Quality Control for Approximate Accelerators by Error Prediction

Daya Shanker Khudia, Babak Zamirai, Mehrzad Samadi, and Scott Mahlke
The University of Michigan

Editor's notes:
How to ensure the output quality is one of the most critical challenges in approximate computing. This paper presents an online quality management system in an approximate-accelerator-based computing environment that can effectively detect and correct large approximation errors.
—Todd Mytkowicz, Microsoft Research

Computation accuracy can be traded off to achieve better performance and/or energy efficiency. The techniques to achieve this tradeoff 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. 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.

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 [9], and discarding high overhead computations [10]. Hardware-based approximation techniques employ neural processing modules [3], [6], analog circuits [3], low-power arithmetic logic units (ALUs) and storage [11], and hardware-based fuzzy memoization [2]. Approximation accelerators [5], [6], [12] 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 central processing unit (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 [4], [10], techniques that build an offline quality model based on the profiling data [4], [6].

However, these techniques have four critical limitations.

- As the output quality is dependent on the input values, therefore, sampling techniques are not capable of capturing all changes of the output

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.
Digital Object Identifier 10.1109/MDAT.2015.2501306
Date of publication: 17 November 2015; date of current version: 18 January 2016.

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.
- Using these quality management techniques, if the output quality drops below an acceptable threshold, there is no way to improve the quality other than reexecuting the whole program on the exact hardware. However, this recovery process has high overhead and it offsets the gains of approximation.
- Previous works [3], [6] 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.
- 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 lightweight output monitoring to ensure more consistent output quality.

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 need 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 reexecutes 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 reexecuted.

The major contributions of this work are as follows.

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

Challenges

The ability of applications to produce results of acceptable output quality in an approximate computing environment is necessary to ensure a positive user experience but quality control faces the following challenges.

Challenges of managing output quality

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 [6], [9], [10] reported that the cumulative distribution function (cdf) of the errors of an approximated application's output follows the curve shown in Figure 1a. Figure 1a 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

[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.

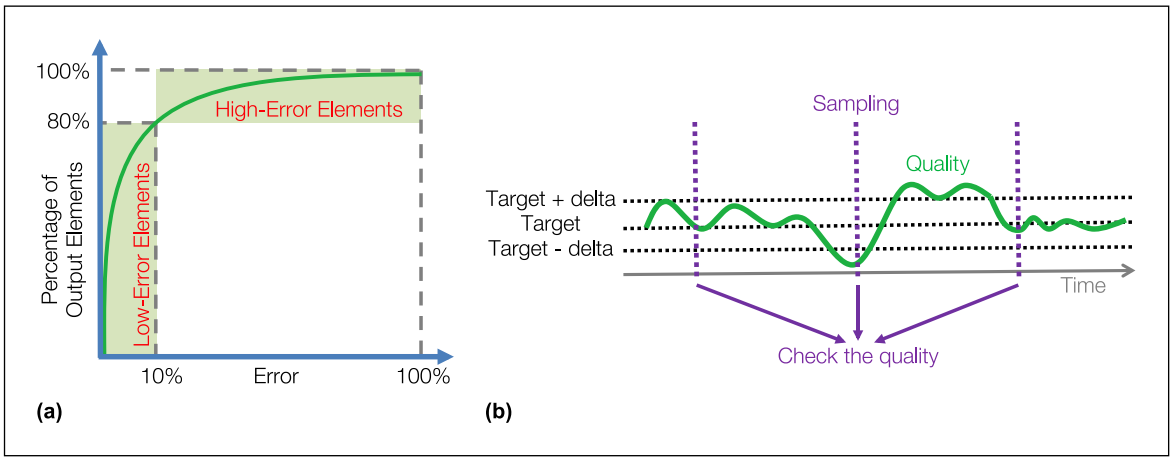

Figure 1. (a) Approximation techniques typically produce a small number of large errors and a large number of small errors. (b) The common technique of sampling to detect quality violations can miss bad quality results.

images by adding errors such that the overall average error is 10% in both images. Figure 2 is the original image. In Figure 2, only 10% of pixels have 100% errors while the rest of pixels are exact. On the other hand, all pixels in Figure 2 have about 10% error. Even though the overall output error is the same for both generated images, errors in Figure 2 are more noticeable than Figure 2 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 [4], [6], [9], [10]. In this case, these techniques must consider the worst case to make sure that the output quality is acceptable. To show this, we experimented with an image processing application called "mosaic" that generates a large image using many small images under a well-known approximation technique called loop perforation. Loop perforation drops iterations of the loop randomly or uniformly.

Under this experiment, average error for 800 images of flowers 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. This shows that the output quality is highly input dependent and 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 lightweight 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 [4], [10] suggest running an application twice (exact and approximate versions) and comparing the results to compute output quality as shown in Figure 1b. Unfortunately, it has high overhead and it is not feasible to monitor all invocations. Running exact and approximate versions at all times will nullify the advantages of using approximation.

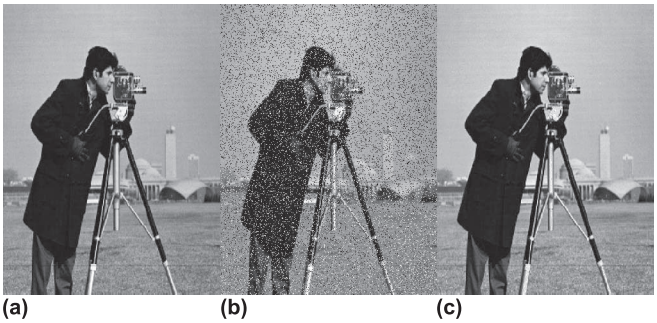

Figure 2. (a) Exact output. In (b) and (c), there is the same average error (90%). However, errors in (b) are large errors, hence, more noticeable.

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.

Design of Rumba

A high-level block diagram of the Rumba system is shown in Figure 3. The offline part of the Rumba system consists 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 application binary.

The execution subsystem of Rumba is also shown in the same figure. The core communicates to the accelerator using input/output (I/O) queues for data transfers from the core to the accelerator and back from accelerator to the core. We augment the approximate accelerator by an error checker module to detect approximation errors. 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 3. The CPU collects these bits from the recovery queue and reexecutes the iterations that their recovery bit is set. Output merger chooses the exact or approximate output as final result. 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 reexecuted.

To overcome the four challenges outlined in an earlier section, Rumba exploits two observations found in the kernels that are amenable to approximation: predictiveness of errors and recovery by selective reexecution.

Lightweight error prediction

Rumba's detection module is based on the observation that it is possible to predict the errors of an approximate accelerator using a computationally inexpensive prediction model. Rumba dynamically employs lightweight checkers (predictors) to detect approximation errors. A threshold on the predicted errors is used to classify errors in the output elements

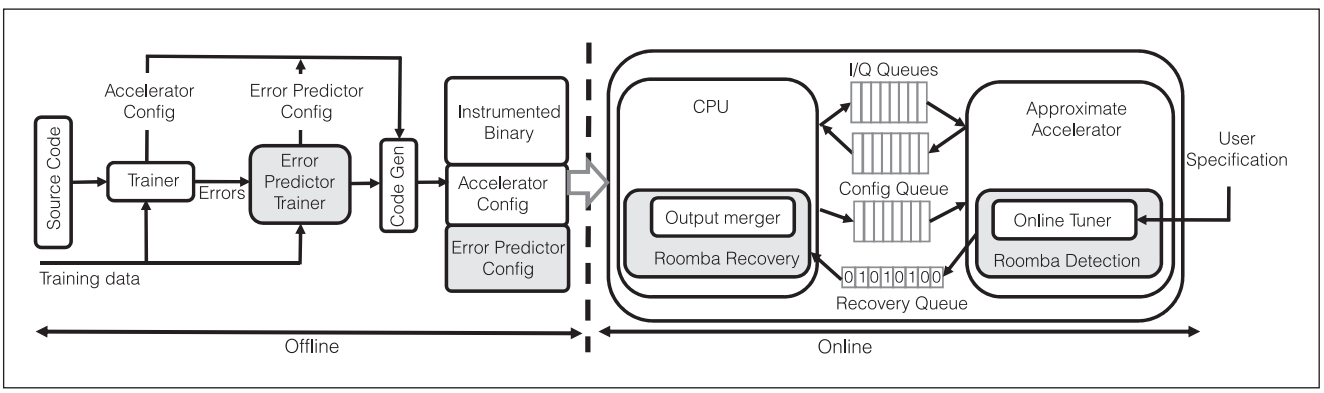

Figure 3. High-level block diagram of the Rumba system. The offline components determine the suitability of an application for the Rumba acceleration environment. The online components include detection and recovery modules. The approximation accelerator communicates a recovery bit corresponding to the ID of the elements to recompute with the CPU via a recovery queue.

as high. Therefore, Rumba targets output elements with high errors as mentioned in Challenge I. Also, since Rumba has lightweight checkers, the checks can be performed online for all the elements of each invocation (Challenge III). Complex but more accurate checkers to detect large approximation will offset the gains of approximation and, thus, are not desirable. 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, we consider linear and decision tree models for error prediction, i.e., the error is predicted using one of these models. For output-based methods, we use a moving-average-based method. The difference between current element and the moving average can be used to detect large errors in a number in the sequence. We used exponential moving average (EMA).

Low-overhead recovery

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 reexecuted without any side effects. Similar characteristics have been previously used in recovering program from external errors simply by reexecuting [7]. 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 and found out that more than 70% of them can be reexecuted without any side effects. It is not a new restriction imposed by Rumba as previously proposed approximate accelerators [3], [6] 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 reexecute that iteration to generate the exact output elements. In this case, there is no need to rerun 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 reexecuted to target Challenge IV. These two techniques are described in detail in [8].

Experimental setup and results

We evaluate Rumba with a neural processing unit (NPU)-style accelerator [6]. Although we evaluate Rumba using an 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 [3], [12]. We use the same hardware parameters as used by the NPU work for modeling the core and the accelerator.

Benchmarks and NPU modeling

We evaluate a set of benchmarks (blackscholes, fft, inversek2j, jmeint, jpeg, kmeans, and sobel) from various domains. The benchmarks represent a mix of computations from different domains and illustrate the effectiveness of Rumba across a variety of computation patterns. The output quality of applications is usually measured by an application-specific error metric [6], [9], [10], and we use metric such as mean relative error or the number of mismatches depending on the benchmark. We target a 90% output quality. This target quality is in commensurate with the previous works in approximate computing [4], [6], [11].

Output quality

Output error. Figure 4 shows the average output error across benchmarks with respect to the number of output elements recomputed 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 in this

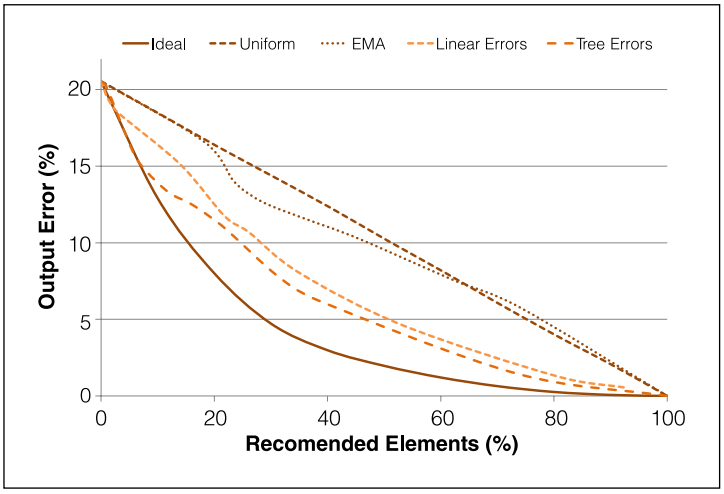

Figure 4. Output error versus the number of recomputed elements.

figure shows the output error, while the *x*-axis shows the number of elements that need to be recomputed to achieve that particular output error. *Uniform* shows the output error when a given percentage of output elements to be fixed are chosen uniformly among all output elements. For example, for recomputing 50% of the elements, *Uniform* selects every other output element and then recomputes it. *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 III.

The techniques that are closest to the *Ideal* line in this plot represent the best possible achievable results. For a point on the *x*-axis, if the corresponding *y* value for a technique is close to the *y* value of *Ideal* at the same *x* point, the technique is closer to the ideal case. Hence, *LinearErrors* and *TreeErrors* are better techniques than *Random*, *Uniform*, and *EMA*, but worse than *Ideal*. The best performing technique (*TreeErrors*) reduces the output error by 2.1× in comparison to an unchecked approximate accelerator (21% to 10%).

Energy consumption and speedup. Rumba (*LinearErrors* or *TreeErrors*) overlaps recovery on CPU with the accelerator execution, and it is able to maintain the same speedup (2.1×) as the NPU. However, to achieve 2.1× error reduction, the Rumba framework (*TreeErrors*) reduces the energy savings, on average, from 3.2× to 2.2× in comparison to an unchecked accelerator. Detailed energy and speedup results are presented in [8].

Checker design space. Figure 5a shows energy cost versus error for different NPU versus NN and decision tree checker configurations for *Blackscholes* benchmark. Each circle on this graph is a combination of a particular configuration for the NPU and a particular configuration of the NN checker. For example, a point on this graph can correspond to an NPU configuration of two hidden layers of 32 neurons each and a checker NN configuration that has eight neurons in a single layer. The points represented by indexed square are the configurations that do not have any checker, i.e., these are the NPU only design points. A single square on this graph shows the error and cost for no accelerator, i.e., all the computations are performed exactly on the CPU. Points shown by a cross sign correspond to an NPU and decision tree of certain depth. On the *x*-axis of this graph is the energy cost of a configuration relative to the CPU. The CPU has a cost of 1 and output error of 0% as shown in the figure. In this figure, output error for a configuration is shown on the *y*-axis. The number in the box corresponding to each design point is the number of neurons in each hidden layer. For example, the box with 64 – >64 and 8 – >8 implies that the NPU has two hidden layers of 64 neurons each and the checker has two hidden layers of eight neurons each. Similarly, the box with 64 – >64 and 5 implies that the NPU has two hidden layers of 64 neurons each and the decision tree checker has a depth of 5. This figure only shows some selective points and labels them. The NN topology space is large thus the NN we consider, for NPU as well as checker, has at most two layers and the number of neurons is restricted to at most 128 neurons in each layer. With this design constraint, in total, 5761 design points are possible for benchmarks, and for *Blackscholes*, all these are plotted in Figure 5b.

Some trends are clearly demonstrated in these figures. First, there is no NPU configuration that is able to achieve output error less than 20% for the *Blackscholes* benchmark. So if 20% error is not acceptable in output then this benchmark cannot be approximated with an NPU accelerator. NPU only designs are efficient in terms of energy but have high error (all are in top left half of the plot). Second, with the combination of the NPU and a checker, we can improve energy efficiency with respect to the CPU and keep the error low as well. Third, a benchmark that produces unacceptable output is able to produce acceptable outputs with a combination of the NPU accelerator and an NN or decision tree checker.

We have explored the NPU-checker design space for all the benchmarks given in earlier. Let us suppose that a user wants to obtain a design that has half the

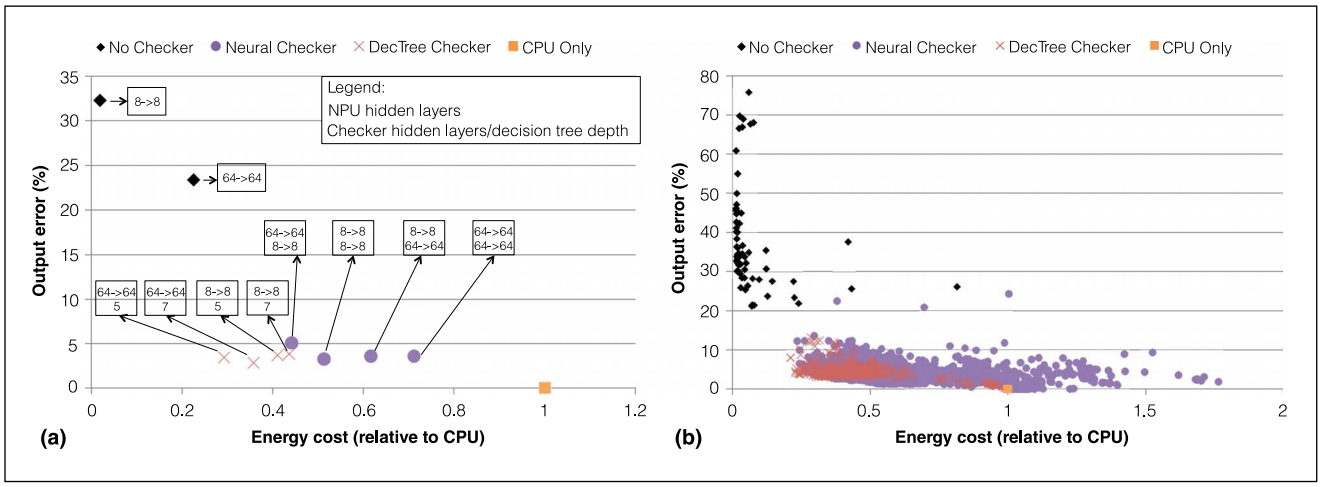

Figure 5. Error and energy cost of selective codesigns of the accelerator and checkers (a) and all explored designs (b).

error of the best NPU-only design for a particular benchmark. We can obtain these best half error designs by analyzing the design space exploration results. An analysis of these results shows that for blackscholes, fft, and inversek2j, the best checker (lowest energy cost) to achieve half error is a neural checker and for jmeint, jpeg, and kmeans, decision tree checker works best. sobel benchmark has very low error with almost all the configuration of the NPU, hence, we do not get much benefits of using a checker with this benchmark. These results demonstrate that the best checker type to achieve 50% less error is application dependent.

Overall, the results show that NPU-checker codesign provides many choices, and a user can pick a design based on the error requirements for a particular application.

Approximate computing can be employed for an emerging class of applications from various domains such as multimedia, machine learning, and computer vision to tradeoff accuracy for better performance and/or energy efficiency. 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 lightweight checkers and corrects them by recomputing individual elements. Across a set of benchmarks from different domains, we show that Rumba reduces the output error by 2.1× 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 3.2× to 2.2× in comparison to an unchecked accelerator. ■

Acknowledgment

This work was supported by the National Science Foundation under XPS Grant CCF-1438996 and by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by the Microelectronics Advanced Research Corp. (MARCO) and the Defense Advanced Research Projects Agency (DARPA). An earlier version of this article was presented at the 42nd Annual International Symposium on Computer Architecture (ISCA 2015) [8].

■ References

[1] A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann, "Using code perforation to improve performance, reduce energy consumption, respond to failures," Massachusetts Inst. Technol. (MIT), Cambridge, MA, USA, 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 Trans. Comput.*, vol. 54, no. 7, pp. 922–927, Jul. 2005.

[3] R. S. Amant et al., "General-purpose code acceleration with limited-precision analog computation," in *Proc. 41st Annu. Int. Symp. Comput. Architect.*, 2014, pp. 505–516.

[4] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using

controlled approximation," in *Proc. Conf. Programm. Lang. Design Implement.*, 2010, pp. 198–209.

[5] Z. Du et al., "Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators," in *Proc. 19th Asia South Pacific Design Autom. Conf.*, 2014, pp. 201–206.

[6] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proc. 45th Annu. Int. Symp. Microarchitect.*, 2012, pp. 449–460.

[7] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August, "Encore: Low-cost, fine-grained transient fault recovery," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitect.*, 2011, pp. 398–409.

[8] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An online quality management system for approximate computing," in *Proc. 42nd Annu. Int. Symp. Comput. Architect.*, 2015, pp. 554–566.

[9] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *Proc. 19th Int. Conf. Architect. Support Programm. Lang. Oper. Syst.*, 2014, pp. 35–50.

[10] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning approximation for graphics engines," in *Proc. 46th Annu. Int. Symp. Microarchitect.*, 2013, pp. 13–24.

[11] A. Sampson et al., "EnerJ: Approximate data types for safe and general low-power computation," in *Proc. Conf. Programm. Lang. Design Implement.*, Jun. 2011, vol. 46, no. 6, pp. 164–174.

[12] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *Proc. 46th Annu. Int. Symp. Microarchitect.*, 2013, DOI: 10.1145/2540708.2540710.

Daya Shanker Khudia is a Software Engineer at Intel Corporation, Santa Clara, CA, USA. His research interests include approximate computing, compilers, and hardware reliability. Khudia has a PhD in computer science and engineering from the University of Michigan, Ann Arbor, MI, USA.

Babak Zamirai is currently working toward a PhD at the Computer Science and Engineering Department, University of Michigan, Ann Arbor, MI, USA. His research interests include approximate computing, neural networks, and neuromorphic accelerators.

Mehrzad Samadi is a Research Fellow at the University of Michigan, Ann Arbor, MI, USA. His research interests include approximate computing, compilers, and automatic parallelization. Samadi has a PhD in computer science and engineering from the University of Michigan (2014).

Scott Mahlke is a Professor in the Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI, USA, where he leads the Compilers Creating Custom Processors Research (CCCP) Group. His research interests include energy-efficient processor design, compilers for multicore processors, and reliable systems. Mahlke has a PhD from the University of Illinois, Champaign, IL, USA (1997). He is a Fellow of the IEEE.

■ Direct questions and comments about this article to Daya Shanker Khudia, Department of Computer Science and Engineering, The University of Michigan, Ann Arbor, MI, USA; dskhudia@umich.edu.