2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

The Anytime Automaton

Joshua San Miguel and Natalie Enright Jerger
University of Toronto
joshua.sanmiguel@mail.utoronto.ca, enright@ece.utoronto.ca

***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 offline or online accuracy models, or 2) present experimental results that demonstrate empirically low error. Unfortunately, these approaches are still unable to guarantee acceptability of *all* application outputs at runtime. We offer a solution that revisits concepts from anytime algorithms. Originally explored for real-time decision problems, anytime algorithms have the property of producing results with increasing accuracy over time. We propose the Anytime Automaton, a new computation model that executes applications as a parallel pipeline of anytime approximations. An automaton produces approximate versions of the application output with increasing accuracy, guaranteeing that the final precise version is eventually reached. The automaton can be stopped whenever the output is deemed acceptable; otherwise, it is a simple matter of letting it run longer. We present an in-depth analysis of the model and demonstrate attractive runtime-accuracy profiles on various applications. Our anytime automaton is the first step towards systems where the acceptability of an application's output directly governs the amount of time and energy expended.**

I. 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 [6], [9], [11], [17], [18], [20], [22]. However, despite the substantial benefits offered by approximate computing, it has not yet earned widespread acceptance to merit adoption in real processors. This is due to the fundamental challenge of providing guarantees on error. The approach in state-of-the-art techniques is to 1) provide offline statistical profiling [3], [6] or online sampling/predictive mechanisms [3], [11], [18] to try to control runtime accuracy, and 2) present experimental results that show how error can lie within some empirical range. However, these approaches are still unable to enforce strong guarantees on acceptability of *all* outputs at runtime. This is a very challenging task, since acceptability of an approximation is inherently subjective. Furthermore, runtime error is dependent on many factors: application algorithm/code, input data, hardware configuration, operating system and runtime environment (e.g., co-executing processes, I/O interaction). Neither the system designers, the programmers nor the users have control over all of these factors. For example, a programmer may implement relaxed synchronization [17] in their application and evaluate the output on an architecture with limited parallelism. However, a user may execute this application on a processor with many cores, yielding far more synchronization conflicts and unacceptable outputs. To earn widespread adoption, it is imperative that system designers and architects find a way to implement approximate computing techniques that can address these challenges.

Our work tackles this problem by revisiting concepts from anytime algorithms [5], [10], [12]. Originally proposed for planning and decision processes in artificial intelligence, anytime algorithms are defined by two key properties: 1) they can be stopped at any time while still producing a valid result, and 2) they guarantee progressively increasing quality over time. We believe that these properties offer a solution to the challenges of approximate computing. However, in prior work, anytime algorithms are built into the derivation of a specific application and are thus difficult to apply to other applications. Our work generalizes the anytime concept to approximate computing such that the acceptability of an approximation is simply defined by how long the user chooses to run the application.

We propose the **Anytime Automaton**, a new computation model that represents an approximate application as a parallel pipeline of anytime computations. Our model enables early availability of the application output: approximate versions of the output are produced with increasing accuracy, guaranteeing that the final precise version is reached eventually. It also enables interruptibility: the automaton can be stopped whenever the current approximate output is deemed acceptable; otherwise, it is simply a matter of letting it run longer. Furthermore, the pipeline organization is able to extract parallelism even out of sequential computations. Whereas state-of-the-art approximate computing techniques employ dynamic accuracy control on code segments of applications [3], [9], [11], [18], our model provides early availability of the whole application output since the accuracy of individual segments does not necessarily translate to accuracy of the whole application. Our model is also valuable in user-interactive environments where acceptability cannot be defined a priori and in real-time environments where absolute time/energy constraints need to be met.

This work was done in part when Joshua San Miguel was at IBM T. J. Watson Research Center [23].

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.54

Imagine typing a search engine query and instead of pressing the `enter` key, you hold it based on the desired amount of precision in the search. With the anytime automaton, we advocate for systems where the acceptability of the output directly governs the amount of time and energy consumed (*hold-the-power-button* computing).

We make the following novel contributions:

- We propose the Anytime Automaton, a new computation model that rethinks the way we use approximate computing, providing the guarantee of improved accuracy over time.
- We evaluate our model on PERFECT [4] and AxBench [6] applications, demonstrating promising results with runtime-accuracy profiles.

II. 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.

A. Approximate Computing

Approximate computing introduces application output error/accuracy as an axis in architectural design, which can be traded 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 [24] and relaxed synchronization [17] (i.e., approximation via lock elision). In hardware, many techniques exploit the physical characteristics of computation [16] and storage elements [7], [13], [20]. Other techniques approximate based on data precision [28], [32] as well as previously seen values [22], [21], [27], [31] and computations [2]. Though these techniques achieve substantial efficiency gains, it can be difficult to reason about accuracy of the application output.

Prior work has proposed offline profiling techniques to build accuracy models [3]. This is also employed via training in neural-based approximations [6], [9], [15], [26]. Though these significantly improve accuracy, offline methods can only draw statistical conclusions and cannot guarantee acceptability of all computations during runtime. Online accuracy control can be implemented via sampling methods [3], [18] or predictive models [11]. However, coverage of sampling and prediction is inherently imperfect and cannot ensure that the accuracy of a given output during runtime is acceptable. Accuracy control is best left to users, since the definition of what is acceptable varies from one case to another. BRAINIAC [9] addresses this using a multi-stage flow of neural accelerators; we later show how such an iterative approach can be generalized via our model. We address these challenges by revisiting concepts from anytime algorithms.

B. 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 [5], [10], [12]. 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 [33]. Contract algorithms make online decisions to schedule their computations to meet a runtime deadline. Researchers have explored optimal scheduling policies for contract anytime algorithms [8], [29] and the error composition of anytime algorithms [33]. 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 [14].

Anytime algorithms derive strong accuracy guarantees 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 motivate a rethinking of how approximate computing is implemented in systems; we introduce a computation model that enables the integration of approximate computing techniques in an anytime way.

III. The Anytime Automaton

The **Anytime Automaton** is a new computation model that enables approximate computing in an anytime way:

1) It generalizes how to apply approximation techniques to computations such that accuracy increases over time and is guaranteed to eventually reach the precise result.
2) It executes these computations as a parallel pipeline such that approximate versions of the whole application output are available early.
3) It enables interruptibility such that execution can be stopped when the current approximate output is deemed acceptable; otherwise, it is a simple matter of running longer, eventually reaching the precise output.

The anytime automaton is valuable in user-interactive environments where acceptability cannot be defined a priori. It is also valuable in real-time systems where applications need to support interruptibility in order to meet hard time/energy constraints. Our model also fits well with conventional approximate computing where the degree of approximation

is dynamically tuned based on accuracy metrics [3], [9], [11], [18]. These accuracy metrics are measured on either 1) the whole application output (which necessitates re-execution of the entire application if accuracy is insufficient), or 2) the outputs of approximate code segments (which does not necessarily translate to the accuracy of the whole application). The pipelined design of our automaton addresses this by providing early availability of the whole application output: starting from low-accuracy approximate versions to the eventual precise version.

As a parallel to dataflow models, the anytime automaton can be viewed as a **data diffusion model**. In the former, information is passed down from computation to computation; in the latter, information is *diffused* (i.e., *updates/versions* of the information are passed down). Imagine that the precise version of the application output is a fluid. Instead of waiting for the fluid to flow in its entirety, its particles are diffused into the output, gradually increasing the concentration of precise information.

In this section, we first provide an overview of the anytime automaton model (Section III-A). We then describe how approximate computing techniques are applied in an anytime way (Section III-B) and show how they are composed into a parallel pipeline (Section III-C). We conclude with a summary example that brings all key concepts together (Section III-D).

A. Model Overview

Figure 1 shows a high-level overview of an anytime automaton. An approximate application is broken down into computation stages with input/output buffers, connected in a directed, acyclic graph. These stages can be arbitrarily large or small. Approximate computing techniques are then applied to each stage in an anytime way. This allows stages to execute in parallel as a pipeline, 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 an 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 via dynamic accuracy metrics, user-specified or enforced by time/energy constraints. In all cases, the user and system designer 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 (i.e., f_1 produces an approximate version of its output and f_2 produces the final precise version). As soon as f_1, g_1, h_1 and i_1 have executed, an approximate output O_{1111} is available, and thus the application can already be stopped here. If the approximate

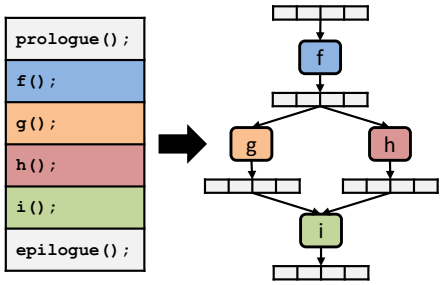

Figure 1: High-level overview of anytime automaton.

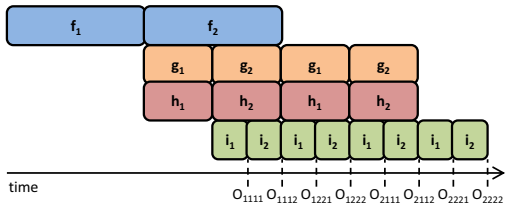

Figure 2: Parallel pipeline of anytime automaton.

output is not acceptable, the pipeline can simply continue executing, progressively improving the output accuracy until the final precise output O_{2222}. 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.

B. Anytime Computation Stages

We describe how to apply an approximate computing technique onto a computation stage f such that it produces an output of increasing accuracy over time. We say that the resulting stage f is an **anytime** computation stage. The general approach is to apply the technique iteratively with decreasing error (Section III-B1). In many cases, a technique can be applied diffusively instead (Section III-B2) to avoid any redundant work introduced by the general iterative approach. In our discussion, a computation stage f with input I and output O is represented as:

$$f(I) \rightarrow O$$

1) Iterative Computations: The general approach is to convert f to an **iterative** anytime computation stage. To do this, f is executed n times sequentially, yielding the set of computations:

$$f_1(I), ..., f_n(I) \rightarrow O$$
$$\forall\, i, \quad f_i(I) \rightarrow O_i$$

where $i \in [1...n]$ and $n > 1$. These computations are executed one after the other starting from f_1 until f_n. Outputs $O_1, ..., O_n$ are produced as each intermediate computation

completes, with each O_i overwriting the previous O_{i-1} in the output buffer upon completion of f_i. The approximate computing technique is applied to each f_i at iteratively increasing accuracy levels such that f_i has greater accuracy than f_{i-1}. The final computation f_n is simply the precise version of f (i.e., the approximation technique is disabled). For example, if applying reduced floating-point precision, f_1 computes f with the lowest precision while f_n computes with the highest. Similarly, BRAINIAC employs this iterative approach in its multi-stage flow of neural accelerators [9]. In this way, f becomes an anytime stage with increasing accuracy over time, eventually reaching the precise output $O_n = O$. Our model imposes Property 1 to ensure that when f_n executes, it is guaranteed to produce the precise output. Most approximate regions of code in applications are pure functions [11]. In the simple example in Figure 2, f, g, h and i are all anytime stages with $n = 2$.

Property 1. *For an anytime computation stage f, each and every intermediate computation $f_1, ..., f_n$ must be a pure function; the computation does not depend on nor modify any external semantic state aside from the data in its input and output buffers.*

In this section, we present examples of how to derive iterative computations using common approximate computing techniques: loop perforation and approximate storage.

Loop Perforation: Loop perforation is a technique that jumps past loop iterations via some fixed stride, trading off lower output accuracy for lower runtime and energy consumption [24]. Loop perforation can be made anytime by iteratively re-executing the perforation with progressively smaller strides. Given a computation stage f, applying loop perforation iteratively involves selecting a set of unique strides $s_1, ..., s_n$. We construct intermediate computations $f_1, ..., f_n$ such that each f_i executes f with perforated loops of stride s_i. The strides are chosen such that $s_i < s_{i-1}$ and $s_n = 1$. This enables accuracy to increase over time and ensures that the final computation f_n computes the precise version of f.

Note that this approach yields redundant work for loop iterations that are common multiples of the selected strides. For example, the instructions at iteration $s_1 \times s_2$ of the loop are executed twice: once in f_1 and again in f_2. Furthermore, if the precise output is needed, all loop iterations executed in previous computations $f_1, ..., f_{n-1}$ must be executed again in f_n. In some cases, this redundant work can be avoided via sampling techniques (since loop perforation is effectively a form of sampling), as we discuss in Section III-B2.

Approximate Storage: Architects have proposed designs for approximate storage elements. They recognize that many applications—particularly multimedia processing—are tolerant to noisy input and output data. Such data can be stored in storage devices with relaxed physical constraints, risking bit failures for improved energy-efficiency. For example, drowsy caches [7] reduce SRAM cell supply voltage, increasing bit failure probability while saving significant energy. Similarly, low-refresh DRAM [13] and approximate phase-change memory [20] allow for efficiency gains at the cost of potential data corruption.

Applying approximate storage techniques iteratively requires a means of controlling the accuracy-efficiency tradeoff of the storage device. For example, the SRAM supply voltages can be dynamically scaled in caches; as voltage increases, the lower the risk of bit failure. With this, $f_1, ..., f_n$ can be defined as executing the compuation f at increasing accuracy levels of storage (e.g., increasing SRAM supply voltage in a drowsy cache). Correctness is ensured by using the nominal (precise) storage operation in f_n. Note that approximate storage techniques are data-destructive; that is to say, when a bit is corrupted in a storage device (e.g., drowsy cache), it remains corrupted even after raising the device accuracy level (e.g., increasing supply voltage). Thus at the beginning of each intermediate computation f_i, the storage device must be flushed (or reinitialized to precise values) so that bit corruptions from f_{i-1} do not degrade the accuracy of f_i. Alternatively, separate storage devices can be used for each of $f_1, ..., f_n$, though this incurs a much larger area cost.

2) Diffusive Computations: As discussed in Section III-B1, iterative computations are effectively re-executions of the baseline computation under varying degrees of approximation. By construction, they introduce redundant work, the amount of which increases as more re-executions are performed. This is because in an iterative stage, each intermediate computation f_i overwrites O_{i-1} (i.e., the result of f_{i-1} that is currently in the output buffer). This negates any useful work done by any preceding computations. It is more desirable for each subsequent f_i to use O_{i-1} and build upon it. In this way, the accuracy of the output improves via useful *updates* from each f_i, as opposed to improving accuracy via *rewrites* from each f_i as in an iterative stage. Accuracy/precision is effectively *diffused* into the output buffer.

We say that such a computation stage f is **diffusive** and represent it as:

$$f_1(I, O_0), ..., f_n(I, O_{n-1}) \rightarrow O$$
$$\forall\, i, \quad f_i(I, O_{i-1}) \rightarrow O_i$$

where $i \in [1...n]$, $n > 1$, and O_0 is the initial value in the output buffer. Unlike an iterative stage, each intermediate computation f_i is dependent on the state of the output buffer O_{i-1} resulting from the computations before it. Note that Property 1 is still satisfied; the only difference is that the output buffer is treated as an input as well. Correctness is ensured by deriving $f_1, ..., f_n$ such that their final aggregate output O_n equals the precise output. In this way, each f_i contributes usefully to the final result (i.e., its intermediate output O_i is necessary for reaching the precise output).

In this section, we go into detail on how to derive diffusive approximations using data sampling techniques and reduced fixed-point precision.

Data Sampling: We describe 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.

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 diffusive computation stage f is **commutative** if it can be represented as:

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

where $\triangle$ is some commutative operation.

It may be undesirable to sample inputs in their default memory order since it gives bias to elements at lower addresses. For a commutative stage f, the final precise output can be computed from any sequential ordering of $x_1, ..., x_n$. A better approach to input sampling is to simply permute the order of the $x_1, ..., x_n$ computations such that:

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

where $p(i)$ is a bijective function (i.e., a one-to-one and onto mapping of i). Each intermediate computation f_i represents a sample of size i of the input set. We say that p is the **permutation** function. As long as p is bijective, the precise output is guaranteed since all x_i computations are still performed exactly once. Later in this section, we discuss various permutation functions and their suitability for different applications. Figure 3 shows an example of anytime histogram construction using input sampling with a pseudo-random permutation. As more input elements are processed over time, the approximate histogram approaches the precise output.

Note that if $\triangle$ is not an idempotent operator (i.e., $\triangle$ is idempotent if $\alpha \triangle \alpha = \alpha$), the output may need to be normalized/weighted using the current sample size and the total population size. For example, consider input sampling on an anytime sum. Addition is not an idempotent operation. If I is a set of random positive integers, then the output value is monotonically increasing. Because of this, the value of $O_{n/2}$, for example, will likely be approximately half of the precise output O_n. To address this, any dependent stages that use O_i should use a weighted O'_i instead:

$$O'_i = O_i \times n/i$$

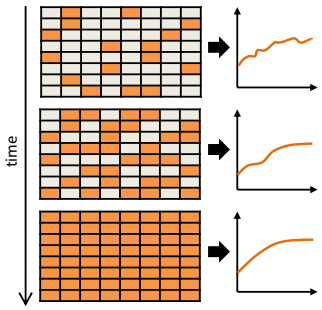

Figure 3: Example of input sampling with a pseudo-random permutation for anytime histogram construction.

Idempotent operations (e.g., bitwise-and, bitwise-or, set-union, set-intersection, min, max) do not require such normalization.

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. This is a special case of a commutative anytime computation; the commutative operation is effectively a union of disjoint sets: $O_{i-1} \cup X_{m(i)}$. Thus it is amenable to sampling.

Unlike input sampling, output sampling permutes the order of the output elements $O[i]$ such that:

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

where $p(i)$ is a bijective permutation function. 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.

Sampling Permutations. We now discuss permutation functions that can be used for both input and output sampling. Depending on the computation, some permutations may be more suitable than others. For example, 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. 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).

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

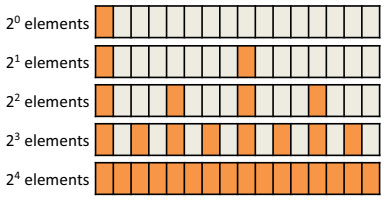

Figure 4: One-dimensional tree sampling permutation example. This shows which indices have been accessed after $2^0, ..., 2^4$ elements are processed.

$i \in [1...n]$. Sequential sampling is 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 (e.g., reduced fixed-point precision in Section III-B2).

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.

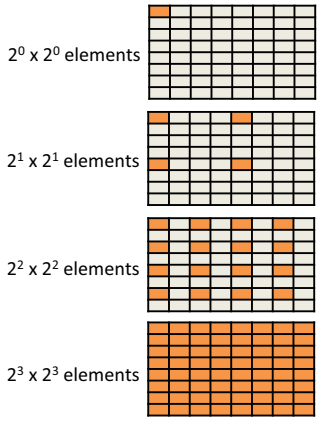

Figure 5: Two-dimensional tree sampling permutation example. This shows which indices have been accessed after $2^0, ..., 2^6$ elements are processed.

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.

Reduced Fixed-Point Precision: Reduced fixed-point (or integer) precision techniques perform computations with only a subset of data bits. This can be viewed as a form of sampling since the bit representation $b_{n-1}...b_0$ of an integer is merely a sum of powers of two:

$$b_{n-1}...b_0 \;=\; b_{n-1} \cdot 2^{n-1} \;+\; ... \;+\; b_0 \cdot 2^0$$

Since addition is a commutative operation, the representation of integer and fixed-point data is amenable to sampling. Computing with reduced precision improves both latency and energy.

This observation extends to other operations that are distributive to addition (e.g., multiplication). For example, a stage f computing an anytime reduced-precision dot product of two vectors I and W can be represented as:

$$\forall\, i, \quad f_i(I, O_{i-1}) = O_{i-1} \;+\; (I \cdot (W \;\&\; 2^{32-i}))$$

where $O_0 = 0$, and elements in the vectors are 32-bit integers or fixed-point values such that $i \in [1...32]$. This computation is effectively applying input sampling on the bits of elements in W. This draws from classic techniques of bit-serial computations [25], [30]. Sampling is performed with a sequential permutation, since the most-significant bits

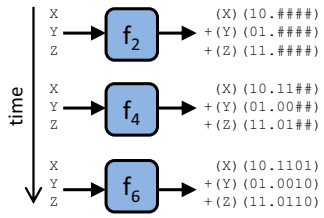

Figure 6: Reduced-precision fixed-point dot product.

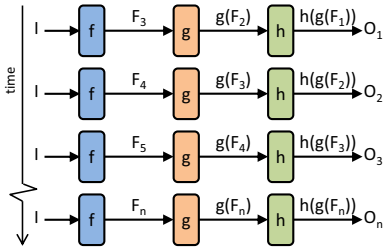

Figure 7: Asynchronous pipeline example.

should be prioritized. This is illustrated in Figure 6 with 6-bit fixed-point data, where I is a vector consisting of values `X`, `Y` and `Z`. The output of f increases in accuracy over time and approaches the precise output as more bits are computed. Note that this computation does not perform any additional work compared to the baseline non-anytime dot product, since integer (or fixed-point) multiplication is computed similarly as a sum of partial products.

C. Anytime Pipeline

In this section, we describe how to compose anytime computation stages into a parallel pipeline. The pipeline enables interruptibility and early availability of the whole application output. Interruptibility is essential in real-time and user-interactive environments, while early availability is essential in systems with dynamic error control (such as Rumba [11]) where error metrics should be applied to the whole output as opposed to the outputs of individual computations in the application. Furthermore, the pipeline can 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 the computations to execute sequentially, as 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.

Without loss of generality, we limit much of the discussion to two computation stages:

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

where g (the child stage) is dependent on f (the parent stage). The automaton is constructed such that Property 2 holds. This enforces a strict producer-consumer relation between parent and child stages and ensures that the parent executes independently of the child.

Property 2. *For an anytime computation stage f, all of its intermediate outputs $F_1,...,F_n$ are stored in a single output buffer, and no other computation stages are allowed to modify this buffer.*

We present two pipeline organizations: asynchronous and synchronous. An asynchronous pipeline (Section III-C1) is the general approach for composing stages, allowing them to run independently in parallel while still guaranteeing the eventual precise output. A synchronous pipeline (Section III-C2) leverages the distributivity of diffusive stages to avoid redundant computations.

1) Asynchronous Pipeline: An **asynchronous** pipeline is the general approach to composing multiple computation stages. Stages simply execute concurrently and independently of each other. If f is an anytime computation, then g can be computed on any or all intermediate F_i outputs such that:

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

where $g(F_i) \rightarrow G_{Fi}$. At any point in time, g processes whichever output F_i happens to be in the buffer. We say that this is an asynchronous pipeline since no synchronization is necessary between f and g to ensure correctness; the only requirement is that g is eventually computed on $F_n = F$ to produce the precise output $G_{Fn} = G$. Thus 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 $j < i$. The pipeline is constructed such that Property 3 holds, ensuring that g processes no other possible outputs aside from $F_1,...,F_n$. Note that correctness is still ensured even if f is not anytime (i.e., $n = 1$); thus the pipeline supports non-anytime stages.

Property 3. *For an anytime computation stage f, all of its intermediate outputs $F_1,...,F_n$ are written into its output buffer atomically.*

An example is shown in Figure 7. The outputs of f flow through the pipeline, producing final outputs $O_1,...,O_n$ with progressively increasing accuracy. At any point in time, g simply processes the most recent available output of f. The precise output is eventually reached since both g and h eventually compute on F_n.

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

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

where $g_j(F_i, G_{Fi,j-1}) \rightarrow G_{Fi,j}$ and $G_{F1,0} = ... = G_{Fn,0} = G_0$.

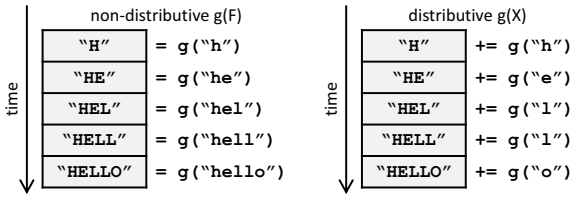

Figure 8: Example of distributive computation stage g.

As defined by the sequential ordering of $f_1, ..., f_n$, the output F_i is always produced after F_{i-1}. Thus we can guarantee that F_n is the eventual output of f. Following from this, $g(F_i)$ (and its output G_{Fi}) must always come after $g(F_{i-1})$ (and its output G_{Fi-1}). And similar to f, the sequential ordering of $g_1, ..., g_m$ within each $g(F_i)$ enforces that $g_j(F_i)$ (and its output $G_{Fi,j}$) always comes after $g_{j-1}(F_i)$ (and its output $G_{Fi,j-1}$). This guarantees that $G_{Fn,m}$ (which equals the precise output G) is the eventual output of $g(F)$.

2) Synchronous Pipeline: A synchronous pipeline prevents redundant computations when the parent stage f is diffusive and the child stage g is distributive over the computations in f. Assume f is a diffusive anytime stage that can be represented as:

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

where $\Diamond$ is some left-associative operator, and $x_i(I) \to X_i$. With a diffusive f, $X_1, ..., X_n$ are effectively the updates to the output F. We say that g is **distributive** over f if:

$$g(F) = g(F_0 \Diamond X_1 \Diamond ... \Diamond X_n) = g(F_0) \Diamond g(X_1) \Diamond ... \Diamond g(X_n)$$

As in the asynchronous pipeline, g can be computed on any or all intermediate F_i outputs such that:

$$g(F_1), ..., g(F_n) \to G$$

where $g(F_i) \to G_{Fi}$. However, since g is distributive, taking a closer look at each $g(F_i)$ and $g(F_{i-1})$, we can see that g performs redundant work:

$$g(F_{i-1}) = g(F_0) \Diamond g(X_1) \Diamond ... \Diamond g(X_{i-1})$$
$$g(F_i) = g(F_0) \Diamond g(X_1) \Diamond ... \Diamond g(X_{i-1}) \Diamond g(X_i)$$

Figure 8 shows an example where f is generating a string letter-by-letter (i.e., $\Diamond$ is the concatenation operator), and g capitalizes each letter in this string. If the current string value is F_i (e.g., "hel"), then computing $g(F_i)$ would involve capitalizing all letters, even the ones that were already processed previously in F_{i-1} (e.g., "he"). Since g is distributive, it only needs to capitalize each newly added letter X_i (e.g., "l"). Other examples of distributive computations include sorting/searching over a growing set of elements or matrix multiplication over addition. Thus composing distributive and diffusive stages via an asynchronous pipeline can yield redundant computations.

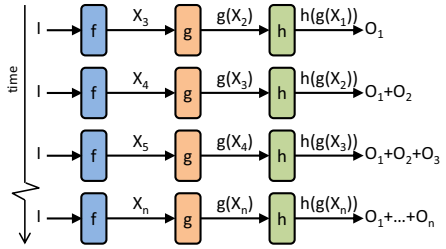

Figure 9: Synchronous pipeline example.

To address this, we can form a **synchronous** pipeline between f and g. We simply expose the intermediate updates X_i and redefine $g(F) \to G$ to $g^S(X) \to G$ such that:

$$\forall\, i, \quad g^S(X, G^S_{Fi-1}) = G^S_{Fi-1} \Diamond g(X_i)$$

where $G^S_{F0} = g(F_0)$. Unlike g, which takes in the output of f (i.e., F), g^S instead takes in the updates to F (i.e., X) as input. In the asynchronous pipeline, only $g(F_n)$ is needed to compute the precise output. However, in the synchronous pipeline, all $g^S(X_1), ..., g^S(X_n)$ are necessary. For this reason, f and g^S must synchronize such that f does not overwrite X_i with X_{i+1} before $g^S(X_i)$ begins executing. This forms a synchronous pipeline where each $f_i(I)$ can execute in parallel to $g^S(X_{i-1})$. Note that with such a pipeline, $g^S(X_1), ..., g^S(X_n)$ all contribute usefully towards the final precise output. An example is shown in Figure 9. The intermediate updates X flow through the pipeline instead of the outputs F as in the asynchronous pipeline (Figure 7).

D. Summary

We conclude this section with an example that summarizes key concepts of our model. Figure 10 shows an example application with two computation stages: $f(I) \to F$ and $g(F) \to G$. Stage f processes input sensor information to generate a matrix of fixed-point data (shown as `[AA.BB]`). Stage g is dependent on f; it computes the dot product of F with some matrix `[C]`.

As Figure 10 shows, in the *baseline* application, f and g simply execute one after the other. To construct an anytime automaton, we apply some approximate computing technique—say, reduced fixed-point precision—on f. The general approach is to apply the technique iteratively (*f iterative*); first with half-precision generating `[AA]` (f_1) then with full-precision generating `[AA.BB]` (f_2) if accuracy is not acceptable. Half-precision yields lower latency for both f and g. However, if it yields unacceptable error, both f and g need to be recomputed at full-precision, resulting in longer runtime overall. We recognize that f_2 is independent of the half-precision invocation of g. Thus we construct an asynchronous pipeline to allow the computation stages to execute in parallel (*f iterative, asynchronous pipeline*), reducing overall runtime. In this way, data is effectively passed down through a parallel pipeline.

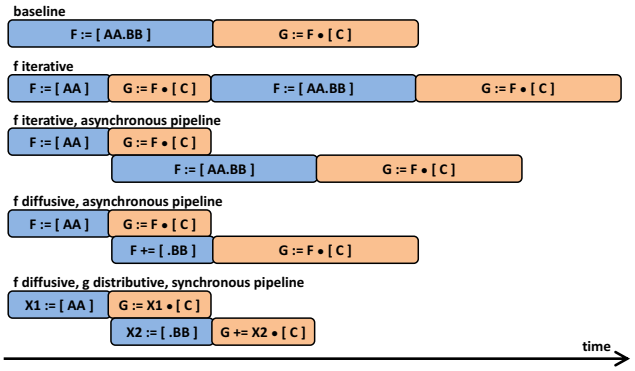

Figure 10: Example comparing varying anytime automaton organizations. Stage f applies anytime reduced-precision approximations to produce output matrix F. Dependent stage g computes dot product on F to produce output G.

By recognizing properties of common approximate computing techniques and application computations, we can minimize redundant work when constructing anytime automata. Since f is merely generating fixed-point values at varying precision, it can be constructed as a diffusive anytime stage. This implies that f_2 can use the current state of its output buffer (which is the output of f_1) to update the current output value without having to recompute at full-precision. Specifically, f_2 simply needs to add the rest of the bits `[.BB]` to the output of f_1 `[AA]`. Thus constructing f as a diffusive stage improves performance further (*f diffusive, asynchronous pipeline*). Note that this does not affect the latency of the full-precision invocation of g. This is because g is oblivious to f's diffusivity, so it needs to perform the full-precision computation on `[AA.BB]`. To improve on this, we can construct a synchronous pipeline (*f diffusive, g distributive, synchronous pipeline*), since the dot product in g is distributive over the addition operations of the updates to f. Stage g is modified to take as input the output buffer updates (X_1 and X_2 for f_1 and f_2, respectively) instead of the output values themselves (F_1 and F_2), yielding lower overall runtime. In this way, data is effectively diffused (as opposed to passed down) through the entire pipeline. From this example, we see how our model is able to transform approximate applications into automata where accuracy is guaranteed to increase over time.

IV. Evaluation

In this section, we construct anytime automata for various applications and evaluate the runtime-accuracy tradeoff under different approximate computing techniques. We then discuss other design considerations for anytime automata.

A. Methodology

This section describes the methodology and approximate applications that we use in our experiments.

1) Experiments: We perform our evaluation of anytime automata on real machines, demonstrating attractive runtime-accuracy tradeoffs even without specialized hardware. We also simulate the impact of approximate computing techniques, such as reduced-precision operations and approximate storage, to show their impact on error. 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 applications are parallelized (both in the baseline precise execution and in the anytime automaton) to fully utilize the available hardware threads.

2) Applications: We evaluate our anytime automaton model on applications from PERFECT [4], a benchmark suite containing a variety of kernels for embedded computing, and AxBench [6], an approximate computing benchmark suite. We focus on five approximate applications that are widely used, are applicable to real-time computing and have visualizable outputs for our evaluation. We use large image input sets for all applications. We measure accuracy in terms of signal-to-noise ratio (SNR)—a standard metric in image processing—of the approximate output relative to the baseline precise. SNR is measured in decibels (dB) where ∞ dB is perfect accuracy. Since acceptability is naturally subjective, we present sample outputs in our evaluation.

2d convolution (**2dconv**) from PERFECT applies a convolutional kernel to spatially filter an image; in our case, a blur filter is applied. It consists of many dot products, computed for each pixel. This is a common computation in computer vision and machine learning. The application is simple in structure, yielding an anytime automaton with a single diffusive stage. We employ output sampling with a tree permutation in generating the filtered image. We also evaluate reduced fixed-point precision (Section IV-B1) and approximate storage (Section IV-B2) on 2dconv.

Histogram equalization (**histeq**) from PERFECT 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 an asynchronous pipeline. The first stage is diffusive; it 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 diffusive stage generates the high-contrast image using tree-based output sampling.

Discrete wavelet transform (**dwt53**) from PERFECT performs a discretely-sampled wavelet transform on an image. This computation is a common form of data compression. We approximate the transform and then execute the inverse transform precisely; accuracy is measured on the inversed output relative to the original image. Our automaton consists

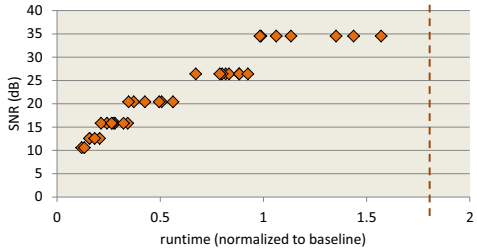

Figure 11: Runtime-accuracy of 2dconv anytime automaton. The vertical line indicates an SNR of ∞ dB.

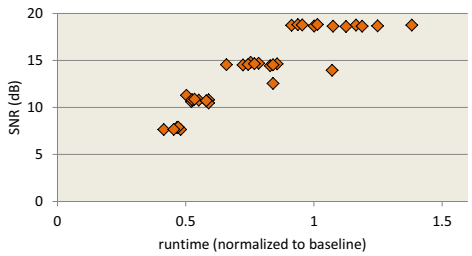

Figure 12: Runtime-accuracy of histeq anytime automaton.

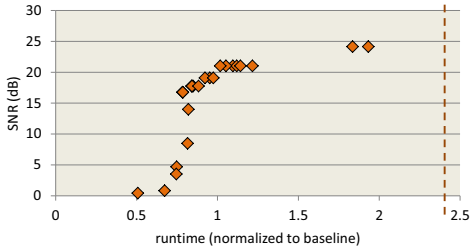

Figure 13: Runtime-accuracy of dwt53 anytime automaton. The vertical line indicates an SNR of ∞ dB.

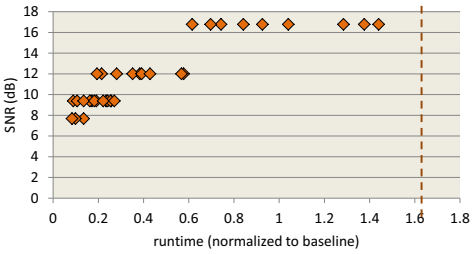

Figure 14: Runtime-accuracy of debayer anytime automaton. The vertical line indicates an SNR of ∞ dB.

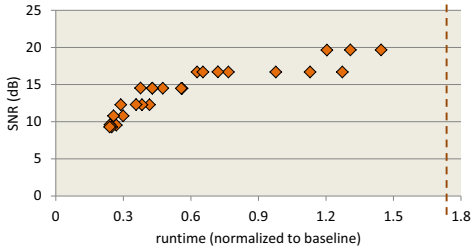

Figure 15: Runtime-accuracy of kmeans anytime automaton. The vertical line indicates an SNR of ∞ dB.

of a single iterative stage that employs loop perforation when processing and transposing pixels.

Debayering (**debayer**) from PERFECT converts a Bayer filter image from a single sensor to a full RGB image. It is commonly used in image sensors for security cameras and x-ray imaging. The structure of the application is similar to 2dconv; the interpolations in debayer are similar to the convolutional filter. As a result, we use a similar single-diffusive-stage automaton with tree-based output sampling.

K-means clustering (**kmeans**) from AxBench performs the k-means algorithm for clustering over the pixels of an image. This is a very common computation in data mining and machine learning. We construct an automaton with two stages in an asynchronous pipeline. The first stage computes the cluster centroids and assigns pixels to clusters based on their Euclidean distances. This is diffusive; we employ anytime output sampling with a tree permutation. The second (non-anytime) stage reduces the centroid computations of the multiple threads from the previous stage.

B. Performance-Accuracy Tradeoff

In this section, we evaluate the performance-accuracy tradeoffs of our anytime automata. The runtime-accuracy results are presented in Figures 11 (2dconv), 12 (histeq), 13 (dwt53), 14 (debayer), and 15 (kmeans). 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 ∞ dB (precise output). This is shown for all applications 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.

As shown in Figures 11 and 14, 2dconv and debayer reap the most benefit 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. This output is visualized in Figure 16, comparing against the baseline precise output. The benchmarks 2dconv and debayer are able to achieve high accuracy at low runtimes because 1) their computations are diffusive, and 2) their pipelines are simple. For both 2dconv and debayer, we employ output sampling, a diffu-

(a) 21% runtime, SNR 15.8dB

(b) baseline precise

Figure 16: Output of 2dconv anytime automaton.

(a) 78% runtime, SNR 16.8dB

(b) baseline precise

Figure 17: Output of dwt53 anytime automaton.

sive anytime approximation that minimizes redundant work. In Sections IV-B1 and IV-B2, we also evaluate reduced-precision operations and approximate storage on 2dconv. The pipelines for 2dconv and debayer are simple since they only consist of one stage. They are not hindered by the presence of non-anytime stages, unlike in histeq and kmeans. Despite the good results, neither 2dconv nor debayer (nor any of the other applications) reach precise output as early as the baseline execution. This is primarily due to poor cache locality from non-sequential sampling permutations. As we discuss later in Section IV-C3, this can be alleviated by architectural optimizations.

As shown in Figure 13, dwt53 has a steep runtime-accuracy curve. The automaton produces unacceptable approximations for over half of the baseline runtime before finally delivering acceptable output. This is due to the iterative loop perforation. Unlike with diffusive sampling where the output constantly increases in accuracy as elements are processed, iterative loop perforation re-executes the computation with progressively larger strides. This results in redundant computations and yields a runtime-accuracy curve that is less smooth. Despite this, with the dwt53 automaton, Figure 17 shows that acceptable output (SNR 16.8 dB) can be reached at only 78% of the baseline runtime.

As shown in Figures 12 and 15, histeq and kmeans do 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 normalization of data structures (as in histeq) or reducing thread-privatized data (as in kmeans). Despite this, both applications produce acceptable outputs at about 60% of the baseline runtime, visualized in Figure 18 for kmeans. 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 the wider design space exploration of anytime automata and new anytime approximation techniques.

(a) 63% runtime, SNR 16.7dB (b) baseline precise

Figure 18: Output of kmeans anytime automaton.

1) Impact of Reduced Fixed-Point Precision: In this section, we evaluate the accuracy of applying reduced-precision operations (integer, in this case) to the 2dconv automaton. Figure 19 shows the SNR using 8-bit (default), 6-bit, 4-bit and 2-bit pixel precisions. The x-axis is the increasing sampling resolution, since output sampling is employed in the 2dconv automaton. After processing all elements (i.e., sample size of 1), for 6-bit and 4-bit precision, output accuracy is 37.9 dB and 24.2 dB respectively. Reduced-precision can be applied in conjunction with sampling while still maintaining reasonable accuracy. Furthermore, reduced-precision operations for integers are diffusive, minimizing redundant computations, as discussed in Section III-B2.

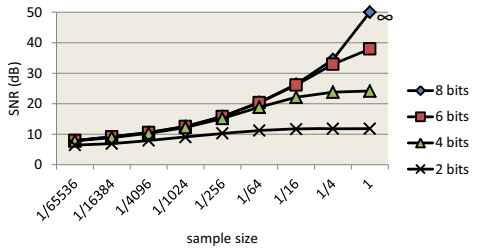

Figure 19: Sample size-accuracy of 2dconv anytime automaton when varying pixel precision.

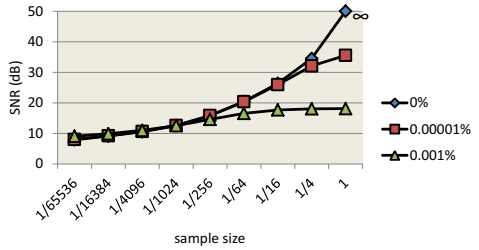

Figure 20: Sample size-accuracy of 2dconv anytime automaton when varying SRAM read upset probability.

2) Impact of Approximate Storage: In this section, we evaluate the use of iterative anytime techniques via approximate storage (low-voltage SRAM [7], in this case). Figure 20 shows how 2dconv accuracy is impacted with varying SRAM bit failure probabilities. We explore read upset probabilities of 0.00001% and 0.001%, the latter of which is estimated to yield up to 90% supply power savings [19]. As shown in the figure, allowing for such read upsets still yields acceptable outputs in many cases. Note that the curves line up at lower sample sizes; this is expected since the number of bit flips is directly related to number of data elements processed so far.

C. Discussion

In this section, we discuss new insights, challenges and research opportunities that emerge when designing architectures for our anytime automaton model.

1) Multi-Threaded Sampling: Our model supports computation stages that are multi-threaded. Though we use non-sequential permutations when sampling, sampling can still be performed by multiple threads. For both the tree and pseudo-random permutations, the permutation function $p(i)$ is bijective and deterministic. Given the base index i which is incremented $1,...,n$, the permutation $p(i)$ simply yields a different (but still deterministic) sequence for accessing elements. It is then straightforward to divide this permutation sequence among threads for sampling. For the tree permutation, we typically want to produce a low resolution output as early as possible. Thus the permutation sequence of p can be divided cyclically; given n threads, a thread that is currently processing the element at $p(i)$ will next access the element at $p(i+n)$. For the pseudo-random permutation, either cyclic or round-robin distribution is acceptable.

2) Pipeline Scheduling: The anytime automaton opens up new interesting challenges in thread scheduling. Given an architecture with limited cores and hardware threads, it can be difficult to decide how many threads to allocate per computation stage. The conventional approach for pipelining is to assign threads to stages such that all stages have similar latencies. However, this may not be suitable for the automaton pipeline.

First, the latency of a computation stage may not be static. An anytime stage f is broken down into intermediate computations $f_1,...,f_n$, whose latencies can vary significantly. In many cases, the latencies increase from f_1 to f_n since the later stages likely perform more computations to achieve higher accuracy. Thus it may be beneficial to reassign threads among stages dynamically. However, this can be difficult since stages are not necessarily synchronized. For example, at one point in time, f_n can be co-executing with g_1, while at another, it can be executing alongside g_2.

Second, thread assignment depends on the desired granularity of anytime outputs. The granularity of outputs is defined by 1) how early the first approximate output is available, and 2) how frequently the approximate outputs are updated as they approach the precise output. Consider the example pipeline in Figure 2. If we want to minimize the amount of time it takes to reach the first approximate output O_{1111}, we need to allocate more threads to the longest stage f. On the other hand, if we want to minimize the amount of time between consecutive outputs O_{1111} and O_{1112}, we need to allocate more threads to the final stage i. Though challenging, pipeline scheduling is merely an optimization problem; correctness is ensured regardless. This motivates the design of architectures with fine-grained, intelligent thread migration/scheduling; this is left for future work.

3) Data Locality: In conventional architectures, the anytime automaton can suffer from poor cache and row buffer locality when sampling with the non-sequential tree and pseudo-random permutations. However, both permutations are deterministic. As a result, simple hardware prefetchers can be implemented to alleviate the high miss rates due to poor locality. The overhead and complexity of such prefetchers is minimal: an address computation unit coupled with the deterministic tree or pseudo-random (e.g., LFSR) counters. Furthermore, thanks to recent advancements in near-data processing [1], input and output data sets can be reordered in-memory, since the sampling permutations are typically static throughout the runtime of the application.

V. Conclusion

We propose the Anytime Automaton, a new computation model that represents an approximate application as a parallel pipeline of anytime computation stages. 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 drawback of state-of-the-art approximate computing techniques: they do not provide guarantees on the acceptability of *all* outputs at runtime. 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. The anytime automaton greatly simplifies (for users and system designers) the process of executing applications in an approximate way. This can catalyze the acceptance of approximate computing in real-world systems and invigorate the design of architectures where output acceptability directly governs the amount of time and energy expended (*hold-the-power-button* computing).

Acknowledgements

The authors thank their internship collaborators at IBM T. J. Watson Research Center [23]: Vijayalakshmi Srinivasan, Ravi Nair and Daniel A. Prener. The authors also thank the anonymous reviewers for their insightful feedback. This work is supported by a Queen Elizabeth II/Montrose Werry Scholarship in Science and Technology, the Natural Sciences and Engineering Research Council of Canada, the Canadian Foundation for Innovation, the Ministry of Research and Innovation Early Researcher Award and the University of Toronto.

References

[1] B. Akin *et al.*, "Data Reorganization in Memory Using 3D-stacked DRAM," in *ISCA*, 2015.

[2] C. Alvarez *et al.*, "Fuzzy memoization for floating-point multimedia applications," *IEEE TOCS*, 2005.

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

[4] K. Barker *et al.*, *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/.

[5] T. L. Dean and M. Boddy, "An analysis of time-dependent planning," in *AAAI*, 1988.

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

[7] K. Flautner *et al.*, "Drowsy caches: simple techniques for reducing leakage power," in *ISCA*, 2002.

[8] A. Garvey and V. Lesser, "Design-to-time real-time scheduling," *IEEE SMC*, 1993.

[9] B. Grigorian *et al.*, "BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing," in *HPCA*, 2015.

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

[11] D. S. Khudia *et al.*, "Rumba: An Online Quality Management System for Approximate Computing," in *ISCA*, 2015.

[12] V. Lesser *et al.*, "Approximate processing in real-time problem-solving," *AI Magazine*, 1988.

[13] S. Liu *et al.*, "Flikker: saving DRAM refresh-power through critical data partitioning," in *ASPLOS*, 2011.

[14] R. Mangharam and A. A. Saba, "Anytime Algorithms for GPU Architectures," in *RTSS*, 2011.

[15] T. Moreau *et al.*, "SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration," in *HPCA*, 2015.

[16] S. Narayanan *et al.*, "Scalable stochastic processors," in *DATE*, 2010.

[17] L. Renganarayana *et al.*, "Programming with relaxed synchronization," in *RACES*, 2012.

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

[19] A. Sampson *et al.*, "EnerJ: approximate data types for safe and general low-power consumption," in *PLDI*, 2011.

[20] A. Sampson *et al.*, "Approximate storage in solid-state memories," in *Proc. Int. Symp. Microarchitecture*, 2013.

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

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

[23] J. San Miguel *et al.*, "A systolic approach to deriving anytime algorithms for approximate computing," IBM Research Report RC25600, Tech. Rep., 2016.

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

[25] A. Sinha and A. Chandrakasan, "Energy efficient filtering using adaptive precision and variable voltage," in *ASIC/SOC*, 1999.

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

[27] M. Sutherland *et al.*, "Texture cache approximation on gpus," in *WAX*, 2015.

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

[29] J. W. S. L. W-K. Shih and J.-Y. Chung, "Fast algorithms for scheduling imprecise computations," in *RTSS*, 1989.

[30] S. White, "Applications of distributed arithmetic to digital signal processing: a tutorial review," *IEEE ASSP*, 1989.

[31] A. Yazdanbakhsh *et al.*, "RFVP: Rollback-free value prediction with safe-to-approximate loads," *TACO*, 2016.

[32] T. Yeh *et al.*, "The art of deception: Adaptive precision reduction for area efficient physics acceleration," in *MICRO*, Dec 2007.

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