i)$ can be represented as:

$$g_1(F_i, G_{Fi,0}), \ldots, g_m(F_i, G_{Fi,n-1}) \rightarrow G_{Fi}$$

where $G_{F1,0} = \ldots = G_{Fn,0} = G_0$, and $g_j(F_i, G_{Fi,j-1}) \rightarrow G_{Fi,j}$. As before, the precise output G_{Fn} is guaranteed since all computations $g_1, \ldots, g_m$ eventually execute on F_n.

4. METHODOLOGY

We perform our evaluation of anytime automata on real machines, demonstrating attractive runtime-accuracy tradeoffs even without specialized hardware. We run experiments on IBM Power 780 (9179-MHD) machines. We use two nodes with four 4.42 GHz POWER7+ cores each, with four-way hyper-threading per core, yielding 32 hardware threads in total. The system consists of 256 KB of L2 cache and 10 MB of L3 (eDRAM) cache per core. All benchmarks are parallelized (both in the baseline precise execution and in the anytime automaton execution) to fully utilize the available hardware threads.

We evaluate our anytime automaton model on benchmarks from PERFECT [2], a suite containing a variety of kernels for embedded computing. We use large image input sets and measure accuracy in terms of the signal-to-noise ratio (SNR) of the approximate output relative to the baseline precise output. We focus on three approximate benchmarks that are widely used, are applicable to real-time computing and have visualizable outputs.

2dconv. 2D convolution applies a convolutional kernel to spatially filter an image; in our case, a blur filter is applied. This is common in computer vision and machine learning. The benchmark is simple in structure, yielding a single-stage anytime automaton. We employ output sampling with a tree permutation in generating the filtered image.

histeq. Histogram equalization enhances the contrast of an image using a histogram of image intensities. This is common in satellite and x-ray imaging. We construct an automaton with four computation stages in the pipeline. The first stage builds a histogram of pixel values using anytime pseudo-random input sampling, similar to the example in Figure 3. The second and third stages are not anytime; they construct a normalized cumulative distribution function from the histogram. The fourth stage generates the high-contrast image using tree-based output sampling.

debayer. This benchmark converts a Bayer filter image from a single sensor to a full RGB image, common in image sensors for security cameras and x-ray imaging. The computation performs simple interpolations; we use a single-stage automaton with tree-based output sampling.

5. EVALUATION

In this section, we present the performance-accuracy tradeoffs of our anytime automata. The runtime-accuracy results are shown in Figures 7 (2dconv), 8 (histeq) and 9 (debayer). These plots are generated from multiple runs, executing each automaton and halting it after some time to evaluate its output accuracy. The x-axis is the runtime of the automaton normalized to the baseline precise execution. The y-axis is our accuracy metric SNR in decibels. We later show example image outputs to relate SNR to image quality. The vertical line indicates the point where SNR reaches infinity (precise output). This is shown for all benchmarks except for histeq, where precise output is reached at 6× the runtime of the baseline; this is high due to non-anytime computations as discussed later. From our runtime-accuracy results, our model maintains the universal and most important trend in that accuracy increases over time and eventually reaches precise output. Though the rate of increase varies among benchmarks, this trend is maintained.

As shown in Figures 7 and 9, 2dconv and debayer benefit greatly from the anytime automaton model. At only 21% of the baseline runtime, 2dconv is able to produce an output with an SNR of 15.8 dB, which may be acceptable in certain use cases. The outputs are visualized in Figures 10 and 12 for 2dconv and debayer respectively, comparing against the baseline precise output. The benchmarks are able to achieve high accuracy at low runtimes because their computations are well-suited for sampling and their pipelines are simple. Despite the good results, neither 2dconv nor debayer reach precise output as early as the baseline execution. This is pri-

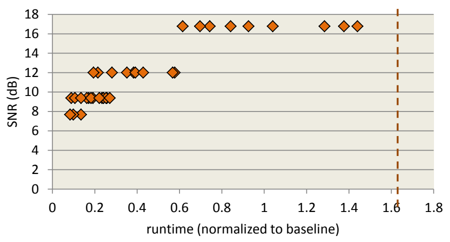
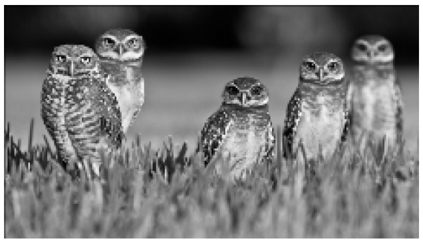
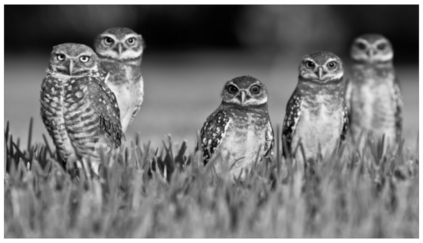


Figure 9: Runtime-accuracy of debayer anytime automaton. The vertical line indicates an SNR of infinity.

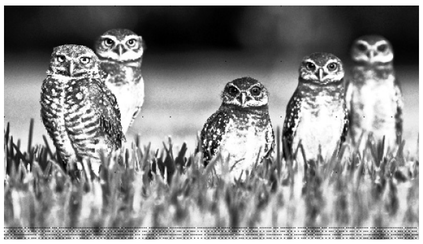


(a) 21% runtime, SNR 15.8dB

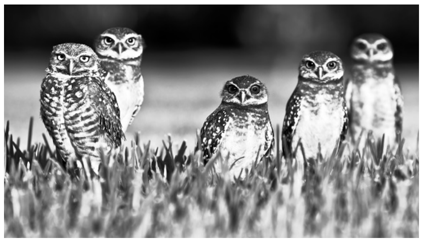


(b) baseline precise

Figure 10: Output of 2dconv anytime automaton.



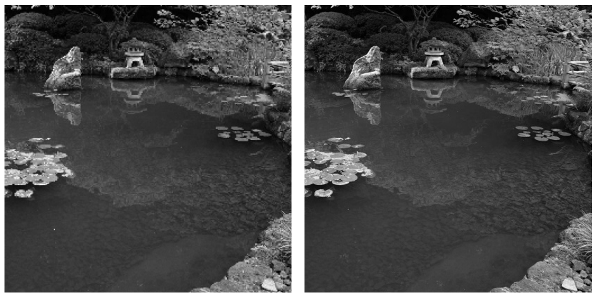
(a) 66% runtime, SNR 14.6dB



(b) baseline precise

Figure 11: Output of histeq anytime automaton.

marily due to poor cache locality from non-sequential sampling permutations. This can be alleviated via architectural optimizations, which we leave for future work.



(a) 21% runtime, SNR 12.0dB (b) baseline precise

Figure 12: Output of debayer anytime automaton.

As shown in Figure 8, histeq does not perform as well as 2dconv and debayer. This is due to the presence of non-anytime stages. Non-anytime stages are common for performing small (typically sequential) tasks such as normalizing data structures and reducing thread-privatized data. Despite this, histeq is able to produce outputs of acceptable quality (in certain cases) at only 66% of the baseline runtime, visualized in Figure 11. Note also that though some computation stages are not anytime in our design, it may still be possible to make them anytime using other methods. This motivates future research avenues in wider design space exploration of anytime automata and new anytime approximate computing techniques.

6. CONCLUSION

We propose the Anytime Automaton, a novel computation model that represents an approximate application as a parallel pipeline of computation stages with anytime sampling. This allows the application to execute such that 1) it can be interrupted at any time while still producing a valid approximate output, and 2) its output quality is guaranteed to increase over time and approach the precise output. This addresses the fundamental drawbacks of state-of-the-art approximate computing techniques: 1) they do not provide any accuracy guarantees at runtime, and 2) they require complex rollback or training mechanisms to control the trade-off between accuracy and performance/energy. With the anytime automaton model, the application can be stopped at any point that the user is satisfied, expending just enough time and energy for an acceptable output. If the output is not acceptable, it is a simple matter of letting the application run longer.

7. REFERENCES

[1] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions on Computers*, 2005.

[2] 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/.

[3] T. L. Dean and M. Boddy, "An analysis of time-dependent planning," in *Proceedings of the National Conference on Artificial Intelligence*, 1988.

[4] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proc. Int. Symp. Microarchitecture*, 2012.

[5] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power," in *Proc. Int. Symp. Computer Architecture*, 2002.

[6] A. Garvey and V. Lesser, "Design-to-time real-time scheduling," *IEEE Transactions on Systems, Man and Cybernetics*, 1993.

[7] B. Grigorian, N. Farahpour, and G. Reinman, "BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing," in *Proc. Int. Symp. High-Performance Computer Architecture*, 2015.

[8] E. J. Horvitz, "Reasoning about beliefs and actions under computational resource constraints," in *Workshop on Uncertainty in Artificial Intelligence*, 1987.

[9] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An Online Quality Management System for Approximate Computing," in *Proc. Int. Symp. Computer Architecture*, 2015.

[10] V. Lesser, J. Pavlin, and E. Durfee, "Approximate processing in real-time problem-solving," *AI Magazine*, vol. 9, no. 1, pp. 49–61, 1988.

[11] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: saving DRAM refresh-power through critical data partitioning," in *Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2011.

[12] R. Mangharam and A. A. Saba, "Anytime Algorithms for GPU Architectures," in *Proceedings of the Real-Time Systems Symposium*, 2011.

[13] 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 *Proc. Int. Symp. High-Performance Computer Architecture*, 2015.

[14] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," in *Proceedings of the Conference on Design Automation and Test in Europe*, 2010.

[15] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener, "Programming with relaxed synchronization," in *Proc. Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, 2012.

[16] M. Samadi, J. Lee, D. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning approximation for graphics engines," in *Proc. of the Int. Symp. on Microarchitecture*, 2013.

[17] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *Proc. Int. Symp. Microarchitecture*, 2013.

[18] J. San Miguel, M. Badr, and N. Enright Jerger, "Load value approximation," in *International Symposium on Microarchitecture*, 2014.

[19] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conf. on Foundations of software engineering*, 2011, pp. 124–134.

[20] 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 *Proc. of the Int. Symp. on Computer Architecture*, 2014.

[21] M. Sutherland, J. San Miguel, and N. Enright Jerger, "Texture cache approximation on gpus," in *Workshop on Approximate Computing Across the Stack*, 2015.

[22] 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 VLSI Systems*, 2000.

[23] J. W. S. L. W-K. Shih and J.-Y. Chung, "Fast algorithms for scheduling imprecise computations," in *Proceedings of the Real-Time Systems Symposium*, 1989.

[24] T. Yeh, P. Faloutsos, S. Patel, M. Ercegovac, and G. Reinman, "The art of deception: Adaptive precision reduction for area efficient physics acceleration," in *Int. Symp. on Microarchitecture*, Dec 2007.

[25] S. Zilberstein, "Operational Rationality through Compilation of Anytime Algorithms," Ph.D. dissertation, Technion - Israel Institute of Technology, 1982.