An Approximate Communication Framework for Network-on-Chips

Yuechen Chen, *Student Member, IEEE* and Ahmed Louri, *Fellow, IEEE*

Abstract—Current multi-/many-core systems spend large amounts of time and power transmitting data across on-chip interconnects. This problem is aggravated when data-intensive applications, such as machine learning and pattern recognition, are executed in these systems. Recent studies show that some data-intensive applications can tolerate modest errors, thus opening a new design dimension, namely, trading result quality for better system performance. In this article, we explore application error tolerance and propose an approximate communication framework to reduce the power consumption and latency of network-on-chips (NoCs). The proposed framework incorporates a quality control method and a data approximation mechanism to reduce the packet size to decrease network power consumption and latency. The quality control method automatically identifies the error-resilient variables that can be approximated during transmission and calculates their error thresholds based on the quality requirements of the application by analyzing the source code. The data approximation method includes a lightweight lossy compression scheme, which significantly reduces packet size when the error-resilient variables are transmitted. This framework results in fewer flits in each data packet and reduces traffic in NoCs while guaranteeing the quality requirements of applications. Our cycle-accurate simulation using the AxBench benchmark suite shows that the proposed approximate communication framework achieves 62 percent latency reduction and 43 percent dynamic power reduction compared to previous approximate communication techniques while ensuring 95 percent result quality.

Index Terms—Approximate communication, error control, power consumption, network-on-chips (NoCs)

1 Introduction

Network-on-chips (NoCs) are becoming standard communication solutions for connecting cores, caches and memory controllers on chips [1], [2], [3], [4]. Current multi-core chips typically have hundreds of cores, and future projections call for thousands of cores [5], [6], [7]. With significant improvement in system performance through exploiting parallelism on multi-core chips, the state-of-the-art NoC design can soon become a communication bottleneck and struggle to deliver packets in a power-efficient manner [8], [9]. Recent studies have shown that NoC power consumption can reach up to 40 percent of the overall chip power [10], [11], [12]. Consequently, there is a need for innovative power and latency reduction techniques for future NoC designs.

Recent research shows that several approximate computing applications, such as pattern recognition, image processing, and scientific computing, can tolerate modest errors while yielding acceptable results [13], [14], [15], [16], [17], [18]. However, conventional NoC designs for multicore processors transmit all data with absolute accuracy, which is unnecessary for such approximate computing applications. Transmitting data with excessive accuracy consumes excess power and increases the network latency. These observations suggest new design space in which data accuracy can be sacrificed to some extent to achieve better network performance.

In this work, we propose an approximate communication framework for NoCs in which the data in a packet are compressed based on the error tolerance of the application to reduce power consumption and network latency. The proposed approximate communication framework includes a software-based quality control method and a hardware-based data approximation method implemented in the network interface (NI). Before the execution of the application, the quality control method uses the code analyzer to identify error-resilient variables and calculate the error tolerance of each variable based on the application's quality requirement on results. We introduce new instructions, namely approximate load and store, to indicate error-resilient variables with error tolerance values in the assembly code. When an approximate load or store is executed, the network interface (NI) compresses the data and generates an approximated packet based on the error tolerance of the variable, using the proposed data approximation method. As a result, the proposed approximate communication framework decreases the amount of data transmitted, resulting in significant improvements in power consumption and network latency compared to conventional NoC designs.

Specifically, the contributions of this work include the following:

- We propose an approximate communication framework that employs a hardware-software co-design to trade result quality for better network performance.

- *The authors are with the Department of Electrical and Computer Engineering, George Washington University, Washington, DC 20052. E-mail: {yuechen, louri}@gwu.edu.*

Manuscript received 14 Oct. 2019; revised 6 Jan. 2020; accepted 15 Jan. 2020. Date of publication 22 Jan. 2020; date of current version 3 Feb. 2020. (Corresponding author: Yuechen Chen.) Recommended for acceptance by M. Kandemir. Digital Object Identifier no. 10.1109/TPDS.2020.2968068

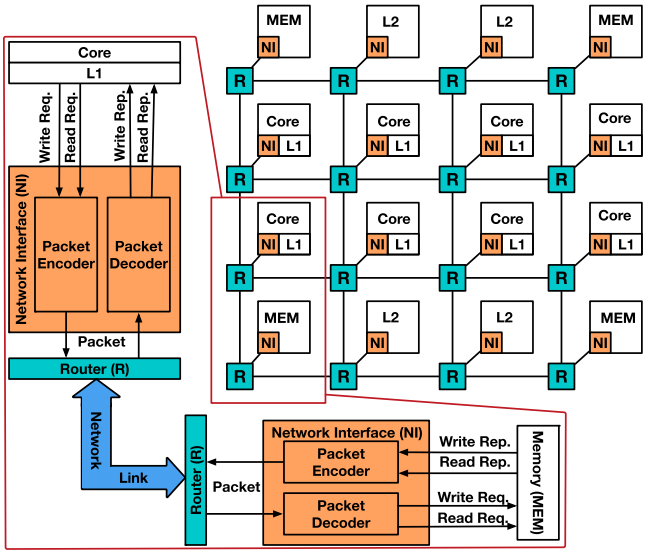


Fig. 1. Conventional network interface (NI) design: Read request (Req.) is triggered by cache read miss. Read reply (Rep.) carries the data from memory (MEM)/L2 cache node to the core/L1 cache node. Write request (Req.) is triggered by cache write miss and carries the data to the MEM/L2 cache node. Write reply(Rep.) is send back to the core/L1 cache node to confirm a successful memory write.

- On the software side, the framework uses a quality control method to automatically identify error-resilient variables and calculate the error tolerances for the variables.
- On the hardware side, we augment the conventional network interface with compression/decompression modules to significantly reduce packet size.
- The performance evaluation of our proposed framework shows that it reduces network latency and dynamic power consumption by 62 and 43 percent, respectively, compared to previous approximate communication techniques [19], [20] while ensuring 95 percent result quality.

2 Motivation and Challenges

2.1 Motivation

On-Chip Communication is Costly. The increasing scale of data movement in emerging big-data applications results in heavy NoC communication loads. For tasks in parallel applications, communication takes more time than computation does as the number of compute elements (processor cores) in the system increases [21], [22]. Communication is also more costly than computation in terms of energy consumption [21], [22], [23].

Fig. 1 shows a conventional Network Interface (NI) design [24], [25], [26] in a multi-core system with L2 shard cache. In this system, when a cache miss occurs during a memory load operation, a read request packet will be sent to the memory or the shared cache through the NoC. The memory or shared cache will use a read reply packet to send the required data back to the core. When a cache miss occurs during a memory store operation, the data is incorporated into a write request packet and sent to the memory or shared cache through the NoC. After the memory or shared cache has received the data, a write reply is sent back to the core to confirm a successful memory write.

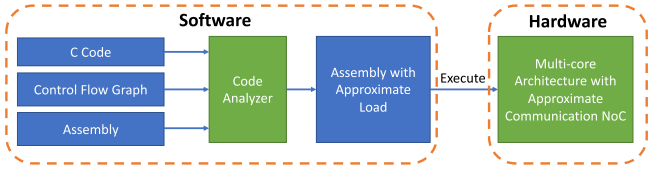


Fig. 2. High-level workflow of the approximate communication framework.

Error Tolerance of Applications. Approximate computing applications leverage the perceptual limitation of users to trade result accuracy for less power consumption or execution time [16], [17], [27], [28]. For example, even though the approximated output of an image processing application is not 100 percent numerically correct, the difference is unnoticeable to the end user. Therefore, program designers provide the result error threshold and evaluation metric when developing approximate computing applications to ensure result quality. The evaluation metric defines how the quality of the result is measured by the user (e.g., relative error, pixel difference). The result error threshold defines the amount of error that can be tolerated by the user. This situation inspires us to exploit the error budget of such applications to improve communication performance in terms of power consumption and latency.

2.2 Challenges

Ensuring Result Quality. The ability to control the errors incurred during approximation is required to ensure result quality [29], [30]. Unrestricted approximation can lead to disastrous consequences, such as erroneous results or program crashes [31], [32], [33]. The previous quality control methods [9], [19], [20], [34] require program designers to specify the error-resilient variables and their error tolerances in approximate computing applications. These methods rely on human engineering to decide the error-tolerance of each error-resilient variable, which can lead to unpredictable quality loss. Therefore, quality control remains a big challenge for approximate communication.

Low Overhead of Approximation Logic. An approximate communication technique requires approximation logic (including hardware support) to compress packets based on the quality requirements for the data. Such approximation logic is located at the critical path, and has strict requirement on overheads in terms of latency and area. The implementation needs to be carefully designed so that these overheads of approximation do not exceed the performance gains achieved through approximate communication.

3 Approximate Communication Framework

The essence of the proposed approximate communication framework is to carefully exploit the error tolerance of applications to reduce packet sizes by compressing data with an acceptable quality loss. Reducing the number of bits used decreases the number of flits inside a packet, which will result in a reduction in power consumption and network latency. Fig. 2 shows the high-level workflow of the proposed approximate communication framework. This framework includes a code analyzer for quality control and a hardware design for approximate communication in NoC

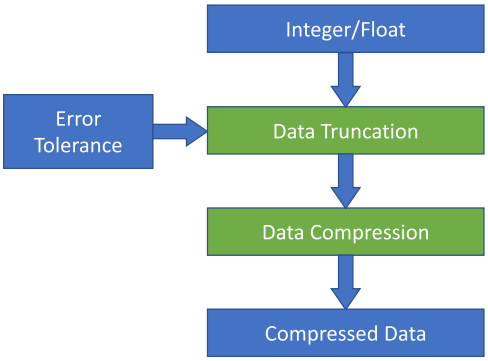


Fig. 3. Approximate data compression workflow.

to enhance the performance of the interconnects. The code analyzer calculates the error tolerance for each variable that is loaded or stored based on the quality requirements of an application before the application is executed. In the approximate communication NoC design, data packets are compressed in the NI based on the error tolerances of the corresponding data values using a lossy compression method during the execution of the application.

The workflow of the framework is described as follows: when a C code is compiled, the compiler generates the assembly code and control flow graph. The code analyzer identifies the error-resilient values and calculates the error tolerance for each value in the approximable code sections. Then, the load/store operations for each variable in the assembly code are replaced with approximate load/store operations, each associated with a quality requirement as calculated by the code analyzer. Finally, the assembly code with the approximate load/store operations is executed in the proposed architecture with approximate communication. We discuss lossy data compression for floating point and integer values based on a given data error threshold in Section 3.1. Then, we present an example of lossy data compression in Section 3.2. In Section 3.3, we discuss the calculation of the error tolerance for each load and store instruction based on the quality requirement specified for the results.

3.1 Approximate Data Compression

Fig. 3 shows the workflow for approximate data compression. The first step is to truncate the integer or floating point data based on the error tolerance. In this paper, we define the error tolerance as the maximum relative error that a data can accept. Eq. (1) shows the definition of error tolerance, where $\tilde{a}$ is approximated a and E_r is relative error

$$E_r = \frac{|a - \tilde{a}|}{a} \leq error\ tolerance. \tag{1}$$

Eqs. (2) and (3) show the representation of single precision floating point value based on IEEE 754 standard [35]

$$float = (-1)^S \times mantissa \times 2^{exp} \tag{2}$$

$$mantissa = 2^0 + \sum_{k=1}^{23} X_k 2^{-k}\ (X_k = 0\ or\ 1). \tag{3}$$

Based on Eqs. (2) and (3), the mantissa always starts with one.

According to the IEEE 754 standard [35], when a data point is represented in the floating point format, the first bit of the mantissa is omitted. We observe that when c bits (of the 23-bit mantissa) are protected, the maximum relative error on this floating point data value will be $\sum_{k=c+1}^{23} 2^{-k}$, which is less than 2^{-c} according to the sum of the geometric sequence ($\sum_{k=1}^{n} ar^{k-1} = a(1-r^n)/1-r$, where a is the first term, n is the number of terms, and r is the common ratio in the sequence). Therefore, using Eq. (3), we can deduce the following expression for the data error tolerance:

$$error\ tolerance = 2^{-n}\ (1 \leq n \leq 23). \tag{4}$$

In Eq. (4) above, the data error tolerance is a number between 0 and 1, and n is the number of most significant bits (MSBs) in the mantissa of this floating point value. In a floating point data value, the 1-bit sign and the 8-bit exponent (a total of 9 bits) are also critical bits, which must be transmitted. Thus, by truncating $23 - n$ bits, we can ensure the value's relative error is less than 2^{-n}. For example, to satisfy a data error tolerance of 10 percent for any floating point value, we can truncate 18 least significant bits (LSBs), resulting in a maximum relative error of 6.25 percent.

Eq. (5) shows the representation of a signed integer. In a signed integer, the MSB represents the sign, and the remaining 31 bits represent the value

$$int = \sum_{k=0}^{31} X_k 2^k\ (X_k = 0\ or\ 1). \tag{5}$$

We observe that when n bits (of the 31 LSBs) are truncated, the maximum error caused by truncation will be $\sum_{k=0}^{n} X_k 2^k\ (X_k = 0\ or\ 1)$. Thus, we can use Eq. (6) to calculate the number of bits (n) to be truncated for a given error tolerance

$$error\ tolerance = \frac{\sum_{k=0}^{n} X_k 2^k}{\sum_{k=0}^{31} X_k 2^k}\ (X_k = 0\ or\ 1). \tag{6}$$

With this data truncation method, an integer with a small absolute value requires a larger number of MSBs to achieve the same error threshold than is required for an integer with a large absolute value. For example, for an integer value of 100, 29 MSBs need to be transmitted to ensure 5 percent error tolerance. On the other hand, for an integer value of 5,418, only 28 MSBs need to be transmitted to achieve the same data error tolerance (5 percent).

To overcome this problem, we compress the data using the frequent data pattern compression method. In previous research, the frequent data pattern compression mechanism (Table 1) has been proposed [36] and extended to NoCs with a low-overhead compression and decompression

TABLE 1
Frequent Pattern Encoding [36]

Code	Pattern Encoded	Data Size After Encoding
000	Zero run	3 bits
001	4-bit sign-extended	4 bits
010	1-byte sign-extended	8 bits
011	Halfword sign-extended	16 bits
100	Halfword padded with a zero halfword	16 bits
101	Two halfwords, each 1-byte sign-extended	16 bits
111	Uncompressed word	32 bits

Code	Frequent Pattern			
000	0x00	0x00	0x00	0x00
001	0x00	0x00	0x00	00000XXX
	0xff	0xff	0xff	11111XXX
010	0x00	0x00	0x00	0XXXXXXX
	0xff	0xff	0xff	1XXXXXXX
011	0x00	0x00	0XXXXXXX	XXXXXXXX
	0xff	0xff	1XXXXXXX	XXXXXXXX
100	XXXXXXXX	XXXXXXXX	0x00	0x00
101	0x00	0XXXXXXX	0x00	0XXXXXXX
	0xff	1XXXXXXX	0xff	1XXXXXXX

Fig. 4. Frequent pattern replacement table: X represents a bit that can be either 0 or 1. 0xff represents eight 1-bits. 0x00 represents eight 0-bits.

mechanism [9], [37]. In this work, we adopt this mechanism to compress the approximated data. The essence of frequent data pattern compression method is to eliminate zeros and ones in the MSBs and LSBs for both integer and floating value without effecting the accuracy of the value. We develop a frequent pattern replacement table based on Table 1. The table in Fig. 4 shows the static set of frequent patterns and the codes. In this table, the notation X represents a bit that can be either 0 or 1. 0xff and 0x00 are two hexadecimal numbers, which represent eight 1-bits and eight 0-bits. The data compressor checks every piece of data and attempts to match its pattern. If the pattern matches, the data compressor will replace the pattern with the corresponding code. The 0 or 1 represented by X will not be changed during the compression process.

3.2 Example of Approximate Data Compression

Fig. 5 shows an example of the proposed approximation technique for two integers (I1 and I2) and two floating point numbers (FP1 and FP2). In this example, 17 LSBs of FP1 and 19 LSBs of FP2 are truncated at the source node to ensure that the floating point values satisfy error thresholds of 3 and 10 percent, respectively. Since these two floating point numbers do not match any pattern, the data cannot be compressed. When the floating point numbers reach the destination, the error rates of the approximated floating point data are 0.2 and 1.9 percent, within the error thresholds of 3 and 10 percent, respectively. For the integer numbers, 20 LSBs of I1 are truncated at the source node, whereas no LSB is truncated for I2 because no error on this value can be tolerated. Then, the

Data Example With Error Threshold Requirement	FP1 = 35.89 Error Threshold: 3%	FP2 = 90.52 Error Threshold: 10%	I1 = 5544320 Error Threshold: 10%	I2 = 48 Error Threshold: 0%
Data Example in Binary	0x420F8F5C	0x42B50A3D	0x00549980	0x00000030
Truncated Data	010000100000111	0100001010110	000000000101	0x00000030
Compressed Data	010000100000111	0100001010110	011101	01000110000
Received Data	0x420E000	0x42B00000	0x00500000	0x00000030
Received Data in Decimal	FP1 = 35.5 Error: 1.09%	FP2 = 88.0 Error: 2.78%	I1 = 45543424 Error: 5.4%	I1 = 48 Error: 0%

Fig. 5. Working example of approximate data compression.

data compression logic matches I1 with pattern 011 and replaces the zeros in the MSBs with the corresponding code. Similarly, I2 is found to match pattern 010, and the zeros in the MSBs are replaced with the corresponding code. When the compressed data arrive at the destination, the receiver NI converts the data back into integer values and fills in the truncated bits with zeros. We observe that the errors on the resulting integer numbers, namely, 5.4 and 0 percent, are again within the corresponding error thresholds.

3.3 Code Analysis

The main function of the code analyzer is to identify the error-resilient values in the approximable code sections and calculate the corresponding error tolerances based on the quality requirements of the application. To achieve this goal, the code analyzer needs to analyze the syntax of the source code and break down all operations into basic operations (addition, subtraction, multiplication, and division). Notably, the compiler can generate a control flow graph (CFG) that describes the function of the source code in terms of basic operations. Therefore, we utilize the CFG to identify the error-resilient values in the code and calculate the error tolerances.

Fig. 6 shows the workflow of the code analyzer by means of an example. The code analyzer needs three files: the source code, the CFG and the assembly code. First, the code analyzer searches for approximable functions in the C code, CFG and assembly code. The approximable section is highlighted with orange boxes in the figure.

Second, the code analyzer identifies all variables and results in the approximable section in the C code, CFG, and assembly code. In the figure, we highlight the result and the result quality for this example in yellow boxes. We also point out the mov operation on variable a, b and c in the assembly code using yellow arrows.

Algorithm 1. Error Threshold Calculation (ETC)

```
1:  function ETC (Data_Dependency_Graph G, start_V)
2:    let Q be a queue
3:    start_V.error tolerance ← result error tolerance
4:    Q.enqueue(start_V)
5:    while Q is not empty do
6:      v ← Q.dequeue
7:      for all edges w in G.adjacentEdges(v) do
8:        error ← calculate_error(v.error_tolerance, v.operation)
9:        if w.error_tolerance = empty then
10:         w.error_ tolerance ← error
11:         Q.enqueue(w)
12:       end if
13:       if w.error_tolerance > error then
14:         w.error_ tolerance ← error
15:       end if
16:     end for
17:   end while
18: end function
```

Third, the analyzer builds a data dependency graph (highlighted in the green box), which shows the calculation process for each variable, based on the CFG. For the example in the figure, we can see that the intermediate variable $D.21740$ is equal to the product of b and c and that the result d is equal to the sum of a and $D.21740$.

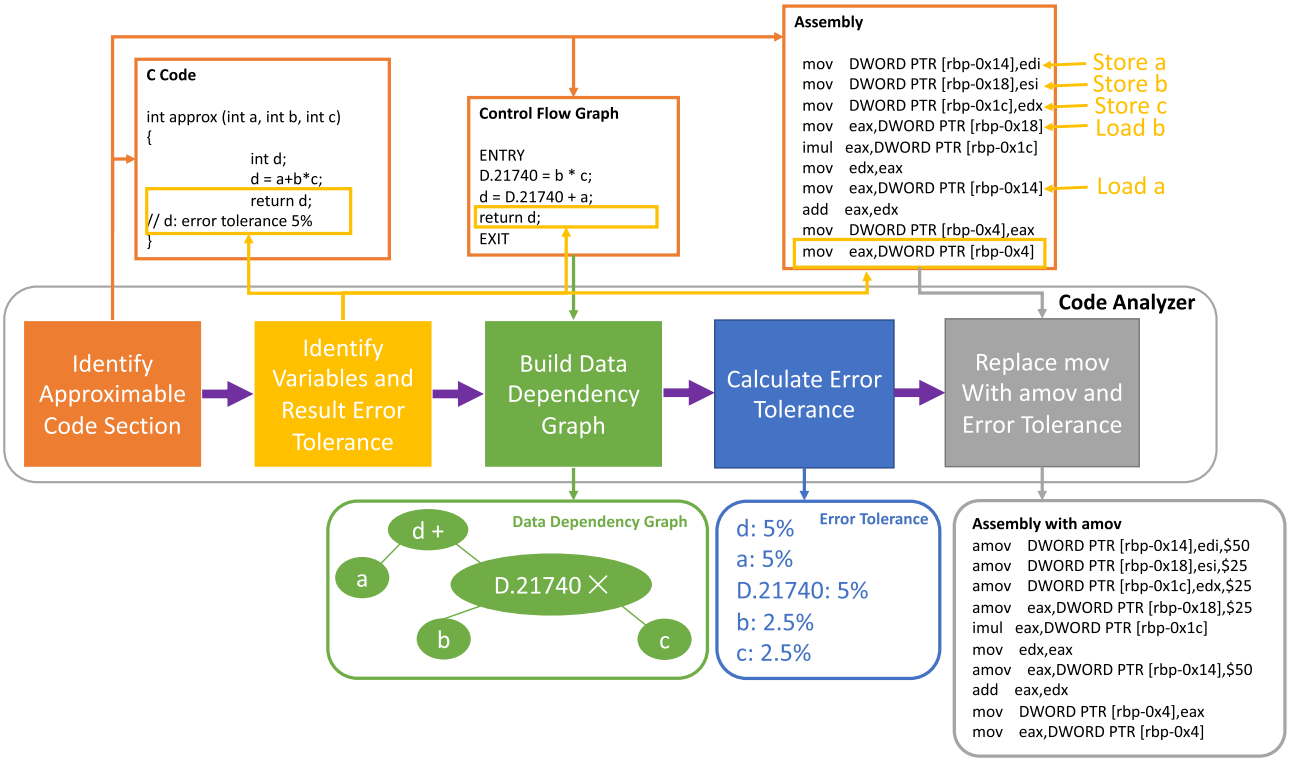


Fig. 6. Code analyzer workflow: The purple arrow indicates the workflow of the code analyzer. The inputs of the code analyzer are C code, control flow graph, and assembly. The output of the analyzer is the assembly with amov instructions, which is highlighted in the gray box. (The source code is compiled by GNU C Compiler (GCC) for X86 instruction set architecture. The control flow graph and assembly code are generated by GCC. D.21740 is an intermediate variable created by compiler).

Fourth, the code analyzer traverses the data dependency graph and updates the error threshold for each variable. The calculated error thresholds are highlighted in the blue box in Fig. 6. The error threshold for each intermediate data value is calculated using Algorithm 1. The error threshold calculation (ETC) algorithm traverses the data dependency graph in a manner similar to that of breadth-first search (BFS). Different from the conventional BFS algorithm, at line 13, we allow the error tolerance to be updated when the current error tolerance is smaller than the previous one to prevent erroneous calculation results. The error tolerance for each variable is calculated based on the error tolerance for the result and the various operations applied.

Eq. (7) describes the addition and subtraction of $\tilde{a}$ and $\tilde{b}$, where $\tilde{a}$ and $\tilde{b}$ are transmitted through approximate communication and $\tilde{c}$ is the result

$$\tilde{c} = \tilde{a} \pm \tilde{b}. \tag{7}$$

From Eq. (1), we can get

$$E_{rc} = \frac{|c - \tilde{c}|}{c} \leq error\ tolerance \tag{8}$$

$$\tilde{a} = a + E_{ra} \times a \tag{9}$$

$$\tilde{b} = b + E_{rb} \times b. \tag{10}$$

When a and b have the same relative error ($E_{ra} = E_{rb} = E_{rab}$), we can get Eq. (11) by combining Eqs. (7), (9) and (10)

$$\tilde{c} = (a + E_{rab} \times a) \pm (b + E_{rab} \times b). \tag{11}$$

By combining Eqs. (8) and (11) with $c = a \pm b$, we find that the relative error (E_{rab}) for a and b are equal to the relative error (E_{rc}) for c. Therefore, $E_{rc} \leq error\ tolerance$ is ensured when $E_{rab} \leq error\ tolerance$ for the addition and subtraction operation. For example, $\tilde{a}$ and $\tilde{b}$ can each contain less than 5 percent relative error when $\tilde{c}$ can tolerate 5 percent relative error.

Eq. (12) describes the multiplication of $\tilde{a}$ and $\tilde{b}$, where $\tilde{a}$ and $\tilde{b}$ are transmitted through approximate communication and $\tilde{c}$ is the result

$$\tilde{c} = \tilde{a} \times \tilde{b}. \tag{12}$$

With the same theory, we can get Eq. (13) for the multiplication, where a and b are fully accurate variables and E_{rab} is relative error

$$\tilde{c} = (a + E_{rab} * a) * (b + E_{rab} * b). \tag{13}$$

By combining Eqs. (8) and (13) with $c = a \times b$, we find that for the multiplication operation, $E_{rc} = (1 + E_{rab})^2 - 1$. Therefore, $E_{rc} \leq error\ tolerance$ is ensured when $-1 + \sqrt{1 + E_{rab}} \leq error\ tolerance$ for the multiplication operation. For example, $\tilde{a}$ and $\tilde{b}$ can each contain less than 2.5 percent relative error when $\tilde{c}$ can tolerate 5 percent relative error.

Eq. (14) describes the division of $\tilde{a}$ and $\tilde{b}$, where $\tilde{a}$ and $\tilde{b}$ are transmitted through approximate communication and $\tilde{c}$ is the result

$$\tilde{c} = \tilde{a}/\tilde{b}. \tag{14}$$

For the division operation, we can get and Eq. (15), where a and b are fully accurate variables and E_{ra}, E_{rb} are relative error

$$\tilde{c} = (a + E_{ra} * a)/(b + E_{rb} * b). \tag{15}$$

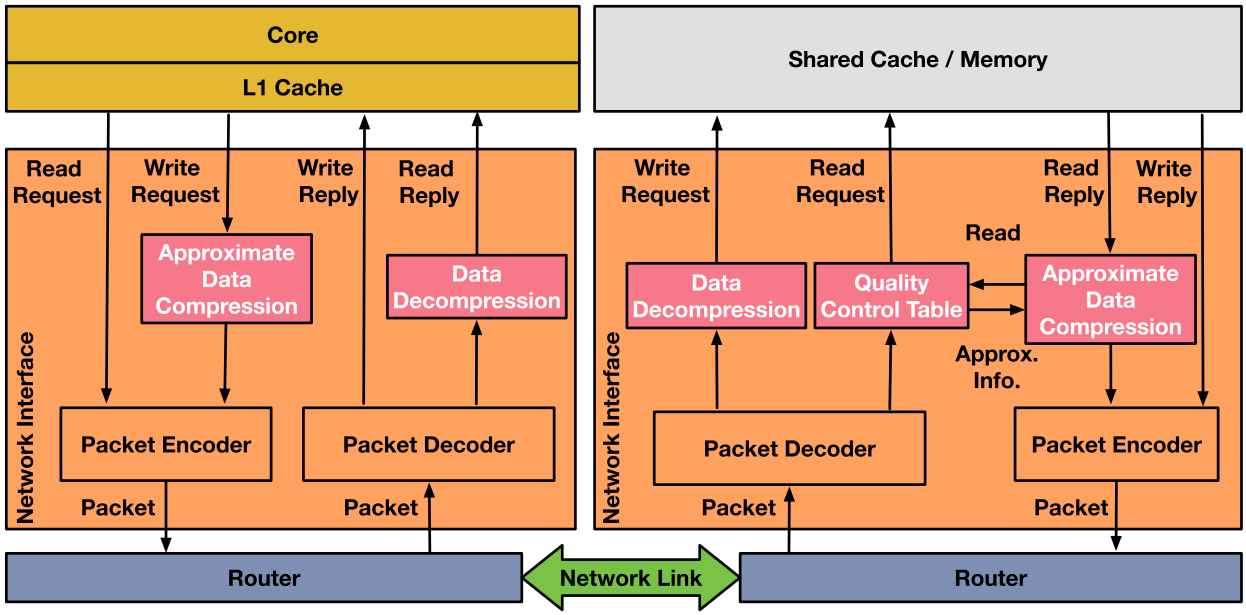


Fig. 7. NoC architecture with approximate communication: The quality control table records the approximation information, including addresses, data error tolerances and int/float indicators, to enable the data approximation logic to identify and compress approximable data. The data approximation logic truncates and compresses data based on the data error tolerances and data types. The data decompression logic recovers the data and fills in truncated bits with zeros after a packet has arrived at the destination NI. The packet encoder generates packets and injects those packets into the network. The packet decoder ejects packets from the network and extracts data from packets.

By combining Eqs. (8) and (15) with $c = a/b$, we find that for the division operation, $E_{rc} = |1 - (1 + E_{ra})/(1 + E_{rb})|$. $E_{rc} = 0$ when a and b have the same relative error ($E_{ra} = E_{rb} = E_{rab}$) for the division operation. The condition ($E_{ra} = E_{rb} = E_{rab}$) is hard to satisfy during the execution of the program and result error is depends on the difference between E_{ra} and E_{rb}. Since the difference of E_{ra} and E_{rb} is small when both E_{ra} and E_{rb} are small, we use $-1 + \sqrt{1 + E_{rab}} \leq error\ tolerance$ to reduce the value of E_{ra} and E_{rb}. For example, $\tilde{a}$ and $\tilde{b}$ can each contain less than 2.5 percent relative error when $\tilde{c}$ can tolerate 5 percent relative error. Therefore, the difference between E_{ra} and E_{rb} are limited to less than 2.5 percent for the division operation.

Finally, the code analyzer selects and replaces the mov instructions in the assembly code with approximate mov instructions. The selected mov instructions load and store the variable, which locates at the leaf of the data dependency graph (e.g., a, b, c). We introduce a new type of approximate move instruction ($amov$ dist, src, error threshold) into the X86 instruction set for the network to compress approximable packets. The error threshold in the $amov$ instruction is multiplied by 10^3 to eliminate decimal point. The final result is highlighted in the gray box, where 50 indicates a 5 percent error tolerance and 25 indicates a 2.5 percent error tolerance. In the next section, we will discuss how the network approximates packets using these approximate move instructions.

4 Architectural Design of the Approximate Communication Framework

Fig. 7 presents a high-level overview of the proposed approximate communication framework along with the proposed hardware design. We modify the baseline NI to include the following additional components: the approximate data compression logic, the data decompression logic and a quality control table.

When an L1 cache miss is caused by an approximate store operation, a write request is generated by the L1 cache with the approximation information. The approximation information includes the address, data type (int/float) and error tolerance. The write request is compressed by the approximate data compression logic at the core/L1 cache node based on the error tolerance of the data. Then, the write request is encoded by the packet encoder and injected into the network. When the shared cache/memory node receives the write request packet, the data decompression module recovers the truncated data and adds zeros to the truncated part to maintain the original data format. Then, the write request is sent to the shared cache/memory, and a write replay is generated and transmitted to the core/L1 cache node to confirm the transmission.

When an L1 cache miss is caused by approximate load operation, a read request is issued by the L1 cache with the approximation information. Then, the read request is sent to the packet encoder to generate a read request packet. When the shared cache/memory node receives the read request, the approximation information is extracted from the packet and inserted into the quality control table. When the read reply is generated by the shared cache/memory, the approximate data compression logic checks the approximation information in the quality control table and truncates the data in accordance with the approximation information. Then, the packet encoder prepares the read reply packet and injects it into the network. When the read reply packet arrives at the core/L1 cache node, the data decompression module recovers the data.

The detailed design of the data approximation logic is discussed in Section 4.1. The detailed design of the data decompression logic is discussed in Section 4.2. The detailed design of the quality control table is discussed in Section 4.3.

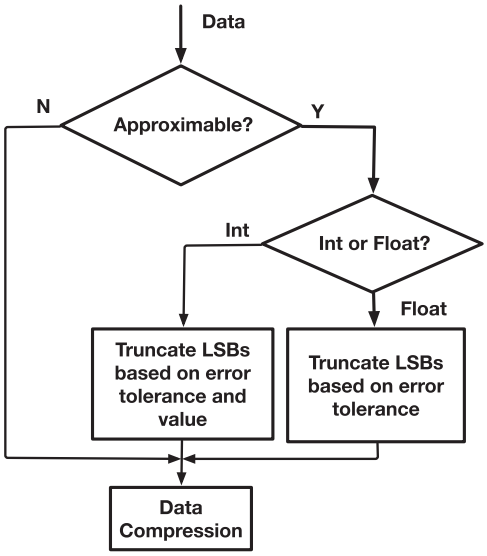


Fig. 8. Data approximation logic flow chart.

4.1 Approximate Data Compression

Fig. 8 shows the flow chart describing the operation of the approximate data compression logic. In the shared cache/memory node, the data approximation logic distinguishes approximable packets from accurate packets depending on the approximation information stored in the quality control table. To obtain approximation information from the quality control table, the data approximation logic sends a read packet, which contains the address, to the quality control table. If the reply message contains approximation information, then the data approximation logic checks whether the data value is an integer or floating point value. If it is an integer value, then the data approximation logic truncates the data using Eq. (6). If it is a floating point value, then the data approximation logic truncates the LSBs by comparing the data's error tolerance against the error thresholds in Table 2, which are derived from Eq. (4). The data approximation logic will always choose the closest error threshold lower than the data's error tolerance to ensure data quality. For example, if the data's error tolerance is 5 percent, then the data approximation logic will choose a data error threshold of 3.125 percent (truncate 18 LSBs) to ensure the data quality. After the data are truncated, the data compressor checks for frequent patterns, as shown in Fig. 3, and compresses the data. Finally, the read reply is sent to the packet encoder and injected into the network.

TABLE 2
Relationship Between the Float Error Threshold and the Number of Truncated LSBs

Float Error Threshold	Truncated LSBs
0.25	21
0.125	20
0.0625	19
0.03125	18
0.015625	17
0.0078125	16
0.00390625	15
0.001953125	14
0.000976563	13
0.000488281	12
0.000244141	11
0.00012207	10
1.52588E-05	7
1.90735E-06	4
0	0

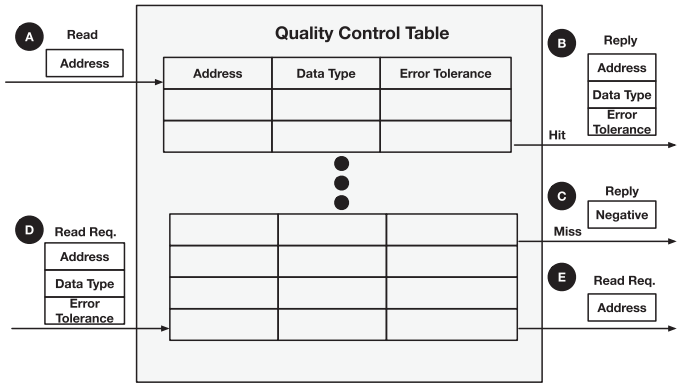


Fig. 9. Hardware design of the quality control table.

In the core/L1 cache node, the data approximation logic distinguishes approximable packets based on the approximation information in the write requests. After the data for a given request are truncated, the data approximation logic deletes the approximation information in the write request and begins to compress the data. After data compression, the write request is sent to the packet encoder and injected into the network.

4.2 Data Decompression

To decompress the data, the decompression logic scans the data and recognizes the codes for specific frequent patterns. Then, the logic recovers the truncated data based on the recognized frequent patterns. After the truncated data are decompressed, the logic checks the length of each piece of data and fills in the truncated parts with zeros. After all data have been recovered, the read reply or write request is sent to the core/L1 cache node or the shared cache/memory node, respectively.

4.3 Quality Control Table

Fig. 9 shows the hardware design of the quality control table at the shared cache/memory node. The quality control table consists of 3 columns of approximation information: addresses, data types, and error thresholds. The data type column contains 1 bit to describe whether the data are integer (1) or floating point (0) in nature.

In the NI of the shared cache/memory node, the data approximation logic receives a read packet (A) containing an address in the quality control table from which to acquire the approximation information for the data. If the address matches an entry in the quality control table, then the table sends a read reply packet with the corresponding approximation information, which includes the address, data type, and error tolerance (B). Then, the corresponding entry is deleted from the table after the reply packet (B) is sent. Otherwise, a negative signal (C) is sent to the data approximation logic to indicate that these data require accurate transmission. When a read request packet (D) arrives at the quality control table, the

TABLE 3
Simulation Environment Setup

NoC Parameters	8 × 8 2D mesh 8 virtual channels Wormhole routing X-Y routing
System Parameters	64 on-chip cores @2 GHz 32 kB L1 instruction cache 32 kB L1 data cache 4-way associative 64-bank fully shared 16 MB L2 cache
Quality Loss Threshold	5%
Approximate Communication Techniques Used For Evaluation	AxBA [19] Quality Control Framework [20] Proposed Framework

table first checks whether it contains approximation information. If so, the table extracts and registers the approximation information. If the requested data contains multiple data types, multiple entries in the quality control table are occupied to store approximation information. Then, the quality control table forwards the read request (E), which now contains only the address, to the shared cache/memory.

5 Experimental Setup

We evaluated the performance of the proposed approximate communication framework using the GEM5 simulator [38] and the AxBench benchmark suite [39]. By running the AxBench benchmarks on the modified GEM5 simulator, we could evaluate the network latency. We used DSENT [40] to capture the dynamic power consumption of the network. The detailed settings for the GEM5 simulator are shown in Table 3.

AxBench [39] is an approximate computing benchmark suite with annotations for approximable code sections and results (Table 4). We analyzed the workloads based on AxBench and implemented an error injection function in the benchmark to simulate the errors introduced by the approximate communication technique. We integrated the approximate data compression logic into the GEM5 simulator to simulate the lossy data compression. In the simulation, our target was 95 percent result quality.

6 Evaluation and Analysis

We evaluate the proposed approximate communication framework from four perspectives: approximation performance, network latency, dynamic power consumption, result quality and overhead. We compare the proposed approximate communication framework with AxBA [19] and the quality control framework [20]. AxBA includes a base-delta compression scheme and a quality control method that requires the program designer to annotate the approximable values and their error tolerances. The quality control framework includes a truncation-based compression scheme and a quality control scheme that allows the network to automatically adjust the quality of transmission after the result error is measured. AxBA, quality control framework and previous approximate communication techniques [9], [19], [20], [41] rely on human engineer to identify error resilient variable in the source code to ensure result quality. In this work, we propose a packet compression method that combines data truncation with frequent pattern compression. To control the result quality, we propose the use of a code analyzer instead of the programmer's annotations. The code analyzer scans the approximable code sections to identify the error-resilient variables and calculate their error tolerances.

6.1 Approximation Performance Analysis

We evaluate the performance of the proposed approximation framework with the following three experiments. The first experiment evaluates the compression rate under a given data error threshold. The high-performance data approximation method can achieve a better compression rate than the low-performance method with the same data error threshold. The second experiment evaluates the number of approximated data packets under a given application quality loss. A high-performance quality control technique can approximate more data packets than the low-performance technique with the same application quality loss threshold. The third experiment evaluates the data compression rate under a given application quality loss. A high-performance approximation framework can achieve a higher compression rate than the low-performance method with the same application quality loss threshold.

Fig. 10 shows the relationship between data compression rate and error threshold when blackscholes benchmark is approximated using different data approximate methods. The data error thresholds are ranging from 1 percent error to 10 percent with a 1 percent stride. As shown in Fig. 10, the proposed framework achieves a significantly higher compression rate than both AxBA and quality control framework. The quality control framework yields a low data compression rate due to the inefficient data compression scheme when approximating integer numbers. The proposed data approximation technique can significantly

TABLE 4
AxBench Benchmark Suite [20], [39]

Benchmark	Input Data Size	Result Data Size	Evaluation Metric
blackscholes	64k floating point (fp) values	64k floating point (fp) values	Average relative error
fft	5k random fp numbers	5k fp values	Average relative error
inversek2j	100k random (x,y) points	100k (x,y) points	Average relative error
jmeint	10k pairs of 3D triangles	10k Boolean values	# of mismatches
jpeg	512 * 512 pixel image	512*512 pixel image	Average pixel diff.
kmeans	512 * 512 pixel image	512*512 pixel image	Average pixel diff.

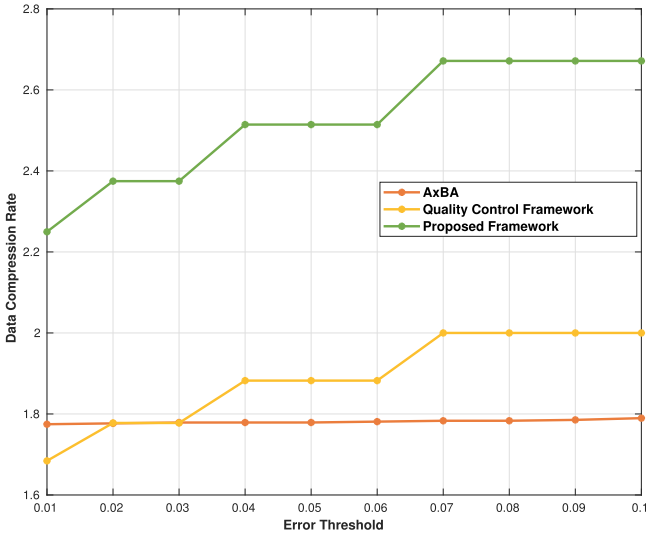


Fig. 10. Data compression rate vs. data error threshold for blackscholes benchmark.

reduce the data size for integer numbers, which improves the compression rate. Moreover, with the error threshold increases, the compression rate increases dramatically for the proposed framework, whereas the AxBA has little improvement. The base-delta compression scheme used by AxBA requires a large number of bits to represent the difference between the base and approximable data. When the delta value is large, increasing data error tolerance has less effect on reducing the size of delta. As a result, the compression rate improvement is limited for AxBA when the data can tolerate more errors.

Fig. 11 shows the percentage of approximated data packets using different quality control techniques when the application can tolerate 5 percent quality loss. The percentage of approximated data packets (P_{ad}) is calculated using Eq. (16), where N_{ad} is the number of approximated data packets and N_d is the total number of data packets

$$P_{ad} = \frac{N_{ad}}{N_d}. \tag{16}$$

As shown in Fig. 11, the proposed quality control technique can approximate more data packets than AxBA and quality control framework. The quality control methods in previous approximate communication frameworks rely on a human engineer to identify the error-resilient variable, which limits

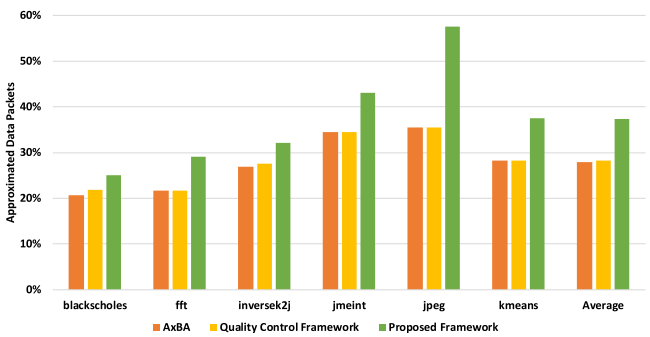


Fig. 11. The percentage of approximated data packets. Quality loss threshold is 5%.

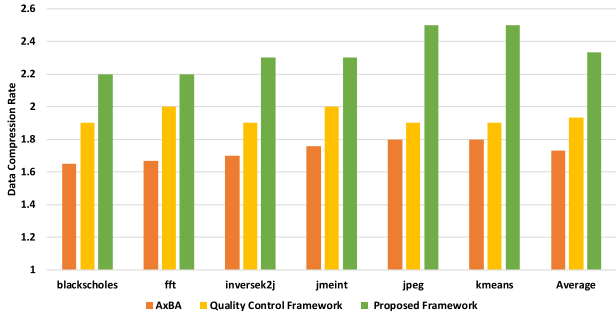


Fig. 12. The data compression rate. Quality loss threshold is 5%.

the number of approximated data packets. On the other hand, the proposed framework discovers error-resilient variables through code analysis and leads to an average of 10 percent more approximated data packets than previous works. The proposed framework can approximate 37 percent of the data packets on average, whereas AxBA and the quality control framework can only approximate 27 and 28 percent of the data packet, respectively. The largest percentage of approximated data packets is achieved (57 percent) when the jpeg benchmark is executed on the proposed architecture.

Fig. 12 shows the compression rate for transmitted data when the quality loss threshold has been set to 5 percent. From Fig. 12, the proposed framework achieves a 2.33 compression rate on average, which is higher than the 1.93 and 1.73 achieved by the quality control framework and AxBA, respectively. Therefore, the proposed framework achieves better performance than AxBA and quality control framework. The high compression rate and more approximated data packets result in lower network latency and power consumption, which will be further discussed in Section 6.2.

6.2 Network Performance Analysis

In this section, we evaluate the NoC performance in terms of network latency and dynamic power consumption using AxBench. The network power consumption includes static power and dynamic power. Static power is the leakage power related to each hardware component. Dynamic power is related to on-chip traffic intensity. Since the proposed approximate communication framework significantly reduces the size of the transmitted packet, we specifically use dynamic power to evaluate the power savings.

6.2.1 *Network Latency*

Fig. 13 shows the evaluation results for the average network latency normalized with respect to AxBA [19]. The network

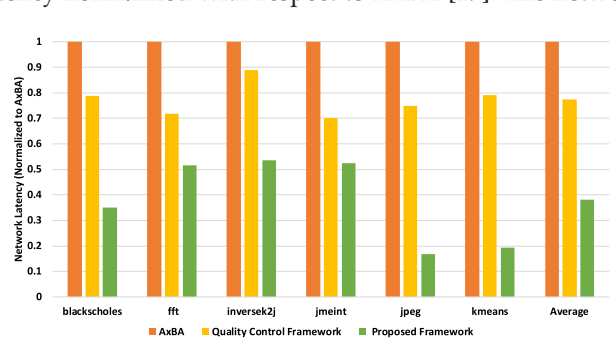


Fig. 13. Network latency: The results are normalized with respect to AxBA (lower is better). Quality loss threshold is 5%.

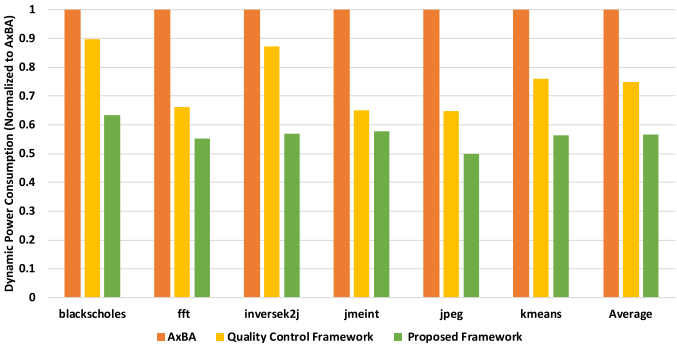


Fig. 14. Dynamic power consumption: The results are normalized with respect to AxBA (lower is better). Quality loss threshold is 5%.

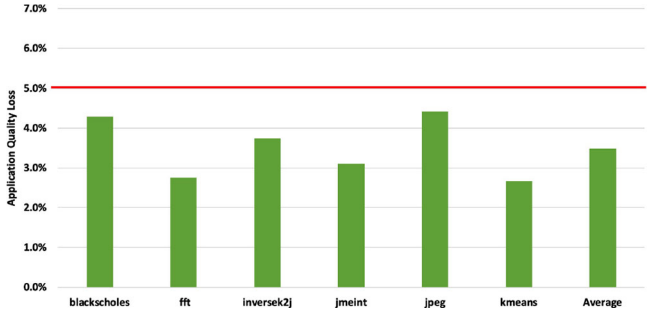


Fig. 15. Application quality loss. The red line represents the application's quality requirement on result.

latency is defined as the number of clock cycles elapsed between the sending of a packet at the source node and the successful delivery of the packet at the destination. Thus, the network latency includes the time consumed in three procedures, packet generation at the source node, packet transmission in the network, and data extraction at the destination node. The packet generation process includes approximate data compression and packet encoding process at the source NI. The data extraction process includes packet decoding and data decompression or updating the quality control table at the destination NI. We compare our proposed design with AxBA and the quality control framework. As shown in Fig. 13, the proposed approximate communication framework achieves an average network latency reduction of 62 percent compared to AxBA. The largest network latency reduction in the experiment is achieved for the jpeg benchmark (83 percent reduction), while the smallest network latency improvement is obtained for inversek2j (46 percent). The proposed framework can achieve such a large latency reduction on the jpeg benchmark due to the high percentage of approximable data packets and the high compression ratio. With the code analyzer, the proposed framework can approximate 57 percent of the data packets, as seen from Fig. 11, while AxBA and the quality control framework can approximate only 35 percent of the data packets. Since most of the approximable data packets in the jpeg benchmark contain integer data, the frequent pattern compression approach used in the proposed framework can achieve the best compression ratio, as seen from Fig. 12. On the other hand, the quality control framework, which converts integers to floating point values to eliminate zeros in the MSBs, achieves its lowest compression ratio on this benchmark (1.9 compared with 2.5 for the proposed framework). As a result, the proposed framework is able to achieve latency reductions of 64, 48, 47, and 80 percent on the blackscholes, fft, jmeint, and kmeans benchmarks, respectively, compared with AxBA.

6.2.2 Dynamic Power Consumption

Fig. 14 shows the amounts of dynamic power consumed by the different approximate communication techniques in comparison to the proposed framework for a target result quality of 95 percent. The dynamic power consumption includes the dynamic power consumed by both NI and NoC. As shown in this figure, the proposed framework achieves an average dynamic power reduction of 43 percent compared with AxBA [19]. The largest dynamic power reduction in this experiment is achieved for the jpeg benchmark (50 percent reduction), while the lowest dynamic power improvement is obtained for blackscholes (36 percent reduction). For the same reason mentioned in the network latency analysis, the proposed framework is able to significantly reduce the number of flits per packet while ensuring the quality of the result. Therefore, the dynamic power consumption of the proposed framework is reduced by 44, 43, 42, and 43 percent on the fft, inversek2j, jmeint, and kmeans benchmarks, respectively, compared with AxBA.

6.3 Result Quality Analysis

The application quality loss is measured using the application-specific metrics given in Table 4. Fig. 15 shows the application quality loss for different benchmarks. The red line in this figure represents the error budget of the approximate computing application. The figure shows that the proposed quality control method incurs quality loss of 4.2, 2.7, 3.7, 3.1, 4.4, and 2.6 percent on the blackscholes, fft, inversek2j, jmeint, jpeg, and kmeans benchmarks, respectively. As shown in Fig. 15, the quality loss for all the benchmarks are less than 5 percent and can be tolerated by the approximate computing application. Fig. 16 compares the accurate result for the jpeg benchmark with the approximate result obtained when the proposed framework is used. The difference between two outputs is negligible and unrecognizable by human vision.

6.4 Overhead Analysis

We implemented the proposed framework with 128 entries in the quality control table using Verilog to evaluate the area, static power, and latency overhead (Table 5). The framework is synthesized with 32 nm technology using Synopsys Design

(a) Accurate Result (b) Approximate Result

Fig. 16. Jpeg benchmark result comparison: The threshold of quality loss is 5%. The result difference is 4.4%.

TABLE 5
Design Overhead for Each NI

Frameworks	AxBA [19]	Quality Control Framework [20]	Proposed Framework
Area	9.28 μm^2	8.23 μm^2	4.79 μm^2
Power	2.9 mW	2.6 mW	1.7mW

Vision software. The synthesis results show that for each NI, the framework incurs an area overhead of 4.79 μm^2, which is less than 1 percent of the total NoC area. Compared to quality control framework [20] and AxBA [19], the proposed framework reduces area overhead by 42 and 48 percent, respectively. When the supply voltage is 1.0 Volt, the framework incurs a power overhead of 1.7 mW for each NI, which is only 0.025 percent of the NoC static power consumption. Compared to quality control framework [20] and AxBA [19], the proposed framework reduces static power overhead by 35 and 41 percent, respectively. Regarding the latency overhead, we find that compression and decompression, as well as searching and updating the quality control table, each require one cycle. Therefore, two cycles are added for each write request, read request, and write reply packet. Three cycles are added for each read reply packet.

7 Related Works

Designing an energy-efficient on-chip interconnection is critical for the current multi-/many-core processors to achieve better performance [4], [21], [23], [42], [43]. Although many techniques have been proposed in the pursuit of efficient NoC design [37], [44], [45], [46], [47], [48], approximate communication is considered to be the most effective way to improve network performance when an application can tolerate modest errors [8], [9], [19], [49], [50], [51], [52], [53], [54]. With reduced accuracy during communication, the approximate communication framework significantly reduce the time and power consumed compared to previous NoC optimization techniques [44], [45], [46], [47]. Various approximate communication techniques have been proposed to enhance the performance of the interconnect [8], [9], [19], [41], [51], [53]. [8] conducted a survey on three promising techniques, namely, compression, relaxed synchronization, and value prediction, to address communication bottleneck issues in massively parallel systems for approximate computing applications. The authors of [9] and [19] proposed lossy data compression techniques to further reduce the size of the error resilience data before packet injection. Specifically, [9] utilized the similarity between two pieces of data and [19] leveraged the value difference between two pieces of data to compress data blocks. In [51], [53], the authors proposed to reduce network congestion by dropping data in the packet before injected into the network. In [53], the authors explored the application's error threshold and proposed an approximation method to drop data accordingly. In [55], [56], [57], the authors proposed several algorithm to reduce the size of query data.

In the approximate communication framework, a quality management system ensures that the data error can be tolerated by the approximate computing application [31], [32], [33]. The proposed approximate communication techniques [9], [19], [41], [51], [53] include a software-based quality management framework, which allows program designer to assign the error threshold. Significant approximation errors can be eliminated during approximate computation when a lightweight result checking system, such as Rumba [34], is used. ApproxIt [58] proposed a run-time quality calibration scheme to control the quality of an approximate computing application with an iterative method. Approxilyzer [59] provides a solution for the quality management system to quantify the quality impact of a single-bit error. In [60], the authors suggested that the result-error can be controlled by managing the input error. However, these quality control frameworks require the program designer to specify the approximable variables, which limits the approximate communication technique to further improve NoC performance. The proposed quality control framework determines error-resilient value through code analysis, which further enhances NoC performance with acceptable quality loss.

Table 6 shows the major difference between the proposed work and our previous works [20], [41]. The data approximation technique in DEC-NoC [41] applies the error control code (ECC) to the most significant bits of a floating-point number (float) to reduce the cost of error correction during transmission. The data approximation technique in quality control framework [20] truncates float to reduce the size of a packet. However, these techniques [20], [41] are insufficient to support the applications consisting of integer numbers because of the overheads of converting integer numbers to floating-point numbers. This paper proposes a data truncation technique, coupled with frequent pattern compression, to reduce the data size for both integer and floating-point numbers. The quality control method in previous works [20], [41] only relies on human engineering to discover approximable variable. Therefore, previous techniques limit the amount of approximable data during on-chip communication. In the proposed framework, we design a code analyzer to automatically identify all the possible error-resilient values in the program, which takes full advantage of approximate communication. As a result, the proposed framework can achieve better performance of on-chip communication without violating the quality requirement of applications.

8 Conclusion

In this work, we propose an approximate communication framework consisting of an approximation technique and a quality control method for power-efficient and low-latency NoCs. We designed a software-based code analyzer to

TABLE 6
Comparison Between Proposed Framework and Previous Works

Frameworks	DEC-NoC [41]	Quality Control Framework [20]	Proposed Framework
Data Approximation Method	Protecting MSBs in float	Truncate LSBs in float	Lossy Compress both int and float
Quality Control Method	Manually Control	Manually Control	Automatically Control

analyze approximable code sections. This code analyzer identifies the approximable variables and calculates their error tolerances based on the quality requirements of the application. We also proposed an approximate communication design for NoCs in which floating-point and integer numbers are compressed based on the error tolerance of the variables. The proposed approximate communication framework increases the number of approximated packets, thus achieving better network performance, while ensuring that the result quality meets the application's requirements. We compared the proposed framework with previous approximate communication techniques called AxBA and quality control framework. Our detailed evaluation showed that the proposed approximate communication framework reduces dynamic power consumption and network latency by 43 and 62 percent, respectively, compared to AxBA.

Acknowledgments

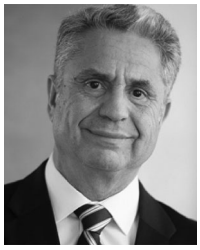
This research is supported by US National Science Foundation Grant CCF-1812495, CCF-1740249 and CCF-1513923.

References

[1] W. J. Dally and B. Towles, "Route packets, not wires: On-chip inteconnection networks," in *Proc. 38th Des. Autom. Conf.*, 2001, pp. 684–689.
[2] G. De Micheli, C. Seiculescu, S. Murali, L. Benini, F. Angiolini, and A. Pullini, "Networks on chips: From research to products," in *Proc. 47th Des. Autom. Conf.*, 2010, pp. 300–305.
[3] R. Ho, K. W. Mai, and M. A. Horowitz, "The future of wires," *Proc. IEEE*, vol. 89, no. 4, pp. 490–504, Apr. 2001.
[4] L. Benini and G. De Micheli, "Networks on chips: A new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, Jan. 2002.
[5] N. Abeyratne *et al.*, "Scaling towards kilo-core processors with asymmetric high-radix topologies," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit.*, 2013, pp. 496–507.
[6] B. Bohnenstiehl *et al.*, "KiloCore: A 32-nm 1000-processor computational array," *IEEE J. Solid-State Circuits*, vol. 52, no. 4, pp. 891–902, Apr. 2017.
[7] O. Villa *et al.*, "Scaling the power wall: A path to exascale," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 830–841.
[8] F. Betzel, K. Khatamifard, H. Suresh, D. J. Lilja, J. Sartori, and U. Karpuzcu, "Approximate communication: Techniques for reducing communication bottlenecks in large-scale parallel systems," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 1:1–1:32, Jan. 2018.
[9] R. Boyapati, J. Huang, P. Majumder, K. H. Yum, and E. J. Kim, "APPROX-NoC: A data approximation framework for network-on-chip architectures," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 666–677.
[10] M. B. Taylor *et al.*, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, Mar./Apr. 2002.
[11] Y. Hoskote, S. Vangala, A. Singha, N. Borkar, and S. Borkar, "A 5-GHz mesh interconnect for a teraflops processor," *IEEE Micro*, vol. 27, no. 5, pp. 51–61, Sep./Oct. 2007.
[12] H. Wang, L.-S. Peh, and S. Malik, "Power-driven design of router microarchitectures in on-chip networks," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2003, Art. no. 105.
[13] 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. Microarchit.*, 2007, pp. 394–406.
[14] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 8, no. 3, pp. 273–286, Jun. 2000.
[15] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2012, pp. 449–460.
[16] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, Mar. 2016.
[17] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Des. Test*, vol. 33, no. 1, pp. 8–22, Feb. 2016.
[18] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *Proc. 18th IEEE Eur. Test Symp.*, 2013, pp. 1–6.
[19] J. R. Stevens, A. Ranjan, and A. Raghunathan, "AxBA: An approximate bus architecture framework," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2018, pp. 1–8.
[20] Y. Chen and A. Louri, "An online quality management framework for approximate communication in network-on-chips," in *Proc. ACM Int. Conf. Supercomput.*, 2019, pp. 217–226.
[21] K. Bergman *et al.*, "ExaScale computing study: Technology challenges in achieving exascale systems Peter Kogge, editor & study lead," 2008. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.165.6676
[22] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *Proc. Int. Conf. High Perform. Comput. Comput. Sci.*, 2010, pp. 1–25.
[23] F. Zahn, S. Lammel, and H. Frning, "Early experiences with saving energy in direct interconnection networks," in *Proc. IEEE 3rd Int. Workshop High-Perform. Interconnection Netw. Exascale Big-Data Era*, 2017, pp. 33–40.
[24] A. Bakhoda, J. Kim, and T. M. Aamodt, "Throughput-effective on-chip networks for manycore accelerators," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, pp. 421–432.
[25] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2011.
[26] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*. Amsterdam, The Netherlands: Elsevier, 2004.
[27] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *Proc. 50th Annu. Des. Autom. Conf.*, 2013, pp. 113:1–113:9.
[28] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "ApproxANN: An approximate computing framework for artificial neural network," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2015, pp. 701–706.
[29] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2013, pp. 1–12.
[30] X. Li and D. Yeung, "Application-level correctness and its impact on fault tolerance," in *Proc. IEEE 13th Int. Symp. High Perform. Comput. Archit.*, 2007, pp. 181–192.
[31] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2014, pp. 309–328.
[32] T. Wang, Q. Zhang, and Q. Xu, "ApproxQA: A unified quality assurance framework for approximate computing," in *Proc. Conf. Des. Autom. Test Eur.*, 2017, pp. 254–257.
[33] C. Xu *et al.*, "On quality trade-off control for approximate computing using iterative training," in *Proc. 54th Annu. Des. Autom. Conf.*, 2017, pp. 52:1–52:6.
[34] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An online quality management system for approximate computing," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 554–566.
[35] IEEE Standards Committee et al., *754–2008 IEEE Standard for Floating-Point Arithmetic, IEEE Computer Society Standard 2008:517*, 2008.
[36] A. Alameldeen and D. Wood, "Frequent pattern compression: A significance-based compression scheme for L2 caches," Dep. Comput. Sci., Univ. Wisconsin-Madison, 2004. [Online]. Available: http://digital.library.wisc.edu/1793/60388
[37] R. Das *et al.*, "Performance and power optimization through data compression in network-on-chip architectures," in *Proc. IEEE 14th Int. Symp. High Perform. Comput. Archit.*, 2008, pp. 215–225.
[38] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.
[39] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "AxBench: A multiplatform benchmark suite for approximate computing," *IEEE Des. Test*, vol. 34, no. 2, pp. 60–68, Apr. 2017.
[40] C. Sun *et al.*, "DSENT - A tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *Proc. IEEE/ACM 6th Int. Symp. Netw.-on-Chip*, 2012, pp. 201–210.
[41] Y. Chen, M. F. Reza, and A. Louri, "DEC-NoC: An approximate framework based on dynamic error control with applications to energy-efficient NoCs," in *Proc. IEEE 36th Int. Conf. Comput. Des.*, 2018, pp. 480–487.

[42] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Commun. ACM*, vol. 58, no. 7, pp. 56–68, Jun. 2015.
[43] R. Marculescu, U. Y. Ogras, L. Peh, N. E. Jerger, and Y. Hoskote, "Outstanding research problems in NoC design: System, microarchitecture, and circuit perspectives," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 1, pp. 3–21, Jan. 2009.
[44] H. Zheng and A. Louri, "An energy-efficient network-on-chip design using reinforcement learning," in *Proc. 56th Annu. Des. Autom. Conf.*, 2019, pp. 47:1–47:6.
[45] K. Wang, A. Louri, A. Karanth, and R. Bunescu, "High-performance, energy-efficient, fault-tolerant network-on-chip design using reinforcement learning," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2019, pp. 1166–1171.
[46] K. Wang, A. Louri, A. Karanth, and R. Bunescu, "IntelliNoC: A holistic design framework for energy-efficient and reliable on-chip communication for manycores," in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 589–600.
[47] H. Zheng and A. Louri, "EZ-Pass: An energy performance-efficient power-gating router architecture for scalable NoCs," *IEEE Comput. Archit. Lett.*, vol. 17, no. 1, pp. 88–91, Jan. 2018.
[48] H. Kim, P. Ghoshal, B. Grot, P. V. Gratz, and D. A. Jimnez, "Reducing network-on-chip energy consumption through spatial locality speculation," in *Proc. 5th ACM/IEEE Int. Symp.*, 2011, pp. 233–240.
[49] A. B. Ahmed, D. Fujiki, H. Matsutani, M. Koibuchi, and H. Amano, "AxNoC: Low-power approximate network-on-chips using critical-path isolation," in *Proc. 12th Int. Symp. Netw.-on-Chip*, 2018, pp. 1–8.
[50] V. Y. Raparti and S. Pasricha, "DAPPER: Data aware approximate NoC for GPGPU architectures," in *Proc. 12th IEEE/ACM Int. Symp. Netw.-on-Chip*, 2018, pp. 1–8.
[51] L. Wang, X. Wang, and Y. Wang, "ABDTR: Approximation-based dynamic traffic regulation for networks-on-chip systems," in *Proc. IEEE Int. Conf. Comput. Des.*, 2017, pp. 153–160.
[52] M. F. Reza and P. Ampadu, "Approximate communication strategies for energy-efficient and high performance NoC: Opportunities and challenges," in *Proc. Great Lakes Symp. VLSI*, 2019, pp. 399–404.
[53] S. Xiao, X. Wang, M. Palesi, A. K. Singh, and T. Mak, "ACDC: An accuracy-and congestion-aware dynamic traffic control method for networks-on-chip," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2019, pp. 630–633.
[54] V. Fernando, A. Franques, S. Abadal, S. Misailovic, and J. Torrellas, "Replica: A wireless manycore for communication-intensive and approximate data," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 849–863.
[55] H. Wang, Q. Wu, L. Shi, Y. Yu, and N. Ahuja, "Out-of-core tensor approximation of multi-dimensional matrices of visual data," *ACM Trans. Graph.*, vol. 24, pp. 527–535, 2005.
[56] A. Behm, C. Li, and M. J. Carey, "Answering approximate string queries on large data sets using external memory," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 888–899.
[57] A. Das, J. Gehrke, and M. Riedewald, "Approximate join processing over data streams," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2003, pp. 40–51.
[58] Q. Zhang and Q. Xu, "ApproxIt: A quality management framework of approximate computing for iterative methods," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, to be published, doi: 10.1109/TCAD.2017.2775236.
[59] R. Venkatagiri, A. Mahmoud, S. K. Sastry 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. Microarchit.*, 2016, pp. 42:1–42:14.
[60] 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.

Yuechen Chen (Student Member, IEEE) received the master's degree in electrical engineering from the George Washington University, Washington, District of Columbia, in 2016. He is currently working toward the PhD degree with the Department of Electrical and Computer Engineering, George Washington University. His research interests include approximate computing and network-on-chips.



Ahmed Louri (Fellow, IEEE) received the PhD degree in computer engineering from the University of Southern California, Los Angeles, California, in 1988. He is the David and Marilyn Karlgaard endowed chair professor of electrical and computer engineering with the George Washington University, and the director of the High Performance Computing Architectures and Technologies Laboratory. From 2010 to 2013, he served as a program director with the National Science Foundations (NSF) Directorate for computer and information science and engineering. His research interests include area of computer architecture and parallel computing, with emphasis on interconnection networks, optical interconnects for scalable parallel computing systems, reconfigurable computing systems, and power-efficient and reliable Network-on-Chips (NoCs) for multicore architectures. He is currently serving as the editor-in-chief of the *IEEE Transactions on Computers*.