Rumba: An Online Quality Management System for Approximate Computing

Daya S Khudia Babak Zamirai Mehrzad Samadi Scott Mahlke
University of Michigan
{dskhudia,zamirai,mehrzads,mahlke}@umich.edu

Abstract

Approximate computing can be employed for an emerging class of applications from various domains such as multimedia, machine learning and computer vision. The approximated output of such applications, even though not 100% numerically correct, is often either useful or the difference is unnoticeable to the end user. This opens up a new design dimension to trade off application performance and energy consumption with output correctness. However, a largely unaddressed challenge is quality control: how to ensure the user experience meets a prescribed level of quality. Current approaches either do not monitor output quality or use sampling approaches to check a small subset of the output assuming that it is representative. While these approaches have been shown to produce average errors that are acceptable, they often miss large errors without any means to take corrective actions. To overcome this challenge, we propose Rumba *for online detection and correction of large approximation errors in an approximate accelerator-based computing environment. Rumba employs continuous lightweight checks in the accelerator to detect large approximation errors and then fixes these errors by exact re-computation on the host processor. Rumba employs computationally inexpensive output error prediction models for efficient detection. Computing patterns amenable for approximation (e.g., map and stencil) are usually data parallel in nature and Rumba exploits this property for selective correction. Overall, Rumba is able to achieve 2.1x reduction in output error for an unchecked approximation accelerator while maintaining the accelerator performance gains at the cost of reducing the energy savings from 3.2x to 2.2x for a set of applications from different approximate computing domains.*

1. Introduction

Computation accuracy can be traded off to achieve better performance and/or energy efficiency. The techniques to achieve this trade off fall under the umbrella of approximate computing. Algorithm specific approximation has been used in many different domains such as machine learning, image processing, and video processing. Different algorithms in these domains have been approximated by programmers to achieve better performance. Video processing algorithms are good candidates for approximation as occasional variation in results will not be noticeable by the user. For example, a consumer can tolerate occasional dropped frames or a small loss in resolution during video playback, especially when this allows video playback to occur seamlessly. Machine learning and data analysis applications also provide opportunities to exploit approximation to improve performance, particularly when such programs are operating on massive data sets. In this situation, processing the entire dataset may be infeasible, but by sampling the input data, programs in these domains can produce representative results in a reasonable amount of time.

However, algorithm specific approximation increases the programming effort because the programmer needs to write and reason about the approximate version in addition to the exact version. Recently, to solve this issue, different software and hardware approximation techniques have been proposed. Software techniques include loop perforation [1], approximate memoization [11, 31], tile approximation [31], discarding high overhead computations [32, 36], and relaxed synchronization [28]. Furthermore, there are many hardware based approximation techniques that employ neural processing modules [16, 4], analog circuits [4], low power ALUs and storage [34], dual voltage processors [15], hardware-based fuzzy memoization [2, 3] and approximate memory modules [35]. Approximation accelerators [16, 41, 14] utilize these techniques to trade off accuracy for better performance and/or higher energy savings. In order to efficiently utilize these accelerators, a programmer needs to annotate code sections that are amenable to approximation. At runtime, the CPU executes the exact code sections and the accelerator executes the approximate parts.

These techniques provide significant performance/energy gains but monitoring and managing the output quality of these hardware accelerators is still a big challenge. A few of the recently proposed quality management solutions include quality sampling techniques that compute the output quality once in every N invocations [32, 6], techniques that build an offline quality model based on the profiling data [6, 16].

However, these techniques have four critical limitations:

- As the output quality is dependent on the input values, dif-

ISCA'15, June 13-17, 2015, Portland, OR, USA

DOI: http://dx.doi.org/10.1145/2749469.2750371

ferent invocations of a program may produce results of different output qualities. Therefore, sampling techniques are not capable of capturing all changes of the output quality. Moreover, it is highly possible to miss large output errors because only a subset of outputs are actually examined, i.e., monitoring is not continuous. Also, profiling techniques do not work efficiently if the profiling data is not representative of all possible inputs.

- Using these quality management techniques, if the output quality drops below an acceptable threshold, there is no way to improve the quality other than re-executing the whole program on the exact hardware. However, this recovery process has high overhead and it offsets the gains of approximation.
- These techniques measure the quality of the whole output that is usually equal to the average quality of each individual output element, e.g., pixels in an image. Previous works [16, 4] in approximate computing show that most of the output elements have small errors and there exist a few output elements that have considerably large errors, even though the average error is low. These large errors can degrade the whole user experience. For example, having a few pixels with high error in an image can be easily noticed by a user. Existing quality management techniques treat all errors equally but large errors have noticeable effect on the perceivable output quality.
- Tuning output quality based on a user's preferences is another challenge for the hardware-based approximation techniques. Different users and different programs might have different output quality requirements. However, it is difficult to change the output quality of an approximate hardware dynamically.

To address these issues, we propose a framework called Rumba[1], an online quality management system for approximate computing. Rumba's goal is to dynamically investigate an application's output to detect elements that have large errors and fix these elements with a low-overhead recovery technique. Rumba performs continuous light-weight output monitoring to ensure more consistent output quality. Rumba's design is based on the following two observations:

First, approximation error can be accurately predicted by simple prediction models such as *linear*, *decision tree*, and *moving average*. Second, we observe that code regions or functions that are amenable for approximation are often *pure*. Pure code regions just read their inputs and only write to their outputs without modifying any other state. Such sections can be safely re-executed without any side effects. It gives us the benefit of re-executing the loop iterations to fix the erroneous output elements with low overhead.

Rumba has two main components: detection and recovery. The goal of the detection module is to efficiently predict output elements that have large approximation errors. Detection is achieved by supplementing the approximate accelerator with a low-overhead error prediction hardware. The detection module dynamically investigates predicted error to find elements that needs to be corrected. It gathers this information and sends it to the recovery module on the CPU. In order to improve the output quality, recovery module re-executes the iterations that generate high error output elements.

To reduce Rumba's overhead, recovery is done on the CPU in parallel to detection on the approximate accelerator. The recovery module controls the *tuning threshold* to manage output quality, energy efficiency and performance gain. The tuning threshold determines the number of iterations that need to be re-executed.

The major contributions of this work are as follows:

- We explore three light-weight error prediction methods to predict the errors generated by an approximate computing system.
- The ability to manage performance and accuracy trade offs for each application at runtime using a dynamic tuning parameter.
- We leverage the idea of re-execution to fix elements with large errors.
- 2.1x reduction in output error with respect to an unchecked approximate accelerator with the same performance gain. Detection and re-execution decrease the energy savings of the unchecked approximate accelerator from 3.2x to 2.2x.

2. Challenges and Opportunities

The ability of applications to produce results of acceptable output quality in an approximate computing environment is necessary to ensure a positive user experience. Output quality control for approximate programs is important for the wide adaptation of this technology.

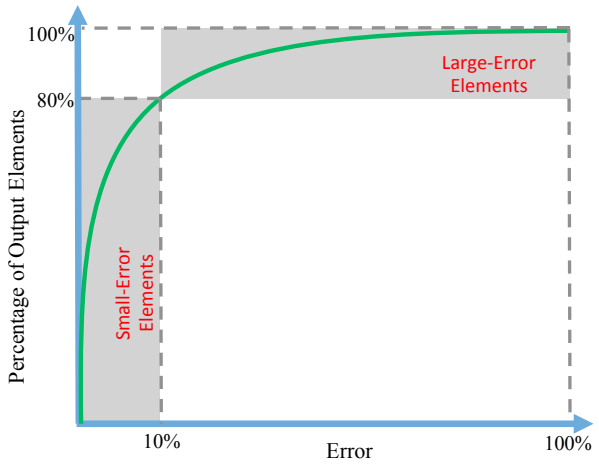

Figure 1: Typical cumulative distribution function of errors generated by approximation techniques. A large number of output elements have small errors while a few output elements have large errors.

[1]The name Rumba is inspired from Roomba®, an autonomous robotic vacuum cleaner. It moves around the floor and detects dirty spots on the floor to clean them.

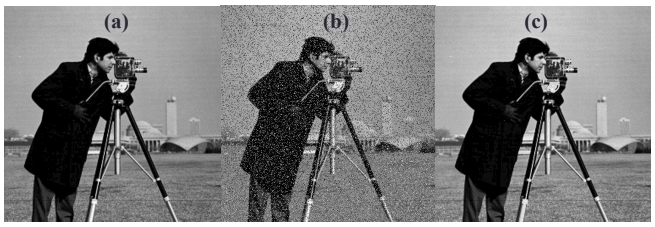

Figure 2: An example of variation in image quality with the changing distribution of errors. Subfigure (a) is the original image without any errors. Ten percent of the pixels in (b) have 100% error while the rest of the pixels are intact. All pixels in (c) have 10% error. Although these two images have the same average quantitative output quality (90%), errors in Subfigure (b) are more noticeable.

2.1. Challenges of Managing Output Quality

The following are the main challenges of output quality management in an approximate computing environment.

Challenge I: Fixing output elements with large errors is critical for user experience. We analyze the distribution of errors in the output elements generated by an application under approximation. Previous studies [16, 31, 32] reported that the Cumulative Distribution Function (CDF) of the errors of an approximated application's output follows the curve shown in Figure 1. Figure 1 shows a typical CDF of errors in output elements when total average error is less than 10%. This figure shows that the most of the output elements (about 80%) have small errors (lower than 10%). However, there are few output elements (about 20%) that have significant errors.

Although the number of elements with large errors is relatively small, they can have huge impact on the user perception of output quality. Figure 2 demonstrates this. In this figure, we generate two images by adding errors such that the overall average error is 10% in both images. Figure 2(a) is the original image. In Figure 2(b), only 10% of pixels have 100% errors while the rest of pixels are exact. On the other hand, all pixels in Figure 2(c) have about 10% error. Even though the overall output error is the same for both the generated images, errors in Figure 2(b) are more noticeable than Figure 2(c) to the end user. This shows that to effectively improve the output quality, a quality management system should reduce the long tail of high errors.

Challenge II: Output quality is input-dependent. Another characteristic of approximate techniques is that output quality is highly dependent on the input [16, 31, 32, 6]. In this case, these techniques must consider the worst case to make sure that the output quality is acceptable. To show this, we run an image processing application called *mosaic* that generates a large image using many small images. The first phase of this application computes the average brightness of all input images. To approximate this phase, a well-known approximation technique called loop perforation [1] is used. Loop perforation drops iterations of the loop randomly or uniformly. Therefore, in this case, instead of computing the average brightness of all the pixels, the approximate version computes the average brightness of a subset of the pixels.

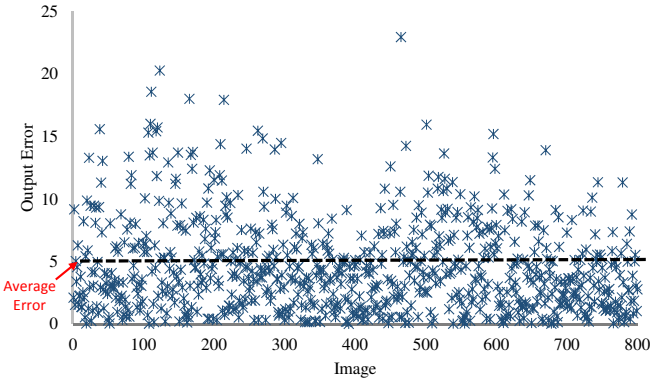

Figure 3: Mosaic application's output error for 800 different images of flowers. This data shows that the output quality is highly input-dependent.

Figure 3 shows the output error for 800 different images of flowers [23]. Average error of all the images is about 5% but there are many images that have output error above the average, up to a maximum of 23%. Therefore, an approximate system in the worst case (23% error) may produce unacceptable quality results. However, if a quality management system can reduce the unacceptable outputs, the aggressiveness of approximate techniques can be increased to get better performance and/or energy savings.

Also, since the output quality is highly input-dependent, previous quality managing systems such as quality sampling or profiling techniques might miss invocations that have low quality. In order to solve this problem, a dynamic light-weight quality management system is required to check the output quality for all invocations.

Challenge III: Monitoring and recovering output quality is expensive. One of the challenges that all approximate techniques have is monitoring the output quality. In order to solve this problem, continuous checks are necessary. Such checks cannot compute the exact output, but instead need to be predictive in nature. Different frameworks [32, 6] suggest running an application twice (exact and approximate versions) and comparing the results to compute output quality. Unfortunately, it has high overhead and it is not feasible to monitor all invocations. Running exact and approximate at all times will nullify the advantages of using approximation.

To reduce this overhead, these frameworks utilize quality sampling techniques that check the quality once in every N invocations of the program. Therefore, if the invocations that are not checked have low output quality, these frameworks will miss them due to the input dependence of output quality (Challenge II).

Challenge IV: Different users and applications have different requirements on output quality. In an approximation system, the user should be able to tune the output quality based on her preferences or program's characteristics. Software-

based approximation techniques are better at tuning the output quality. However, for hardware-based techniques, it is a huge challenge. For example, in a system with two versions of functional units, exact and approximate, it is hard to control the final output quality dynamically.

2.2. Rumba's Design Principles

To overcome the four challenges, Rumba exploits two observations found in the kernels that are amenable to approximation: predictiveness of errors and recovery by selective re-execution.

Predictiveness of Errors: Rumba's detection module is based on the observation that it is possible to accurately predict the errors of an approximate accelerator using a computationally inexpensive prediction model. Figure 5 shows exact output (a Gaussian distribution), approximate output produced by an accelerator and errors in approximation. For this case, it is visually clear that errors are concentrated on certain inputs. Hence, a simple prediction model can separate cases of high errors accurately.

Rumba dynamically employs light-weight checkers to detect approximation errors. A threshold on the predicted errors is used to classify errors in the output elements as high. Therefore, Rumba targets output elements with high errors as mentioned in Challenge I. Also, since Rumba has light-weight checkers, the checks can be performed online for all the elements of each invocation (Challenge III).

Recovery by Selective Re-execution: In computing, a *pure* function or code region only reads its inputs and only affects its outputs, i.e., it does not affect any other state. In other words, pure functions or code regions can be freely re-executed without any side-effects. Similar characteristics have been previously used in recovering program from external errors simply by re-executing [17, 20]. Such functions or code regions naturally occur in many data-parallel computing patterns such as map and stencil. We analyzed the data parallel parts of the applications in Rodinia benchmark suite [12] and found out that more than 70% of them can be re-executed without any side effects. Rumba detects this characteristic using these previous techniques to identify such regions in applications. A more detailed description of recovery is given in Section 3.1. It is not a new restriction imposed by Rumba as previously proposed approximate accelerators [16, 4] require functions or code regions to be pure to be able to map them to an approximate accelerator.

Therefore, if Rumba detects that one of the accelerator invocations generates output elements with large error, the Rumba recovery module can simply re-execute that iteration to generate the exact output elements. In this case, there is no need to re-run the whole program to recover those output elements (Challenge III). Also, using this technique, Rumba can manage the performance/energy gains by changing the number of iterations to be re-executed to target Challenge IV.

3. Design of Rumba

3.1. Overview

Approximation errors can be broadly divided into *large errors* and *small errors*. Approximation accelerators generate a large number of small errors and relatively few large errors as shown in Figure 3. Rumba is a detection and recovery scheme for errors in an approximate computing system. Rumba is specifically designed to detect these large errors by a light-weight checker and then fix these errors. Rumba makes the output of an approximation accelerator computing system acceptable by reducing the long tail of large errors. Alternatively, with Rumba's error correction capabilities, it will be possible to dial up the amount of approximation, thus improving performance and/or energy savings, while still producing user acceptable outputs.

A high-level block diagram of the Rumba system is shown in Figure 4. The offline part of Rumba system consist of two trainers. The first trainer finds the optimal configuration of the approximate accelerator for a particular source code. The second trainer trains a simple error prediction technique based on the errors produced by the accelerator trainer. The configuration parameters for both the approximate accelerator and the error predictor are embedded in the binary.

The execution subsystem of Rumba is shown in the same figure. For the purpose of exposition, we assume that the design of our approximation accelerator is similar to the one proposed by Esmaeilzadeh et al. [16]. However, the same design principles can apply to other accelerator based approximate computing systems. As shown in the figure, the core communicates to the accelerator using I/O queues for data transfers from the core to the accelerator and back from accelerator to the core. Rumba's execution has two components: detection and recovery modules.

The annotated approximate part of the application code gets mapped to the approximation accelerator [16, 4]. We augment the approximate accelerator by an error predictor module to detect approximation errors. A variety of prediction techniques can be used to predict these errors. We explore three light-weight checkers that are implemented using three simple error prediction techniques. These error predictors are described in Section 3.2. Once a check fires, i.e., approximation for that particular output element is larger than a tuning threshold (determined by the online tuner based on user requirements), a recovery bit for the iteration generating that particular element is set in the recovery queue as shown in Figure 4. The CPU collects these bits from the recovery queue and re-executes the iterations that their recovery bit is set. Output merger choses the exact or the approximate output as final result. A more detailed description is in Section 3.3. Another important aspect of Rumba is the dynamic management of output quality and energy efficiency. By controlling the threshold at which the checker fires, Rumba can control the number of iterations to be re-executed. This tuning process is discussed in Section 3.4.

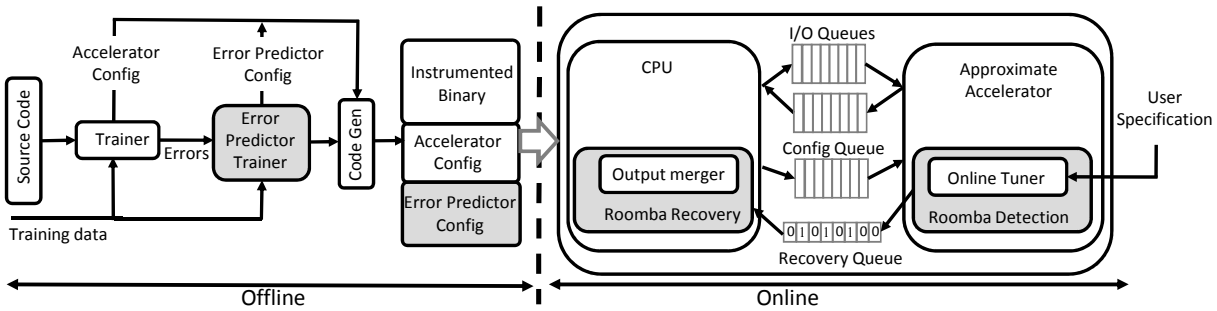

Figure 4: A high-level block diagram of the Rumba system. The offline components determine the suitability of an application for the Rumba acceleration environment. The online components include detection and recovery modules. The approximation accelerator communicates a recovery bit corresponding to the ID of the elements to recompute with the CPU via a recovery queue.

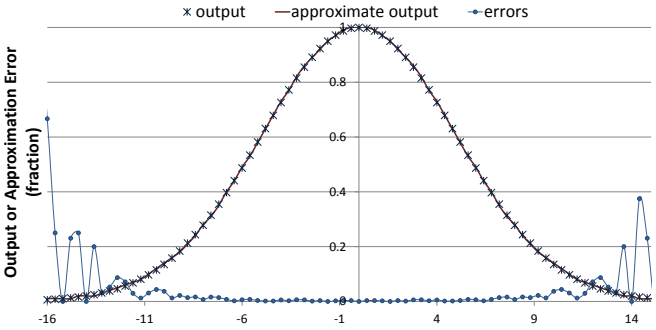

Figure 5: Exact output, approximate output and relative errors in the approximate output. The relative errors in the approximate output are higher for some inputs than the others and are more easily predictable than the output itself.

3.2. Light-weight Error Prediction

An important first step is the inexpensive detection of large approximation errors in output elements. Since it is not known beforehand which output elements will have large errors, runtime checks should be employed for all the output elements. Therefore, the light-weight nature of these checkers is of paramount importance. Complex checkers to detect large approximation will offset the gains of approximation and, thus, are not desirable. A desirable dynamic checker should have low overhead and still be accurate at predicting errors in output elements.

A dynamic checker does not have access to the exact results, hence, the errors in approximate output cannot be computed by comparing with the exact result. Computing exact values is not an option because that negates the benefits of employing an approximation system. The Rumba detection module needs to detect large approximation errors by using inputs to the accelerator or the approximate output produced by the accelerator. We call a method an input-based method if the method calculates errors using the inputs to the accelerator. Similarly, if the errors are detected by just observing the accelerator output, such a method is called an output-based method. For input-based methods, approximation errors can be obtained using a simple predicting model on inputs in the following two ways:

- Errors by Value Prediction (EVP): predict the output using a model and then get the error by comparing it with the approximate accelerator's output.
- Errors by Error Prediction (EEP): predict the errors directly using a model.

In our experiments, we observed that if we use the same Prediction model it is more accurate to predict the errors directly than computing the errors by first predicting the output. We analyzed errors in the approximation of a Gaussian distribution and found that average distance between exact approximation errors and errors obtained by EVP and EEP is 2.5 and 1, respectively, i.e., EEP is more accurate. Therefore, we use simple prediction models to predict errors in the approximation. We explore two input-based methods and one output-based method to detect errors.

3.2.1. Error prediction using a linear model: The first error prediction method is a linear error predictor and is an input-based method. A linear error prediction method predicts error by computing a linear function of inputs to the accelerator. Equation 1 shows the linear function that is calculated to compute the error. The number of terms (x_is) are determined by the number of inputs to the code section that is mapped to the approximate accelerator. A linear model requires relatively simple computations in the form of multiply-add operations. Hence, the online prediction of errors for a particular input does not add much energy overhead. The weights (w_is) and constant c are determined by offline training.

$$err = w_0 * x_0 + w_1 * x_1 \ldots w_{N-1} * x_{N-1} + c \qquad (1)$$

where x_i is the ith input, w_i is the weight for the ith input and c is a constant.

3.2.2. Error prediction using a decision tree: The second error prediction method is a decision tree and is also an example of input-based methods. An example of error prediction using a decision tree is shown in Figure 6. This model contains decision and leaf nodes. The decision nodes typically have two branches and uses one of the inputs to decide on

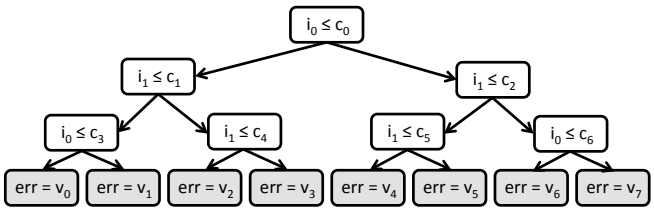

Figure 6: A decision tree with a depth of 3 in decision nodes. For this example, it predicts errors based on two inputs. The leaf nodes (gray) give the approximation errors. The coefficients (c_is and v_is) are determined by offline training.

whether to traverse the left or right child. This process continues until it reaches a leaf node in the tree. Leaf nodes store the predicted error. Training data is used to determine the values of constants used in making decisions at the decision nodes and predicted error at the leaf nodes. The computation required in a decision tree is dependent on the depth of the tree structure. We limit the tree depth to 7 in our experiments. Only comparison operations are required to implement this decision tree and hence it is not a computationally expensive error prediction method.

3.2.3. Error prediction using moving average: The third error prediction model is using moving average as the general trend of data in the sequence. This moving average based method is an output-based method because it just observes the accelerator outputs to find out the erroneous elements. The difference between current element and the moving average can be used to detect large errors in a number in the sequence. In this work, we used Exponential Moving Average (EMA) which can be calculated by the formula shown in Equation 2.

$$EMA = (e * \alpha) + (Previous\ EMA * (1 - \alpha)) \qquad (2)$$

where e = Current element, α = Smoothing factor = $\frac{2}{1+N}$ and N = Number of elements in the history

EMA computes the exponential moving average over a window of output elements and compare it to each output element to compute the difference. If the difference is higher than a tuning threshold, the detection module marks the output element as erroneous.

Once an application is deemed fit for approximation on the accelerator, it is transferred to the accelerator augmented with an error predictor. The dynamic check for each output element is the predicted error greater than a tuning threshold. If this predicted error is greater than a tuning threshold, a large approximation error is suspected and the check fires.

Predictor Hardware: Figures 7(a) and 7(b) show the hardware for the linear error and decision tree error predictors. An approximate accelerator is augmented with these hardware to predict errors. Coefficient buffers are circular buffers and contain weights and constants for the linear model and decision constants and errors for the decision tree model. The coefficients are transferred to these checkers via a *config* queue

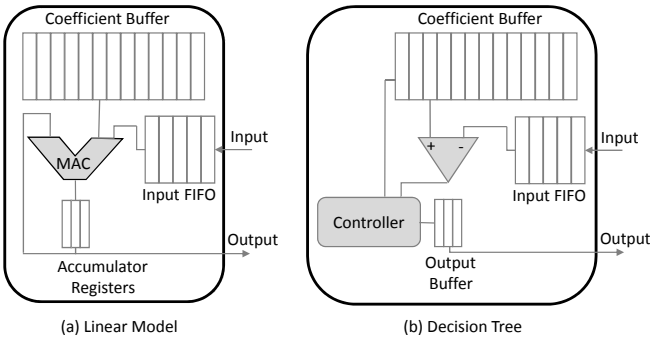

Figure 7: Hardware for the approximation error predictors.

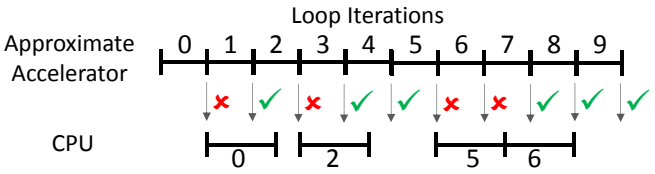

Figure 8: An example of overlapping the re-computation of elements by the CPU with the approximation accelerator. For example, a large error is detected in iteration 0 by the accelerator and the CPU recomputes this iteration while accelerator is working on the execution of iteration 1 and 2.

(the same queue is used to transfer accelerator configuration) between the CPU and the accelerator.

EMA detects large approximation errors by comparing the current approximate outputs with the history of previously computed approximate outputs. The history is represented by EMA, the detection module keeps the EMA and calculates the approximate error in the current approximate output by comparing it with EMA.

3.3. Low-overhead Recovery

Rumba's recovery module on the CPU gets an iteration's recovery bit via the recovery queue. If the corresponding bit of an iteration is set, the recovery module re-executes that iteration and commits the re-computed output while discarding the accelerator output for that input. The results received by the CPU from the approximation accelerator are directly committed to their final destination if the corresponding recovery bit is not set in the recovery queue. This is how Rumba merges approximate outputs from the accelerator with the exact outputs obtained by re-execution on the CPU.

The CPU and the accelerator work in a pipelined fashion, i.e., while accelerator is working on an iteration, the CPU recomputes a previous iteration. An example of such an arrangement is shown in Figure 8. For this example, the checks fire for output elements of iterations 0, 2, 5 and 6. The CPU re-computes iteration 0 while the accelerator is working on iteration 1 and 2. Similarly, re-computation of iteration 2 is overlapped with the execution of 3 and 4 on the accelerator and so on. In such a setup, the CPU can recompute 50% of the output elements, assuming a 2x gain for the accelerator,

Application	Domain	Train Data	Test Data	NN Topology (Rumba)	NN Topology (NPU)	Evaluation Metric
blackscholes	Financial Analysis	5K inputs	5K outputs	3->8->8->1	6->8->8->1	Mean Relative Error
fft	Signal Processing	5K random fp numbers	5K random fp numbers	1->1->2	1->4->4->2	Mean Relative Error
inversek2j	Robotics	10K random (x, y) points	10K random (x, y) points	2->2->2	2->8->2	Mean Relative Error
jmeint	3D Gaming	10K pairs of 3D triangles	10K pairs of 3D triangles	18->32->2->2	18->32->8->2	# of mismatches
jpeg	Compression	220x200 pixel image	512x512 pixel image	64->16->64	64->16->64	Mean Pixel Diff
kmeans	Machine Learning	220x200 pixel image	512x512 pixel image	6->4->4->1	6->8->4->1	Mean Output Diff
sobel	Image Processing	512x512 pixel image	512x512 pixel image	9->8->1	9->8->1	Mean Pixel Diff

Table 1: Applications and their inputs.

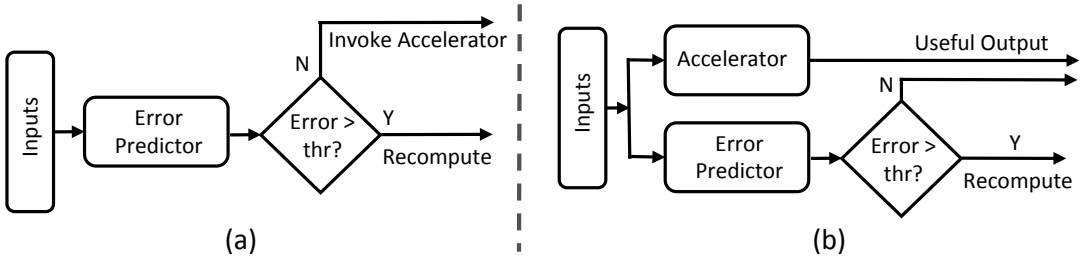

Figure 9: Shows the design choices for the relative placement of input-based detectors with respect to the accelerator. Configuration in part (a) adds delay, thus impacting overall performance, in the path to invoking accelerator. Configuration in part (b) wastes energy on invocations of the accelerator that have large error.

and still keep up with the accelerator provided the elements to recompute are uniformly distributed.

3.4. Online Tuning

The tuning threshold of Rumba is used as a threshold for the dynamic checks to determine if the current output has large error. A larger threshold value will result in fewer iterations to be re-executed. This, in turn, will cause higher energy savings but lower output quality. Rumba's tuning threshold can be determined by user specified requirements either on energy consumption or output quality. Online tuning can be programmed in three modes:

TOQ Mode: In this mode, user specifies the target output quality (TOQ). The goal of this mode is to make sure that all output elements have better quality than TOQ. Therefore, Rumba compares the predicted quality with TOQ and re-execute iterations that have lower quality than TOQ.

Energy Mode: If a user specifies an energy target to achieve, Rumba calculates the number of iterations (*iteration budget*) it can re-execute while staying in the energy budget. For each invocation, it monitors the number of re-executed iterations. If it goes over the iteration budget it stops re-executing and increases the tuning threshold for the next invocation. If the current invocation is finished and Rumba still stays within the iteration budget, the tuning threshold is decreased. This would result in more iterations to be re-executed for the next invocation and thus improves output quality while staying in the same energy budget.

Quality Mode: If a user is more concerned about achieving the best output quality, Rumba maximizes re-execution of iterations on the CPU until the current invocation of accelerator finishes. The accelerator performance gain in comparison to the CPU determines how many elements the CPU can recompute and still keep up with the accelerator. If the CPU is not fully utilized during recovery, it implies that it can fix more iterations so the tuning threshold is increased for the next invocation. If accelerator finishes the current invocation and the CPU still has iterations to re-execute, the tuning threshold for the next invocation is increased. This results in lesser number of re-executions for the next invocation.

3.5. Error Detector Placement

An important design choice for input-based methods is the relative invocation of the error predictor with respect to the accelerator. An input-based detector can be placed in one of the ways shown in Figure 9. Figure 9(a) (Configuration 1) shows the error detector placement before sending the inputs to the accelerator and Figure 9(b) (Configuration 2) shows the error detector placement if the error detector and accelerator simultaneously start working on inputs. These configurations provide different trade-offs in the design space. Configuration 1 saves the unnecessary accelerator invocations, hence saves energy, for the cases when error detector detects an error. However, since error prediction precedes the accelerator invocation, it delays accelerator computation, hence, has performance overhead. Energy is wasted in Configuration 2 for accelerator invocations that have errors greater than the threshold. However, error detector in this configuration does not add any delay in the invocation of the accelerator, hence, does not add to performance overhead. In our experiments, to minimize the impact on performance overhead, we use Configuration 2. Error detector placement for output-based methods is straight forward and should be invoked after accelerator invocation.

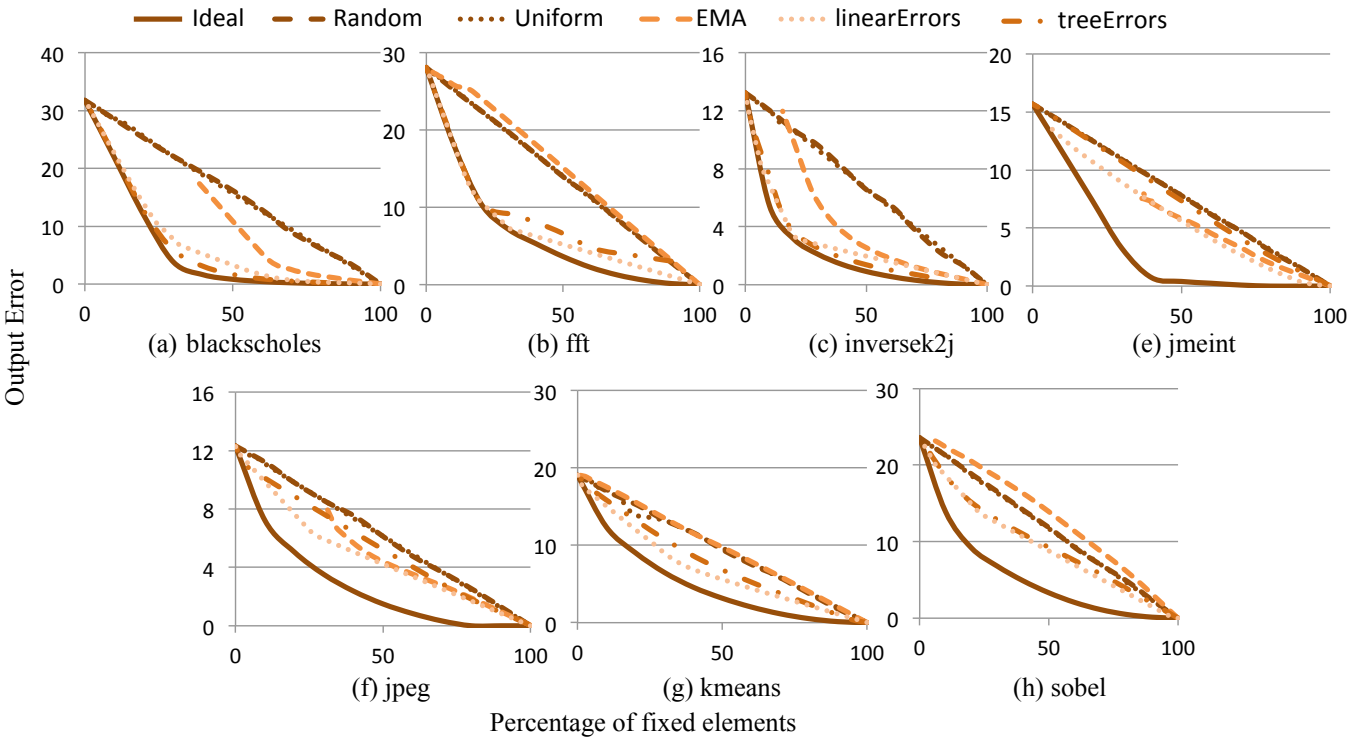

Figure 10: Output error with respect to the number of output elements fixed.

Parameter	Value	Parameter	Value
Fetch/Issue width	4/6	Load/Store Queue Entries	48/48
INT ALUs/FPUs	2/2	L1 iCache	32KB
Load/Store FUs	1/1	L1 dCache	32KB
Issue Queue Entries	32	L1/L2 Hit Latency	3/12 cycles
ROB Entries	96	L1/L2 Associativity	8
INT/FP Physical Registers	256/256	ITLB/DTLB Entries	128/256
BTB Entries	2048	L2 Size	2 MB
RAS Entries	16	Branch Predictor	Tournament

Table 2: Microarchitecutral parameters of an X86-64 cpu used in experiments.

4. Experimental Setup

We evaluate Rumba with a Neural Processing Unit (NPU) style accelerator [16]. Although we evaluate Rumba using a NPU-style accelerator, the design of Rumba is not specific to an accelerator as the core principles can be applied to a variety of approximation accelerators [41, 4]. We use the same hardware parameters as used by the NPU work for modeling the core and the accelerator. The remainder of this section describes the benchmarks, accelerator outputs and energy modeling setup used to evaluate the effectiveness of Rumba.

Benchmarks: We evaluate a set of benchmarks from various domains that map to approximate accelerators. The benchmarks represent a mix of computations from different domains and illustrate the effectiveness of Rumba across a variety of computation patterns. We use the same set of benchmarks as used in the NN accelerators [16, 4]. A brief description of these benchmarks along with their domain, train and test data is given in Table 1. Rumba NN (Neural Network) topology column in the table shows the NN topology used by Rumba. For example, 6->4->4->1 for *kmeans* implies that the NN has 6 inputs, two hidden layers of 4 neurons each and 1 output. The final column in this table shows the NN topology used by the unchecked NPU. In all cases, Rumba's error detection capabilities make it possible to chose a smaller or equal, therefore efficient, NN. The output quality of applications is usually measured by an application specific error metric [32, 31, 16]. This application specific error metric is given in the evaluation metric column in Table 1. We target a 90% output quality. This is in commensurate with the previous works in approximate computing [16, 6, 34].

Accelerator Output: We obtain the accelerator output (approximate output) by implementing NN using pyBrain [37] library. We find the best NN configuration by searching the NN topology space. The best configuration for our case is the smallest NN that does not produces excessive errors. The NN topology space is large thus the NN we consider have at most 2 layers and the number of neurons are restricted to at most 32 neurons in each layer (same restriction as in NPU [16]).

Energy Modeling: We run each application using the GEM5 [7] simulator to calculate the different microarchitectural activities. These activities are fed to McPAT [38], which calculates the baseline energy for the entire application. We use an X86-64 model for the cpu core and the microarchitectural parameters are given in Table 2. The accelerator design is an 8-Processing Elements (PEs) NPU and uses the same parameters for various structures of the PEs as given in the NPU paper [16]. We model the energy of the multiply-and-add for the linear error model and comparator in the same way as in the NPU paper. We calculate the energy for the light-weight checkers separately. The energy of these checkers and the energy of re-computation of elements on the CPU are combined to calculate the total energy for a particular scheme.

5. Evaluation

We evaluate Rumba for output quality, energy savings, false positives and the coverage of large errors. We also analyze the energy savings of Rumba for different target output qualities.

5.1. Output Quality

Output Error: Output errors are measured using the application specific metric given in Table 1 on the whole application output. Figure 10 shows the output error with respect to the number of output elements fixed for different techniques under consideration. Output error is directly related to the output quality. Output error of 5% represents 95% output quality. The y-axis of each plot in this figure shows the output error, while the x-axis shows the number of elements that need to be fixed to achieve that particular output error. *Random* fixes a given percentage of randomly selected output elements. For example, for fixing 10% of the elements *Random* selects 10% of output elements randomly and then recomputes them. Similarly, *Uniform* shows the output error when a given percentage of output elements to be fixed are chosen uniformly among all output elements. *Ideal* has the oracle knowledge about the approximation errors in all the output elements and it uses this oracle knowledge to fix a given percentage of the output elements. The data for *Ideal* is generated by sorting approximation errors in output elements by the error magnitude and then fixing the highest error elements. For example, to obtain output error when 10% of the elements are fixed for the *Ideal* scheme, the top 10% approximation error elements are fixed. Finally, *EMA*, *linearErrors* and *treeErrors* represent the output error when the errors are calculated by using the prediction models described in Section 3.2.

The techniques that are closest to the *Ideal* line in these plots represent the best possible achievable results. For a point on the x-axis, if the corresponding y value for a technique is close to y value of *Ideal* at the same x point, the technique is closer to the ideal case. For *inversek2j*, if 30% elements are fixed, *Ideal*, *Random*, *Uniform*, *EMA*, *linearErrors* and *treeErrors* will have 2.1%, 9.7%, 9.6%, 5.9%, 2.6 and 2.7% output errors, respectively. Hence, *linearErrors* and *treeErrors* are better

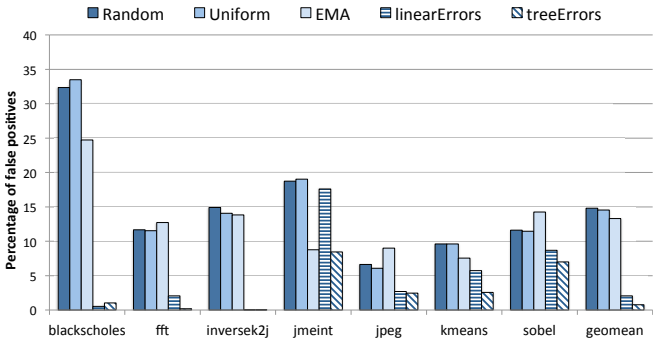

Figure 11: False positives at 90% target output quality. *Ideal* have zero false positives. A low number of false positives for *linearErrors* and *treeErrors* indicate their effectiveness in detecting large approximation errors.

techniques than *Random*, *Uniform* and *EMA* but worse than *Ideal*.

These plots also show that for some benchmarks (e.g., *kmeans*) *linearErrors* performs better and for others (e.g., *blackscholes*) *treeErrors* performs better. Overall, error prediction accuracy of a particular scheme is benchmark dependent.

False Positives: A false positive is a large error detected by a particular scheme that was not actually a large error. An error prediction scheme will have a false positive if the predicted error is high but the actual error is not. It is important to have low numbers of false positives for a technique for it to be practical. A high number would imply that the CPU would need to fix a large number of elements thus partially offsetting the gains of approximation. Figure 11 shows the number of false positives for 90% target output quality, i.e., 10% output error in output elements. For example, the first bar from the left for the *blackscholes* benchmark represents 32% false positives for *Random* if we target 90% output quality. *Random* and *Uniform* have a large percentage of false positives since these techniques randomly and uniformly, respectively, pick approximate output elements to fix and do not have any detection method. *Ideal* does not have any false positives since it has oracle knowledge of the errors in output elements. On average, *Ideal*, *Random*, *Uniform*, *EMA*, *linearErrors* and *treeErrors* have 0%, 14.8%, 14.5%, 13.3%, 2.1% and .76% false positives for 90% target output quality. *linearErrors* and *treeErrors* show a very low percentage of false positives and thus are effective at detecting large approximation errors.

Fixed Elements: Figure 12 shows the number of elements that are need to be fixed (recomputed) to achieve 90% output quality. A lower number of fixes implies that the energy overhead of re-execution on the CPU will be lower. Hence, a technique that fixes lower number of elements to achieve the same quality is better. For example, on average, *Random* requires 41% (29% more than *Ideal*) of the output elements to be fixed to achieve 10% output error. In comparison to *Ideal*, *linearErrors* and *treeErrors* just require 9% and 6% extra elements to be fixed to achieve the same output quality, respectively.

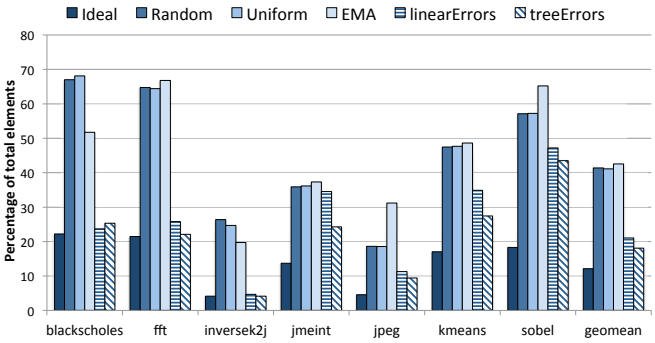

Figure 12: The number of elements that are required to be re-executed for a 90% target output quality.

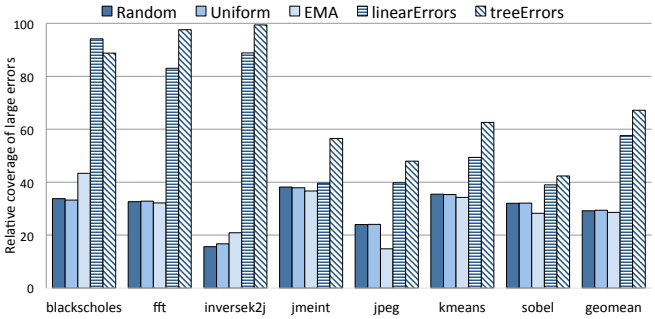

Figure 13: Relative coverage of large errors at 90% target output quality. *Ideal* has 100% coverage.

Large Error Coverage: Relative coverage is defined as the normalized ratio of detected large errors (larger than 20%) and the total number of fixes required. This ratio is normalized with respect to *Ideal*. This shows how good a prediction scheme is with respect to *Ideal*. Figure 13 shows the relative coverage of large errors for 90% target output quality. For example, the first bar from the left for *blackscholes* benchmark represents that relative coverage of *Random* is 29.2%. The relative coverage of scheme is high if it fixes a less number of elements to cover more large error elements for a given target output quality. On average, *linearErrors* and *treeErrors* are able to achieve 57.6% and 67.2% relative coverage, respectively.

5.2. Energy Consumption and Speedup

Figure 14 shows the energy consumed by various techniques in comparison to the CPU baseline for a target output quality of 90%. This figure shows the whole application energy savings. First column (labeled *NPU*) for each benchmark represents the energy savings of the unchecked NPU, i.e., no error checking mechanism is employed. *NPU* [16] reduces, on average, the CPU energy consumption by 3.2x. Note that since *NPU* does not have any fixing mechanism for large errors and so the output application quality is not always 90%. Without fixing any errors, output error, on average, is 20.6%. The other bars from left to right for each benchmark show energy consumed by *Ideal*, *Random*, *Uniform*, *EMA*, *linearErrors* and *treeErrors* schemes, respectively. The energy consumption shown for

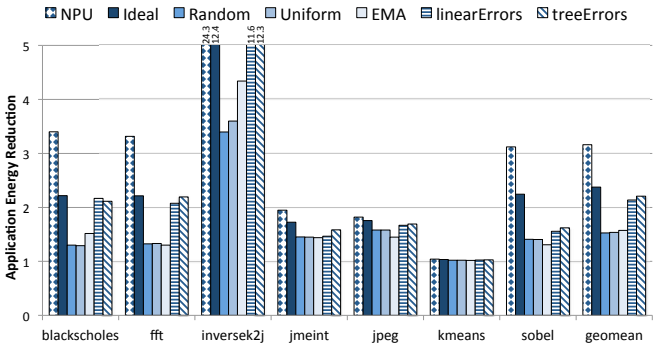

Figure 14: Energy consumption of Rumba, including the cost of re-computation and the energy used for the prediction of large approximation errors. *treeErrors* saves 2.2x energy while the unchecked NPU saves 3.2x energy.

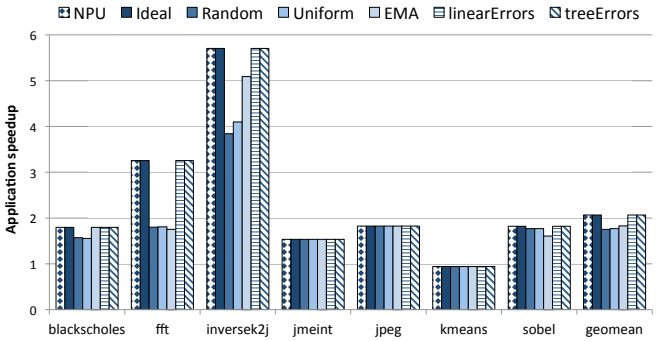

Figure 15: Speedup of each technique with respect to the CPU baseline. Rumba (*linearErrors* or *treeErrors*) maintains the same speedup (2.2x) as the NPU.

each of the schemes in this figure includes energy required to recompute the elements on the CPU and also the energy required for the checkers in the accelerator. As also observed in the NPU work [16], *kmeans* has very little energy gains and achieves slowdown because the code region that gets mapped to the NPU is very small and can be efficiently executed on the CPU itself. Energy savings of *sobel* decrease significantly for *linearErrors* and *treeErrors* schemes because this particular benchmark requires relatively large number of re-executions due to the lower prediction accuracy of errors.

Figure 15 shows the speedup all the schemes described earlier. Each scheme also factors in performance loss due to re-execution on the CPU if the CPU cannot keep up with the accelerator. Since Rumba (*linearErrors* or *treeErrors*) overlaps recovery on CPU with the accelerator execution, it is able to maintain the same speedup (2.1x) as the NPU. Our energy savings and speedup for the NPU baseline are close to the ones given in the NPU paper [16] but do not exactly match as we use different neural network libraries and simulation infrastructure.

Time for prediction: Figure 17 shows the time taken by the two error predictor model normalized with respect to the NPU. For all the benchmarks, *linearErrors* and *treeErrors* require less time than the NPU. Therefore, the predicted error is

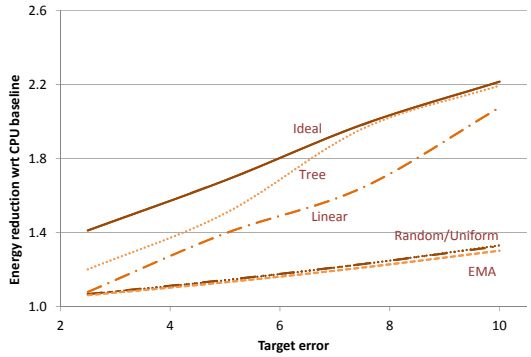

Figure 16: Energy consumption vs target error rate for *fft*.

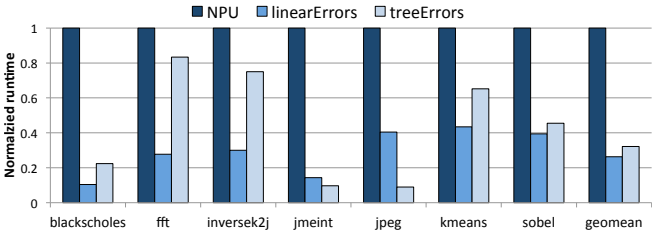

Figure 17: Time used by error prediction models in comparison to the NPU. This is normalized with respect to the NPU. Error prediction model are faster in all the cases, hence, the accelerator never needs to wait for the error prediction model to finish execution.

always available before NPU finishes and the NPU never needs to wait for the error predictor to finish, i.e., error prediction does not slow down the NPU.

Rumba reduces approximation errors overall by 2.1x (20.6% to 10%). Rumba achieves this error reduction while maintaining the same performance improvement as the NPU accelerator but reduces the energy savings from 3.2x to 2.2x in comparison to the unchecked NPU.

5.3. Case Studies

Energy vs. Output Quality: Figure 16 shows the energy consumption of different schemes with varying requirements on output quality for the *fft* benchmark. Energy savings for the unchecked NPU for *fft* is 3.3x. As expected, *Ideal* achieves the best energy savings among all techniques. *treeErrors* achieves energy savings close to the *Ideal* scheme for higher target error rates (> 7%). Note that the gap between *treeErrors* and *Ideal* increases as the demands on output quality increases (greater than 97%). This is because *Ideal* knows exactly which elements to fix to achieve certain target output quality while *treeErrors* (or *linearErrors*) must predict such cases. This causes the false positives for *treeErrors* (or *linearErrors*) to start increasing, requiring more re-computation and more energy consumption. Thus, the gap between *Ideal* and *treeErrors* (or *linearErrors*) is larger at high demands on output quality.

CPU Activity: In this second case study, we show an example of the CPU activity in conjunction with the accelerator. The top half of Figure 18 shows the percentage difference

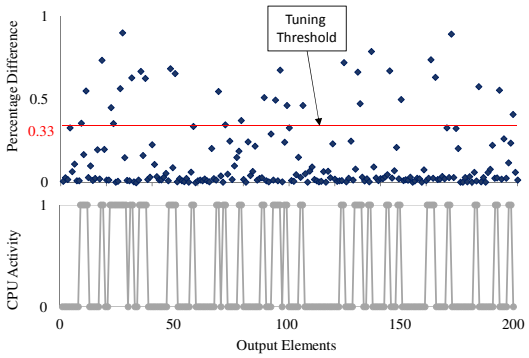

Figure 18: The approximation accelerator and the CPU work in tandem. The CPU works on re-computing detected large error iterations while the accelerator continues with the execution. In this case, 0.33 is the tuning threshold used to achieve 10% target error rate.

of each output element (on y-axis) with *treeErrors* for 200 elements (on x-axis). To achieve 10% target output error, a tuning threshold of 0.33 is required on this percentage difference (y-axis). The bottom half of the figure shows the CPU activity. The accelerator and the CPU work in tandem, i.e., the CPU fixes the detected large approximation errors while the accelerator executes other iterations. Only 30 elements out of these 200 (15%) are above this threshold and thus the CPU can keep up with an approximate accelerator as fast as 6.67x.

6. Related Work

Approximate computing, where the accuracy is traded off for better performance or higher energy efficiency, is a well-known technique. Approximate computing techniques can be broadly classified into two categories: Software-based and hardware-based approaches. Software-based approaches are usually algorithmic modifications and can be utilized without any hardware modifications. Loop perforation [1] is one of the well-known software approximation techniques which skips the iterations of loops randomly or uniformly. Rinard et al. [30] use early phase termination technique to terminate parallel phase as soon as there are too few remaining tasks to keep the processor busy to prevent the processors from being idle and wasting energy. Sartori et al. [36] introduce a software approximation technique which targets control divergence on GPUs. Paraprox [31] is a software framework which detects patterns in data parallel applications and applies different approximation techniques such as loop perforation, approximate memoization, and tile approximation based on the detected patterns. All these software approximation techniques need a quality management system to monitor the output quality and control the aggressiveness of the approximation during execution.

Different hardware approximation techniques have also been proposed to save energy while improving performance. EnerJ [34] proposed hardware techniques such as voltage scaling, width reduction in floating point operations, reducing

DRAM refresh rate, and reducing SRAM supply voltage to reduce energy consumption. Esmaeilzadeh et al. [15] demonstrated dual-voltage operation, with a high voltage for precise operations and a low voltage for approximate operations. The low-voltage pipeline introduces faults in the operations and hence these operation are approximate. We compare against the hardware neural network [16] proposed by the same authors extensively in our results section. Du et al. [14] also use hardware neural networks to trade off accuracy for energy savings. Amant et al. [4] design limited precision analog hardware to accelerate approximable code sections. Other works [39, 41] design different approximate accelerators. Sampson et al. [35] improve memory array lifetime using approximation. Flikker [22] is an application-level technique that reduces the refresh rate of DRAM memories which store non-critical data. The Rumba quality management system can be added to these hardware-based approximation techniques to control and improve their output quality.

There exist a few quality management solutions to control quality in an approximate computing system. Ansel et al. [5] use a genetic algorithm to find the best approximate code that provides the acceptable quality. In this work, the programmer writes runtime low overhead checking functions to verify output quality online. However, Rumba can automatically manage the output quality without programmer's help. CCG [33] is another quality monitoring technique. In this technique, while GPU runs the approximate version, the CPU is responsible to check the quality of a subset of data for the next invocation. To reduce the performance overhead of monitoring, size of the subset that is processed by the CPU is small and thus, CCG's accuracy to predict the output quality is limited. Unlike CCG, Rumba has light-weight checkers and therefore, it can investigate larger subset of the data compared to CCG.

Green [6] is a framework that developers can use to take advantages of approximation opportunities to achieve better performance or reduce energy consumption. Green builds a quality of service model based on the profiling data that gets used at runtime. In order to make sure that output quality is acceptable, Green checks the output quality once in every N invocations. SAGE [32] is an approximation framework for GPUs that automatically generates approximate versions of the input program using skipping atomic expressions, compressing data, and tile approximation. SAGE also uses a similar quality sampling strategy as Green to check the output quality frequently. However, in contrast to these techniques, because of its light-weight checkers, Rumba checks all invocations to reduce large errors and to make sure that the output quality is acceptable for all invocations. Some other techniques [9, 10, 29, 27, 24] statically analyze applications assuming an input distribution to reason about the output quality under approximation. Such techniques do not need sampling but can only handle limited computational patterns and approximation methods.

PowerDial [18] is a framework that dynamically monitors the application's performance during runtime. When the performance drops below target performance, PowerDial will increase the aggressiveness of the approximation to match the performance requirements. Their goal is to maximize accuracy while maintaining application's performance. Several probabilistic reasoning models [25, 10, 8, 26, 11] are also introduced to compute the probability of the output being wrong. In contrast, Rumba dynamically monitors the output quality during runtime and recovers from the large errors generated by an approximation technique.

Hardware reliability for soft computations and approximate computing share the same basic underlying philosophy. Hardware reliability solutions [19, 40, 21] for soft computations aim to allow errors in error tolerant parts of an application with the goal of lowering the cost of reliability. The idea of re-execution has previously been used in the context of reliability to recover against hardware faults [13, 17]. We leverage this idea in the context of recovering against approximation errors and it fits well with the nature of the code regions (pure) that are mapped to approximate accelerators.

7. Conclusions

Approximate computing can be employed for an emerging class of applications from various domains such as multimedia, machine learning and computer vision. Approximate computing trades off accuracy for better performance and/or energy efficiency. However, the quality control of approximated outputs has largely gone unaddressed. In this work, we propose Rumba for online detection and correction of large errors in an approximate computing environment.

Rumba predicts large approximation errors by light-weight checkers and corrects them by recomputing individual elements. Our results demonstrate that Rumba is effective at predicting large errors and follows an ideal case very closely. Across a variety of benchmarks from different domains, we show that Rumba reduces the output error by 2.1x in comparison to an accelerator for approximate programs while maintaining the same performance improvement. To achieve this, the Rumba framework reduces the energy savings, on average, from 3.2x to 2.2x in comparison to an unchecked accelerator.

Acknowledgments

This research is supported by the National Science Foundation XPS grant CCF-1438996 and by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. We would like to thank Gaurav Chadha for suggesting the name Rumba and the fellow members of the CCCP research group for numerous productive discussions. We would also like to thank the anonymous reviewers for their constructive comments and suggestions for improving this work.

References

[1] A. Agarwal, M. Rinard, S. Sidiroglou *et al.*, "Using code perforation to improve performance, reduce energy consumption, and respond to failures," MIT, Tech. Rep. MIT-CSAIL-TR-2009-042, Mar. 2009. [Online]. Available: http://hdl.handle.net/1721.1/46709

[2] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 922–927, 2005.

[3] C. Alvarez, J. Corbal, and M. Valero, "Dynamic tolerance region computing for multimedia," *IEEE Transactions on Computers*, vol. 61, no. 5, pp. 650–665, 2012.

[4] R. S. Amant, A. Yazdanbakhsh, J. Park *et al.*, "General-purpose code acceleration with limited-precision analog computation," in *Proc. of the 41st Annual International Symposium on Computer Architecture*, 2014, p. To Appear.

[5] J. Ansel, Y. L. Wong, C. Chan *et al.*, "Language and compiler support for auto-tuning variable-accuracy algorithms," in *Proc. of the 2011 International Symposium on Code Generation and Optimization*, 2011, pp. 85 –96.

[6] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *Proc. of the '10 Conference on Programming Language Design and Implementation*, 2010, pp. 198–209.

[7] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[8] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain<T>: A first-order type for uncertain data," in *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 51–66.

[9] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *Proc. of the '12 Conference on Programming Language Design and Implementation*, 2012, pp. 169–180.

[10] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *Proc. of the 2013 ACM SIGPLAN international conference on Object-Oriented Systems and applications*, 2013, pp. 33–52.

[11] S. Chaudhuri, S. Gulwani, R. L. Roberto, and S. Navidpour, "Proving programs robust," in *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 102–112.

[12] S. Che, M. Boyer, J. Meng *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. of the IEEE Symposium on Workload Characterization*, 2009, pp. 44–54.

[13] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *Proc. of the 37th Annual International Symposium on Computer Architecture*, Jun. 2010, pp. 497–508.

[14] Z. Du, A. Lingamneni, Y. Chen *et al.*, "Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators," in *Proc. of the 19th Asia and South Pacific Design Automation Conference*, 2014, pp. 201–206.

[15] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 301–312.

[16] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proc. of the 45th Annual International Symposium on Microarchitecture*, 2012, pp. 449–460.

[17] S. Feng, S. Gupta, A. Ansari *et al.*, "Encore: low-cost, fine-grained transient fault recovery," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 398–409.

[18] H. Hoffmann, S. Sidiroglou, M. Carbin *et al.*, "Dynamic knobs for responsive power-aware computing," in *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 199–212.

[19] D. S. Khudia and S. Mahlke, "Harnessing soft computations for low-budget fault tolerance," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2014, pp. 319–330.

[20] M. Kruijf, K. Sankaralingam, and S. Jha, "Static analysis and compiler implementation of idempotent processing," in *Conference on Programming Language Design and Implementation*, Beijing, China, 2012.

[21] X. Li and D. Yeung, "Exploiting soft computing for increased fault tolerance," in *Workshop on Architectural Support for Gigascale Integration*, 2006.

[22] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," in *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 213–224.

[23] Mazaika, "Software solutions for photographic mosaics," http://www.mazaika.com.

[24] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing sequential programs with statistical accuracy tests," *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 2s, pp. 88:1–88:26, May 2013.

[25] S. Misailovic, D. M. Roy, and M. C. Rinard, "Probabilistically accurate program transformations," in *Proc. of the 18th Static Analysis Symposium*, 2011, pp. 316–333.

[26] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *Proc. of the 2006 International Conference on Supercomputing*, 2006, pp. 324–334.

[27] M. Rinard, "Probabilistic accuracy bounds for perforated programs: A new foundation for program analysis and transformation," in *20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, 2011, pp. 79–80.

[28] M. Rinard, "Parallel synchronization-free approximate data structure construction," in *Proc. of the 5th USENIX Workshop on Hot Topics in Parallelism*, 2012, pp. 1–8.

[29] M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou, "Patterns and statistical analysis for understanding reduced resource computing," in *Proc. of the 2010 ACM SIGPLAN international conference on Object-Oriented Systems and applications*, 2010, pp. 806–821.

[30] M. C. Rinard, "Using early phase termination to eliminate load imbalances at barrier synchronization points," in *Proc. of the 22nd ACM SIGPLAN international conference on Object-Oriented Systems and applications*, 2007, pp. 369–386.

[31] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 35–50.

[32] M. Samadi, J. Lee, D. A. Jamshidi *et al.*, "SAGE: Self-tuning approximation for graphics engines," in *Proc. of the 46th Annual International Symposium on Microarchitecture*, 2013, pp. 13–24.

[33] M. Samadi and S. Mahlke, "CPU-GPU collaboration for output quality monitoring," in *1st Workshop on Approximate Computing Across the System Stack*, 2014, pp. 1–3.

[34] A. Sampson, W. Dietl, E. Fortuna *et al.*, "EnerJ: approximate data types for safe and general low-power computation," *Proc. of the '11 Conference on Programming Language Design and Implementation*, vol. 46, no. 6, pp. 164–174, Jun. 2011.

[35] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 25–36.

[36] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications." in *IEEE Transactions on on Multimedia*, 2012, pp. 427–428.

[37] T. Schaul, J. Bayer, D. Wierstra *et al.*, "Pybrain," *The Journal of Machine Learning Research*, vol. 11, pp. 743–746, 2010.

[38] L. Sheng, H. A. Jung, R. Strong *et al.*, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. of the 42nd Annual International Symposium on Microarchitecture*, 2009, pp. 469–480.

[39] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *Proc. of the 39th Annual International Symposium on Computer Architecture*, 2012, pp. 356–367.

[40] A. Thomas and K. Pattabiraman, "Error detector placement for soft computation," in *International Conference on Dependable Systems and Networks*. IEEE, 2013.

[41] S. Venkataramani, V. K. Chippa, S. T. Chakradhar *et al.*, "Quality programmable vector processors for approximate computing," in *Proc. of the 46th Annual International Symposium on Microarchitecture*, 2013, pp. 1–12.