Appears in the Proceedings of the 43th International Symposium on Computer Architecture, 2016

Towards Statistical Guarantees in Controlling Quality Tradeoffs for Approximate Acceleration

Divya Mahajan Amir Yazdanbakhsh Jongse Park Bradley Thwaites Hadi Esmaeilzadeh
Alternative Computing Technologies (ACT) Lab
Georgia Institute of Technology
{divya_mahajan, a.yazdanbakhsh, jspark, bthwaites}@gatech.edu hadi@cc.gatech.edu

***Abstract*—Conventionally, an approximate accelerator replaces every invocation of a frequently executed region of code without considering the final quality degradation. However, there is a vast decision space in which each invocation can either be delegated to the accelerator—improving performance and efficiency–or run on the precise core—maintaining quality. In this paper we introduce MITHRA, a co-designed hardware-software solution, that navigates these tradeoffs to deliver high performance and efficiency while lowering the final quality loss. MITHRA seeks to identify whether each individual accelerator invocation will lead to an undesirable quality loss and, if so, directs the processor to run the original precise code.**

This identification is cast as a binary classification task that requires a cohesive co-design of hardware and software. The hardware component performs the classification at runtime and exposes a knob to the software mechanism to control quality tradeoffs. The software tunes this knob by solving a *statistical optimization problem* that maximizes benefits from approximation while providing statistical guarantees that final quality level will be met with high confidence. The software uses this knob to tune and train the hardware classifiers. We devise two distinct hardware classifiers, one table-based and one neural network based. To understand the efficacy of these mechanisms, we compare them with an ideal, but infeasible design, *the oracle*. Results show that, with 95% confidence the table-based design can restrict the final output quality loss to 5% for 90% of unseen input sets while providing 2.5× speedup and 2.6× energy efficiency. The neural design shows similar speedup however, improves the efficiency by 13%. Compared to the table-based design, the oracle improves speedup by 26% and efficiency by 36%. These results show that MITHRA performs within a close range of the oracle and can effectively navigate the quality tradeoffs in approximate acceleration.

***Keywords*-Approximate computing, accelerators, quality control, statistical guarantees, statistical compiler optimization**

I. INTRODUCTION

With the effective end of Dennard scaling [1], energy efficiency fundamentally limits microprocessor performance [2], [3]. As a result, there is a growing interest in specialization and acceleration, which trade generality for significant gains in performance and efficiency. Recent work shows three orders of magnitude improvement in efficiency and speed with Application Specific ICs [4]. However, designing ASICs for the massive and rapidly-evolving body of general-purpose applications is currently impractical. Programmable accelerators [5]–[7] establish a middle ground that exploit certain characteristics of the application to achieve performance and efficiency gains at the cost of generality. Tolerance to approximation is one such application characteristic. As the growing body of research in approximation shows, many application domains including web search, machine learning, multimedia, cyber-physical systems, vision, and speech recognition can tolerate small errors in computation [8]–[15]. *Approximate accelerators* exploit this application characteristic by trading off computational accuracy for higher performance and better efficiency [16]–[23]. Each invocation of the approximate accelerator improves performance and efficiency but may also lower the quality of the final output. The common practice in approximate acceleration is to *always* invoke the accelerator in lieu of a frequently-executed safe-to-approximate region of code, e.g., a function in a loop. Always invoking the accelerator provides maximum gains from approximation but can potentially lead to an unacceptable *fixed degree* of quality loss. This approach does not provide the flexibility to explore the tradeoffs between quality and gains in performance and efficiency.

This paper tackles this shortcoming and defines a cohesively co-designed hardware-software solution, MITHRA, with components in both compiler and microarchitecture. MITHRA seeks to identify whether each individual accelerator invocation will lead to an undesirable quality loss. If MITHRA speculates that a large quality degradation is likely, it directs the processor to run the original precise code. This paper makes the following contributions by exploring the unique properties, challenges, and tradeoffs in designing MITHRA and defines solutions that address them.

(1) We find that a relatively low fraction of the accelerator invocations lead to large errors and need to be excluded from approximate acceleration. To accomplish the challenging task of filtering out this small subset and still delivering significant gains, we introduce MITHRA– a hardware-software co-design–for controlling the quality tradeoffs. The guiding principle behind MITHRA is that quality control is a binary classification task that either determines to invoke the approximate accelerator or the original precise code for each invocation.

(2) We devise MITHRA, the framework comprising of both hardware and software components. Hardware is devised to perform the binary classification task at runtime. The hardware also provides a knob to the software to control the quality tradeoffs. The software solves a statistical optimization problem to tune the knob. The solution provides statistical guarantees with a high confidence that desired quality loss levels will be met on unseen datasets. This tuned knob is used to pre-train the hardware classifiers.

(3) We evaluate MITHRA using an existing approximate accelerator [16] on a set of benchmarks with diverse error behaviors. We compare our designs with an ideal but infeasible mechanism, referred to as *the oracle*. The oracle maximizes the performance and energy benefits by filtering out the fewest possible accelerator invocations for any level of final quality loss. We devise two realistic instances of hardware classifiers–table-based and neural network based– that aim to mimic the behavior of the oracle. The results show that both of our designs closely follow the oracle in delivering performance and efficiency gains.

For 5% quality loss, with 95% confidence and 90% success

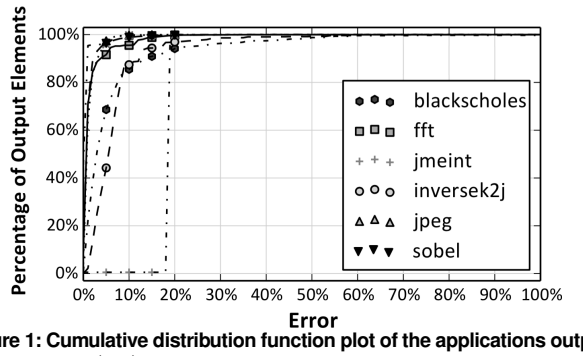


Figure 1: Cumulative distribution function plot of the applications output error. A point (x, y) implies that y fraction of the output elements see error less than or equal to x [16].

rate, the table-based design with eight tables each of size 0.5 KB achieves 2.5× average speedup and 2.6× average energy reduction. The neural design shows similar speedup, however provides 13% better energy reduction. Compared to the table-based design, the ideal oracle with prior knowledge about all invocations achieves only 26% higher speedup and 36% better energy efficiency. Compared to the neural design, the oracle achieves 26% and 19% higher performance and efficiency benefits, respectively. These results suggest that MITHRA makes an effective stride in providing hardware-software solutions for controlling quality tradeoffs. Such solutions are imperative in making approximate acceleration widely applicable.

II. Challenges and Overview

Approximate accelerators trade small losses in output quality for significant performance and efficiency gains [16]–[23]. When a processor core is augmented with an approximate accelerator, the core delegates the computation of frequently executed safe-to-approximate functions to the accelerator. A safe-to-approximate function is a region of code that can tolerate imprecise execution. Instead of executing the function, the core sends the function's inputs to the accelerator and retrieves its outputs. The outputs from the accelerator are an approximation of the outputs that the core would have calculated by executing the original function. The configuration of the accelerator is generated by the compiler.

The accelerator is always invoked in lieu of the original function. Always invoking the accelerator leads to a *fixed* tradeoff between quality and gains in performance and efficiency. The lack of flexibility in controlling this tradeoff limits the applicability of approximate acceleration. Therefore, we devise MITHRA, a hardware-software solution that can effectively control the quality tradeoffs.

A. Challenges and Insights

MITHRA's objective is to provide flexibility in controlling final quality loss and to maximize the performance and energy benefits at any level of quality. MITHRA aims to only filter out those approximate accelerator invocations that cause relatively large quality degradation in the final output. To devise such a solution, we analyze the properties and challenges of a system augmented with an approximate accelerator. Below we discuss the insights that guide the design of MITHRA.

1) What accelerator characteristics can guide the design of MITHRA? We investigate the error distribution in the output of different applications when they undergo approximate acceleration [16]. Figure 1 shows the cumulative distribution function (CDF) plot of final error for each element of the applications output. The application output is a collection of elements e.g. image consists of pixels, vector consists of scalars, etc. As illustrated in the Figure 1, only a small fraction (0%-20%) of these elements see large errors. The main insight is that there is an opportunity to attain significant gains with approximate acceleration while reducing the quality loss. To exploit this opportunity, MITHRA filters out these small fraction of accelerator inputs that lead to relatively large errors.

2) What information is needed and is available to segregate accelerator inputs that cause large error? For each invocation, the core sends a set of inputs (the input vector) to the accelerator and retrieves a set of outputs (the output vector). The input vector here refers to the accelerator input and not the application input. The accelerator output is the function of its input vector and its configuration. The difference between the imprecise accelerator output and the precise output is the accelerator error. Therefore, accelerator error becomes a function of the input vector and the accelerator configuration. For a given application, the accelerator configuration is fixed; making the accelerator error only a function of the input vector. Accelerator inputs provide enough information to determine whether or not a specific accelerator invocation will lead to a relatively large error. This insight simplifies the design of MITHRA, enabling it to use only information that is local to each invocation.

3) How to map the final output quality loss as a local accelerator error? MITHRA uses the accelerator inputs to classify accelerator invocations that cause large error. Therefore, MITHRA makes local decisions based on the accelerator input without a global view of execution and without any knowledge about the manifestation of the accelerator error on the final output. The main challenge is to devise a technique that can find an upper bound on the accelerator error to guide MITHRA's local decisions while considering the final output. To address these challenges, we develop a statistical optimization solver that finds an optimal threshold for the accelerator error. This threshold maximizes accelerator invocations and gains from approximation while providing statistical guarantees that the final quality loss level will be met with high confidence. This mechanism tries to keep the accelerator error below a certain threshold for each invocation. MITHRA considers the local error large if it speculates that any element in the accelerator output vector might have an error larger than the threshold. The threshold forms the knob for controlling the quality tradeoffs.

4) What guarantees are provided for the final output quality loss? The final output quality is mapped onto the local accelerator site as the threshold for the accelerator error. The final quality loss is provided by the programmer requiring either formal or statistical guarantees. In general, providing formal guarantees that the quality requirements will be met on all possible application input datasets is intractable due to the complex behavior of programs and large space of possible inputs. Statistical approach is the most viable alternative to validate quality control techniques.

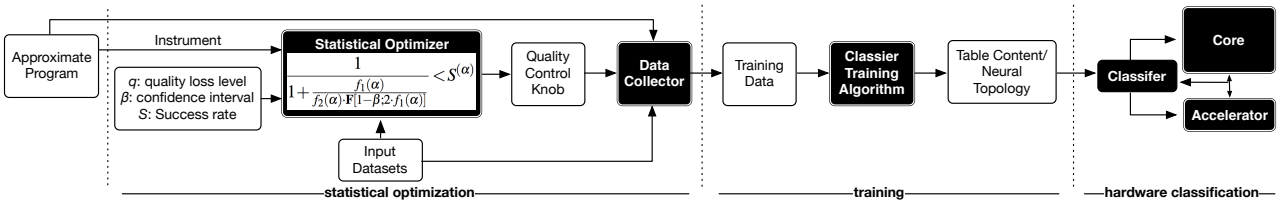


Figure 2: Overview of MITHRA comprising a statistical optimizer, a trainer, and a hardware classifier. The statistical optimizer uses instrumented approximate program and input datasets to tune the quality control knob such that it satisfies a desired quality loss (q) with high confidence (β) and success rate (S). The knob is used to generate the training data for the classifiers. The classifiers operate at runtime to control the quality tradeoffs.

Therefore, we incorporate the Clopper-Pearson exact method to provide statistical guarantees that the desired level of quality loss will be met with high confidence on unseen datasets.

5) What needs to be done for MITHRA in hardware? MITHRA's objective is to identify accelerator invocations that lead to a relatively large quality loss or an error above the threshold. Measuring the error requires running both the original precise code and invoking the accelerator, which will nullify the benefits of acceleration. MITHRA makes this decision without measuring the actual error of the accelerator. This decision making can be accomplished by using a classification algorithm that maps the accelerator inputs to a binary decision. The main challenge is to devise classifiers that can be efficiently implemented in hardware and make quality tradeoff decisions at runtime.

B. Overview

MITHRA's framework, shown in Figure 2, comprises (1) a compiler constituting statistical optimizer and pre-trainer, and (2) a hardware classifier which is a microarchitectural mechanism that resides between the core and the accelerator. The hardware classifier uses the accelerator inputs to make a binary decision of either running the original function on the core or invoking the accelerator. In this paper, we propose and explore two classification algorithms that can be implemented as the microarchitectural instances of MITHRA. The first algorithm is a novel table-based mechanism that efficiently hashes the accelerator input vector to retrieve the decision from a small ensemble of tables. The second technique is a multi-layer perceptron based neural mechanism. Section IV describes these two hardware classifiers.

The hardware classifiers need to be trained in order to make decisions at runtime. The compilation component of MITHRA trains these classifiers to detect accelerator inputs that might produce accelerator error greater than a threshold. This threshold is the quality control knob that is tightened or loosened in accordance to the desired level of final output quality. Optimal threshold is obtained by solving a statistical optimization problem that maximizes accelerator invocation rate and benefits from approximate acceleration for a set of representative application input datasets while keeping the quality loss below a desired level. The optimization provides statistical guarantees that final quality loss will be met with high confidence and success rate. The thresholding mechanism and pre-training is described in Section III.

III. STATISTICAL OPTIMIZATION FOR CONTROLLING QUALITY TRADEOFFS

Approximate accelerators require the programmer to provide an application-specific quality metric and a set of representative input datasets [16]–[23]. MITHRA uses the same information to automatically train the classifiers. This training process constitutes two phases. The first phase, referred to as the *thresholding phase* utilizes profiling information to find a threshold for the accelerator error. This phase converts the global quality loss into a local accelerator error threshold by solving a statistical optimization problem.

The second phase or the *training phase*, generates training information for the hardware classifiers based on the threshold found in the first phase (Section III-B). This training information is incorporated in the accelerator configuration and is loaded in the classifiers when the program is loaded to the memory for execution. This strategy is commensurate with previous works on acceleration (precise or approximate) that generate the accelerator configuration at compilation time and encode it in the binary [6], [7], [16]–[18]. The configurations of both the accelerator and MITHRA are part of the architectural state. Therefore, the operating system must save and restore the configuration data for both the accelerator and MITHRA on a context switch. To reduce context switch overheads, the OS can use the same lazy context switch techniques that are typically used with floating point units [24].

A. Finding the Threshold

The objective of this phase is to tune the quality control knob i.e. find a threshold for the accelerator error. The threshold enables MITHRA to maximize the accelerator invocations for any level of quality loss. The optimized threshold is an upper bound on the error that can be tolerated by the target function from the accelerator to maintain the desired final output quality. To allow an accelerator invocation, the error of each element of the output vector should be below the threshold (th), as shown in Equation 1.

$$\forall o_i \in OutputVector \quad |o_i(precise) - o_i(approximate)| \leq th \qquad (1)$$

Algorithm 1 shows the iterative procedure to find this optimized threshold. In this process, for each intermediate threshold (th), the program (P) can be instrumented to find the final quality loss (q_i) for a set of representative application input datasets (i). Each final quality loss level (q_i) is compared with the desired final quality loss(q). Due to the complex behavior of programs and the large space of possible datasets, some application inputs might fall within the desired final quality loss, while the others might not. Hence, providing formal guarantees that quality requirements will be met on all possible application inputs is intractable. Statistical

approaches are the most viable solutions to validate quality control techniques. Therefore, MITHRA provides statistical guarantees that quality requirements will be met on unseen datasets with high confidence. To provide such guarantees, the algorithm counts the application input datasets that have final output quality (q_i) below or equal to the desired quality loss (q) as shown in Equation 2.

$$\forall i \in inputSet \quad if(q_i(P_i(th)) \leq q) \quad n_{success} = n_{success} + 1 \qquad (2)$$

The number of application outputs that have desired quality loss ($n_{success}$) varies with the threshold (th). For instance, as the threshold is made tighter the number of $n_{success}$ will increase as the output quality loss of each application input (q_i) will decrease. We utilize the $n_{success}$ to compute the binomial proportion confidence interval and success rate using Clopper-Pearson exact method [25] as described below.

Clopper-Pearson exact method. As Equation 3 shows, the Clopper-Pearson exact method computes the one-sided confidence interval of success rate $S^{(q)}$, when the number of application inputs or sample trials, n_{trials}, and the number of successes among the trials, $n_{success}$, are measured for a sample of the population. In Equation 3, F is the F-critical value that is calculated based on the F-distribution [26].

$$\frac{1}{1 + \frac{n_{trials} - n_{success} + 1}{n_{success} \cdot \mathbf{F}[1-\beta; 2 \cdot n_{success}, 2 \cdot (n_{trials} - n_{success} + 1)]}} < S^{(q)} \qquad (3)$$

To further understand Equation 3 and the $S^{(q)}$, we discuss a simple example in which, 90 ($n_{success}$) out of the total 100 (n_{trials}) representative application input datasets generate outputs that have a final quality loss $\leq$ 2.5%. In this example, the lower limit of the 95% confidence interval ($S^{97.5\%}$) is 80.7%. This implies that with 95% confidence we can project that at least 80.7% of unseen input sets will produce outputs that have quality loss level within 2.5%. This projection is conservative because the Clopper-Pearson exact method calculates a conservative lower bound for the confidence interval. The degree of confidence (β) is the probability of the projection being true. The projection based on 95% confidence interval is true with probability of 0.95. The statistical optimization algorithm incorporates this Clopper-Pearson exact method. The optimization iteratively searches for an optimal threshold that maximizes accelerator invocations while providing high confidence that final quality loss will be met on unseen datasets.

The inputs to the algorithm are the program code (P), set of representative input datasets (ρ), quality degradation when accelerator is always invoked ($\mathscr{D}$), desired quality loss level (q), confidence interval (β) and desired success rate (S). The algorithm goes through following steps:

(1) Initialize. Assign a random value to the threshold.

(2) Instrument. Instrument the program to execute both the original function and the accelerator for all invocations of the target function. For each invocation, use the original precise result if the accelerator error exceeds the threshold.

(3) Measure the quality. Run the instrumented program for each application input and measure the final output quality.

(4) Measure the success rate. Calculate the number of inputs that have the final output quality within the desired level (q). Use this number and confidence interval to calculate the success rate (θ) with the Clopper-Pearson exact method.

(5) Adjust the threshold. If the success rate (θ) is less than S, decrease the threshold by a small delta. If the success rate (θ) is greater than S, increase the threshold by a small delta.

(6) Reiterate or terminate. Terminate if success rate (θ) with the last threshold is greater than S and with the current threshold is less than S. Otherwise, go to step (2).

As Section V elaborates, we use a different set of input datasets to validate the selection of the threshold. If the application offloads multiple functions to the accelerator, this algorithm can be extended to greedily find a tuple of thresholds. Due to the complexity of application behavior, this greedy approach will find suboptimal thresholds if the number of offloaded functions increases. After finding the threshold, the compiler profiles application input datasets to generate the training data for the classifiers.

```
Input  : P: Program
         ρ: Input data sets
         𝒟: Quality loss with 100% accelerator invocation
         q: Desired quality loss level
         β: Confidence interval
         S: Desired success rate
Function SuccessRate (q, P, ρ, β, th)
    Initialize num ← 0
    for (∀ ρ_i in ρ) do
        P_i = Instrument (P)
        error = RunMeasureQuality (P_i, ρ_i, th)
        if (error < q) then
            num = num + 1;
    end
    ClopperPearson (num, β, ρ)
    return num
end
Initialize th ← th_0
θ = SuccessRate (q, P, ρ, β, th)
terminate = false
while (terminate == false) do
    if (θ < S) then
        th = th - Δ
    else if (θ > S) then
        th_last = th
        th = th + Δ
    θ_last = θ
    θ = SuccessRate (q, P, ρ, β, th)
    if (θ < S and θ_last > S) then
        terminate = true
        return th_last
end
```

Algorithm 1: Finding the threshold.

B. Training Data for Hardware Classifiers

Once the threshold is determined, hardware classifiers can be pre-trained using representative application inputs. Generating training data requires running the application and randomly sampling the accelerator error. For the sampled invocations, if the accelerator error for all elements in the output vector are less than the threshold, the corresponding input is mapped to invoke the accelerator (binary decision '0'). Conversely, the input vector is mapped to trigger the execution of the original function (binary decision '1') if the accelerator error is greater than the threshold. The training data is a collection of tuples. Each tuple contains an input vector and a binary value. The binary value is 0 if the accelerator error is larger than the threshold and 1 otherwise. For a given input

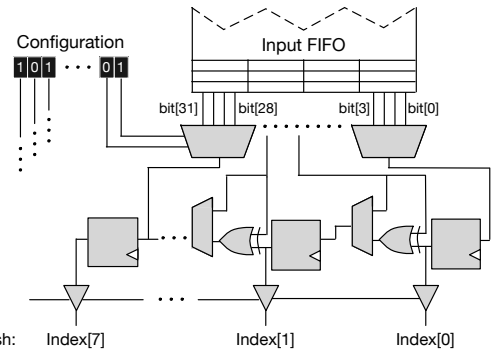


Figure 3: A reconfigurable hash function. Each hash function takes an input vector and generates the index. All the hashes are MISRs but the configuration register decides the input bits they use.

vector, this binary value is the function of accelerator configuration and the threshold. Both the configuration and the threshold are constant for a fixed application and final quality loss. Therefore, a set of representative accelerator input vectors is sufficient to generate the training data. In many cases, a small number of application input datasets is sufficient to generate this training data for classifiers because the target function is hot and is executed frequently in a single application run. For instance, in an image processing application, the target function runs for every pixel in the input image. Even a single 512×512 input image provides 262,144 training data points. The generated training data is agnostic to the type of hardware classifier that needs to be trained. However, the training process depends on classifier. In this paper, we focus on a table-based and neural network based classifier. These classifiers and their training process is detailed in the next section.

IV. Designing Hardware Classifiers for MITHRA

This section provides details about how hardware classifiers of MITHRA are designed to be deployed at runtime. MITHRA's microarchitectural component is a hardware classifier that maps an accelerator input vector with multiple elements to a single-bit binary decision. This binary decision determines whether MITHRA would invoke the accelerator or run the original precise function. This section defines and explores two hardware classifiers for MITHRA, one table-based and one based on neural networks. The table-based classifier mostly utilizes storage for decision making, whereas the neural classifier relies on arithmetic operations.

A. Table-Based Classifier Design

We devise a novel table-based classifier that stores its decisions (single-bit values) in a table, which are indexed by a hash over the elements of the accelerator input vector. We design an efficient circuit to hash the input elements and generate the index aiming to minimize aliasing. Below, we first discuss the hash function and then describe a multi-table design that leverages a small ensemble of tables to achieve better accuracy with limited storage.

1) Generating Index from Multiple Inputs: For the table-based design, the hash function should (1) be able to combine all the elements in the input vector, (2) be able to reduce destructive aliasing as much as possible, (3) be efficiently implementable in hardware, (4) be able to accept a varying number of inputs, and, (5) be reconfigurable to work across different applications. To efficiently satisfy these requirements, we use a hardware structure called Multi-Input Signature Register (MISR) [27] to hash the input elements and generate the table index. A MISR takes in a bit-vector and uses a set of XOR gates to combine the incoming bit-vector with the content of a shift register. Figure 3 illustrates an instance of our MISR hash function. In a MISR, the result of the XORs is stored in a register after a shift operation. As the next input comes in, the MISR repeats the previous step of combining the inputs together. After all the elements of the input vector for a single invocation are processed, the value that remains in the register is the index.

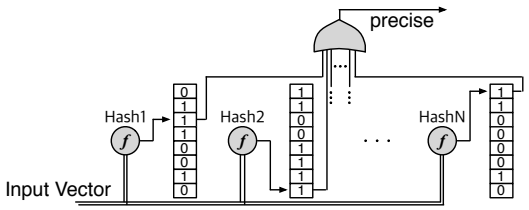


Figure 4: Multi-table based Classifier. All tables are equally sized but each table is indexed with a different MISR or hash configuration.

Number of elements in the accelerator input vector vary across applications. Therefore, the hash function should be able to accept a varying number of inputs. MISRs can naturally combine arbitrary number of input elements. As Figure 3 illustrates, we designed a reconfigurable MISR that supports different combinations of XOR, bit selection, and shift operations to allow the table-based classifier to adapt to the needs of the application. This configuration is decided at compile time for each application and is fixed during execution. The fixed hashing for each application makes the indexing completely deterministic.

One of the challenges with using MISRs to index the tables is that its content changes as the input elements arrive. These transient changes in the index of the tables can cause high energy consumption. Therefore, we connect the MISRs through a series of tri-state gates to the tables. The tri-state gates are inactive until the arrival of the last input element, to prevent the transient state of the MISRs from affecting the table. A counter counts the number of received input elements and activates the tri-state gates when the last element arrives. We send accelerator inputs to both the accelerator and the classifier simultaneously assuming that in most cases the classifier will decide to invoke the accelerator. This strategy is in line with the earlier insight that only a small fraction of the invocations require invoking the original precise code.

2) Multi-Table Classifier: A single-table requires a large number of entries to maximize the benefits of approximate acceleration while reducing quality loss. The reason being that only a small fraction of the input combinations need to be filtered out from accelerator invocation. This characteristic makes it harder for only one small table to segregate inputs that cause relatively large errors because of destructive aliasing. When aliasing occurs and multiple inputs collide on the same entry, the bias is always toward invoking the accelerator. This bias may impair the ability of the smaller tables to distinguish the inputs, thereby causing relatively large quality losses.

To address this issue, we devise a multi-table classifier illustrated in Figure 4. The design consists of multiple equally-

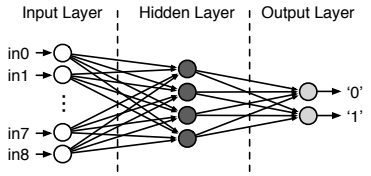


Figure 5: The neural classifier takes in accelerator inputs and generates two outputs. The output neuron with the larger value is the final outcome.

sized tables and each entry in the table is a single-bit value. The hash function for each table is a different MISR configuration. These configurations are selected from a pool of 16 fixed MISR configurations that exhibit least similarity i.e. they map same input to different table indices. This configuration pool is independent of the application. The compiler assigns the first table with the MISR configuration that incurs least aliasing. The second table is assigned a different MISR that has least amount of aliasing and the combination of the two tables provides least false decisions. The compiler repeats this step for the third, fourth, etc., tables. We developed this greedy algorithm since the decision space is exponentially large. Using different hash functions for each table lowers the probability of destructive aliasing in the tables. As the input elements arrive, all the MISRs generate indices in parallel and the corresponding values are read from the tables. Since the bias in each single table is toward invoking the accelerator, MITHRA directs the core to run the original function even if a single table determines that the precise code should be executed. Therefore, the logic for combining the result of the tables is just an OR gate. The multi-table design is similar to Boosting in machine learning that combines an ensemble of weak learners to build a powerful classifier.

B. Neural Classifier Design

We also explore the use of neural networks to control the quality tradeoffs. While the table-based design utilizes storage for controlling the quality tradeoffs, the neural design leverages arithmetic operations for the same task. The neural classifier spends some of the gains achieved in performance and efficiency to obtain a higher quality in the results.

We use multi-layer perceptrons (MLPs) due to their broad applicability. An MLP consists of a fully-connected set of neurons organized into layers: the input layer, any number of hidden layers, and the output layer (Figure 5). A larger, more complex network offers greater accuracy, but is likely to be slower and dissipate more energy. To strike a balance between accuracy and efficiency, we limit neural design to three layer networks comprising one input layer, one hidden layer, and one output layer. Furthermore, we only consider neural networks with 2, 4, 8, 16, and 32 neurons in the hidden layer even though more neurons-per-layer are possible. The neural classifier takes in the same number of inputs as the accelerator and always contains two neurons in the output layer. One neuron represents the output '0' and the other represent the output '1'. The output neuron with the larger value determines the final decision. We train [28] these five topologies and choose the one that provides the highest accuracy with the fewest neurons. During an invocation, as the core sends the input elements to neural network, which is executed on a specialized hardware and decides whether or not to invoke the accelerator. The next subsection describes how these hardware classifiers are trained using the threshold specific training data.

C. Training the Classifiers

Table-based classifier is pre-trained offline. At runtime the table-based design is updated as we discuss later in this section. Updating the neural design online requires more computation power and may incur significant overheads. Therefore, for the neural design we follow the same workflow as the neural processing units (NPUs) [16]–[18], [22], [23] and train the neural network offline.

1) Training the Table-Based Classifier: The offline training of the table-based design initially sets all the table entries to '0' enabling a 100% accelerator invocation. For each training tuple, we calculate the hash for each accelerator input vector to identify its corresponding table entry. If a particular accelerator input vector leads to an error larger than the threshold, the corresponding table entry is set to '1'. In the case of aliasing, the table entry will be set to '1' even if only one of the aliased inputs results in an error larger than the threshold. This training strategy is conservative and avoids the bias towards invoking the accelerator since most of the accelerator inputs lead to small errors. The same procedure is extrapolated to train the ensemble of tables. After pre-training, we compress the content of these tables using the Base-Delta-Immediate compression algorithm [29] and encode the compressed values in the binary.

Online training for the table-based design. After deploying the pre-trained table-based design, we use the runtime information to further improve its accuracy. We sample the accelerator error by running both the original precise code and the accelerator at sporadic intervals. After sampling the error, the table entry is updated according to the same procedure used in pre-training. In addition to generating the hash and updating the table entries, the online table update requires a few arithmetic operations to calculate the error and compare it with the threshold.

2) Training the Neural Network Design: Similar to the prior works [16]–[18], [22], [23], we use offline training the neural classifier. An alternative design could train the neural design concurrently with in-vivo operation. Online training could improve accuracy but would result in runtime overheads. To mitigate these overheads, an online training system could offload neural training to a remote server on the cloud.

D. Instruction Set Architecture Support

We add a special branch instruction to the ISA that invokes the original code if MITHRA decides to fall back to the precise code. Hence, the branch is taken if the hardware classifier speculates that the original precise function should be executed. This special branch instruction is inserted after the instructions that send the inputs to the accelerator. The overhead of this instruction is modeled in our evaluations.

V. EVALUATION

A. Experimental Setup

Cycle-accurate simulation. We use the MARSSx86 x86-64 cycle-accurate simulator [30] to measure the performance of the accelerated system augmented with MITHRA. The processor is modeled

Table I: Benchmarks, their quality metric, input data sets, and the initial quality loss when the accelerator is invoked all the time.

Benchmark	Description	Type	Application Error Metric	Input Data	Compilation Dataset	Validation Dataset	NPU Topology	Error with Full Approximation
blackscholes	Math model of a financial market	Financial Analysis	Avg. Relative Error	4096 Data Point from PARSEC Suite	250 Distinct	250 Distinct	6->8->8->1	6.03%
fft	Radix-2 Cooley-Tukey fast fourier	Signal Processing	Avg. Relative Error	2048 Floating Point Numbers	250 Distinct	250 Distinct	1->4->4->2	7.22%
inversek2j	Inverse kinematics for 2-joint arm	Robotics	Avg. Relative Error	10000 (x, y) Coordinates	250 Distinct	250 Distinct	2->8->2	7.50%
jmeint	Triangle intersection detection	3D Gaming	Miss Rate	10000 Pairs of 3D Triangle Coordinates	250 Distinct	250 Distinct	18->32->8->2	17.69%
jpeg	JPEG encoding	Compression	Image Diff	512x512-Pixel Color Image	250 Distinct	250 Distinct	64->16->64	7.00%
sobel	Sobel edge detector	Image Processing	Image Diff	512x512-Pixel Color Image	250 Distinct	250 Distinct	9->8->1	9.96%

after a single-core Intel Nehalem to evaluate the performance benefits over an aggressive out-of-order architecture[1]. We use NPU [16] as the approximate accelerator to evaluate MITHRA. The NPU consists of eight processing elements that expose three queues to the processor to communicate inputs, outputs, and configurations. The simulator is modified to include ISA-level support for the NPU[2]. This support consists of two enqueue and two dequeue instructions and a special branch instruction for executing the original function. The processor uses the same architectural interface and FIFOs to communicate the configuration of classifiers. Classifiers receive the inputs as the processor enqueues them in the accelerator FIFO. We use GCC v4.7.3 with -O3 to enable compiler optimization. The baseline in our experiments is the benchmark run solely on the processor with no approximate acceleration.

We augmented MARSSx86 with a cycle-accurate NPU simulator that also models the overheads of hardware classifiers. For the table-based mechanism, these overheads include, cycles to decompress the content, generate the indices, index into the table, and finally generate the decision. We use the NPU to execute the neural design which adds extra cycles and energy to the overall system.

Energy modeling. We use McPAT [31] for processor energy estimations. We model the NPU energy using results from McPAT, CACTI 6.5 [32], and [33]. The cycle-accurate NPU simulator provides detailed statistics that we use to estimate the energy of both the accelerator and the neural classifier. For estimating the energy of the table-based design, we implement the MISRs in Verilog and synthesize them using Synopsys Design Compiler (G-2012.06-SP5). We use Synopsys PrimeTime (F-2011.06-SP3-2) to measure the energy cost of the MISRs after synthesis. The synthesis library is the NanGate 45 nm Open Cell Library–an open source standard cell library. We also use the same synthesis procedure to measure the cost of arithmetic operations that are required to decompress the content in each table. We use CACTI 6.5 to measure the energy cost of accessing the tables. The processor, hardware classifier, and the accelerator operate at 2080 MHz at 0.9 V and are modeled at 45 nm technology node. These settings are in line with the energy results in [33] and [16].

[1] **Processor:** Fetch/Issue Width: 4/6, INT-ALUs/FPUs: 3/2, Load/Store FUs: 2/2, ROB Size: 128, Issue Queue Size: 36, INT/FP Physical Registers: 256/256, Branch Predictor: Tournament 48KB, BTB Sets/Ways: 1024/4, RAS Entries: 64, Load/Store Queue Entries: 48/48, Dependence Predictor: 4096 Bloom Filter, ITLB/DTLB Entries: 128/256 **L1:** 32KB Instruction, 32KB Data, Line Width: 64bytes, 8-Way, Latency: 3 cycles **L2:** 2MB, Line Width: 64bytes, 8-Way, Latency: 12 cycles **Memory Latency:** 50 ns

[2] **NPU**: Number of PEs: 8, Bus Schedule FIFO: 512x20-bit, Input FIFO: 128x32-bit, Output FIFO: 128x32-bit, Config FIFO: 8x32-bit **NPU PE**: Weight Cache: 512x33-bit, Input FIFO: 8x32-bit, Output Reg File: 8x32-bit, Sigmoid LUT: 2048x32-bit, Multiply-Add Unit: 32-bit Single-Precision FP

Table II: Size of compressed table-based and neural classifiers.

Benchmark	Size of Table-based Design after Compression (KB)	Neural-base Design	
		Size (KB)	Neural Topology
blackscholes	0.25	0.57	6->4->2
fft	0.25	0.10	1->4->2
inversek2j	0.29	0.10	2->4->2
jmeint	0.25	1.47	18->16->2
jpeg	3.70	0.79	64->2->2
sobel	3.20	0.22	9->4->2

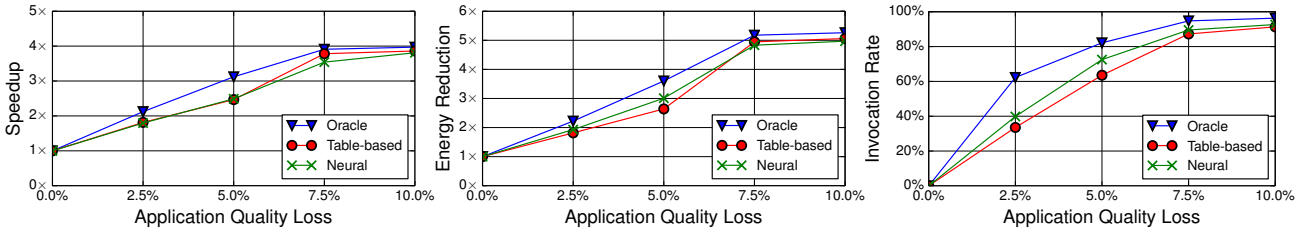
Classifier configurations. For the main results in this section, we use a table-based design that consists of eight tables, each of size 0.5 KB. This design is the result of our Pareto analysis, presented in Section V-B2. The topology of the neural classifier varies across benchmarks (Table II).

Benchmarks. We use AxBench, a publicly available benchmark suite (http://www.axbench.org) that is used in [16], [17]. These benchmarks come with NPU topology and we use them without making any NPU-specific optimizations for utilizing MITHRA. These benchmarks represent a diverse set of application domains, including financial analysis, signal processing, robotics, 3D gaming, compression, and image processing. Table I summarizes each benchmark's application domain, input data, NPU topology, and final application error levels when the accelerator is invoked for all inputs without MITHRA. We use each benchmark's application-specific error metric to evaluate MITHRA. The initial error with no quality control and full approximation ranges from 6.03% to 17.69%. This relatively low initial error makes the quality control more challenging and the diversity of the application error behavior provides an appropriate ground for understanding the tradeoffs in controlling quality with MITHRA.

Input datasets. We use 250 distinct datasets during compilation to find the threshold and train MITHRA. We use 250 different unseen datasets for validation and final evaluations that are reported in this section. Each dataset is a separate typical program input, such as a complete image (see Table I).

B. Experimental Results

In this paper, we devise an optimized hardware for the table-based and neural based classifiers, and provide the necessary microarchitectural support. The table-based and neural based designs can also be implemented in software. To justify the hardware implementation of these classifiers we implement these algorithms in software and measure the corresponding application



(a) Speedup vs Application Quality Loss (b) Energy Reduction vs Application Quality Loss (c) Invocation Rate vs Application Quality Loss

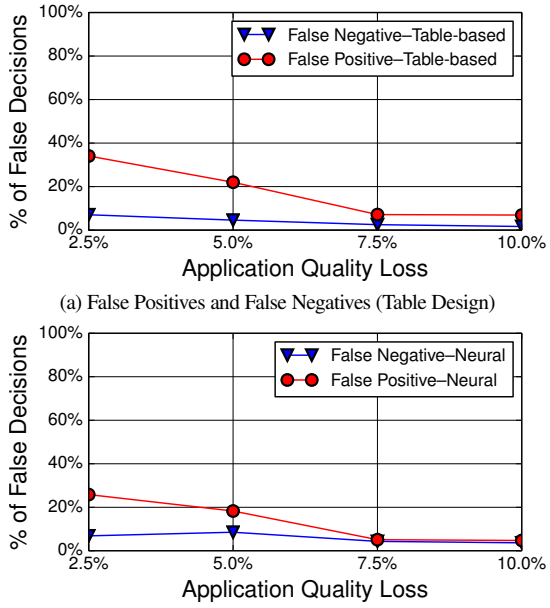
Figure 6: We compare the mean (a) speedup, (b) energy reduction and (c) invocation rate across all the benchmarks for the oracle, table-based and neural designs for varying levels of final application output quality loss for 95% confidence interval and 90% success rate.

runtimes. The software implementation of the table-based and neural classifiers slow the average execution time by 2.9× and 9.6×, respectively. These results confirm the necessity of a co-designed hardware-software solution for quality control.

1) Controlling Quality Tradeoffs with MITHRA*:* The primary goal of MITHRA is to control quality tradeoffs while preserving maximum benefits from approximate acceleration. We build an *ideal oracle design* as a gold standard to measure the efficacy of our realistic designs At any level of quality loss, the oracle *always* achieves the maximum performance and energy benefits by only filtering out the invocations that produce an accelerator error larger than the threshold. Therefore, MITHRA's objective can be redefined as a design that closely mimics the achievements of the oracle in delivering speedup and energy reduction.

Performance and energy benefits. Figure 6a and Figure 6b shows the speedup and energy reduction when the quality tradeoffs are controlled by the oracle, the table-based design, and the neural design. These speedups and energy reductions are the *geometric mean* across all the benchmarks. We present the per-benchmark trends in Figure 8, and discuss them below. All the numbers in Figures 6b, 6a are presented for 90% success rate and 95% confidence interval. This result implies that with 95% confidence, we can project that at least 90% of unseen input sets will produce outputs that have quality loss level within the desired level. To obtain these results, 235 (out of 250) of the test input sets produced outputs that had the desired quality loss level. As expected, the oracle delivers the highest benefits. The table-based design and the neural design both closely follow the oracle. These results show the efficacy of both our designs in controlling the quality tradeoffs. With 5% final output quality loss, the table-based design provides 2.5× average speedup and 2.6× average energy reduction. In comparison to the table-based design, the neural design yields similar performance benefits while providing 13% more energy gains. Compared to the table-based design, the oracle achieves 26% more performance and 36% more energy benefits. Similarly, compared to the neural design, the oracle delivers 26% more speedup and 19% more energy reduction. These results suggest that both classifier designs can effectively control the tradeoff between quality and the gains from approximate acceleration.

Accelerator invocation rate. To better understand the trends in performance and energy reduction, we examine the accelerator invocation rate with MITHRA in Figure 6c. The invocation rate is the percentage of target function invocations that are delegated



(a) False Positives and False Negatives (Table Design)

(b) False Positives and False Negatives (Neural Design)

Figure 7: False positive and false negative decisions for (a) table-based and (b) neural classifier for varying quality losses at 95% confidence interval and 90% success rate.

to the accelerator. When the invocation rate is 100%, the target function is always executed on the accelerator. When the invocation rate is 0%, the function always runs on the precise core. Gains from approximate acceleration are directly proportional to invocation rate and MITHRA aims to maximize these gains for any level of quality loss. Figure 6c shows the invocation rate when the quality tradeoffs are controlled by the oracle, the table-based, and the neural design. As expected, the oracle provides the highest invocation rate due to its prior knowledge about all the invocations. The table-based and the neural designs obtain invocation rates that closely follow the oracle. As the quality loss level tightens, the invocation rate declines for all the three designs. For 5% quality loss level, the table-based design achieves 64% and the neural design achieves 73% average invocation rate. The oracle is only 29% and 13% higher than the table-based and neural designs, respectively. Even though the table-based design shows a lower invocation rate than the neural design, the performance achieved by both the designs are similar since the neural design generally incurs a higher cost for generating the result.

Per-benchmark analysis. Figure 8 illustrates the speedup, energy

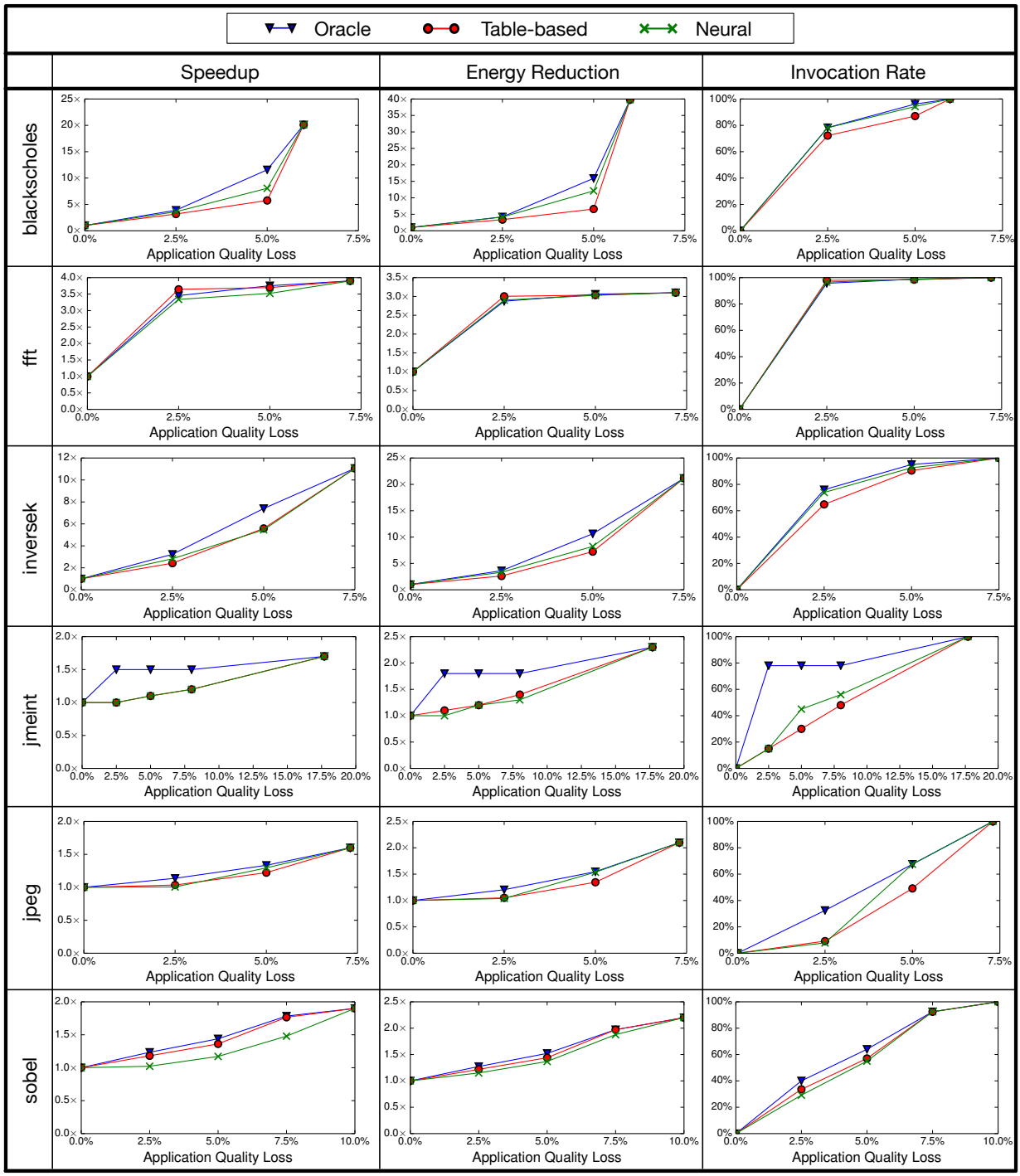
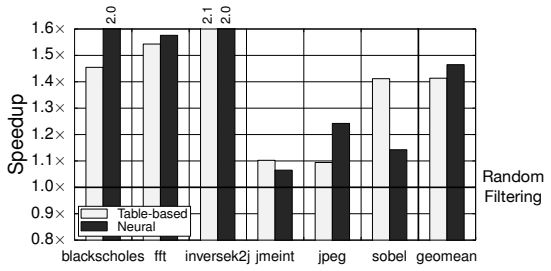
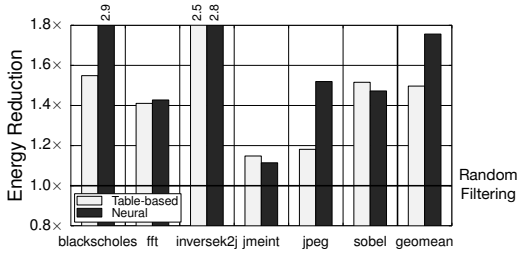


Figure 8: Speedup, energy reduction, and invocation rate for individual benchmarks at 95% confidence interval and 90% success rate.

reduction, and invocation rate for each individual benchmark with 95% confidence and 90% success rate with varying desired quality levels. Similar to the mean results, the majority of benchmarks closely follow the oracle for both the designs. Two benchmarks, jmeint and jpeg, reveal interesting trends. In both cases, the neural design significantly outperforms the table-based design in terms of invocation rate. This phenomenon is the result of large number of elements in the accelerator input vector (64 inputs for jpeg and 18 inputs for jmeint). This leads to high hash conflicts and hence the table-based design is less effective in segregating inputs that incur large quality losses. Therefore, it conservatively falls back to the original precise code to achieve better quality. Another observation is that even though jmeint achieves higher invocation rate with the neural design, the gains from approximation are similar to the table-based design. The neural design for jmeint requires a relatively large neural network with 18 input, 16 hidden, and 2 output



(a) Speedup with MITHRA (Baseline: Random Filtering)



(b) Energy Reduction with MITHRA (Baseline: Random Filtering)

Figure 9: (a) Speedup and (b) energy reduction for 95% confidence interval and 90% success rate compared to random filtering at 5% quality loss. The baseline is approximate acceleration with random filtering.

neurons that outweighs the benefits of higher invocation rate.

False positives and false negatives. To further understand the operation of our classifiers, we examine their false decisions. Figure 7a and 7b shows the percentage of these false decisions (positive and negative) for the table-based and the neural design, respectively. The false positives comprise the input cases that should have been run on the accelerator according to the oracle, but are identified as potential high-error cases by classifiers and are executed using the original precise function. Conversely, the false negatives comprise those input cases, which should have been run using the original function according to the oracle but are missed by classifiers and are run on the accelerator. Figure 7 shows the ratio of the false decisions to the total invocations averaged over all benchmarks. With 5% quality loss, the table-based design makes 22% false positive and 5% false negative decisions. The neural design makes 18% false positive and 9% false negative decisions. The low rate of false negatives demonstrates the high efficacy of both designs in filtering out inputs that lead to large quality degradations. Moreover, the false negatives are significantly lower than the false positives with both designs because hardware classifiers adopt a conservative approach towards controlling quality trade-offs; hence, prioritizing quality over benefits from approximation.

Comparison with random filtering. We compare our input-conscious techniques to a simple random filtering technique. In this technique, the decision to delegate a function invocation to the accelerator is random, irrespective of the inputs. Figure 9 shows the speedup and energy reduction with both of our techniques relative to the random filtering at 5% quality loss. The trends are similar for other quality levels. Compared to random filtering, the table-based design delivers 41% average speedup and 50% average energy reduction. With the neural design, these figures increase to 46% average speedup and 76% average energy reduction. The speedup is as high as 2.1× (inversek2j with the table-based design) and the maximum energy reduction grows to 2.9× reduction (blackscholes with the neural design). These results collectively confirm the importance of focusing and capturing the inputs that lead to large quality losses.

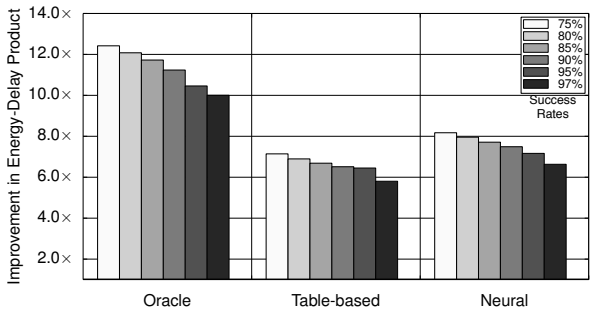


Figure 10: Trends in the Energy-Delay product for varying success rate with 95% confidence interval and at 5% quality loss level.

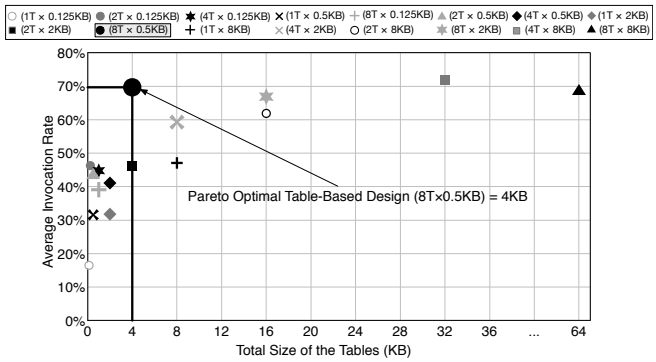


Figure 11: Pareto analysis for the table-based MITHRA at 5% quality loss. The (aT×bKB) is a configuration with a parallel tables each of size b KB. Our default configuration, (8T×0.5KB), is Pareto optimal.

Varying success rate with 95% confidence. MITHRA aims to provide statistical guarantees with high confidence that quality levels will be met. The main results focus on a confidence interval of 95% and attain 90% success rate. As the Clopper-Pearson method in Section III describes, for different confidence intervals the success rate is dependent on $n_{success}$ which in turn is dependent on the threshold selected. For 5% quality loss level, we vary the threshold so as to obtain a sweep of success rates for 95% confidence interval. Figure 10 presents the improvement in energy-delay product for these success rates. As the success rate increases, implying that there is a higher probability that quality levels will be met, the benefits from approximation decrease. Higher success rate provides higher statistical guarantee and therefore comes at a higher price. The results show the MITHRA effectively enables the programmer to control both the level of quality loss and also the degree of statistical guarantee they desire.

2) Pareto Analysis for the Table-Based Classifier: The two main design parameters of the table-based design are the number of parallel tables and the size of each table. We vary these two parameters to explore the design space of table-based MITHRA and perform Pareto analysis to find the optimal configuration. Figure 11 illustrates the Pareto analysis for 5% quality loss. The trends are similar with other levels of quality loss. The design that is denoted by (aT×bKB) represents a configuration with a parallel tables, each of size size b kilo bytes. We explore 16

different designs, a set of all the combinations of (1T, 2T, 4T, 8T) parallel tables and (0.125KB, 0.5KB, 2KB, 4KB) table sizes. The *x*-axis in Figure 11 captures uncompressed size in kilo bytes. The *y*-axis is the average accelerator invocation rate across all benchmarks. We use the average invocation rate because the invocation rate directly determines the speedup and efficiency benefits. In Figure 11, the optimal design minimizes the size of the predictor (left on the *x*-axis) and maximizes the accelerator invocation rate (up on the *y*-axis). As Figure 11 illustrates, the design with eight parallel tables, each of size 0.5KB is the Pareto optimal design. This design space exploration shows that both the number of tables and the size of each table have a significant impact on the accelerator invocation rate. Due to destructive aliasing, a smaller table (i.e., 0.125KB) is unable to discriminate the inputs that cause large errors. As a result, smaller tables fail to preserve the benefit of acceleration. On the other hand, larger table sizes (i.e.,8KB) do not provide benefits beyond a certain point because destructive aliasing cannot be completely eliminated even with 8KB tables. Hence, we use the Pareto optimal point with 8 tables each of size 0.5KB as our default configuration. The configuration with larger number of tables provide higher benefits even if the size of each table is small as the chance of destructive aliasing decreases since each table uses a distinct hash function.

3) Data Compression for the Table-based MITHRA*:* Based on the Pareto analysis, the optimal table-based design is eight tables, each of size 0.5 KB. Therefore, the total uncompressed size of this design is 4 KB. We observe that there are large trails of 0s in the tables. This insight provides an opportunity to compress the table and reduce the necessary memory state of the table-based design. To compress the tables, we use the low-overhead and low-latency Base-Delta compression technique [29] that has been recently proposed for cache line compression. The Base-Delta compression and decompression algorithms require only vector addition, subtraction, and comparison operations. We arrange the tables in rows of 64 B size to employ this cache line compression mechanism. Table II shows the compression results for each benchmark from the original uncompressed size of 4 KB. The sparser the contents of the table, the higher the compression ratio. These results show that blackscholes, fft, inversek2j, and jmeint achieve 16× size reduction. However, sobel and jpeg do not benefit from compression due to the complexity of the inputs and the high density of the contents of the tables. Table II also shows the size of the neural MITHRA for each application and their topology. In most cases, after compression, the sizes of MITHRA is less than 1 KB.

VI. Related Work

A growing body of work has explored leveraging approximation for gains in performance and energy [8]–[10], [10]–[12], [16], [17], [19], [22], [23], [34]. Our work, however, focuses on a hardware-software mechanism to control quality tradeoffs for approximate accelerators. Several techniques provide software-only quality control mechanisms for approximate computing that either operate at compile-time [13], [14], [35]–[37] or runtime [10]–[12], [22], [38]. In contrast, we define MITHRA, that uses hardware to control quality tradeoffs at runtime and provides necessary compiler support for the proposed hardware designs. Below, we discuss the most related works.

Compile-time techniques to control quality tradeoffs. Rely [35] is an approximate programming language that requires programmers to mark variables and operations as approximate. Given these annotations, Rely combines symbolic and probabilistic reasoning to verify whether the quality requirements are satisfied for a function. To provide this guarantee, Rely requires the programmer to not only mark all variables and operations as approximate but also provide preconditions on the reliability and range of the data. Similar to Rely, the work in [37] proposes a relational Hoare-like logic to reason about the correctness of approximate programs. The work in [36] uses Bayesian network representation and symbolic execution to verify probabilistic assertions on the quality of the approximate programs given the input distributions. Given a quality requirement, Chisel [14] uses Integer Linear Programming (ILP) to optimize the approximate computational kernels at compile time. The approximation model in these works is based on architectures that support approximation at the granularity of a single instruction [9]. The work in Stoke [39] focuses on reducing the bit width of floating-point operations at compile-time, trading accuracy for performance. While these techniques focus approximation at the fine granularity of single instruction, we focus on coarse-gain approximation with accelerators. In a concurrent work [40], determining the quality control knob is cast as an optimization problem; however, the work neither uses statistical techniques nor provides hardware mechanisms for quality control. The above approaches do not utilize runtime information or propose microarchitectural mechanisms for controlling quality tradeoffs, which is a focus of our work.

Runtime quality control techniques. Green [12] provides a code-centric programming model for annotating loops for early termination and substitution of numerical functions with approximate variants. Sage [10] and Paraprox [11] provide a set of static approximation techniques for GPGPU kernels. Both techniques utilize the CPU to occasionally monitor the quality of the final outputs and adjust the approximation level. ApproxHadoop [41] uses statistical sampling theory to control input sampling and task dropping in approximate MapReduce tasks. Light-Weight Checks [38] requires the programmer to write software checks for each approximately accelerated function and the precise function is run if the check fails. Rumba [42], concurrent to an earlier version of this work [43], only proposes microarchitectural mechanisms that use decision trees and linear models for predicting the accelerator error value. Since Rumba does not offer the necessary compiler support, it does not map the final output quality to the local decision on the accelerator call site. The lack of compiler support impedes Rumba from providing concrete statistical guarantees for the final output quality. Rumba also relies on error value prediction (regression) that is significantly more demanding and less reliable than the MITHRA's binary classification solution.

Unlike these techniques that either rely only hardware or software checks, we define a cohesively co-designed hardware-

software technique for controlling quality tradeoffs that leverages runtime information and compiler support to maximize the gains from approximate acceleration and provide statistical guarantees.

VII. Conclusion

Approximate accelerators deliver significant gains in performance and efficiency by trading small losses in quality of the results. However, the lack of a hardware-software mechanism that control this tradeoff limit their applicability. In this paper, we describe MITHRA, a hardware-software solution for controlling the quality tradeoffs at runtime. MITHRA provides only statistical quality guarantees on unseen data. The acceptability of such guarantees is still a matter of debate and investigation. However, it is clear that the applicability of approximate computing requires moving beyond traditional and formal quality guarantees. In fact, such guarantees are to some extent accepted for service level agreements in data centers. Also, the widely used machine learning algorithms also rely on similar statistical guarantees. This work takes an initial step in controlling the quality tradeoffs for approximate accelerators; aiming to open a path for their adoption.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. We thank Anandhavel Nagendrakumar for his contributions to energy measurements. We also thank Balaji Chandrasekaran and Hardik Sharma for their feedback on the text. This work was supported in part by a NSF award CCF#1553192, Semiconductor Research Corporation contract #2014-EP-2577, and gifts from Google and Microsoft.

References

[1] R. H. Dennard *et al.*, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, 1974.

[2] N. Hardavellas *et al.*, "Toward dark silicon in servers," *IEEE Micro*, 2011.

[3] H. Esmaeilzadeh *et al.*, "Dark silicon and the end of multicore scaling," in *ISCA*, 2011.

[4] R. Hameed *et al.*, "Understanding sources of inefficiency in general-purpose chips," in *ISCA*, 2010.

[5] S. Gupta *et al.*, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *MICRO*, 2011.

[6] G. Venkatesh *et al.*, "Conservation cores: Reducing the energy of mature computations," in *ASPLOS*, 2010.

[7] V. Govindaraju *et al.*, "Dynamically specialized datapaths for energy efficient computing," in *HPCA*, 2011.

[8] M. de Kruijf *et al.*, "Relax: An architectural framework for software recovery of hardware faults," in *ISCA*, 2010.

[9] H. Esmaeilzadeh *et al.*, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012.

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

[11] M. Samadi *et al.*, "Paraprox: Pattern-based approximation for data parallel applications," in *ASPLOS*, 2014.

[12] W. Baek *et al.*, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *PLDI*, 2010.

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

[14] S. Misailovic *et al.*, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *OOPSLA*, 2014.

[15] J. Park *et al.*, "AxGames: Towards crowdsourcing quality target determination in approximate computing," in *ASPLOS*, 2016.

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

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

[18] A. Yazdanbakhsh *et al.*, "Neural acceleration for gpu throughput processors," in *MICRO*, 2015.

[19] S. Venkataramani *et al.*, "Quality programmable vector processors for approximate computing," in *MICRO*, 2013, pp. 1–12.

[20] Z. Du *et al.*, "Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators," in *ASP-DAC*, January 2014.

[21] B. Belhadj *et al.*, "Continuous real-world inputs can open up alternative accelerator designs," in *ISCA*, 2013.

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

[23] T. Moreau *et al.*, "SNNAP: Approximate computing on programmable socs via neural acceleration," in *HPCA*, 2015.

[24] NetBSD Documentation, "How lazy FPU context switch works," 2011. [URL] http://www.netbsd.org/docs/kernel/lazyfpu.html

[25] C. Clopper *et al.*, "The use of confidence or fiducial limits illustrated in the case of the binomial," *Biometrika*, pp. 404–413, 1934.

[26] M. H. DeGroot, *Probability and Statistics*. Chapman & Hall, 1974.

[27] B.-H. Lin *et al.*, "A fast signature computation algorithm for lfsr and misr," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 19, no. 9, pp. 1031–1040, Sep 2000.

[28] D. E. Rumelhart *et al.*, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, 1986, vol. 1.

[29] G. Pekhimenko *et al.*, "Base-delta-immediate compression: practical data compression for on-chip caches," in *PACT*, 2012.

[30] A. Patel *et al.*, "MARSSx86: A full system simulator for x86 CPUs," in *DAC*, 2011.

[31] S. Li *et al.*, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[32] N. Muralimanohar *et al.*, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO*, 2007.

[33] S. Galal *et al.*, "Energy-efficient floating-point unit design," *IEEE TC*, 2011.

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

[35] M. Carbin *et al.*, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *OOPSLA*, 2013.

[36] A. Sampson *et al.*, "Expressing and verifying probabilistic assertions," in *PLDI*, 2014.

[37] M. Carbin *et al.*, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *PLDI*, 2012.

[38] B. Grigorian *et al.*, "Dynamically adaptive and reliable approximate computing using light-weight error analysis," in *AHS, 2014 NASA/ESA Conference on*. IEEE, 2014, pp. 248–255.

[39] E. Schkufza *et al.*, "Stochastic optimization of floating-point programs with tunable precision," in *PLDI*, 2014.

[40] X. Sui *et al.*, "Proactive control of approximate programs," in *ASPLOS*, 2016.

[41] I. Goiri *et al.*, "Approxhadoop: Bringing approximations to mapreduce frameworks," in *ASPLOS*, 2015.

[42] D. S. Khudia *et al.*, "Rumba: An online quality management system for approximate computing," in *ISCA*, Jun. 2015.

[43] D. Mahajan *et al.*, "Prediction-based quality control for approximate accelerators," in *Workshop on Approximate Computing Across the System Stack (WACAS) in conjunction with ASPLOS*, Mar. 2015.