Prediction-Based Quality Control for Approximate Accelerators

Divya Mahajan Amir Yazdanbakhsh Jongse Park Bradley Thwaites Hadi Esmaeilzadeh
Georgia Institute of Technology

Abstract

Approximate accelerators are an emerging type of accelerator that trade output quality for significant gains in performance and energy efficiency. Conventionally, the approximate accelerator is always invoked in lieu of a frequently executed region of code (e.g., a function in a loop). However, always invoking the accelerator results in a fixed degree of error that may not be desirable. Our core idea is to predict *whether each individual accelerator invocation will lead to an undesirable quality loss in the final output. We therefore design and evaluate predictors that only leverage information local to that specific potential invocation. If the predictor speculates that a large quality degradation is likely, it directs the core to run the original precise code instead. We use neural networks as an alternative prediction mechanism for quality control that also provides a realistic reference point to evaluate the effectiveness of our table-based predictor. Our evaluation comprises a set of benchmarks with diverse error behavior. For these benchmarks a table-based predictor with eight tables each of size 0.5KB achieves 2.6× average speedup and 2.8× average energy reduction with a 5% error requirement. The neural predictor yields 4% and 17% larger performance and energy gains, respectively. On average, an idealized oracle predictor with prior knowledge about all invocations achieves only 26% more performance and 37% more energy benefits compared to the table-based predictor.*

1. Introduction

With the effective end of Dennard scaling, per-transistor speed and efficiency improvements are diminishing [10]. Energy efficiency now fundamentally limits microprocessor performance. As a result, there is an increasing interest in specialization and acceleration that trade generality for significant gains in performance and efficiency. Designing application-specific ICs may provide three orders of magnitude improvement in efficiency and speed. However, designing ASICs for the massive and rapidly-evolving body of general-purpose applications is currently impractical. Programmable accelerators, such as FPGAs and GPUs, provide a middle ground that exploit some characteristic of the application domain to achieve performance and efficiency gains at the cost of generality. For instance, FPGAs exploit copious fine-grained irregular parallelism but perform poorly when complex and frequent accesses to memory are required. GPUs exploit data-level parallelism and SIMT execution but lose efficiency when threads diverge.

There is an emerging type of accelerators, *approximate accelerators* [12, 2, 36, 9, 4, 14, 20] that exploit application's tolerance to inexact computation. As the growing body of recent work in approximation shows [8, 11, 26, 1, 28, 35], many classes of applications including web search, data analytics, machine learning, multimedia, cyber-physical systems, vision, and speech recognition can tolerate small errors in computation. For these classes of applications, trading off computation accuracy can potentially lead to gains in performance and energy efficiency.

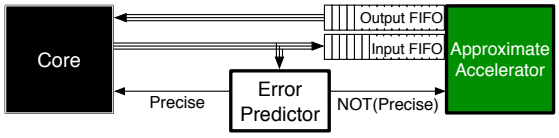


Figure 1: Architectural overview of our quality control approach.

Conventionally, the approximate accelerator is always invoked in lieu of a frequently executed region of code, e.g., function in a loop. Always invoking the accelerator results in a fixed degree of error that may not conform with the user requirements limiting its applicability. The core idea behind this work is *to predict* whether or not each individual accelerator invocation will lead to an undesirable quality loss in the final output. To realize this idea, we design and evaluate predictors that only leverage information local to that specific potential invocation. If the predictor speculates that a large quality degradation is likely, it directs the processor core to run the original precise code thereby reducing the final output error. The predictor is in command of all accelerator invocations.

We are inspired by the large body of work on branch prediction, value prediction, and load-store dependence prediction. The unique property of this work is the use of prediction for quality control in approximate acceleration. This work builds on the prior work on prediction and extends it to an emerging direction of approximate computing. Furthermore, even though there have been several works on software-based quality control [26, 15, 3], we address the lack of microarchitectural mechanisms that leverage runtime information for quality control.

2. Overview

As shown in Figure 1, the error predictor sits between the core and the accelerator and determines whether the core should execute the original function or invoke the accelerator. To realize this approach, we answer the following questions:

1) What insights from the accelerator behavior can guide the predictor design? To understand the approximate accelerator behavior, we investigate different application's error distribution when approximated with NPUs. Figure 2 depicts the CDF (cumulative distribution function) plot of the error incurred by each element of the application output. The application output comprises groups of elements–an image consists of pixels; a vector consists of scalars; etc. The CDF shows that only a small fraction (0%-20%) of the output elements see large errors. This finding shows that a prediction mechanism that can filter out these cases not only can eliminate large errors but also can preserve the significant gains from approximate acceleration.

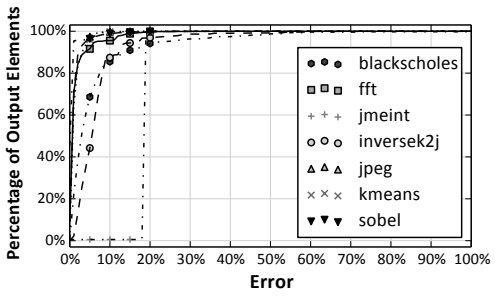


Figure 2: Cumulative distribution function (CDF) plot of the applications output error. A point (x, y) indicates that y fraction of the output elements see error less than or equal to x.

2) Given a final quality requirement, how much error in the accelerator is acceptable? As shown in Figure 1, the predictor only has the information provided to the accelerator. As the accelerator only approximates a region of the code, the predictor can only make local decisions based on the input vectors with no knowledge of how this local error manifests in the final application output. The main challenge is to devise a compilation or a runtime system that can guide the predictor's local decisions while considering the final program output. This challenge is strongly correlated with another challenge, which is, how much accelerator error is acceptable given a final quality requirement. Hence, the prediction-based quality control should provide a knob to the compiler or the runtime system to navigate the tradeoff between the quality and performance-efficiency gains. To address these challenges, we develop a thresholding mechanism that guides the predictor. Section 4 provides this algorithm and elaborates on how the predictor is trained.

3) What accelerator information is needed and is available to perform the prediction? The approximate accelerator accelerates a function with well-defined inputs and outputs. The prediction mechanism defined here leverages this input-output interface of the accelerator. As the accelerator error contributes to the final output error and hence a mechanism to control accelerator error can control the final output error. However, for a given application, the accelerator is constant which makes the accelerator error only a function of the input vector. This insight is significant because unlike branch predictors that rely on the history of previous branches, our error predictors cannot use history information as it would require running both the accelerator and the original code. Our predictor utilizes the accelerator input vector to predict if a particular invocation gives undesirable error. This mechanism can administer the accelerator error by limiting the accelerator invocations to the inputs that give error within a bound.

4) What is the prediction algorithm? We explore two types of predictors: a table-based predictor and a neural predictor. Both prediction mechanisms comprise two phases - training and runtime. For the table-based design, in both the phases the predictor uses a simple circuitry to generate a hash from the accelerator inputs. During the predictor-training phase the hash generated indexes into an entry in the predictor table and this hash is used to fill the predictor contents. The predictor is trained to invoke the accelerator if the accelerator error is below the threshold and trained to run the original function if the accelerator error is above the threshold. Hence, this training generates a configuration that maps a hash to a single bit decision. The predictor needs to be retrained for different applications as accelerator error is a function of the application. During the runtime phase, a hash is calculated for the accelerator inputs to index into the prediction table to give a decision. Hash collisions are imminent and this destructive aliasing in the tables can lead to a significant decrease in the prediction accuracy. We address this challenge with careful hash function design that tries to minimize conflicts between different input sets and the use of multiple prediction tables(a small ensemble of predictors, Section 3 provides further details). In the case of neural predictor, we use a multilayer perceptron that takes in the input vector and produces a single-bit output as the prediction. The neural predictor is trained during compile time using a set of representative training inputs and the configuration is used at runtime to train the predictor and finally produce the prediction. Section 3 describes these predictors is detail.

3. Predictor Design for Quality Control

As mentioned before, the accelerator provides a well defined input-output interface and the quality-control predictor can utilise this interface by taking in an input vector and produce a single-bit prediction. This section discusses the design and implementation of a table-based and a neural predictor.

3.1. Table-based Predictor

A table-based predictor stores the predictions (single-bit values) in a table indexed by a hash over the elements of the input vector. In this section we first discuss the hash function and then explore a multi-table predictor that aims to strike a balance between prediction accuracy and the predictor size.

3.1.1. Variable-Size Multi-Input Prediction

This hash function should be able to (1)combine all the elements in the input vector, (2)reduce destructive aliasing as much as possible, (3)be efficiently implementable in hardware, (4) accept a varying number of inputs. To efficiently satisfy these requirements, we use a hardware structure called Multi-Input Signature Register (MISR) [18] that hashes the input elements and generates the table index. 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. The result of the XOR operation is stored in the 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 inputs are processed, the value remaining in the register is the index.

This index is the result of combining all the input elements that the core sends to the accelerator for a single invocation. We optimistically send these inputs to both the accelerator and the predictor assuming that in most cases the predictor will decide to invoke the accelerator. This strategy is in line with

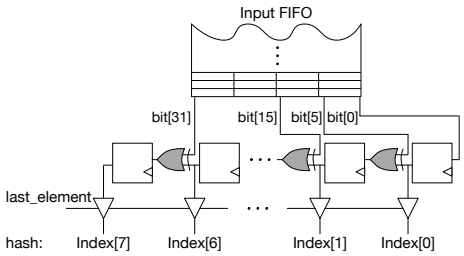


Figure 3: An example hash function. Each hash function takes an input vector with multiple elements and generates the index. All the hashes are MISRs but they select different bits from the inputs.

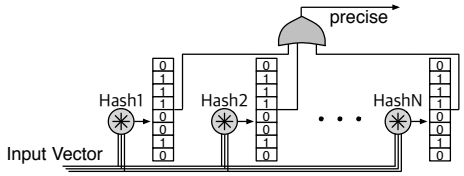


Figure 4: Multi-table predictor. All the tables are equally size but each table is indexed with a different hash function. All the hash functions are are MISRs. However, each MISR is configured differently.

the earlier insight that only a small fraction of the invocations will require invoking the original precise function.

3.1.2. Multi-Table Prediction

Using a single table for prediction requires large table sizes to provide acceptable prediction accuracy. The root cause is that only a small fraction of the input combinations need to execute the original precise code. This phenomenon makes the prediction harder with only one small table due to destructive aliasing. As multiple inputs hash to the same index, in a single table, the bias is always towards invoking the accelerator. However, the predictor needs to distinguish the inputs that will cause large quality loss and avoid invoking the accelerator.

We take inspiration from prior work on branch predictors that use multiple tables [31, 32] or combine different predictors [16]. Figure 4 illustrates the general architecture of our multi-table predictor. The predictor consists of multiple equally-sized tables. The entries in the tables are all single-bit values. In this predictor, the hash function for each table is a different MISR. Figure 3 illustrates one example MISR configuration. Using different hash functions for different tables lowers the probability of destructive aliasing in all of the tables. As the input elements arrive, the MISRs generate the indices in parallel and then the prediction values are read from the tables. The next step is combining the predictions from different tables. Since the bias in each single table is toward invoking the accelerator, the predictor directs the core to run the original function even if a single table predicts to run the original function. Therefore, the logic for combining the result of the tables is an OR gate. Our results shows that our multi-table predictor achieves similar accuracy levels to a more sophisticated neural predictor.

3.2. Neural Predictor

Neural networks are powerful prediction and modelling tools. We explore their for our prediction-based quality control task. Prediction with neural networks have higher latency and requires more computational power than the table-based predictor. The neural predictor spends some of the the performance and efficiency gains for higher quality of results. Using neural networks not only provides another design point, but also provides a reference point to evaluate the accuracy and effectiveness of our table-based predictor. There are a variety of neural networks in the literature. For our task, we use multi-layer perceptrons (MLPs) due to their broad applicability.

3.3. Table-based and Neural Predictor Training

As mentioned in the section 4, during compilation or runtime we calculate the threshold that represents the final output requirements at the local accelerator level. During the training period both the accelerator and the original function are run to obtain the accelerator error. If the accelerator error is above the threshold the predictor is trained to run the original function '1' else the accelerator '0'. However, the training is also dependent on the predictor mechanism. We define the training for both table-based and neural predictor.

Table-based predictor. Initially all the table entries are set to a '0' implying a 100% accelerator invocation. However, if a particular accelerator input vector gives an error above the threshold, the table entry pointed by the hash generated by this input set is set to '1'. This implies that even if a single input set pointing to a particular hash gives an undesirable error, that hash entry in the table is set to '1'. This avoids the bias towards invoking the accelerator as most of the inputs give a small accelerator error. Finally, this same methodology is extended to train an ensemble of tables. In the case of a table-based predictor, we compress the content of the tables and encode it in the binary using the Base-Delta-Immediate compression algorithm [23].

Neural Predictor. We only consider neural networks with 2, 4, 8, 16, and 32 neurons in the hidden layer even though more layers and more neurons-per-layer are possible. We train these five topologies with back-propagation [24] and choose the one that provides the highest accuracy with the fewest neurons.

4. Compiler-Support for Quality Control

As mentioned before, only a certain region of code(a target function) is offloaded to the accelerator. For each invocation of this target function, the predictor decides whether to use the accelerator or run the original precise function. The predictor makes this decision based on the available local information (input vector) with no knowledge of how those decision will affect the application's final output quality. That is because applications behave in complex manners and the final output is not yet calculated when the predictor is making its decisions. However, before the predictor can make its decision there is a need to train the predictor such that it can make decisions during the actual runtime. The challenge here is to guide and train the predictor based on the final output quality requirement. This work provides the insight that profiling information can effectively provide the proper guidance and training for the

predictor. Profiling provides the necessary global view of the application and quality degradation. The key is to exploit this global view and enable the predictor to make local decisions.

4.1. Compilation Workflow for Predictor Training

For the profiling and training, the programmer should first provide the application-specific quality requirement, quality metric, and training input sets to the compiler. The remaining predictor training constitutes two phases. In the first phase, using all the requirements and metrics provided by the user, the compiler runs a heuristic algorithm that leverages profiling to determine which input combinations to the target function can be executed on the accelerator while statistically satisfying a given quality requirement (Section 4.2). Using this information, the compiler trains the predictor (Section 4.3). As the profiling and training are a part of the offline compilation, the trained predictor configuration is incorporated in the accelerator configuration.

4.2. Finding Threshold

The degree of acceptable quality loss is provided as a compiler option. Quality is an application-specific metric. As is commensurate with other works on approximation, the programmer needs to provide a function that measures the final quality degradation. To perform profiling, the programmer also provides a set of representative input sets similar to those in the application test suite. Given this information, we use thresholding to identify the function inputs for which invoking the approximate accelerator will eventually lead to quality loss greater than the requirement.

In this technique, the premise is that the accelerator error should not exceed a given threshold. That is, if the accelerator will lead to a larger error than the threshold on a certain input, it should not be invoked on that input and the original function should be executed. Equation 1 defines this premise and shows that the error on all the output elements should be below the threshold to allow invoking the accelerator.

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

For a given output quality, the threshold is the upper bound on the error that can be tolerated from the accelerator. In other words, the threshold is the maximum local error that the target function can impose on the execution. With this definition, the compiler's task is to find the threshold such that the final quality requirement is statistically satisfied. To find the threshold, we develop an algorithm that iteratively searches for the optimal threshold that conservatively maximizes the number of accelerator invocations subject to the quality requirements. That is, the algorithm aims to maximize the acceleration benefits. As Section 5 elaborates, we use a different set of inputs to validate the selection of the threshold.

4.3. Training Predictor

Once the threshold is determined, in the second phase the predictor can be trained. Training the predictor requires running the application and randomly sampling the accelerator error. For the sampled invocations, if the accelerator error on all the output elements is less than the threshold, the predictor will be trained to invoke the accelerator. Otherwise, the predictor will be trained to trigger the execution of the original function.

Whether or not to invoke the accelerator is a local decision that is merely based on the accelerator error. Hence, this decision is a function of the accelerator configuration, its inputs, and the threshold. The accelerator configuration does not change for an application and the the threshold is constant for compilation targeting a quality requirement. Therefore, training of the predictor is only dependent on the inputs that are sent to the accelerator. Given this insight, we pre-train the predictor through profiling. Profiling is used to generate the training input vectors for the predictors from the application training datasets. Each accelerator input vector is mapped to a single bit decision depending on the results of the equation 1 for that particular input vector. As the results in Section 5 show, this training methodology provides statistical quality guarantees with high confidence. Finally, we add a special branch instruction to the ISA that invokes the original code instead of the accelerator if the predictor predicts large error for that particular invocation.

Table 1: Benchmarks, their quality metric, input data sets, and the initial quality loss when the accelerator is invoked all the time.

Benchmark	Application Error Metric	Input Data	NPU Topology	Error
blackscholes	Avg. Relative Error	4096 Data Point from PARSEC Suite	6->8->8->1	6.03%
fft	Avg. Relative Error	2048 Random Floating Point Numbers	1->4->4->2	5.48%
inversek2j	Avg. Relative Error	10000 (x, y) Random Coordinates	2->8->2	7.29%
jmeint	Miss Rate	10000 Random Pairs of 3D Triangle Coordinates	18->32->8->2	17.69%
jpeg	Image Diff	220x200-Pixel Color Image	64->16->64	6.91%
sobel	Image Diff	220x200-Pixel Color Image	9->8->1	9.96%

5. Evaluation

Cycle-accurate simulation. We use the MARSSx86 x86-64 cycle-accurate simulator [22] to measure the performance of the accelerated system, which is augmented with our prediction-based quality control mechanisms. The processor is modeled after a single-core Intel Nehalem to evaluate the performance benefits over an aggressive out-of-order architecture[1]. The NPU consists of eight processing elements and exposes three queues to the processor to communicate the inputs, the outputs, and the configurations. 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 also augmented MARSSx86 with a cycle-accurate simulator for NPU that also models the overheads of the prediction-based quality control.

Energy modeling. We use McPAT [17] for processor energy estimations. We model the energy of the NPU using results from McPAT, CACTI 6.5 [21], and [13]. The cycle-accurate

[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-entry 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

NPU simulator provides detailed statistics that are used to estimate its energy and the neural predictor. For estimating the energy of the table-based predictor, we implement the MISRs in Verilog and synthesize them. We use CACTI 6.5 to measure the energy cost of accessing the tables. The processor, the predictors, and the accelerator operate at 2080 MHz at 0.9 V and are modeled at 45 nm technology node.

Benchmarks. Table 1 summarizes the information about each benchmark: application domain, input data, NPU topology, and final application error levels when the accelerator is invoked all the time without quality control. Each benchmark requires an application-specific error metric, which is used in our compilation and evaluations. The initial error with no quality control and full approximation ranges from 5.48% to 17.69%. This relatively low initial error makes the quality-control more challenging and the diversity of the application error provides an appropriate ground for understanding the tradeoffs in prediction-based quality control.

Input datasets. We use 40 distinct input datasets for the experiments. We use 20 datasets during compilation to find the threshold and train the predictor. We use 20 different unseen datasets for validation and final evaluations that are reported in this section. The datasets are typical program inputs, such as images, or random values (see Table 1).

5.1. Experimental Results

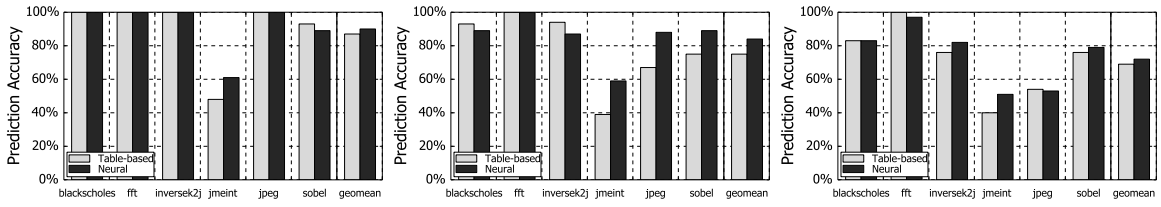
Prediction accuracy. We use an oracle predictor as an idealized point of reference to measure the accuracy of our predictors. The oracle predictor provides maximum benefits from approximate acceleration while satisfying the error requirements as it uses the threshold to demarcate the inputs that run on the accelerator and the ones that run on the core. Figure 5a, Figure 5b, and Figure 5c compare the prediction accuracy of the table-based and neural predictor when the error requirements are 7.5%, 5.0%, and 2.5%. In these experiments, for the table-based predictor we use the Pareto-optimal configuration comprising of eight tables, each of size 0.5KB. The neural predictor requires a different topology for each application. These topologies are reported in Table 1. The table-based predictor provides an average of 75% of prediction accuracy for the 5.0% error requirement; only 9% short of the neural predictor which is a sophisticated compute-heavy predictor.

Another factor useful in the evaluation of the prediction-based quality control is the accelerator invocation rate. The invocation rate is the ratio of the number of the accelerator invocations to the total number of times the target function is executed. Figure 6 shows the invocation rate of the table-based, neural, and oracle predictor with the three error requirements. Note that the accelerator invocations determined by the predictors satisfy the specified error requirements. As the error requirements are tightened (from 7.5% down to 2.5%), we see a general decrease in both the prediction accuracy and the invocation rate. To meet the stricter requirements, the predictor conservatively runs the original function more frequently. The average prediction accuracy decreases from 87% to 69% for the table-based predictor and 90% to 72% for the neural predictor as the error requirements tightens from 7.5% to 2.5%. The average invocation rate drops from 88% to 42% for the table-based predictor and 90% to 57% for the neural predictor.

Except for two of the benchmarks, jmeint and sobel, the other benchmarks satisfy the 7.5% error requirement even with 100% accelerator invocation rate. In these cases, the predictor simply predicts '0'. We report the accuracy of the predictor in these cases as 100% and do not engage the predictor at all. The jmeint benchmark is an outline and has the highest initial error rate of 17.69% with 100% invocation rate. This characteristic is due to the complex control flow behavior in this benchmark. As the requirements become stringent, the predictor needs to mostly execute the original precise function. For jmeint with an error requirement of 2.5%, the table-based predictor executes the original function 90% of the time to satisfy the error requirements. Benchmark fft on the other hand, provides a very high prediction accuracy of 99% for all error requirements. This high prediction accuracy is due to the fact that the accelerator inputs that produce large errors on the accelerator have similar properties. These accelerator inputs benefit from the constructive aliasing that the MISR-based hash function provides. The neural predictor outperforms the table-based prediction with relatively small margins. However, neural networks are sophisticated prediction algorithms that require both energy and compute resources. The neural predictor also provide a realistic reference point to asses the efficacy of the much simpler table-based predictor.

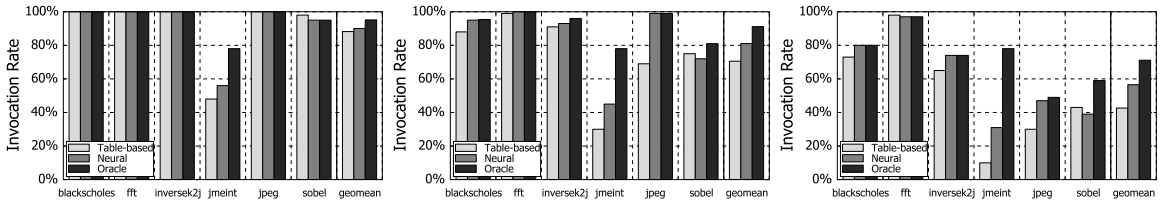
> As the results show, the table-based predictor provides comparable accuracy to the more sophisticated neural predictor. These results suggest that our table-based algorithm can be efficiently used for quality-control in approximate acceleration.

Performance and energy benefits. The prediction-based quality control aims to retain the maximum possible performance and energy efficiency benefits while statistically satisfying the desired error requirements. Figure 7 and Figure 8 respectively show the speedup and the energy benefits for the specified error requirements. Comparing Figure 6, Figure 7, and Figure 8, clearly shows that the performance and energy efficiency benefits have a direct correlation with the invocation rate. As discussed when the invocation rate is 100%, the predictor is turned off and does not impose any overheads. In Figure 7, and Figure 8, the FullApprox bar represents the acceleration with no quality control. Figure 7a shows the application speedup when the accelerator is executed for the approximated region while satisfying the 7.5% error requirement. In this case, the predictor is only engaged for jmeint and sobel. Therefore, the average speedup of 4× and 5.5× are observed for the table-based and neural predictors respectively. However, as shown in Figure 7b and Figure 7c, the speedup reduces to 3× for 5.0% and 2× for 2.5% error requirement. The reduced invocation rate for stricter error requirements accounts for this decrease in the benefits obtained from approximate accelerator. Even though the table-based predictor has a lower invocation rate than the neural predictor due to the lower prediction accuracy, the speedup obtained from the predictors are similar since the neural predictor incurs a higher cost owing to its sophisticated technique and hence negating some of the benefits obtained from approximate acceleration. The energy reduction results show similar trends. The average energy reduction is 5× for 7.5% and drops to 2× for 2.5% error requirement. Compared to the speedup, the energy benefits decrease with a higher rate as the error requirements become tighter since the energy reduction ratio per an accelerator invocation is greater than the speedup. In our experiments, the



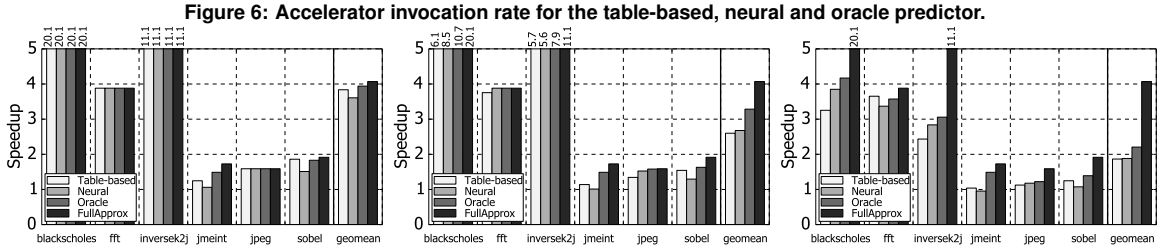
(a) 7.5% error requirement (b) 5.0% error requirement (c) 2.5% error requirement

Figure 5: Prediction accuracy in comparison to the oracle predictor.



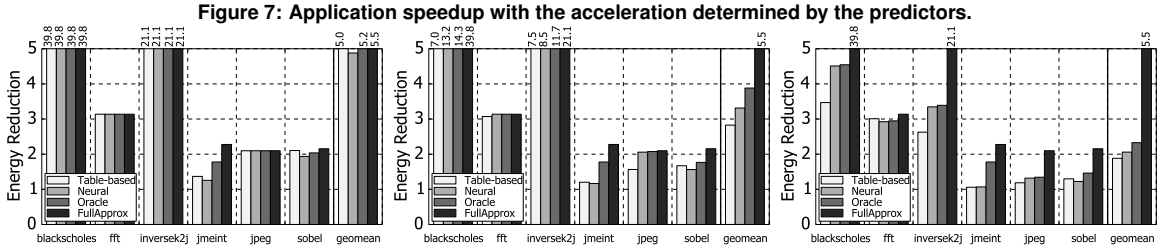
(a) 7.5% error requirement (b) 5.0% error requirement (c) 2.5% error requirement

Figure 6: Accelerator invocation rate for the table-based, neural and oracle predictor.



(a) 7.5% error requirement (b) 5.0% error requirement (c) 2.5% error requirement

Figure 7: Application speedup with the acceleration determined by the predictors.



(a) 7.5% error requirement (b) 5.0% error requirement (c) 2.5% error requirement

Figure 8: Application energy reduction with acceleration determined by the predictors.

	Table Predictor			Neural Predictor		
	7.5%	5.0%	2.5%	7.5%	5.0%	2.5%
blackscholes	0.952 ± 0.038	0.91 ± 0.075	0.933 ± 0.061	0.952 ± 0.038	0.952 ± 0.038	0.952 ± 0.038
fft	0.91 ± 0.075	0.933 ± 0.061	0.952 ± 0.038	0.91 ± 0.075	0.952 ± 0.038	0.952 ± 0.038
inversek2j	0.952 ± 0.038	0.933 ± 0.061	0.952 ± 0.038	0.952 ± 0.038	0.933 ± 0.061	0.952 ± 0.038
jmeint	0.91 ± 0.075	0.91 ± 0.075	0.91 ± 0.075	0.841 ± 0.103	0.841 ± 0.103	0.864 ± 0.095
jpeg	0.818 ± 0.11	0.887 ± 0.086	0.91 ± 0.075	0.818 ± 0.11	0.818 ± 0.11	0.91 ± 0.075
sobel	0.841 0.103	0.887 ± 0.086	0.952 ± 0.038	0.91 ± 0.075	0.887 ± 0.086	0.952 ± 0.038

Table 2: Table-based and neural predictor's 95% confidence interval.

neural predictors benefits from the existing NPU infrastructure and hence when a neural accelerator is unavailable, neural predictors are not that effective. However, our table-based predictor achieves similar benefits with no dependencies to the acceleration technique.

> Although the neural predictor has a slightly higher prediction accuracy and invocation rate than the table-based predictor, both predictor yield similar benefits in terms of speedup and energy efficiency due to the high cost of the neural predictor.

Statistical Quality Guarantees The predictors provide statistical guarantees that the quality requirement will be satisfied. That is, with high probability, the error requirements will be met on unseen data that is drawn from the same distribution of the inputs that are used for selecting the threshold and training the predictors. To provide confidence in our statistical guarantees, we randomly divide the program input set into two subsets, training and validation. Each subset in our experiments have 20 distinct input sets. We use one subset to train the predictors and the other to validate the predictors' accuracy. Previous works on approximate computing [33, 19, 26, 25, 30] that deal with quality control also provide statistical guarantees. In general, providing formal guarantees that the quality requirements will be met on all possible inputs is still an open research problem. Due to the complex behavior of programs and the large space of possible inputs, the statistical approaches to validate quality control techniques are the common practice in approximate computing [33, 19, 26, 25, 30]. Section 6 discusses these related works in more detail.

To assess the confidence in the statistical guarantees, we measure the 95% central confidence intervals using the 20 unseen validation datasets. We adopt the same methodology that is used in [26, 25] for measuring Baysian confidence intervals. To avoid bias when measuring the confidence

intervals, we assume that the prior distribution is uniform. Consequently, the posterior will be a Beta distribution, $BETA(k+1, n+1-k)$, where n is the number of program input datasets (number of observed samples) and k is the number of datasets on which the approximated program satisfies the quality requirement [34]. Table 2 depicts the confidence intervals for all the three error requirements, which are 7.5%, 5%, and 2.5%. An interval (p_1, p_2) in the figure shows that with probability between p_1 and p_2, the prediction-based quality control will satisfy the requirement on unseen data. The confidence in these probabilities is 95%.

In most cases, the quality requirement is satisfied on all the validation data sets. In these cases, the probability interval is (0.91, 0.99). In the case of jmeint and jpeg that the quality requirements are not satisfied with few of the data sets, in the worst case the probability interval is (0.72, 0.90). Even in these cases the probability of satisfying the error requirements is significantly high that confirms the effective ness of our prediction-based quality control.

6. Related Work

Several studies have shown that diverse classes of applications are tolerant to imprecise execution [7, 27, 12]. A growing body of work has explored leveraging approximation at the the circuit and architecture level for gains in performance, energy, and resource utilization [8, 11, 26, 1, 12, 36, 28, 2, 26, 3, 14, 20]. Our work; however, lies at the intersection of (1) quality control techniques for approximate computing and (2) microarchitectural techniques for prediction and speculation. Several techniques provide software-only quality control mechanisms for approximate computing that either operate at compile-time [33, 19, 6, 29, 5, 30] or runtime [3, 26, 25, 15, 14]. In contrast, we define a prediction-based microarchitectural mechanism for quality control at runtime that exposes a knob to the compiler. We also develop the necessary compiler support for the proposed hardware mechanism.

7. Conclusion

Approximate accelerators are an emerging type of accelerators that trade output quality for significant gains in performance and energy efficiency. However, the lack of microarchitectural mechanisms that control this tradeoff limit their applicability. In this paper, we described a prediction-based microarchitectural mechanism for quality control in these accelerators. The work in this paper provides microarchitectural mechanisms to assist quality control, an imperative feature for approximate acceleration to become viable.

References

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

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

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

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

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

[6] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *OOPSLA*, 2013.

[7] M. de Kruijf and K. Sankaralingam, "Exploring the synergy of emerging workloads and silicon reliability trends," in *SELSE*, 2009.

[8] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *ISCA*, 2010.

[9] Z. Du *et al.*, "Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, January 2014.

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

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

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

[13] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *IEEE TC*, 2011.

[14] B. Grigorian, N. Farahpour, and G. Reinman, "BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing," in *HPCA*, 2015.

[15] B. Grigorian and G. Reinman, "Dynamically adaptive and reliable approximate computing using light-weight error analysis," in *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*. IEEE, 2014, pp. 248–255.

[16] R. E. Kessler, "The Alpha 21264 Microprocessor," *MICRO*, 1999.

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

[18] B.-H. Lin, S.-H. Shieh, and C.-W. Wu, "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.

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

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

[21] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO*, 2007.

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

[23] G. Pekhimenko *et al.*, "Base-delta-immediate compression: practical data compression for on-chip caches," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 377–388.

[24] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, 1986, vol. 1, pp. 318–362.

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

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

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

[28] A. Sampson *et al.*, "Approximate storage in solid-state memories," in *MICRO*, 2013.

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

[30] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," in *PLDI*, 2014.

[31] A. Seznec, "Analysis of the o-geometric history length branch predictor," in *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, 2005, pp. 394–405.

[32] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *Journal of Instruction Level Parallelism*, vol. 8, pp. 1–23, 2006.

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

[34] A. Tamhane and D. Dunlop, "Statistics and data analysis," in *Prentice-Hall*, 2000.

[35] B. Thwaites *et al.*, "Rollback-free value prediction with approximate loads," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 493–494.

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