XMeter: Finding Approximable Functions and Predicting Their Accuracy

Riad Akram, *Member, IEEE*, Shantanu Mandal, *Member, IEEE*, and Abdullah Muzahid, *Member, IEEE*

Abstract—Approximate computing has significant potential to improve the efficiency of a computing system. Numerous techniques have been proposed in literature. Virtually, all of them require programmers to either experiment with every instance of a specific type of code region exhaustively to find approximable code regions or annotate such regions manually. Both approaches are error-prone and can lead to missed opportunities. Therefore, we propose *XMeter* to automatically find and quantify approximable code regions. XMeter, first, analyzes the application code statically using a novel algorithm based on memory location updates. Also, XMeter provides a deep learning-based predictor to predict the accuracy of the application when different code regions are approximated. Our proposed scheme does not require the programmer to experiment exhaustively for all possible error rates and types of approximation techniques. Moreover, the scheme does not require any domain knowledge and is not specific to any approximation technique. Therefore, it is general enough to be applicable for any approximation technique. We developed XMeter using LLVM and experimented with 10 applications. We analyzed 43 approximable functions and found 21 to be highly tolerant of errors. We validated our results using 4 well-known approximation techniques and showed that XMeter can predict an application's accuracy accurately.

Index Terms—Approximate computing, accuracy, static analysis, machine learning

1 Introduction

Approximate computing is a promising approach to improve performance and energy efficiency. This style of computing trades-off accuracy to gain those benefits. Approximate computing lends itself to many application domains that have an inherent tolerance towards inaccuracy. For example, video encoders are designed to give up perfect accuracy for faster encoding and smaller videos [1]. Machine learning algorithms are designed to produce probabilistic models that are not 100 percent accurate. Many scientific computations (e.g.,n body simulations [2]) are inherently inaccurate in that they are designed to produce an approximation to an ideal result.

Different approximation techniques have been proposed in literature [3]. For example, loop perforation is an approximation technique to skip (occasionally) some iterations of a loop in order to speed up the application [4]. Parrot transformation skips an entire function execution and instead, invokes a hardware neural network with the function's parameters to get an approximate result [5]. Paraprox replaces a data parallel kernel with a parameterized approximate kernel and tunes it at runtime to satisfy some quality metric [6]. There are techniques such as barrier [7] and lock approximation [8] that aim at reducing synchronization overhead by relaxing some semantics (e.g.,allowing a long waiting thread to skip a critical section). Besides these, there are techniques that relax the accuracy of data itself. For example, load value approximation learns general patterns of values of some data, generates approximate values using the learned patterns, and uses those values (instead of the original ones) in order to speed up the application [9]. Approximate storage proposes to store data approximately in solid state memories to reduce the complexity and wear out of the underlying memory cells [10]. Doppleganger cache groups tags of multiple similar cache lines with a single data block in order to reduce data storage [11]. Venkatagiri *et al.* [12] applied software approximation techniques for summarizing videos in unmanned aerial vehicles and showed that such techniques can have minimal effect on the application's resiliency to soft errors (of hardware).

Various approximation techniques differ from each other substantially. However, they all require the *ability to find code that is amenable to approximation*. To find such code, some existing proposals focus on every code region of a specific type (e.g.,loop, critical section, barrier etc.) [4], [7], [8], [13]. Others require the programmers to annotate some potential code based on domain expertise [5], [11], [14]. Programmers need to experiment with each instance of the potential code region exhaustively to find out the actual approximable code regions and their accuracy characteristics. We argue that such an approach is ad hoc, labor intensive, and not scalable to larger programs. Moreover, programmers need to repeat the process entirely for each new approximation technique. Thus, in order to make approximation techniques practical and usable, we need an approach that, given an application, automatically finds code regions amenable to approximation, can predict those regions' accuracy characteristics without

- *Riad Akram is with Intel Corporation, Santa Clara, CA 95054-1549 USA. E-mail: riad.cse@gmail.com.*
- *Shantanu Mandal and Abdullah Muzahid are with the Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843 USA. E-mail: {shanto, abdullah.muzahid}@tamu.edu.*

Manuscript received 19 Nov. 2019; revised 9 June 2020; accepted 10 June 2020. Date of publication 25 June 2020; date of current version 9 June 2021. (Corresponding author: Abdullah Muzahid.) Recommended for acceptance by R. Wang. Digital Object Identifier no. 10.1109/TC.2020.3005083

exhaustive experimentation, and does not rely on the details of any specific type of approximation technique.

Towards the aforementioned goals, we propose *XMeter*. XMeter analyzes application code statically to find out approximable code regions. The static analysis algorithm traverses application's call graph multiple times to find functions where any approximation technique can be easily applied. The analysis determines whether a function can be easily approximated or not by checking the memory locations that are updated inside the function. In addition, XMeter provides a deep learning based predictor that can be used to predict accuracy of the application when different code regions are approximated. The predictor can predict accuracy based on how the application behaves during a few (e.g., 3) error rates at the approximable code region. The application's behavior during a few error rates is called the Accuracy Signature of the approximable code region. Thus, the predictor does not require the programmer to experiment exhaustively for all possible error rates and all possible types of approximation techniques. In other words, XMeter provides a general predictor that can be used in the context of any approximation technique. In essence, XMeter provides programmers the ability to go over a set of approximable code regions and use the general predictor to determine the accuracy characteristics of those regions for any error rate (which in turn can help programmers to decide the most suitable code region to approximate). Thus, XMeter makes the following contributions:

- *Automatic Approach:* XMeter can find approximable functions automatically without any domain knowledge.
- *Novel Static Analyzer:* XMeter proposes a *novel* static analysis algorithm to find approximable functions. The analysis is based on call graph traversals and memory locations that are updated inside the functions.
- *General Predictor:* XMeter provides a *novel* deep learning based predictor that can predict accuracy of an application when a function is approximated. The predictor takes the Accuracy Signature of the function and predicts application accuracy at any error rate. The predictor is not specific to any particular approximation technique. XMeter is the *first* technique to predict accuracy for approximable functions using deep learning.

We developed XMeter using LLVM [15]. We evaluated it using a total of 10 applications - seven from AxBench [16] benchmark suit, Lulesh [17] and MILCmk [18] from Lawrence Livermore National Lab, and gzip [19]. We analyzed 43 functions in total. Our results showed that 21 functions are highly tolerant of errors i.e.,the application accuracy does not drop below 80 percent even at a high error rate in these functions. We also found 12 functions not to be tolerant of any error at all i.e.,the application accuracy drops below 50 percent even at a very low error rate. We validated XMeter's accuracy predictor against 4 well known approximation techniques - function memoization [20], loop perforation [4], task skipping [21], [22], and precision reduction [23]. Our validation results showed that XMeter is highly accurate and effective in predicting accuracy. especially for low to moderate approximation rates.

The rest of the paper is organized as follows: Section 2 describes some related work; Section 3 explains the main idea; Section 4 provides implementation details; Section 5 discusses the results, and finally, Section 6 concludes.

2 Related Work

Due to growing interest in the field of approximate computing, many techniques have been proposed to sacrifice accuracy for improved power, performance and scalability. Loop perforation [4], for example, skips some iterations of a loop to improve performance. Laurenzano *et al.* [13] use canary inputs that are representations of the full inputs and different approximation techniques to find the best approximation technique within an accuracy threshold. It does so automatically.

A number of work uses dynamic monitoring to keep accuracy within a limit. For example, SAGE [22] provides transparent automated approximate programs using offline compilation and run-time monitoring to keep output quality within a certain limit. In the offline phase, approximate programs are created using different optimizations targeting atomic operations, input data packing and thread fusion. Mitra *et al.* [24] use tunable approximation levels in approximable blocks of a program to find best performing approximation setting. But they have to rely on user to identify approximable blocks. Sui *et al.* [25] address control problem of tunable approximate programs, formulate the problem and use machine learning technique to create error and cost model and finally use this model to find best approximation settings that improved performance or power. Rumba [26] provides runtime detection and recovery modules to keep output quality in check with performance improvement and energy efficiency. Programmers have to annotate the code that will be mapped to approximation accelerator. Mahajan *et al.* [27] propose to classify dynamically whether approximating a function using neural networks can lead to unacceptable output quality. If so, the technique refrains from approximating the function. For the initial classification, a tabular predictor or a neural network can be used. Hashim *et al.* [28] propose ApproxHPVM, a portable compiler for accuracy-aware optimizations. The key idea is to map output accuracy limit to accuracy limits for individual approximable computations, autotune the code to satisfy the limits using a combination of approximation techniques and hardware. Our proposed approach could complement the autotuning phase of ApproxHPVM.

Some approximation techniques only target data. For exampe, Doppelgänger [11] presents a cache that approximates similar values across cache to reduce cache size and energy without performance degradation and too much accuracy loss. Programmers have to annotate programs to identify the data to be approximated. Miguel *et al.* [9] provide a load value estimator which provides estimated value when there is a load cache miss for improved performance, power with acceptable accuracy. Programmers have to annotate program to mark the data that can be approximated. Boston *et al.* [29] provide programming language support for programmers to specify probability of approximating a code region for improved performance within output accuracy threshold. EnerJ [14] abandons automation as

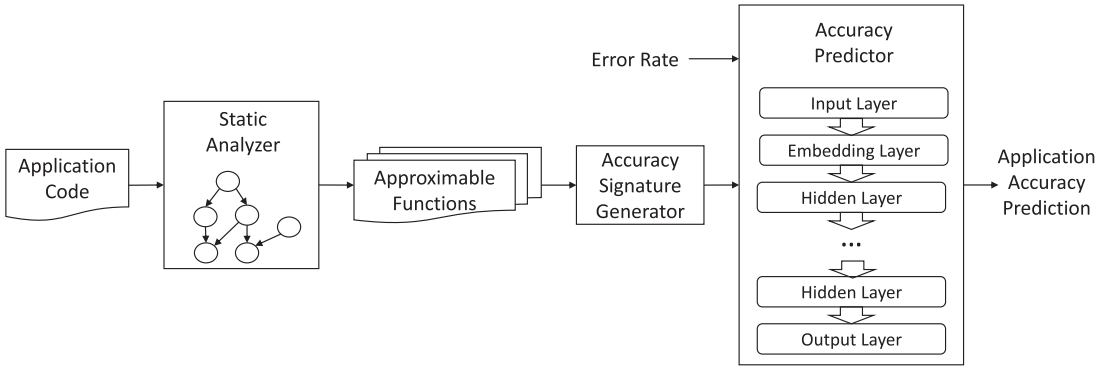

Fig. 1. System diagram of XMeter.

programmers need to specify manually and explicitly which part of programs can be approximated to gain energy efficiency with inaccurate computations. ExpAX [30] provides a framework to automate approximate programming by allowing programmers to implicitly declare which data and operations of programs are approximable by explicitly declaring the acceptable output quality. It proposes a static analysis based on the output expectation to determine safe-to-approximate operations. ACCEPT [31] provides a compiler analysis based on data type annotations about whether a data needs to be precise or not. The static analysis declares a code as approximable if no modification to precise data is visible outside. Thus, XMeter's static analyzer has resemblance with both ACCEPT and ExpAX. However, we argue that XMeter is more advanced than either of them because XMeter does not require any prior knowledge about data preciseness or expectations.

A large number of work target functions for approximation. Rely [32] requires programmers to manually specify functions with reliability requirement to be approximated. Axilog [33] provides a language annotations for designers to approximate hardwire design for energy efficiency with acceptable output loss. Esmaeilzadeh *et al.* [5] and Zhenghao *et al.* [34] propose to approximate certain functions of a program using a neural network based hardware. Yazdanbakhsh *et al.* [35] propose to use neural accelerator in GPU to approximate programs in order to gain performance by sacrificing accuracy. In the last two work, approximable code regions have to be annotated by programmers.

Approxilyzer [36] and gem5-Approxilyzer [37] are the closest proposals to our scheme. They provide a framework to quantify the impact of approximation automatically. They quantify the effect of a single bit flip in a register operand of an instruction. Different instructions are clustered together based on their data and control flow. The techniques experiment with one instruction per cluster to quantify the output quality. Any other instruction from the same cluster is predicted to have the same output quality. There are two major differences between these techniques and our proposed scheme, XMeter. First, both Approxilyzer techniques find approximable instructions, and are, therefore, ISA and architecture dependent. On the other hand, XMeter finds approximable functions by statically analyzing the code. Therefore, XMeter is architecture independent and provides a higher level insight about approximable functionalities of a program. Second, Approxilyzer techniques provide a coarse grained prediction about whether an approximable instruction leads to a detectable or silent data corruption. In contrast, XMeter can predict the exact output accuracy for a given error rate (thanks to the machine learning based accuracy predictor).

3 XMeter: Main Idea

XMeter consists of three major components - Static Analyzer, Accuracy Signature Generator and Performance Predictor. The overall system diagram is shown in Fig. 1. Next, we are going to elaborate on each component.

3.1 Static Analyzer

Static analyzer finds code regions whose computations can be approximated. Designing such an analyzer requires us to resolve two issues - (i) what is the granularity of code regions to analyze, and (ii) what should be the properties of such regions.

3.1.1 Granularity of Code Regions

A code region can be of different granularities. It can be a sequence of instructions, basic block, loop, critical section, function etc. We choose function as the code region granularity because it provides the right balance between scope and semantic boundary. Moreover, any approximation technique applicable to a smaller granularity (e.g.,loop, critical section etc.) can be easily covered by this choice. For example, if a particular function is approximable, we can apply techniques such as loop perforation, barrier approximation, lock approximation etc. to any loop, barrier or lock inside the function. If the function has multiples loops, barriers or locks, we can apply the approximation technique to all of them or some of them. Similarly, techniques that relax data accuracy can focus on the data accessed inside the function.

3.1.2 Properties of Code Regions

One of the goals of XMeter is to find functions whose computations can be easily approximated. Let us consider a function $\mathcal{F}$. An approximation technique essentially replaces $\mathcal{F}$ with $\mathcal{F}'$ such that $\mathcal{F}'$ computes similar results. Let us assume that $\mathcal{C}$ is the set of *communication points* through which $\mathcal{F}$ interacts with the rest of the program. If $\mathcal{F}$ is side effect free (i.e.,it does not modify any non-local data and communicates with its callers by returning some result),

then $\mathcal{C}$ contains only the return value. If $\mathcal{F}$ modifies reference parameters, then $\mathcal{C}$ also includes the reference parameters. If $\mathcal{F}$ modifies some arbitrary non-local data (that is not passed as a reference parameter), then $\mathcal{C}$ includes that non-local data too. If $\mathcal{C}$ is a large set containing many arbitrary non-local data, the effect of approximation can permeate to the rest of the program through any member of $\mathcal{C}$. Such a scenario makes it harder to constrain the application's accuracy degradation. Therefore, many techniques (such as parrot transformation [5]) approximate $\mathcal{F}$ only if $\mathcal{C}$ does not contain arbitrary non-local data. In other words, many techniques choose to approximate $\mathcal{F}$ if $\mathcal{C}$ contains only return value (in case of side-effect free function) or reference parameters of $\mathcal{F}$. Therefore, XMeter also focuses on such $\mathcal{F}$'s only.

3.1.3 Detailed Algorithm

The formal algorithm is shown in Algorithm 1. At the high level, the static analyzer of XMeter works by traversing the call graph $\mathcal{G}$ of a program multiple times. During the first traversal (Line 1-12), XMeter initializes various flags and sets. For each function $\mathcal{F}$, it keeps a flag, called Approximable ($\mathcal{A}_\mathcal{F}$). If $\mathcal{F}$ does any memory allocation, deallocation, blocking system call or I/O activities, XMeter initializes $\mathcal{A}_\mathcal{F} \leftarrow FALSE$ (Line 2-4). However, if $\mathcal{F}$ does not do any such operation, XMeter first populates the communication set $\mathcal{C}_\mathcal{F}$ (Line 8). $\mathcal{C}_\mathcal{F}$ contains all non-local variables that are being modified inside $\mathcal{F}$ as well as its return value. XMeter initializes $\mathcal{A}_\mathcal{F} \leftarrow TRUE$ if $\mathcal{F}$ does not write to any non-local variable other than its reference parameters (Line 10). Otherwise, XMeter initializes $\mathcal{A}_\mathcal{F} \leftarrow FALSE$ (Line 12). After the first traversal, other traversals are done in Depth First Search (DFS) order to reduce the total number of traversals. During those traversals, XMeter updates $\mathcal{A}_\mathcal{F}$ by doing a logical AND of approximable flags of all the functions called by $\mathcal{F}$ (Line 19). In other words, XMeter keeps a function as approximable if each of its called functions is also approximable. Otherwise, XMeter updates the flag to make $\mathcal{F}$ non-approximable. Communication set of $\mathcal{F}$ is also updated similarly to include the communication set of each called function (Line 20). XMeter continues to traverse the call graph until there is no change in any approximable flag (Lines 21-22). A rough sketch to prove that Algorithm 1 is sound and complete is provided in Section 3.4.

Algorithm 1. Find Approximable Functions of a Program

Input: Call graph $\mathcal{G}$
Output: Approximable functions
1 **for** *Each* $\mathcal{F}$ *in* $\mathcal{G}$ **do**
2 **if** $\mathcal{F}$ *contains memory (de)allocation, blocking system call or I/O* **then**
3 $\mathcal{A}_\mathcal{F} \leftarrow FALSE$;
4 $\mathcal{C}_\mathcal{F} \leftarrow \emptyset$;
5 **else**
6 $\mathcal{P}_\mathcal{F} \leftarrow$ Reference parameters of $\mathcal{F}$;
7 $\mathcal{R}_\mathcal{F} \leftarrow$ Return value;
8 $\mathcal{C}_\mathcal{F} \leftarrow$ Variables of non local stores and return value;
9 **if** $\mathcal{C}_\mathcal{F} \setminus (\mathcal{P}_\mathcal{F} \cup \mathcal{R}_\mathcal{F}) = \phi$ **then**
10 $\mathcal{A}_\mathcal{F} \leftarrow TRUE$;
11 **else**
12 $\mathcal{A}_\mathcal{F} \leftarrow FALSE$;
13 $terminate \leftarrow$ FALSE;
14 **while** $terminate = FALSE$ **do**
15 $terminate \leftarrow$ TRUE;
16 **for** *Each* $\mathcal{F}$ *in* $\mathcal{G}$ *in DFS order* **do**
17 $\mathcal{CL}_\mathcal{F} \leftarrow$ Functions called from $\mathcal{F}$ in $\mathcal{G}$;
18 $\mathcal{A'}_\mathcal{F} \leftarrow \mathcal{A}_\mathcal{F}$;
19 $\mathcal{A}_\mathcal{F} \leftarrow \mathcal{A}_\mathcal{F} \wedge (\bigwedge_{\mathcal{H} \in \mathcal{CL}_\mathcal{F}} \mathcal{A}_\mathcal{H})$;
20 $\mathcal{C}_\mathcal{F} \leftarrow \mathcal{C}_\mathcal{F} \cup (\bigcup_{\mathcal{H} \in \mathcal{CL}_\mathcal{F}} (\mathcal{C}_\mathcal{H} \setminus \mathcal{R}_\mathcal{H}))$;
21 **if** $\mathcal{A'}_\mathcal{F} \neq \mathcal{A}_\mathcal{F}$ **then**
22 $terminate \leftarrow terminate \wedge FALSE$;
23 **return**

3.2 Accuracy Signature Generator

Static analyzer of XMeter provides a list of approximable functions. One of the goals of XMeter is to predict application accuracy for any arbitrary error rate at an approximable function $\mathcal{F}$. In order to facilitate the prediction process, we introduce the notion of Accuracy Signature ($AS_\mathcal{F}$). Intuitively, $AS_\mathcal{F}$ acts as a hint about how approximation of $\mathcal{F}$ impacts application accuracy by providing accuracy corresponding to a few error rates. Accuracy predictor takes $AS_\mathcal{F}$ to predict accuracy corresponding to any arbitrary error rate at $\mathcal{F}$. With this view in mind, we can define $AS_\mathcal{F}$ as a vector $[a_i]_{i=1}^n$, where a_i is the application accuracy when $\mathcal{F}$ is approximated at an error rate e_i. n is kept sufficiently small (e.g., $n = 3$ is found to be adequate in our experiments in Section 5.4) so that XMeter does not amount to an exhaustive experimentation technique at all possible error rates.

To generate $AS_\mathcal{F}$, XMeter emulates the behavior of an approximation technique at $\mathcal{F}$. Since $\mathcal{F}$ interacts with the rest of the program with its communication set $\mathcal{C}_\mathcal{F}$, the effect of approximation can be emulated by injecting error at each of the communication points in $\mathcal{C}_\mathcal{F}$. This emulation technique makes XMeter a general approach that can be used in the context of any approximation technique. Error injection is done by adding some instrumentation code for each member of $\mathcal{C}_\mathcal{F}$. Recall that $\mathcal{C}_\mathcal{F}$ contains return value and/or reference parameters that are modified inside $\mathcal{F}$. In case of the return value, an error is injected at the return statement whereas for a reference parameter, an error is injected at the last update of that parameter. Last update is determined by the instrumentation code at runtime. To inject error e_i at a value v, XMeter replaces v with $v \times (1 \pm e_i)$. The final accuracy for e_i is taken as the average from both error points. In our experiments, error rate 1, 50 and 100 percent are found to generate effective $AS_\mathcal{F}$ that can boost prediction ability of Accuracy Predictor. Thus, XMeter records application accuracy corresponding to low (1 percent), medium (50 percent), and high (100 percent) error rate at $\mathcal{F}$ to generate $AS_\mathcal{F}$. Finally, to ensure execution of error injection code, software test fuzzing can be used [38].

3.3 Accuracy Predictor

Accuracy predictor is essentially a deep neural network with an input layer followed by an embedding layer, h hidden layers and an output layer (Fig. 2). The predictor takes $AS_\mathcal{F}$ and an error rate as inputs. Each input number is passed through the embedding layer which converts it into a d dimensional vector. The purpose of the converting each input number into a d dimensional vector is to place similar signature (or error rate) into nearby points in a d dimensional

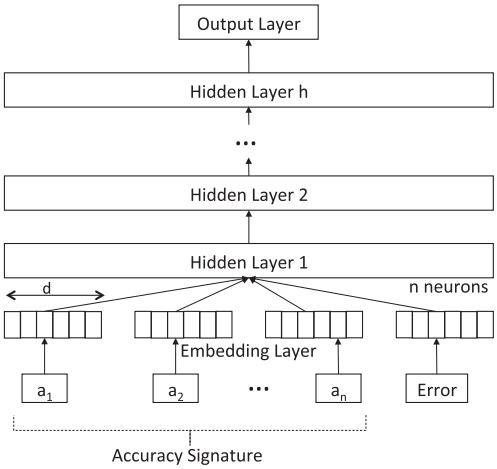

Fig. 2. Accuracy predictor of XMeter.

space. Similar technique has been used in natural language processing to capture semantic similarity among words [39]. d dimensional vectors are passed through h hidden layers and finally an output layer. Each hidden layer consists of n neurons. Since output layer produces a single value (i.e., accuracy prediction), it contains a single neuron. Hidden layer neurons use hyperbolic tangent activation function [40] while the output layer neuron uses ReLU activation function. We choose these activation functions because they provided the best accuracy in our experiments. Since predicting application accuracy can be considered as a regression problem, XMeter uses Mean Squared Error (MSE) as the loss function for the predictor. Note that the choice of MSE as a loss function does not imply that an application's accuracy degradation needs to be expressed in MSE. In fact, the choice of loss function is orthogonal to the application's accuracy metric. In order to determine the value of h, n, d, we exhaustively experiment with different combinations of values and pick the network architecture that provides the best accuracy. More on this in Section 5.4.

3.4 Proof Sketch

Here, we are going to show that Algorithm 1 is both sound and complete.

Theorem 1. Algorithm 1 is sound i.e., if a function calls a non-approximable function directly or indirectly, the function will be marked as non-approximable.

Proof. We can prove this theorem using proof by contradiction. Suppose, we have a call graph as shown in Fig. 3. Here, *Root* represents the main function, and V_0 represents another function that calls function V_n through a chain of other functions such as V_1. For the sake of contradiction, let us assume that the Algorithm 1 marks V_n as

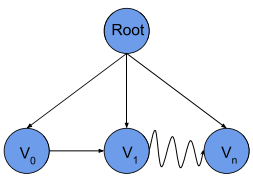

Fig. 3. An example call graph for Theorem 1.

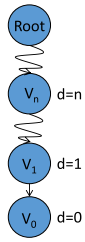

Fig. 4. An arbitrary call graph with information about leaf node.

non-approximable (i.e., $\mathcal{A}_{V_n} = FALSE$) but V_0 as approximable (i.e., $\mathcal{A}_{V_0} = TRUE$). Since V_0 can be marked as approximable only if all of its called functions are approximable (Line 17-19 in Algorithm 1), V_i is approximable. Following the same argument, all of the called functions of V_1 should be marked as approximable. If we follow the call chain from V_1 to V_n and apply the same argument, V_n should be marked as approximable. However, this contradicts our initial assumption that V_n is non-approximable. Thus, the algorithm must not mark V_0 as approximable if one of its called functions is non-approximable. In other words, all the functions marked as non-approximable by Algorithm 1 are correctly marked so. Therefore, the remaining functions that are marked approximable are correctly marked too. □

Theorem 2. Algorithm 1 is complete i.e., if an approximable function exists, the algorithm will find it.

Proof. We can prove the completeness of Algorithm 1 using induction. Let us consider the call graph in Fig. 4. As before, *Root* represents the main function. Assume that for a node, d represents the longest distance (excluding loop induced repetition) of that node from a leaf node in the call graph. Thus, for a leaf node V_0, we will have $d = 0$. Also assume that V_1 and V_n have $d = 1$ and $d = n$ respectively. □

Let us consider an arbitrary approximable function. It could be a leaf or internal node in the call graph. In other words, an approximable function can have $d = 0$ or $d > 0$. Without loss of generality, let us assume that V_0 with $d = 0$ is an approximable function. Algorithm 1 will mark V_0 as approximable in Line 6-10 during the initial loop. Since V_0 is a leaf node, it will remain approximable during the iterations of the second loop. Thus, Algorithm 1 will find approximable function V_0.

Now, we will prove that Algorithm 1 can find an approximable function even if its $d > 0$. Without loss of generality, assume that both V_1 and V_n are approximable (i.e., all of their called functions are approximable too). During the traversal of the call graph in DFS order, Line 17-19 will check if the called functions of V_1 are approximable. Since $d = 1$ (i.e., V_1 calls some leaf functions directly), Algorithm 1 will mark it as approximable during the first iteration of the second loop. Following same logic, if an approximable functions has $d = 2$, Algorithm 1 will mark it as approximable by at most 2 iterations of the second loop. Continuing this way, for the approximable function V_n with $d = n$, Algorithm 1 will mark it as approximable by at most n iterations of the second loop. Thus, Algorithm 1 can find an approximable function for any value of d.

4 Implementation Details

Both the static analyzer and accuracy signature generator of XMeter are written in C++ using LLVM [15]. Given an application, the static analyzer provides a list of approximable functions and their communication sets. The accuracy signature generator iterates over each approximable function and injects errors at a number of different rates in each communication point. This results in a number of different versions of the source code - one for each approximable function and error rate. XMeter compiles and executes each such version and records the corresponding output accuracy. Thus, XMeter collects accuracy signature data for each approximable function.

The accuracy predictor is written in Keras [41] with TensorFlow [42] as the back end. In order to generate training data for the accuracy predictor, we randomly split all approximable functions (found by the static analyzer) into training and testing set. The training set contains 80 percent approximable functions while the rest are in the testing set. For each approximable function in the training set, XMeter injects errors (similar to that of accuracy signature generator) from 1 to 100 percent with 1 percent increment. XMeter collects application's output accuracy corresponding to each error rate. Thus, if we have x approximable functions in the training set, we collect $100x$ accuracy data, which are, then, used to generate the training data for the predictor. After the accuracy predictor trains on the training data, validation is done to asses the prediction accuracy using the testing set approximable functions. We plan to release our code in a public repository after the paper is published.

Current implementation of XMeter excludes any function, $\mathcal{F}$, that modifies arbitrary non-local data such as global and static data. Such a function can be easily supported by making the following changes. *First*, XMeter needs to use points-to analysis [43] to determine the complete communication set, $\mathcal{C}_{\mathcal{F}}$. Note that, $\mathcal{C}_{\mathcal{F}}$ will include the non-local communication points of the callee functions too. Lines 5-12 of Algorithm 1 will no longer be required. *Second*, XMeter needs to inject errors at each of the communication points of $\mathcal{C}_{\mathcal{F}}$ during the error injection process. The rest of the components of XMeter can be applied without any modification.

5 Evaluation

In this section, we will first outline the experimental setup followed by the effectiveness of our static analyzer. Finally, we will show some results related to the accuracy predictor of XMeter.

5.1 Experimental Setup

We used ten applications for evaluation. Seven of them are from AxBench [16] benchmark suite, two applications (Lulesh and MILCmk) are from Lawrence Livermore National Lab [17], and the last one, Gzip [19], is an open source application. AxBench is widely used in approximate computing studies [5], [44]. Lulesh and MILCmk are high performance computing benchmarks used in large supercomputers. Lastly, Gzip is a large utility application used for compressing files. Accuracy is calculated by comparing the final results of an application with its golden results (i.e., results obtained from the original program). The applications and their accuracy metrics are shown in Table 1. Similar metrics have been used in prior work [45]. For each application, we used two inputs to demonstrate the generality of our findings. Default inputs of AxBench are referred to as *Input1* whereas *Input2* refers to manually generated second set of inputs. For Lulesh, we used *problem size 13* as Input1 and *problem size 7* as the Input2. For MILCmk, we used array lengths from 2^6 to 2^8 as Input1 and from 2^6 to 2^11 as Input2. For Gzip, we used two random text files - small one as Input1 and the other as Input2. All experiments were performed on a 7 node cluster with 100 cores and 345 GB memory. For training the accuracy predictor, we used a learning rate of 0.001 with Adam optimizer. We trained up to 200 epochs. We used 3 hidden layer neural network with 20 dimensional embedding layer in the front and an output layer at the end as our accuracy predictor. Each hidden layer contains 512 neurons. We used 3 error rates to generate accuracy signatures. These parameters are chosen experimentally as shown in Section 5.4.

TABLE 1
Accuracy Metric Used for Applications

Application	LOC	Description	Accuracy Metric
Sobel	325	Sobel edge detector	Image difference
Kmeans	495	K-means clustering of pixels	Image difference
Jpeg	1,320	Image encoding	Image difference
Blackscholes	378	Financial modeling	Mean relative error
FFT	197	Fast Fourier Transform	Mean relative error
Inversek2j	137	Inverse kinematics	Mean relative error
Jmeint	741	Triangle intersection detection	Miss rate
Lulesh	7,240	Hydrodynamics simulation	Mean relative error
MILCmk	5,000	Quantum lattice	Vector difference
Gzip	8,614	Compression algorithm	Compression ratio

5.2 Effectiveness of Static Analyzer

XMeter found a total of 43 approximable functions in 10 applications. A brief characterization of each application and its approximable functions is provided below. Since different inputs tend not to affect the characterization much, we omit input specificity in most cases.

5.2.1 Sobel

XMeter found two approximable functions - `convolution` and `makeOpMem` (Figs. 5a and 5b). `convolution` is a side effect free function that calculates the convolution of each pixel. It shows almost a linear relationship between accuracy and error. Therefore, this function can tolerate low-to-moderate errors. The other approximable function, `makeOpMem`, is not a side effect free function. It makes operational memory for each pixel using the surrounding pixels. Accuracy for `makeOpMem` tends to decrease with increased error rate with some irregularities. However, the accuracy plot of `makeOpMem` is steeper than that of `convolution`. Therefore, `makeOpMem` can tolerate less error than `convolution`.

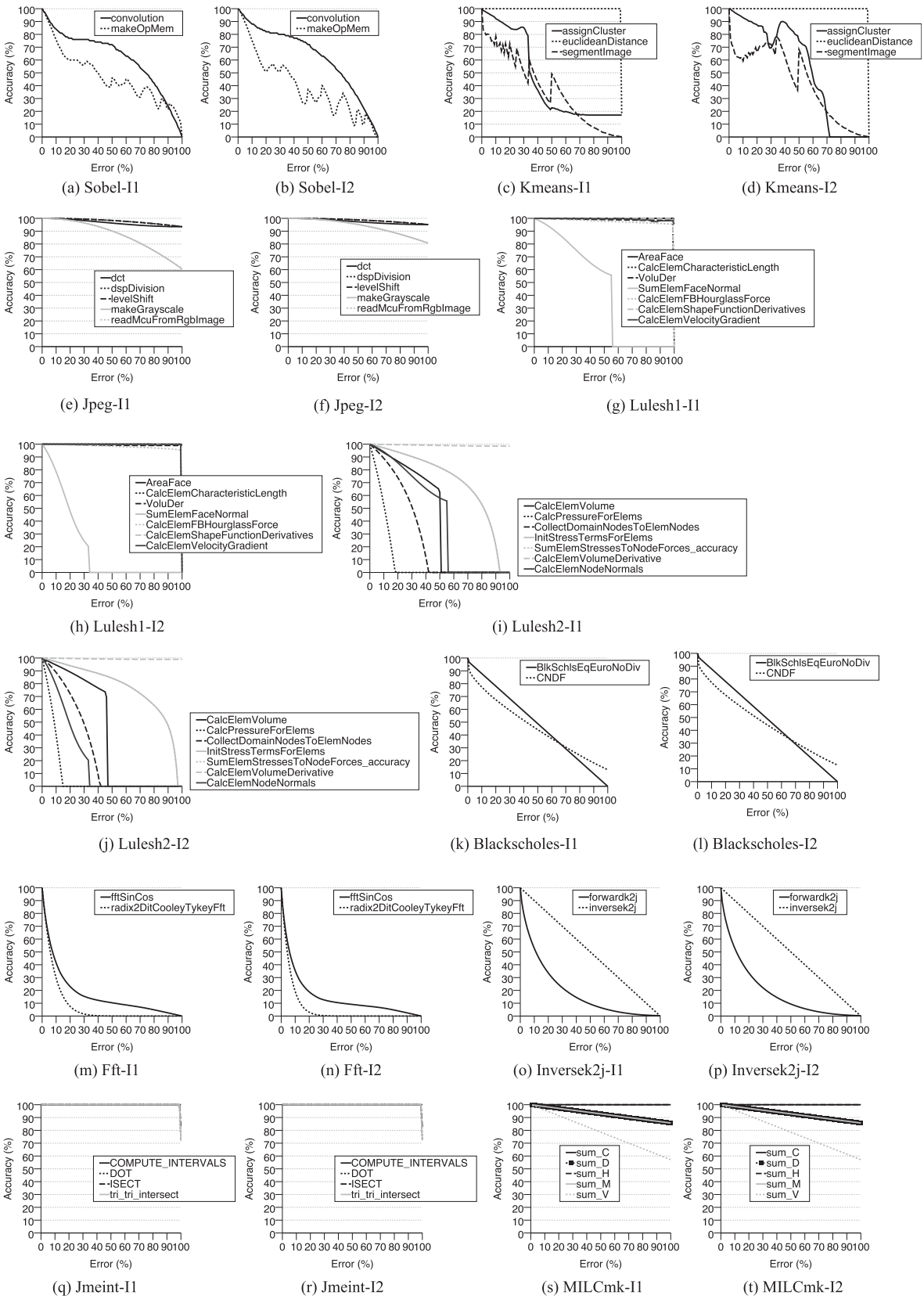

Fig. 5. Accuracy versus error graphs for all approximable functions. I1 and I2 indicates input1 and input2, respectively.

5.2.2 Kmeans

XMeter found three approximable functions - `euclideanDistance`, `assignCluster`, and `segmentImage` (Figs. 5c and 5d). `euclideanDistance` calculates euclidean distance of a pixel from the center of a cluster and suffers no accuracy loss except at 100 percent error rate. The other two approximable functions call only side effect free functions and modify their parameters. While `assignCluster` has high accuracy

(above 80 percent) for up to 30 percent error rates, segmentImage suffers from high accuracy loss (up to 40 percent) even at a low rate. Thus, cluster assignment and euclidean distance calculation can be approximated to a high degree without compromising the final accuracy.

5.2.3 Jpeg

XMeter found five approximable functions in Jpeg application (Figs. 5e and 5f). Among them, dspDivision is the only side effect free function. Every function except makeGrayscale shows almost fixed accuracy across different error rates. makeGrayscale can cause accuracy degradation up to 40 percent for input1 and 20 percent for input2. So, in general, the functions of Jpeg have a high tolerance towards errors.

5.2.4 Lulesh

XMeter found 14 approximable functions (Figs. 5g, 5h, 5i and 5j). CalcElemFBHourglassForce is the largest function (in terms of lines of code). The function calculates hourglass force for each element which is then applied to each node's forces. Intuitively, such a function should tolerate inaccuracy. This intuition is supported in the graph also. The function's accuracy ranges from 99 to 95 percent across different error rates. All functions except SumElemFaceNormal exhibit similar behavior. SumElemFaceNormal has a drastic drop in accuracy even at a low error rate. Clearly, this function cannot tolerate any error at all. Among the 7 approximable functions shown in Figs. 5i and 5j, SumElemStressesToNodeForces and CalcElemVolumeDerivate have high tolerance towards errors. InitStressTermsForElems can suffer from up to 20 percent loss in accuracy for up to 50 percent error rate. Thus, this function has a moderate tolerance towards errors. Other functions cannot tolerate error at all and suffer from rapid degradation in accuracy.

5.2.5 Blackscholes

XMeter found two approximable functions (Figs. 5k and 5l). CNDF is a side effect free function that calculates cumulative distribution function of the standard normal distribution. At a low error rate, it has high accuracy i.e., above 80 percent. At higher error rates, accuracy drops to 20 percent. So, this function cannot tolerate much error. BlkSchlsEqEuroNoDiv is another approximable function which calls only side effect free functions. Accuracy drops linearly for different error rates. This implies that this function can tolerate only low error rates.

5.2.6 FFT

XMeter found two approximable functions - fftSinCos and radix2DitCooleyTykeyFft (Figs. 5m and 5n). fftSinCos modifies two of its parameters. It calculates *twiddle factor* in Fast Fourier Transform algorithm. The accuracy quickly drops to below 20 percent. This is true for both inputs. So, this function cannot tolerate any error without significantly degrading the accuracy. radix2DitCooleyTykeyFft is the second approximable function that calls side effect free functions and modifies five of its parameters. Similar to the previous function, this function also causes a rapid drop of accuracy even at a very low error rate.

5.2.7 Inversek2j

XMeter found two approximable functions - inversek2j and forwardk2j (Figs. 5o and 5p). None of them are side effect free functions. inversek2j calculates starting point of a joint system based on the given end position. At a low error rate, it has an accuracy of 90 percent. The accuracy drops to 0 percent at higher error rates. Accuracy loss is almost linear. So, this function can tolerate low error rates. The other approximable function, forwardk2j, quickly cause an accuracy drop to 0 percent. So, this function is not tolerant to any error.

5.2.8 Jmeint

XMeter finds four approximable functions - COMPUTE_INTERVALS, DOT, ISECT, and tri_tri_intersect (Figs. 5q and 5r). Injecting errors to those functions does not cause any drop of accuracy except at 100 percent error rate. So, these functions are highly tolerant to errors.

5.2.9 MILCmk

XMeter finds 5 approximable functions (Figs. 5s and 5t). Each of the functions are computing sums used in quantum lattice calculation. Thus, intuitively, they should be able to tolerate significant inaccuracy. This is clearly demonstrated in the graphs. Except for sum_v, others are highly error tolerant. sum_v can tolerate low to moderate errors.

5.2.10 Gzip

We found 4 approximable functions in Gzip - bi_reverse, gen_codes, make_table, and updcrc. However, none of them are error tolerant. Even at a low error rate, Gzip produces a compressed file which cannot be decompressed. In other words, Gzip's outputs are completely corrupted.

Summarized Results

- We found 43 approximable functions.
- Among them, 21 functions are highly error tolerant (i.e., they have over 80 percent accuracy even at a high error rate) and 12 functions are not tolerant to error at all (i.e., they have less than 50 percent accuracy even at a low to moderate error rate).
- Different inputs do not have any effect on how an approximable function impacts application's accuracy. So, approximation behavior is largely input independent.

5.3 Effectiveness of Accuracy Predictor

5.3.1 Accuracy in Testing Set

We randomly choose 7 approximable functions (i.e., 20 percent of all approximable functions) as testing set. After training, we used accuracy predictor to predict for each of those functions. Fig. 6 shows the results. Accuracy predictor is quite accurate for 4 functions - AreaFace, CNDF, Dot, and dspDivision. The prediction error is 20 percent or less for these functions.

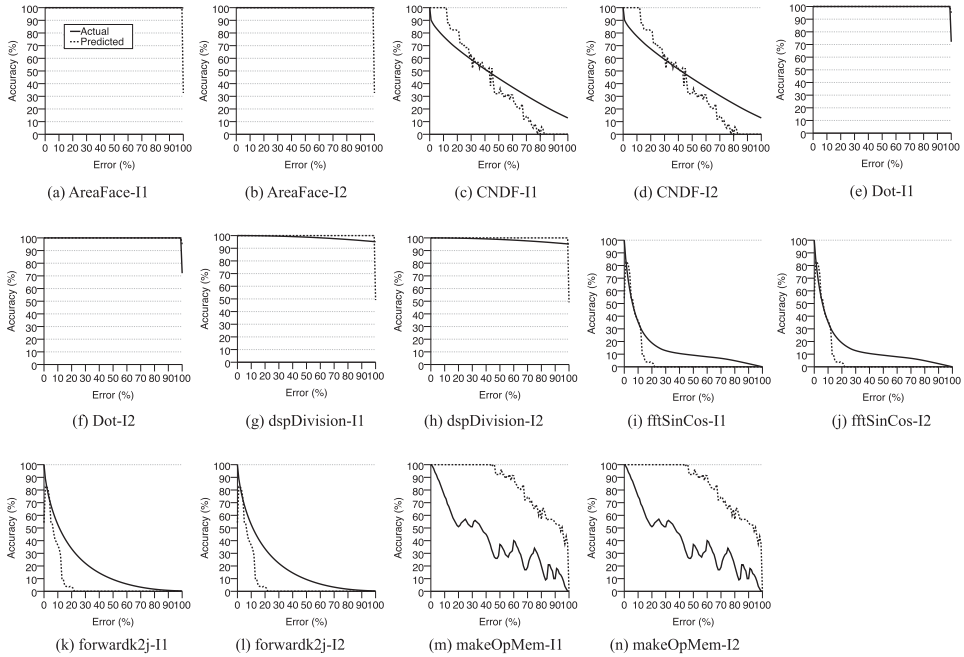

Fig. 6. Actual and predicted accuracy for testing set functions. I1 and I2 indicates input 1 and 2, respectively.

For `fftSinCos` and `forwardk2j`, the predicted graph follows the actual accuracy graph during lower error rates. At higher error rates (such as 30 percent or more), prediction error is higher (as high as 30 percent). Among the seven functions, `makeOpMem` suffers from the largest prediction error (as high as 70 percent). For all functions, the prediction accuracy remains unchanged across inputs. This is expected since accuracy behavior is largely independent of inputs.

5.3.2 Validation With Approximation Techniques

To validate XMeter's prediction ability, we experimented with 4 approximation techniques - function memoization [20], loop perforation [4], task skipping [21], [22], and precision reduction [23]. For each technique, we applied high, medium, and low rates of approximation to the most error tolerant approximable function of each application and calculated the application's output accuracy. Table 2 shows the selected functions. We compared the calculated accuracy against XMeter's predicted accuracy. Although we experimented with both inputs, we show results only for Input1 because Input2 produces similar results.

TABLE 2
Functions Used for Validating XMeter's Prediction Ability

Application	Function
Sobel	`Convolution`
Kmeans	`euclideanDistance`
Jpeg	`makeGrayScale`
Blackscholes	`BlkSchlsEqEuroNoDiv`
FFT	`fftSinCos`
Inversek2j	`inversek2j`
Jmeint	`DOT`
Lulesh	`VoluDer`
MILCmk	`sum_v`

Function Memoization. To memoize a function $\mathcal{F}$ at a rate of $r\%$, we record the return value (or reference parameters' values, whichever is appropriate) from the last invocation of $\mathcal{F}$, randomly $r\%$ of the times. During the next invocation of $\mathcal{F}$, the program uses the recorded value(s) instead of executing the function. We calculate how much error is injected due to memoization and what the final accuracy of the application is. We denote 90, 40, and 10 percent as high, medium, and low rates of memoization. Figs. 7a, 7b, and 7c show the results for different memoization rates. Except for Sobel and Kmeans, XMeter is quite accurate in predicting accuracy (less than 20 percent prediction error) for different memoization rates. Prediction is more accurate at low to moderate memoization rates. Sobel, Kmeans, and MILCmk suffer from higher prediction error especially at high memoization rate.

Loop Perforation. To perforate a loop at a rate of $r\%$, we randomly drop $r\%$ iterations of the loop. We calculate error injected due to the perforation and the corresponding accuracy of the application. Note that loop perforation is possible only if the approximable function has a loop. We consider 10, 40, and 90 percent perforation rates as low, medium, and high respectively. We are able to perforate loops in the error tolerant functions of 5 applications. Blackscholes, Inversek2j and Jmeint does not have any loop in the most error tolerant approximable functions. Figs. 7d, 7e, and 7f show the results. At low and moderate perforation rate, XMeter predicts fairly accurately (with less than 20 percent error) except for FFT and Sobel. At high perforation rates, we observe accurate prediction in 3 applications - Sobel, Kmeans, and Jpeg.

In summary, XMeter's prediction is quite accurate at low and moderate perforation rates.

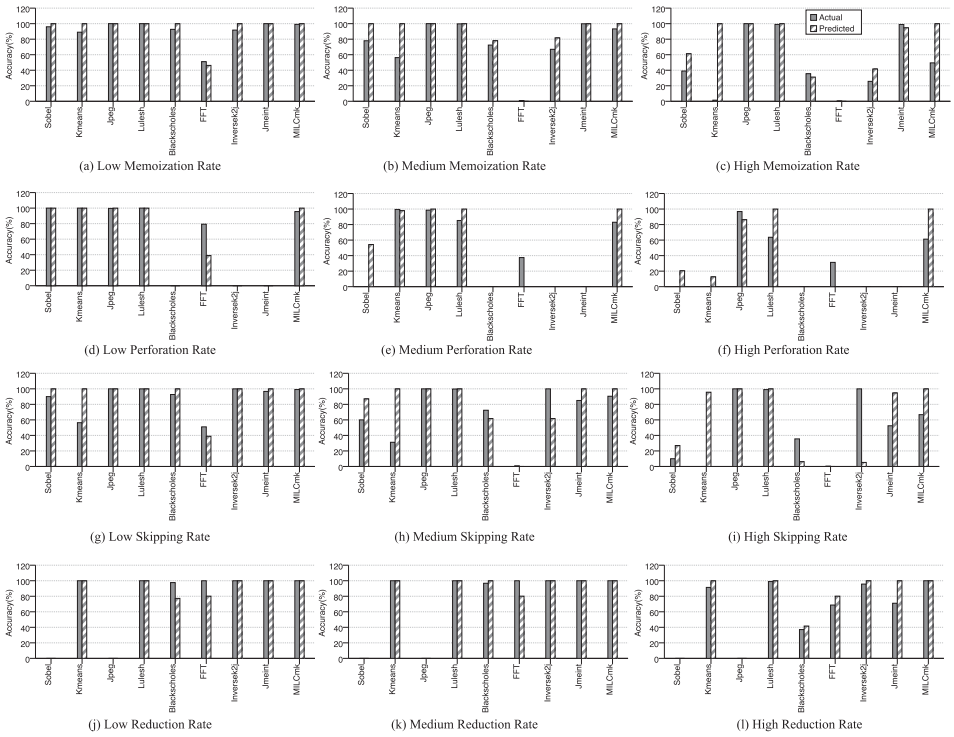

Fig. 7. Graphs validating XMeter's prediction against different approximation techniques.

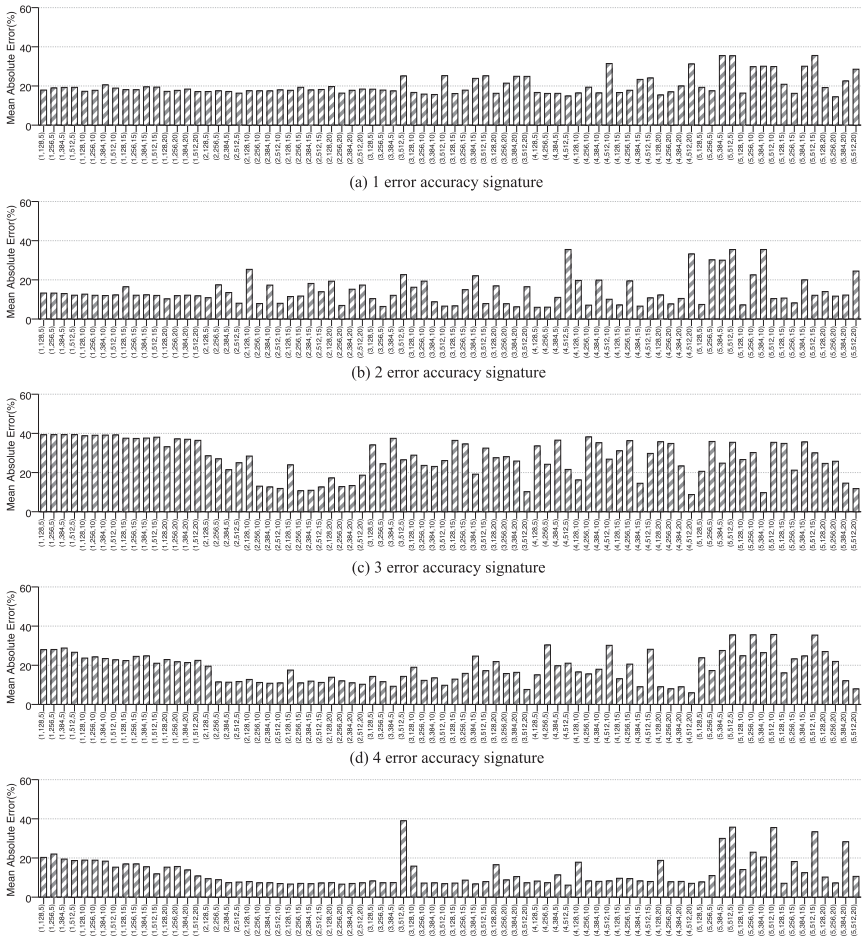

Fig. 8. Mean absolute error in prediction for different neural network and accuracy signature configurations. Each network configuration is shown as (hidden layers, neurons per hidden layer, dimension).

Task Skipping. We consider a function as a task. We can skip a task randomly at a rate of $r\%$. As before, we calculate error injected due to skipping and the corresponding final accuracy of the application. We consider 10, 40, and 90 percent skipping rates as low, medium, and high respectively. Figs. 7g, 7h, and 7i show the results. At low and medium rates, XMeter predicts fairly accurately (with less than 20 percent difference) except for Sobel, Kmeans and Inversek2j. At high skipping rate, XMeter predicts three of the applications (i.e., Sobel, Jpeg, and Lulesh) quite accurately while others suffer from a significant prediction error.

Precision Reduction. We experimented by reducing precision of floating point numbers. More specifically, we reduce the number of digits after the decimal point. We denote 9, 4, and 1 digit after the decimal point as low, medium and high reduction of precision respectively. As before, we calculate injected error and final accuracy of each application. Figs. 7j, 7k, and 7l show the results for precision reduction. Since the error tolerant functions of Sobel and Jpeg do not return any floating point number, we do not apply this technique to those two functions. XMeter has a highly accurate prediction for every program at every reduction rate except for Jmeint at high reduction rate. For Jmeint, prediction error is around 25 percent.

Summarized Results

- XMeter predicts quite accurately for low to medium approximation rates. The results are consistent across different approximation techniques and inputs.
- XMeter suffers from high prediction errors at high approximation rate. As before, this is true regardless of approximation techniques and inputs.

5.4 Neural Network and Accuracy Signature

To determine the best configuration for our accuracy predictor, we experiment with both the neural network and accuracy signature configuration. We vary the hidden layers (h) from 1 to 5, the neurons per hidden layer (n) from 128 to 512 (with 128 increment) and the embedding layer dimension (d) from 5 to 20 (with 5 increment). This leads to a total of 80 different network configurations. We vary the number of error rates in the accuracy signature between 1, 2, 3, 4, and 11. The error rates are (1), (1, 10), (1, 50, 100 percent), (1, 25, 50, 100), and (1, 10, 20, 30, ...,100 percent) respectively. We trained and tested each configuration with the same training and testing set used in Section 5.3.1. The mean absolute error in predicting testing set is shown in Figs. 8a, 8b, 8c, 8d and 8e. Each network is shown as (h, n, d). We observe that more error rates in the accuracy signature leads to higher prediction accuracy except in smaller networks. This is because larger networks suffer from overfitting when there is a lower number of error rates. We should note that more error rates imply higher number of profiling runs for constructing the accuracy signature. Therefore, to strike a balance between accuracy and profiling, we choose 3 error rates as our default accuracy signature configuration (Fig. 8c). In that case, network (3, 512, 20) and (4, 512, 20) provide the lowest mean absolute error which is 10.2. We choose (3, 512, 20) as the default configuration of the accuracy predictor.

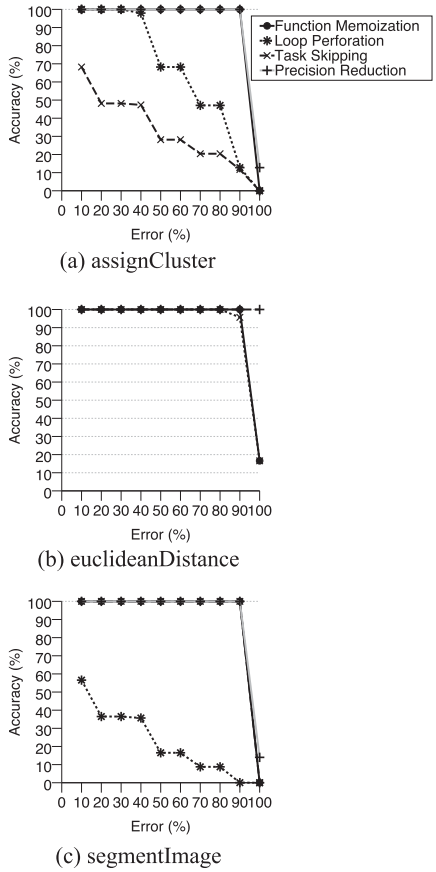

Fig. 9. XMeter can be used to select an approximation technique.

5.5 Approximation Technique Selection

XMeter can be an effective tool in selecting approximation techniques. To demonstrate such a use, we show the predicted accuracy for different approximation techniques for the approximable functions of Kmeans in Fig. 9. We draw several conclusions. *First,* function memoization can cause the least accuracy degradation. *Second,* `euclideanDistance` is the most tolerant of errors irrespective of approximation techniques., given a target accuracy, say at least 80 percent, we can decide which function should be approximated and how. For example, `segmentImage` is the largest function among the three and it can be approximated using function memoization at a rate as high as 90 percent.

5.6 Time

Training data generation requires collecting accuracy profiles from a number of approximable functions at multiple error rates. Thus, multiple invocations of those functions are needed. Fortunately, this step is done only once. In our experiments, this step takes around a day to finish. The accuracy predictor takes less than 15 minutes to train. After training is complete, the accuracy predictor can be used just by generating an approximable function's accuracy signature. Accuracy signature generation and accuracy prediction takes less than a minute to finish. Thus, XMeter can be an effective tool to get a quick estimate about approximation opportunity in an application.

6 Conclusion

Approximate computing has a significant potential to improve the power, performance, and scalability of a computing system. However, prior work requires exhaustive experimentation with every instance of a specific type of code region for each approximation technique or manual annotation of approximable code regions (which requires domain expertise). Both approaches pose a non-trivial impediment in widespread adoption of approximation techniques. Therefore, we propose *XMeter* to automatically identify code sections where approximation can be used and predict corresponding accuracy. XMeter first identifies potential approximable functions using a novel static analysis algorithm and provides a deep learning based predictor to predict accuracy for any arbitrary error rate. We analyzed 43 approximable functions in 10 applications. We found 21 functions to be highly tolerant of errors whereas 12 functions are found to be not tolerant of any error at all. Other functions have moderate tolerance towards errors. Our validation results showed that XMeter is accurate in predicting accuracy especially at low to moderate approximation rates. Thus, XMeter is a fast, effective and promising approach. We believe that by relieving programmers the burden of finding approximable code regions, XMeter can pave the way of widespread adoption of approximation techniques.

Acknowledgments

The authors would like to thank the reviewers for spending their valuable time on this article and provide useful feedback. The authors would also like to thank the members of PALab group. The collaboration and discussion among the group members were invaluable for this project. Finally, this work was supported by the startup package provided by Texas A&M University and NSF under Grant No. 1652655.

References

[1] x264. [Online]. Available: http://www.videolan.org/x264.html
[2] S. Aarseth, *Gravitational N-Body Simulations: Tools and Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2003.
[3] T. Moreau *et al.*, "A taxonomy of general purpose approximate computing techniques," *IEEE Embedded. Syst. Lett.*, vol. 10, no. 1, pp. 2–5, Mar. 2018.
[4] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proc. 19th ACM SIGSOFT Symp. and the 13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 124–134.
[5] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2012, pp. 449–460.
[6] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *Proc. 19th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2014, pp. 35–50.
[7] M. C. Rinard, "Using early phase termination to eliminate load imbalances at barrier synchronization points," in *Proc. 22nd Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst. Lang. Appl.*, 2007, pp. 369–386.
[8] R. Akram, M. M. Ul Alam, and A. Muzahid, "Approximate lock: Trading off accuracy for performance by skipping critical sections," in *Proc. IEEE 27th Int. Symp. Softw. Rel. Eng.*, 2016, pp. 253–263.
[9] J. S. Miguel, M. Badr, and N. E. Jerger, "Load value approximation," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2014, pp. 127–139.
[10] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2013, pp. 25–36.
[11] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, "Doppelgänger: A cache for approximate computing," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2015, pp. 50–61.
[12] R. Venkatagiri *et al.*,"Impact of software approximations on the resiliency of a video summarization system," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2018, pp. 598–609.
[13] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang, "Input responsiveness: Using canary inputs to dynamically steer approximation," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2016, pp. 161–176.
[14] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2011, pp. 164–174.
[15] The LLVM compiler infrastructure. [Online]. Available: http://llvm.org
[16] A. Yazdanbakhsh, D. Mahajan, P. Lotfi-Kamran, and H. Esmaeilzadeh, "AxBench: A multi-platform benchmark suite for approximate computing," *IEEE Des. Test*, vol. 34, no. 2, pp. 60–68, Apr. 2017.
[17] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 updates and changes," Lawrence Livermore National Laboratory, Livermore, CA, *Tech. Rep. LLNL-TR-641973*, Aug. 2013.
[18] CORAL benchmarks. [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/
[19] gzip. [Online]. Available: https://www.gzip.org/
[20] Approximate memoization. [Online]. Available: https://github.com/IntelLabs/iACT
[21] A. Raha, S. Venkataramani, V. Raghunathan, and A. Raghunathan, "Quality configurable reduce-and-rank for energy efficient approximate computing," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2015, pp. 665–670.
[22] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning approximation for graphics engines," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2013, pp. 13–24.
[23] T. Yeh, P. Faloutsos, M. Ercegovac, S. Patel, and G. Reinman, "The art of deception: Adaptive precision reduction for area efficient physics acceleration," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2007, pp. 394–406.
[24] S. Mitra, M. K. Gupta, S. Misailovic, and S. Bagchi, "Phase-aware optimization in approximate computing," in *Proc. Int. Symp. Code Gener. Optim.*, 2017, pp. 185–196.
[25] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali, "Proactive control of approximate programs," in *Proc. 21st Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2016, pp. 607–621.
[26] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An online quality management system for approximate computing," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Architecture*, 2015, pp. 554–566.
[27] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and H. Esmaeilzadeh, "Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Architecture*, 2016, pp. 66–77.
[28] H. Sharif *et al.*, "ApproxHPVM: A portable compiler ir for accuracy-aware optimizations," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, 2019, Art. no. 186.
[29] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability type inference for flexible approximate programming," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Lang. Appl.*, 2015, pp. 470–487.
[30] J. Park, X. Zhang, K. Ni, H. Esmaeilzadeh, and M. Naik, "ExpAX: A framework for automating approximate programming," Georgia Tech, Comput. Sci., Tech. Rep. GT-CS-14-05, Jul. 2014.
[31] A. Sampson *et al.*, "ACCEPT: A programmer-guided compiler framework for practical approximate computing," Univ. Washington, Comput. Sci. Eng., Tech. Rep. UW-CSE-15–01-01, 2015.
[32] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Lang. Appl.*, 2013, pp. 33–52.
[33] A. Yazdanbakhsh *et al.*,"Axilog: Language support for approximate hardware design," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2015, pp. 812–817.
[34] Z. Peng *et al.*,"AXNet: ApproXimate computing using an end-to-end trainable neural network," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2018, pp. 1–8.

[35] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Neural acceleration for GPU throughput processors," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2015, pp. 482–493.
[36] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 1–14.
[37] R. Venkatagiri *et al.*,"gem5-approxilyzer: An open-source tool for application-level soft error analysis," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2019, pp. 214–221.
[38] J. Neystadt, "Automated penetration testing with white-box fuzzing," 2008. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/software-testing/cc162782(v=msdn.10)?redirectedfrom=MSDN
[39] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1532–1543.
[40] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proc. 27th Int. Conf. Mach. Learn.*, 2010, pp. 807–814.
[41] F. Chollet *et al.*, "Keras," 2015. [Online]. Available: https://github.com/fchollet/keras
[42] M. Abadi *et al.*,"TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 265–283.
[43] B. Steensgaard, "Points-to analysis in almost linear time," in *Proc. 23rd ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 1996, pp. 32–41.
[44] T. Moreau, F. Augusto, P. Howe, A. Alaghi, and L. Ceze, "QAPPA: A framework for navigating quality-energy tradeoffs with arbitrary quantization," Carnegie Mellon Univ., Comput. Sci. Eng., Tech. Rep. CMU/CSE-17–03-02, Mar. 2017.
[45] I. Akturk, K. Khatamifard, and U. R. Karpuzcu, "On quantification of accuracy loss in approximate computing," in *Proc. 12th Annu. Workshop Duplicating Deconstructing Debunking*, 2015.

Riad Akram (Member, IEEE) received the PhD degree in computer science from the University of Texas at San Antonio, San Antonio, Texas, in 2017. He is a software engineer at Intel Corporation, Santa Clara, California. His research focuses on approximate computing and its application in parallel programs. He is currently working on performance debugging of large scale software.

Shantanu Mandal (Member, IEEE) is working toward the PhD degree with the Department of Computer Science and Engineering, Texas A&M University, College Station, Texas. His research focuses on using machine learning to synthesize programs automatically.

Abdullah Muzahid (Member, IEEE) received the PhD degree in computer science from the University of Illinois, Urbana-Champaign, Champaign, Illinois, in 2012. He has been serving as an assistant professor with the Department of Computer Science and Engineering, Texas A&M University, College Station, Texas since Fall 2018. Before that, he worked as an assistant professor with the Department of Computer Science of the University of Texas at San Antonio, San Antonio, Texas. His research broadly focuses on various aspects of computer architecture and systems. More specifically, he is interested in multiprocessor architecture, parallel programming, programming models, debugging, program analysis and synthesis. Recently, he is interested in applying machine learning to solve various system-related issues. He received the NSF CAREER Award, in 2017 and Intel PhD Fellowship, in 2010.