FTApprox: A Fault-Tolerant Approximate Arithmetic Computing Data Format

Ye Wang and Jian Dong✉
School of Computer Science and Technology
Harbin Institute of Technology
Harbin, China
{yewang, dan}@hit.edu.cn

Qian Xu and Gang Qu
Department of Eletrical and Computer Engineering
and Institute of Systems Research
University of Maryland, College Park, USA
{qxu1234, gangqu}@umd.edu

***Abstract*—Approximate computing (AC) is an effective energy-efficient method for error-resilient applications. The essence behind AC is to reduce energy consumption by slightly sacrificing computation accuracy purposefully while providing quality-acceptable results. On the other hand, soft error is a common problem during program execution and may cause unacceptable outputs or catastrophic failure to the system. As AC introduces errors and soft errors are mitigated by fault-tolerant mechanisms, they have conflict goals and contradictory approaches. To the best of our knowledge, there is no previous efforts to consider the two at the same time. In this paper, we study the problem of AC with soft errors in order to guarantee the safe execution of the program while reducing energy (by AC). More specifically, we propose FTApprox, a fault-tolerant approximate arithmetic computing data format, to enable the detection and correction of SEs. As an approximate data format, FTApprox can use 16 bits to approximate any 32-bit integers and fixed-point numbers, and will select only the most significant part of operands for AC at runtime. Energy saving is obtained by converting 32-bit arithmetic operations to 8-bit operations. Meanwhile, for soft errors such as random bit flips, FTApprox not only can detect all single bit flips and most 2-bit flips, it can also correct most of these errors. The experimental results show that FTApprox has significant resistance against soft errors while providing 66.4%-79.6% energy saving.**

Keywords—Approximate computing, Data format, Energy efficiency, Fault-tolerant computing.

I. Introduction

Advancements in Internet of Things (IoT), mobile computing and artificial intelligence are increasing the demand for higher energy efficiency. Meanwhile, for many modern applications such as image recognition, deep neural network and signal processing, results with certain amount of errors can still meet users' accuracy requirement and thus are acceptable [1]. This observation motivates the development of approximate computing (AC), which is a design methodology that utilizes the intrinsic error resilience of applications to break the lower bound of the energy consumption at the cost of reducing the computing accuracy [2]. In recent years, researchers proposed many successful AC techniques at different design levels [3]. Some of the research outcomes have already made impact on industry, including the tensor processing unit (TPU) in Google's deep learning chip [4].

Due to the demand of high robustness against different problems in real-world applications, how to ensure the output quality of AC is a great concern in the long-term goal of implementing AC into commercial or industrial applications. Soft Error (SE, also called Transient Error) is one of the most serious reliability problems in real time computing. Unlike the systematic errors deliberately introduced by AC design, SE is caused by unpredictable physical conditions like high-energy neutrons from cosmic rays, alpha particles from the packaging material, or some other environment factors. SE usually shows its destructiveness through unexpected bitflips. Authors in [5] proposed that a typical 6T-SRAM cell in 45-nm technology has an soft error rate of 1095 errors per million cells per billion hours at sea-level, and this rate increases nearly 500 times at an altitude of 40,000 feet. Without error detection or correction mechanisms, bitflips are easy to incur Silent Data Corruptions (SDCs), which will lead the applications to erroneous results and even crush in application or system level.

This paper is the first attempt to take soft error into consideration in AC. We notice that, although AC is usually deployed on applications which have intrinsic error resilience, SE will introduce unacceptable errors and significantly degrade the output quality. According to our test in Section IV, some applications which should work well with AC, such as DNN and IDCT, show significant quality degradation when SE occurs. At this point, traditional fault-tolerant mechanisms are facing new challenges when being deployed on AC. Firstly, the traditional fault-tolerant mechanisms such as Error Correcting Code (ECC) usually have large overheads, which will jeopardize AC's optimization in energy. Secondly, AC arithmetic circuits usually have different logic from traditional designs, which make traditional fault-tolerant mechanisms may not work correctly in AC. Therefore, a fault-tolerant mechanism designed for AC needs to be highly customized with low energy, storage and performance overhead.

In this paper, we propose FTApprox, a fault-tolerant approximate arithmetic computing data format. Here we limit the type of soft error to bitflips **on storage**, and we only consider the cases of 1-bit and 2-bit flips because the probability becomes extremely low for three or more flips to occur at the same time. As an approximate data format, FTApprox can use 16 bits to approximate any 32-bit integers or fixed-point numbers, and can select only the most significant part of operands into computation at runtime. Energy saving is obtained by converting 32-bit arithmetic operations to 8-bit. FTApprox is also orthogonal to approximate ALU designs, so it can be applied to any 8-bit accurate or approximate ALUs. Meanwhile, FTApprox can detect and correct all 1-bit flip and most of 2-bit flips. Our experimental results show that FTApprox can provide 66.4%-79.6% energy saving and 50% storage saving while holding significant resistance against soft error.

II. Fault-Tolerant Approximate Arithmetic

In this section, we present the main components of the proposed FTApprox data format.

A. Overview

The structure of FTApprox is shown in Fig.1. There are three components: 8-bit data part, 7-bit control signal and a parity bit. The data part represents the most significant valid digits of the original binary data, and the control signal is a

fault-tolerant design which represents the original exponential order of the data part. The parity bit provides a parity check coverage to detect bitflip(s) occurrence in the data part.

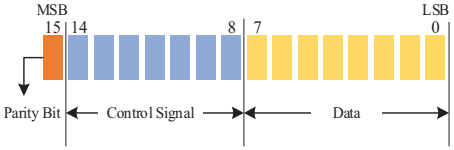

Fig. 1. FTApprox data format.

B. Approximate data

The 0-7 bits in FTApprox is the data part. This part saves the most significant valid part of the original binary data. To convert a 32-bit fixed point number to FTApprox, this number will be segmented to 8 blocks while each block contains 4 bits, as shown in Fig.2. Similar with the definition of valid digits in mathematics, we have following definitions:

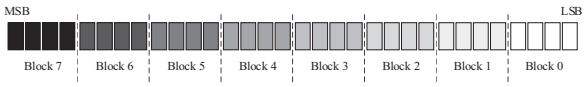

Fig. 2. Segmentation of 32-bit fixed point numbers

Definition 1: A bit is called the *most significant valid bit*, if and only if this bit is the first "1" from the left side.

Definition 2: A block is called the *Most Significant Valid Block (MSVB)*, if and only if it contains the most significant valid bit.

In FTApprox, only the MSVB and its right neighbor block, 8 bits in total, will be saved in the data part, and the other bits will be truncated. A special scenario is that when the MSVB is the most-right block (Block 0), there is no block on its right so we will save the Block 1 and Block 0 into the data part. Here we use a fast lightweight technique combining with the naïve truncation method to reduce the error's bias. While truncating the original number, the MSB of the truncated part will be read. If this bit is "1", then the last bit of the data part will be set to 1, and if this bit is "0", the truncated part will be directly dropped without any adjustments. This process can be seen as a bit OR operation.

C. Control signal

Although the data part saves the most two significant valid blocks, the blocks' location information has been broken, that is, only given the data part, one cannot tell where these blocks belong to in the original data. Therefore, we design a control signal to represent the order of the data part.

According to the location of the MSVB, there are 8 different scenarios (Block 0 to Block 7). So the control signal needs to have at least 3 bits ranging from 000 to 111, respectively representing that the MSVB is the Block 0 to Block 7 in the original data. However, although 3 bits are enough to represent all possible scenarios, it has no resistance against soft error. Once a control signal 111 suffers from a 1-bit flip and turns to 011, the data part which should belong to the Block 7 will be wrongly believed to be locating in Block 3. This will bring a huge error that this data will be underestimated by 2^{16} times smaller than the real value. To protect the control signal from soft error, we extend this part to 7 bits with a least hamming distance as 4, which is shown in Table I.

In this table, each codeword corresponds to a situation and the location of the data part can be easily gotten by reading the first 3 bits (the underlined part). These 8 codewords are called

TABLE I MSVB and corresponding control signal

MSVB	Control Signal	MSVB	Control Signal
Block 0	000 0000	Block 4	100 1011
Block 1	001 1110	Block 5	101 0101
Block 2	010 1101	Block 6	110 0110
Block 3	011 0011	Block 7	111 1000

Legal Control Signals. The minimum hamming distance is 4 so that the control signal can *correct 1 bitflip or detect 2 bitflips*. For any loaded signal with no error or only 1 bitflip, it will be automatically mapped back to the nearest legal signal. When there are 2 bitflips, the loaded signal will have the same least distances with at least two legal signals. In this scenario, because the original correct control signal cannot be determined for sure, an error detected signal will be sent to the system instead of mapping the loaded signal to any possible candidate. After getting an error detected signal, users can utilize any pre-defined lightweight data recovery techniques, such as reloading from lower level storage, to recover the correct data. In this paper, we restrict our discussion within providing a soft error correction or detection method for approximate computing. The recovery techniques after receiving the error detected signal are beyond the scope.

D. Parity bit

The control signal can prevent the original location of the data part from being mistaken because of soft errors, but the data itself is still exposed to soft errors. Considering the low overhead requirement of protecting the approximate data part, here we use the 1-bit parity check to provide error detection ability. The parity of the data part will be computed and saved in the parity bit. At runtime, the parity consistency will be checked to detect bitflips in the data part. However, this parity check can only detect 1-bit flip. If 2 bitflips occur in the data part at the same time, the parity check cannot detect it. Fortunately, in real world environments, most of soft bitflips are 1-bit flip. The possibility of simultaneously occurring 2-bit flips is quite low, and it is much more rare that both of the bitflips occur in the same FTApprox number's data part. Therefore, this parity can still detect most of soft errors. We will further evaluate the effect of soft error in the Section IV.

E. Example

Here we illustrate the transfer process with a numerical example in Fig. 3.

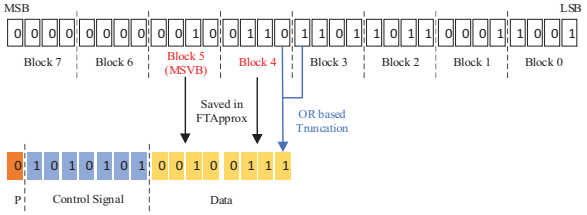

Fig. 3. Transfer from 32-bit fixed point number to FTApprox.

First, the MSVB will be found as Block 5, then Block 5 together with its right neighbor, Block 4, will be selected into the data part. Next, the leftmost bit of Block 3 and the rightmost bit of Block 4 will be processed by a bit OR operation and the output will be saved in the rightmost bit of the data part. The other bits will be truncated and dropped. Finally, the corresponding control signal of Block 5 can be figured out through Table I, and the parity bit can be easily computed and be saved in FTApprox. Compared with the original value, this truncation brings an relative error for 0.37%

III. Runtime Behavior of FTApprox

In this section, we will illustrate the runtime computing process with FTApprox. In this paper, we use normal accurate full adders and multipliers to more clearly illustrate the computing process of FTApprox. The runtime addition should process with three main steps: parity check, control signal mapping and arithmetic computing, as shown in Fig.4.

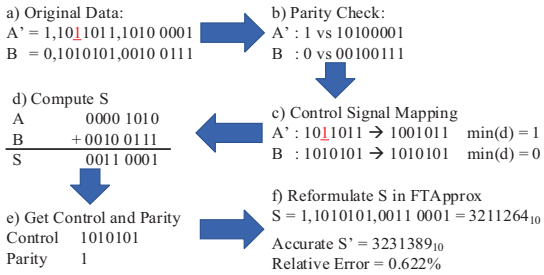

Fig. 4. Runtime addition process with FTApprox.

Assuming that there are two binary numbers:

A_0 = 0000,0000,0000,1010,0001,0011,1000,0100
B_0 = 0000,0000,0010,0110,1101,1011, 0001,1001

and both of them have been represented in FTApprox and noted as A and B as shown in Fig.4. a). At beginning, a bitflip occurs on the fourth bit to the left of A and turns it to A'.

First, A' and B will pass the parity check as their parity bits are both consistent with their data parts. Then, the loaded control signals of A' and B will be mapped to their nearest legal control signals in Table I with the minimum hamming distance min(d). In this process, the bitflip in A' will be automatically corrected and A' will be converted back to A. Then according to the first three bits of the control signals in A and B, the orders of the data parts will be determined and the addition will start. Finally, because there is no carry overflow, S will have the same MSVB location with the larger operand B. Therefore the control signal of S will be the same as B and the parity bit can be computed. Compared with the accurate result S' of A_0+B_0, the relative error is only 0.622%. Here we use addition as an example. When it turns to other operations like multiplication, the only difference is in the step d) where the addition will be replaced by 8-bit multiplication. Once there is carry overflow, the corresponding control signal of S can still be easily get from the table of legal control signal.

IV. Experimental Results

In this section, we evaluate FTApprox in terms of accuracy and energy saving. We first compare the power and area of the parity module, control module and ALUs. Next, we use 5 well-known applications: IDCT, DNN, *k*NN, SVM and IFFT, to compare the computing accuracy and evaluate the fault-tolerant ability of FTApprox. Finally, we compare the energy efficiency between FTApprox and AIF. The arithmetic units and modules are built in Verilog and synthesized using Cadence RTL Compiler with FreePDK 45nm library. The applications are implemented in Matlab 2018b.

A. Overhead of Arithmetic Units and Modules

Table II shows the hardware design parameters for the computing units and modules. The Parity Module is used to check the parity consistency of operands and compute the parity of an output. The Control Module is used to map a loaded signal to its nearest legal control signal, and get the exponential order of the data part according to the first 3 bits. In arithmetic operations, the parity module and the control module will be executed twice because both of the two operands need to be checked. Therefore, the energy saving is created by turning 32-bit addition and multiplication to 8-bit, while the AC overhead mainly comes from double executing the check modules in each arithmetic operation.

TABLE II Overhead of arithmetic units and control module

Units	Cells	Area	Power(nW)
8-bit Adder	91	212.12	752614.84
32-bit Adder	498	1116.93	4818821.94
8-Bit Multiplier	377	1037.62	2829745.1
16-Bit Multiplier	4916	15126.01	34033690.3
Parity Module	7	32.85	46556.53
Control Module	301	666.4	1268531

B. Accuracy with Different Soft Error Rate

To compare the accuracy of FTApprox and AIF, we implemented them in five well-known applications whose inputs are injected random soft error at two different error rates. The error rate *E1* is 1% for 1-bit flip and 0.01% for 2-bit flips which means that among all the test operands, 1% of them will be injected with one bitflip and 0.01% of them has two bitflips, and the other error rate *E2* is 2% for 1-bit flip and 0.04%f for 2-bit flips. If one operand is injected with a 1-bit flip, then one of the bits in this operand will be randomly selected and modified from 1(0) to 0(1). We do not take 3 or more bitflips into consideration because it has ultra-low possibility to occur in the same operand. Here we assume that the pre-defined data recovery technique is reloading. After detecting an unrecoverable soft error in storage, the system will reload the data from its lower level storage. For example, detecting a soft error in L1 Cache will trigger a reload from the L2 Cache. We take the results of these applications with no approximation as the baseline and the results are shown in Table. III.

In IDCT application, we executed a DCT process with FTApprox and AIF on Lena.jpg, and then processed the result with IDCT. We compared the peak signal-to-noise ratio (PSNR) in Table III and the output figures and noise distributions under the error rate *E1* are shown in Fig.5. When there is no soft error, the FTApprox and AIF have the same PSNR as they share a similar approximate degree. However,

TABLE III Accuracy of FTApprox and AIF with Different Soft Error Rate

Benchmarks	Error Metric	Baseline	No soft Error		*E1*:1-bit Flip: 1% 2-bit Flips: 0.01%		*E2*:1-bit Flip: 2% 2-bit Flips: 0.04%	
			FTApprox	AIF	FTApprox	AIF	FTApprox	AIF
IDCT	PSNR	\	90.32	90.32	55.24	37.84	51.27	28.45
IFFT	PSNR	\	323.48	323.48	58.01	37.60	49.92	28.52
DNN	Accuracy	94.5%	94.4%	94.4%	94.09%	75.56%	92.75%	60.94%
KNN	Accuracy	93.3%	93.3%	93.3%	93.3%	93%	93.3%	92.6%
SVM	Accuracy	95.72%	95.72%	95.72%	95.72%	95.19%	95.72%	95.1%

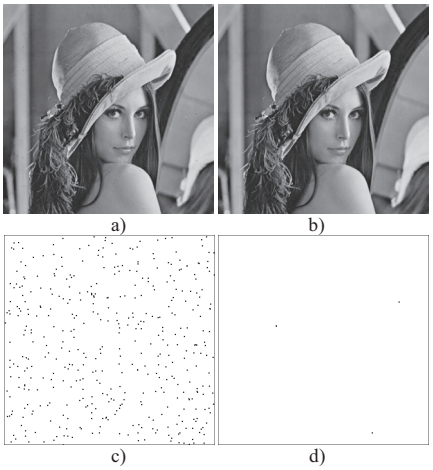

Fig. 5. IDCT results of AIF and FTApprox under soft error rate *E1*. a) and b) are the output figures of AIF and FTApprox (PSNR of a = 37.84, PSNR of b =55.24). c) and d) are the noise distribution of a) and b).

AIF shows significant quality degradation and much more noise pixels on the output picture when there exists random bitflips, while FTApprox have only several noise pixels. This is because FTApprox can correct or detect most of soft errors, but when both of the 2 bitflips occur in the data part of the same pixel, FTApprox cannot detect this error and this pixel behaves as a noise. We also implemented Lena.jpg in an FFT-IFFT process. The results are listed in the second raw of Table III and show the same regularity.

For DNN, KNN and SVM, we define the error metric as the percentage of miss-classified inputs. We take MNIST data as the training and testing data set. The DNN is structured as 784 nodes in the input layer, 100 nodes in the hidden layer and 10 nodes in the output layer. The training inputs is 60k and the testing inputs is 10k. The KNN application is a single-NN classifier and the SVM application is a straightforward SVM classifier with linear kernel. Since KNN and SVM will take very long time to classify the whole MNIST, to speed up the experiment, we only use part of the data. Noted that the training process contains no soft error, the error is only occurs in the runtime classification process.

When there is no soft error, FTApprox and AIF both show high classification accuracy and negligible degradation compared with the baseline among all 3 applications. For DNN, when the soft error occurs, AIF shows a degraded accuracy that decreases from 94.4% to 75.56% and 60.94%, while FTApprox effectively resists the soft error and contains a high accuracy for 94.09% and 92.75%. For KNN and SVM, while FTApprox still has improvement on accuracy, the soft error only causes a small accuracy degradation in AIF. This mainly has two reasons. On the one hand, complex models may suffer more from soft error because the error caused by bitflips has a long error propagation path while computing and cause significant difference in the output. On the other hand, the DNN in our experiment has no special layers such as convolutional or pooling layers which are able to filter some noise. The KNN and SVM applications are not as sensitive as the DNN we used and have shorter computing path, therefore the injected error does not significantly show the influence on the final accuracy.

C. Energy Saving

We simulate the energy saving of AIF and FTApprox and the results are shown in Fig. 6. Here we do not consider the energy consumption of reloading the data with unrecoverable error because it is negligible compared with the application's consumption.

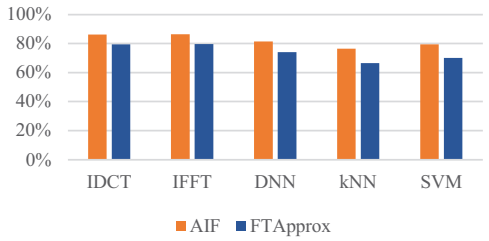

Fig. 6. Energy Saving of AIF and FTApprox among 5 Benchmarks

Among all applications, the energy saving of AIF is from 79.4% to 86.2%. Compared with AIF, the energy saving of FTApprox is 8% less on average, ranging from 66.4%-79.3%. This gap mainly comes from the additional control and parity module. AIF only has a slight overhead for implementing approximation function only, that is selecting the most significant blocks at runtime, but has no resistant to soft error. FTApprox both provides the approximate and fault-tolerant ability and the additional modules are more complex than AIF. Therefore the AC and fault-tolerant overhead of FTApprox is slightly higher than AIF.

V. CONCLUSION

In this paper, we analyzed the serious fault-tolerant requirement in approximate computing and proposed FTApprox, a fault-tolerant approximate arithmetic data format. FTApprox can dynamically select the most significant valid bits of any 32-bit fixed point operands at runtime, represent a 32-bit operand in 16 bits and provide significant energy saving by converting 32-bit operations to 8-bit. Moreover, FTApprox can correct, or at least detect all 1-bit flip and most of 2-bit flips, which makes it the first AC design that can tolerant soft error. Our experiments show that, FTApprox can provide 66.4%-79.6% energy saving, while showing good robustness against soft errors. This is the first work that takes soft error into consideration in approximate computing design. In this paper, we restrict our discussion in approximate data format under soft bitflips in storage. When it turns to approximate hardware and software, together with considering hard/unrepairable errors and application/system failures, there might be more challenges and design space for approximate computing, which is also our future work.

REFERENCES

[1] N. Zhu, W. L. Goh, W. Zhang, et al, "Design of Low-Power High-Speed Truncation-Error-Tolerant Adder and Its Application in Digital Signal Processing," in *IEEE Transactions on VLSI Systems*, 2010.

[2] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, no. 4, pp. 1-33, May 2016.

[3] M. Gao and G. Qu, "Estimate and Recompute: A Novel Paradigm for Approximate Computing on Data Flow Graphs," in *IEEE Transactions on CAD of Integrated Circuits and Systems*, 2020.

[4] W. Liu, C. Gu, G. Qu and M. O'Neill, "Approximate Computing and Its Application to Hardware Security," in *Cyber-Physical Systems Security. Springer, Cham*, 2018, pp. 43-67.

[5] A. Vijayan, S. Kiamehr, et al, "Online Soft-Error Vulnerability Estimation for Memory Arrays and Logic Cores," in *IEEE Transactions on CAD of Integrated Circuits and Systems*, 2018.

[6] M. Gao, Q. Wang, A. S. K. Nagendra and G. Qu, "A novel data format for approximate arithmetic computing," in *ASP-DAC*, 2017.