Distributed Edge Computing Infrastructure with Low Hardware Cost, Performance Evaluation and Reliability

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Bingzhe Li

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Prof. David J. Lilja

August, 2018

Acknowledgements

First, I would like to express my sincere gratitude to my advisor, Prof. David J. Lilja, for the continuous financial support and for his patient, motivation and extensive personal guidance throughout my entire Ph.D. study at the University of Minnesota. As my advisor, he taught me more than I could ever give him credit for here, especially the capacity of critical thinking I learned from him. As he demonstrated to me, by his example, what a good researcher should be. Throughout these years that he has been my advisor, he has served as an incredible inspiration and talented mentor in my educational development.

I also would like to thank Prof. David Du, Prof. Pen-Chung Yew and Prof. Soheil Mohajer at the University of Minnesota, for their support as members of my Ph.D. committee and for their invaluable comments and suggestions.

As a member of the Center for Research in Intelligent Storage (CRIS), my sincere thank my industry project mentors, Farnaz Toussi and Clark Anderson (currently with IBM), for strong support and great efforts in collaborating on each research project. I also want to thanks group members, CRIS members and friends, Manas Minglani, M.Hassan Najafi, Qianqian Fan, Yaobin Qin, Jinfeng Yang, Peng Li, Xiang Cao, Zhichao Cao, Jim Diehl, Ziqi Fan, Xiongzi Ge, Alireza Haghdoost, Weiping He, Fenggang Wu, Hao Wen, Ming-hong Yang, Baoquan Zhang, Zhichao Cao, Guanda Wu, Bin Li, Han Zhang, Peng Liu, Bo Peng, Tengtao Li, Zhaoxin Liang, Jiaxi Hu, Bo Yuan, Weikang Qian for their support and help during my Ph.D study.

Finally, I would like to thank NSF (grant no. IIP-1439622 and grant no. CCF-1408123) and the member companies for funding my projects.

Dedication

To my family, who always stay with me and support me. Especially to my Mom (Xiuzhen Jia), my dad (Zengjiang Li) and my wife (Qiannan Li), for their advice, their faith and their endless love.

Abstract

Edge computing is one method of pushing the applications, data and computation components away from the centralized system. Each edge computing component has its own computation ability and storage capacity. And they also can communicate with the center server through the cloud and Internet to send the pre-computed data to the center. With increasing the interests of Internet of Things (IoT), more and more edge computing nodes will be attached to the cloud and will build a distributed edge computing infrastructure. In the infrastructure, the center server with storage systems is capable of managing the data and storing the captured data into its storage systems. Moreover, tens or hundreds of edge computing nodes are attached to the infrastructure cloud. Each edge computing node has the computation ability to pre-compute some data captured by the back-end sensors and sends the processed data to the center server via the infrastructure cloud. In this thesis, three aspects of the distributed edge computing infrastructure are investigated, the low hardware cost design for edge computing node, performance evaluation for the central server and the reliability of the central server.

First, for each edge computing component, the hardware cost is an extremely important factor due to the limited power supply and computation ability. Stochastic computing is a promising technology to achieve the low power and area designs. Taking neural networks as the applications at edge computing nodes, we proposed different arithmetic operations in stochastic domain for neural networks to achieve the low hardware cost design for those edge computing components.

Second, with adding more and more edge computing nodes, the network and storage traffic will be increased tremendously in the central server side. Therefore, it is significant to know how much workloads (number of edge computing nodes) the central server can tolerate for system designers. In our work, we proposed replayer tools which are capable of replaying traces to the target system in order to measure the performance of the target system. By doing so, it will be clear to know the ability of the system whether it can tolerate the workload or not and guide the system designer to add more resource to the cenral server such as more storage devices and changing higher frequency CPUs.

Finally, the central server in the infrastructure receives the data sent from each edge computing node and stores them into its storage system. Therefore, it is important to protect the data and to achieve the high performance. In this thesis, we proposed new RAID-6 codes to improve the write and degraded read performance while keeping the reliability of the central server.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Edge computing is one method of pushing the applications, data and computation components away from the centralized system. Each edge computing component has its own computation ability and storage capacity. And they also can communicate with the center server through the cloud and Internet to send the pre-computed data to the center. With increasing the interests of Internet of Things (IoT), more and more edge computing nodes will be attached to the cloud and will build a distributed edge computing infrastructure. In the distributed edge computing infrastructure as shown in Fig. 1.1, the center server with storage systems is capable of managing the data and storing the captured data into its storage systems. Moreover, tens or hundreds of edge computing nodes are attached to the infrastructure cloud. Each of the edge computing nodes has the computation ability to pre-compute some data captured by the back-end sensors and sends the processed data to the center server via the infrastructure cloud. In this thesis, three aspects of the distributed edge computing infrastructure are investigated, the low hardware cost design for edge computing node, performance evaluation for the central server and the reliability of the central server.

First, for each edge computing component, the hardware cost is an extremely important factor due to the limited power supply and computation ability. However, with the application becoming more and more complex such neural networks, the requirement of the computation ability for those edge computing is tremendously increased under the limited power supply. Stochastic computing (SC) [1] is one of such technology that has the advantages of low power consumption and small areas. The stochastic computing

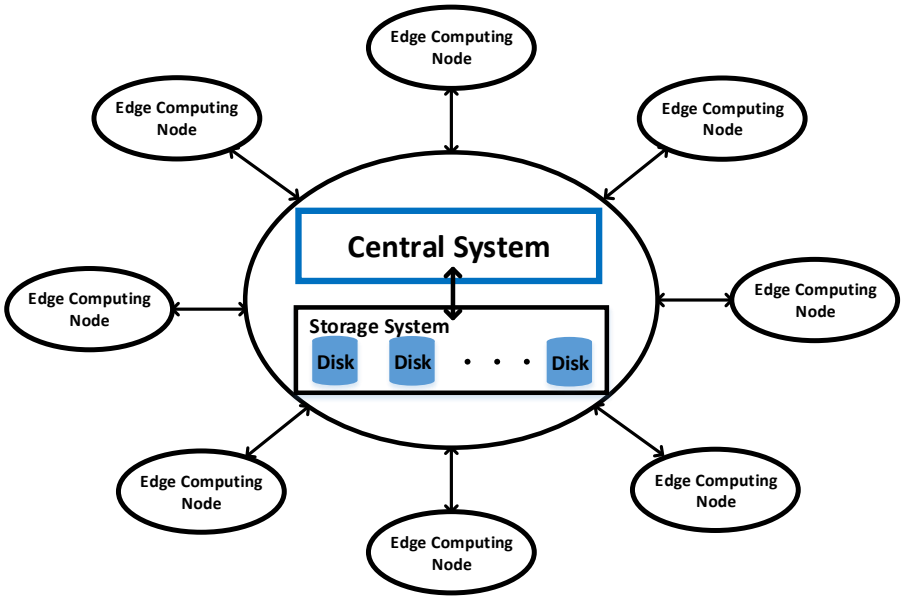


Figure 1.1: The distributed edge computing infrastructure

uses simple logic gates to implement complicated computations. For example, an AND gate in stochastic computing is able to achieve the multiplication operation. Multiple finite state machines implemented by flip-flops can accomplish tanh() and exponential functions, etc. In addition, since the stochastic computing uses bit streams to express the conventional binary values, it provides a good fault tolerate ability. Therefore, stochastic computing is an opportune method to solve those area, fault tolerance and power issues in the back-end applications.

Neural networks, which are designed as a computational model based on neurons, are becoming prevalent in many areas. With growing interests in neural networks, people are beginning to focus on hardware implementations to achieve low-power back-end applications rather than only being limited to software implementations. Some prior works [2][3] are implemented by FPGA. However, those implementations are restricted to implement on mobile devices or Internet-of-things (IoT) due to their high power and large area. Additionally, the power and area of neural networks will increase tremendously with growth of application complexity. Therefore, the requirement of low power

and small area will become more urgent for neural network applications, especially in the power starvation and high noisy environments like edge computing. Therefore, in this thesis, we mainly focus on the neural network applications at the edge computing nodes.

Second, with adding more and more edge computing nodes, the network and storage traffic will be increased tremendously in the central server side. Therefore, it is significant to know how much workloads (number of edge computing nodes) the central server can tolerate for system designers. Trace-driven replayers are used for debugging or mimicking the behavior of real applications on difference environments including large scale-out systems. Previous trace-driven replayers [4][5][6][7][8] are only investigated for either the network or the storage system. In the past, people could ignore the impacts of either storage or network while studing the other since there was a big performance gap between them. However, the scenario has changed as fast storage devices (e.g., NAND flash) become prevalent. Some file system based replayers [9] focused on both network and storage impacts. They only replayed network traces to the target system and used the storage receiver (e.g., iSCSI target) to generate the corresponding storage I/O requests. This method is applicable to the case when all storage I/O requests are generated by the requests delivered by network. However, some storage requests may be independent from network traffic and they are generated by the storage system internally (e.g., log I/Os triggered by I/Os from clients) in the storage and network environment. Therefore, to generate the same workloads as the original applications and to accurately evaluate the performance of target network-storage systems, it is necessary to replay storage and network traffic separately while maintaining some appropriate connection. In our work, we proposed replayer tools which are capable of replaying traces to the target system in order to measure the performance of the target system. By doing so, it will be clear to know the ability of the system whether it can tolerate the workload or not and guide the system designer to add more resource to the cenral server such as more storage devices and changing higher frequency CPUs.

Finally, the central server in the infrastructure receives the data sent from each edge computing node and store them into its storage system. Therefore, it is important to protect the data and to achieve the high performance. Redundant Array of Independent Disks (RAID) [10] as a storage technology uses multiple disks to build a faster and a

more reliable disk system. With the development of the disks, the storage capacity has been steadily increasing but the disks' failure has remained a big concern [11][12]. Therefore, researchers are investigating different methods for recovering from the failure of the RAID systems. RAID code is considered one of the key methods to recover from the failure system. In this thesis, we propose new RAID codes called Partial Stripe code (PS-code) which gives a significantly better write performance and the degraded mode read performance over the recently released RAID codes [13][14][15][16]. The PS-code and Tier-code are vertical RAID codes. By using new labeling algorithms they reduce the number of the parity updates and reads for contiguous writes and thus improve the write and degraded read performance while keeping the reliability of the central server.

In this thesis, there are five major contributions categorized into three aspects as shown in the following.

- The first aspect contribution is about applying stochastic computing into the distributed edge computing nodes and the purpose is to design low-cost hardware architectures for those edge computing nodes.
  - A new stochastic adder called *pos_neg_adders* is proposed for the low-cost stochastic neural network implementations. The proposed architecture can reduce the hardware resource requirements significantly for implementing matrix multiplication while keeping the fault tolerant ability in stochastic neural networks and maintaining reasonable recognition error rates of neural networks.
  - A new stochastic neuron with the hardware-oriented activation function is proposed. It not only well adapts the new hardware-oriented activation function, but also further reduces hardware cost compared to previous stochastic neuron architectures.
  - A stochastic quantized matrix multiplier is proposed with SUC-Adders for the quantized neural networks. The method can efficiently and accurately achieve high accuracy of partial matrix multiplication by only using several AND and OR gates.

- For the second aspect, a new trace replayer is proposed for debugging and evaluating performance of distributed edge computing environments including network and storage workloads.

- A new RAID-6 code called Tier-code is proposed that optimizes the write performance and degraded mode read operation for the reliability of the distributed edge computing environments. Tier-code achieving the least hardware cost and the highest throughput in the VLSI implementation and software testing, respectively. ASIC synthesis results show the hardware implementation of Tier-code has the least cost in terms of area, power, frequency and energy.

The rest of this thesis is organized as follows. The stochastic computing applied for edge computing node is presented in Chapter 2. Chapter 3 introduces the performance evaluation tool for the distributed edge computing environment. In Chapter 4, we propose new erasure codes for the reliability of the distributed edge computing environment. Finally, the summary of total contributions of this thesis and future work are concluded in Chapter 5.

Chapter 2

Edge Computing with Low-Cost Hardware Design

In this section, we mainly focus on how to design a low-cost hardware architecture for edge computing nodes. As mentioned in Section 1, neural networks are becoming prevalent in many areas. With growth of the complexity of neural network applications, the hardware cost of neural networks is increased significantly. Therefore, the requirement of low power and small area will become more urgent for neural network applications. In this section, for each computing node, we use the stochastic computing to implement neural networks hardware design.

2.1 Background

Stochastic computing is a promising technology to achieve the low power and area designs. In stochastic computing domain, a stochastic bit steam is a way to express a floating value, which consists of a sequence of binary digits. The value expressed by a stochastic bit stream is computed from the primary statistic of the bit stream or the probability of any given bit in the stream being a logic '1'. Two types of encoding formats are used in stochastic bit streams, unipolar and bipolar [17]. They can express a real number x in $[0, 1]$ or $[-1, 1]$, respectively. For the unipolar coding format, the probability of having '1's in a bit stream X is $P(X) = x$. On the other hand, for the bipolar format, the probability of seeing '1's in a bit stream X is $P(X) = (x+1)/2$ [18].

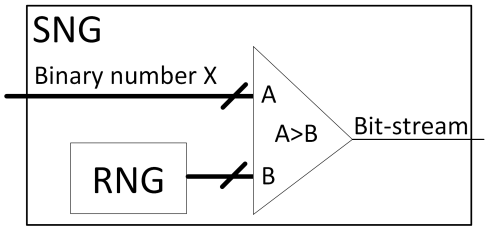


Figure 2.1: A stochastic number generator (SNG) consists of a random number generator (RNG) and a comparator.

Furthermore, to express values which are larger than 1 or smaller than -1, scaled functions are used to scale values into the range of $[-1, 1]$ or $[0, 1]$. Then, the scaled values can be displayed by bit streams. The scaled functions can be found in Eq. 2.1 and Eq. 2.2.

$$P(X) = x/N \tag{2.1}$$

$$P(X) = \frac{x/N + 1}{2} \tag{2.2}$$

where N is the scaling coefficient to make x/N located in [-1, 1] or [0, 1].

2.1.1 Stochastic Number Generator

Stochastic number generator (SNG) in Fig. 2.1 is used to generate bit-streams for computation, which is conventionally implemented using random number generators (RNGs) and comparators. To implement the random number generators, linear feedback shifted registers (LFSRs) have been used with only pseudo randomness. Some stochastic generators [19][20][21][22] have been proposed to shorten the required stream length, in which certain patterns are generated sequentially to increase the accuracy of stochastic computing at short stream lengths. Moreover, by using spintronic devices, previous works [23][24][25][26][27][28] have been introduced to reduce the hardware cost of SNG and have achieved promising performance regarding both power and area consumptions.

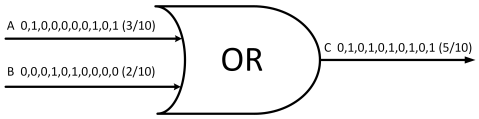


Figure 2.2: OR gate for unipolar addition with unipolar coding.

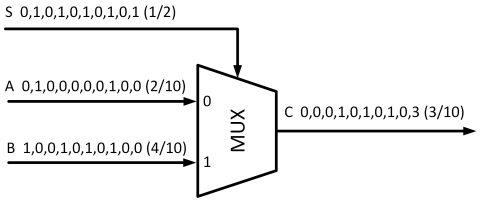


Figure 2.3: MUX gate for scaled addition with unipolar coding.

2.1.2 Stochastic Addition

Dickson *et al.* [29], and Qian *et al.* [30], introduced two methods of adding stochastic bit streams. Dickson *et al.* used a simple standard OR gate to perform approximate addition illustrated in Fig. 2.2. An example in this figure shows the output of OR gate, 5/10, is exactly the result of adding input values, 2/10 and 3/10. According to Eq. 2.3, performing an add operation using OR gate introduces an extra AB term which is considered as an error to the result. The AB term can be ignored when A $\ll$ 1 and B $\ll$ 1.

$$C = AorB = A + B - AB \tag{2.3}$$

Qian *et al.* implemented scaled adding operation using a simple standard MUX. According to Eq. (2.4) when the select line of the MUX is stochastic stream with 0.5 probability, the normal add operation changes to $C = (A + B)/2$ which scales the normal addition results down two times. Fig. 2.3 shows an example of performing scaled addition.

$$C = A \cdot S + B \cdot (1 - S) \tag{2.4}$$

To generalize the simple two-input adder to a multi-input adder, Eq. 2.3 and Eq. 2.4 can be extended to Eq. 2.5 and Eq. 2.6. In the ideal case for Eq. 2.6, N can be any positive

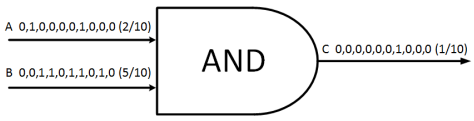


Figure 2.4: An example of multiplication using AND gate.

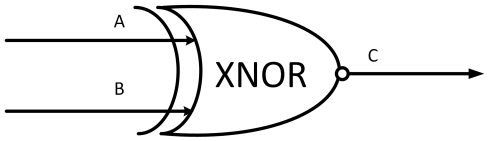


Figure 2.5: XNOR gate for multiplication of bit streams in bipolar format

integer. However, if we are restricted to only use 2-input MUXs in implementation of multi-input adder, N should be a power of 2.

$$C = A_1 or A_2 or A_3 or ... or A_N \tag{2.5}$$

$$C = \frac{1}{N}\sum_{i=1}^{N} A_i \tag{2.6}$$

2.1.3 Stochastic Multiplication

In the stochastic domain multiplication can be implemented just by a simple AND gate for the unipolar (Fig. 2.4) and an XNOR gate for the bipolar representation of bit streams (Fig. 2.5) [31].

To show how an XNOR gate can implement the bipolar multiplication, suppose A, B and C are the encoded bipolar results according to the main bipolar encoding equation. So, the XNOR operation in Eq. 2.7 is change to Eq. 2.8. Considering $A = (1 + a)/2$, $B = (1 + b)/2$ and $C = (1 + c)/2$, by simplifying Eq. 2.8, we have $c = ab$.

$$C = AB + (1 - A)(1 - B) \tag{2.7}$$

$$\begin{aligned} \frac{c+1}{2} &= \frac{a+1}{2}\frac{b+1}{2} + (1 - \frac{a+1}{2})(1 - \frac{b+1}{2}) \\ &= \frac{ab+1}{2} \end{aligned} \tag{2.8}$$

2.2 Low Cost and Highly Reliable Architecture for Neural Network Classifiers

In this section, new stochastic matrix multiplication and activation functions are proposed. By using those, we exploit stochastic bit streams in neural networks to implement the classification of the handwritten digit recognition application completely. Therefore, the stochastic neural network is implemented with finite state machine-based (FSM) stochastic sigmoid and Rectified linear unit (ReLU) functions and the novel stochastic large matrix multiplications.

2.2.1 Motivation

A simple presentation of the matrix multiplication performed in neural networks is shown in Eq. 2.9 where each element in the result matrix C is calculated according to Eq. 2.10.

$$\mathbf{C} = \begin{bmatrix} A_{11} & \cdots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nm} \end{bmatrix} \times \begin{bmatrix} B_{11} & \cdots & B_{1k} \\ \vdots & \ddots & \vdots \\ B_{m1} & \cdots & B_{mk} \end{bmatrix} \tag{2.9}$$

$$C_{ij} = \sum_{s=1}^{m} A_{is} \cdot B_{sj} \tag{2.10}$$

Based on Eq. 2.10, the element C_{ij} of the matrix multiplication in Eq. 2.9 is the sum of the multiplications of elements A_{is} and B_{sj}. In the stochastic domain, multiplication can be implemented easily by a simple two-input AND gate. However, for the addition operations required in computing each element of the result matrix, $\mathbf{C}$, there are usually a large number of addition operations proportional to the size of the number of neurons in neural networks.

As an example, the first layer of neural networks uses the MNIST handwritten digit image dataset [32] as inputs, where the input image size is 28×28. Thus, a matrix multiplication needs to be performed in the first layer, which consists of 784 addition operations for computing one element. As discussed in Section 2.1, there are two types of stochastic adders, MUXs and ORs. The n-input scaled adder defined in Eq. 2.6 always scales the result down n times and the OR gate introduces an extra error with

the term A*B in Eq. 2.3. For implementing such large number of additions, it seems to be challenging for both the MUX and the OR gates.

For the MUX gate, considering the example presented in Fig. 2.6 in which the expected result of adding 784 input values is 0.8 while we want to use 1024-bit streams to perform stochastic additions. In the ideal case, when a MUX is being used as the stochastic adder and there is no correlation between different input bit streams, the stochastic result is about 0.8/784=0.00102. This value is too small to be represented even with 1024-bit streams because the output stream will only have a single '1' out of 1024 bits as explained in Fig. 2.6. Representing such small values can make the system vulnerable even to a very small number of errors. For example, assume in this case a soft error has caused one of the '0' bits in the output bit stream to flip to '1' and so d_{ij} changes to $2/1024 \approx 0.00195$. Now if we scale up this value 784 times and compare with the expected result it shows about $(1.53125 - 0.8)/0.8 \approx 91.4\%$ error rate. Therefore, the MUX gate is not suitable for such a large number of stochastic additions.

For the OR gate, in order to perform the stochastic addition accurately, we scale the inputs of the OR gate down to properly small values (in our case, scale factor $N/2 = 16$) to be able to ignore the effect of the extra AB term in Eq. 2.3 and to avoid large soft errors as MUX adder. For example, if the result of 0.4+0.4 is going to be computed directly by OR gate, it can introduce about 0.4*0.4/(0.4+0.4)=20% error rate just by A*B term. However, if we first scale the input values down 10 times and then scale the result back, the error rate can reduce from 20% to 0.04*0.04/(0.04+0.04)=2%. From the soft error point of view, assume that the sum of 784 values using OR gate is 0.08 which contains about $1024 * 0.08 \approx 82$ '1's in a 1024-bit stream. If a noise causes one bit of the stream to flip from '0' to '1', the output error rate will be just $(83/1024 - 0.08)/0.08 \approx 1.32\%$. Therefore, an OR gate with a proper scaling factor for inputs can be a suitable addition circuit and can help the neural networks to experience very small error rates from both the soft error and also the extra AB term of Eq. 2.3 in matrix multiplications.

2.2.2 Uni_pos_neg adder

The circuits of the stochastic neural networks proposed in this section use both positive and negative values. However, the unipolar representation of stochastic numbers can

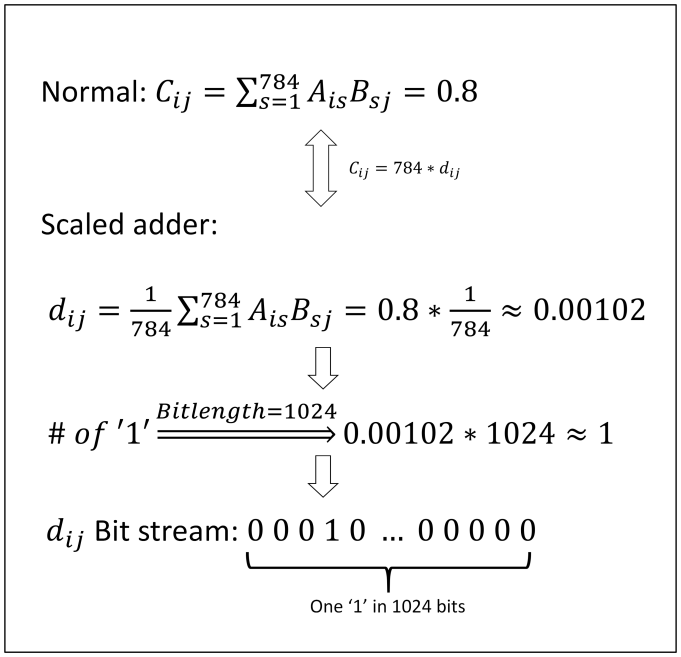


Figure 2.6: One example for transforming between a bit stream and a real number by using scaled adders

only represent pure positive or pure negative values. Since the stochastic tanh function accepts input bit streams in only bipolar format, the output bit streams generated from stochastic matrix multiplication needs to be also in that format. Thus, in this section we propose an *Uni_pos_neg* adder module for implementing the matrix multiplication structure which can automatically handle the format conversion process.

Considering Eq. 2.10 as an example, it computes one element of the result matrix of multiplying two matrices. The element C_{ij} in the result matrix equals to the sum of the products of A_{is} and B_{sj}. Since the products of A_{is} and B_{sj} contain both positive and negative values, we cannot directly use the OR gate as the stochastic adder. Generally, A in neural networks comes from image inputs or outputs of sigmoid function which has been scaled to [0, 1]. B_{sj} is regarded as constant value which is computed from the training process. So, the sign of the product of A_{is} and B_{sj} in neural networks totally

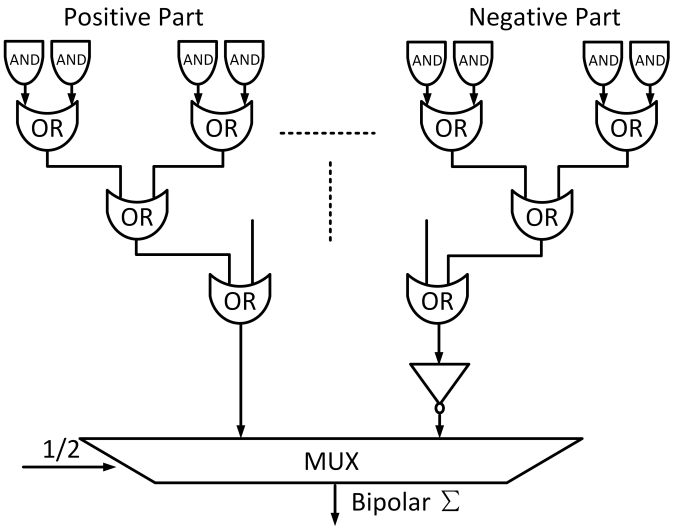


Figure 2.7: Uni-pos-neg-Adder: Big matrix multiplication hardware structure with positive and negative inputs

depends on the sign of B_{sj}, which has been known after training neural networks.

$$POS_{ij} = \sum M_{pos} = \sum_{pos} A_{is} \cdot B_{sj} \tag{2.11}$$

$$NEG_{ij} = \sum M_{neg} = \sum_{neg} A_{is} \cdot B_{sj} \tag{2.12}$$

In the Uni_pos_neg adder, the products are first divided into two parts: negative part and positive part, based on the sign bit of the binary value, B_{sj}. For the positive part, we directly encode the binary value as a unipolar bit stream, then use AND gates to multiply A_{is} and B_{sj}, and go through OR gates to sum all positive products up. For the negative part, at first we consider them as positive values and do the same operations as the positive part. Then, at the final stage of Fig. 2.7 invert the value computed in the negative part to unify the encoding format. In this way the results of Eq. 2.11 and Eq. 2.12 will be obtained after going through some ANDs and OR addition tree structure seen in Fig. 2.7.

In Eq. 2.11 and Eq. 2.12, POS_{ij} and NEG_{ij} are sums of the positive and the negative products in the matrix multiplication with unipolar encoding, respectively. Then, we

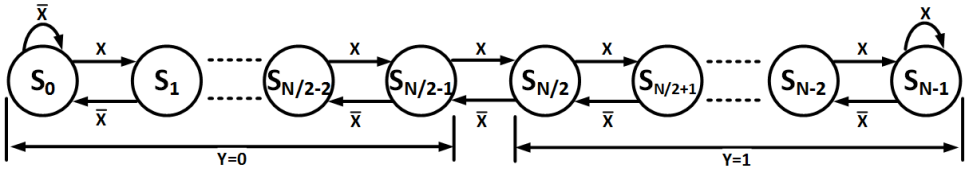


Figure 2.8: Finite state machine of stochastic Tanh function (N must be larger than 2 since when N=2, the FSM will become $Y = X$ or $Y = \bar{X}$)

invert the negative part NEG_{ij} and use a MUX adder to sum POS_{ij} and the inverted NEG_{ij} as seen in Fig. 2.7 to obtain the *sum* variable of Eq. 2.13. Comparing this variable with the bipolar expression $P(X) = (x + 1)/2$, *sum* is the bipolar encoding of C_{ij}, where $C_{ij} = POS_{ij} - NEG_{ij}$. The inverter and MUX automatically transfer two unipolar values, POS_{ij} and NEG_{ij}, into the bipolar format version of $POS_{ij} - NEG_{ij}$. Thus, the bipolar format of C_{ij} is computed by above process. By repeating the above process, the Uni_pos_neg adder can produce the result of matrix multiplications.

$$sum = \frac{POS_{ij} + (1 - NEG_{ij})}{2} = \frac{1 + (POS_{ij} - NEG_{ij})}{2} \tag{2.13}$$

After training the neural networks, the number of negative and positive weights has been decided. So, the number of the gates in the positive part and negative part for each neuron shown in Fig. 2.7 is decided in a neural network classifier.

2.2.3 Sigmoid Function

Brown and Card in [17] proposed a stochastic implementation of the hyperbolic tangent function using a finite state machine (FSM) (Fig. 2.8). Li *et al.* [31] mathematically proved that FSM-based stochastic tanh function works correctly. The FSM presented in Fig. 2.8 implements Eq. 2.14 where N is the number of states in the FSM, and both x and y are the bipolar coding format of the bit streams X and Y, respectively.

$$Y = tanh\frac{N}{2}X \tag{2.14}$$

The experiments performed by Brown and Card indicate that the stochastic tanh function will better approximate the tanh function with larger values of N. Note that $N/2$

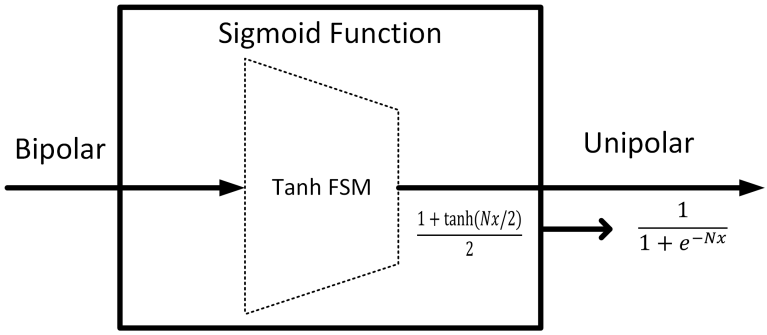


Figure 2.9: Sigmoid function implementation with Bipolar input and Unipolar output from tanh function

($N > 2$) in Eq. 2.14 is the coefficient of the input which increases the value of the input by a factor of $N/2$ during computation of tanh function.

In order to implement the sigmoid function we make some transformations according to Eq. 2.15, Eq. 2.16 and Eq. 2.17. Based on these transformations the sigmoid function can be considered as a bipolar encoding of the tanh function. As seen in Fig. 2.9, both the input and the output of the tanh function are in bipolar format. However, to match with the restrictions of the neural networks we obtain the sigmoid function with a bipolar input and a unipolar output based in Eq. 2.17.

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.15}$$

$$\begin{aligned} \frac{1}{1+e^{-x}} &= \frac{e^{x/2}}{e^{x/2} + e^{-x/2}} \\ &= 1/2(1 + \frac{e^{x/2} - e^{-x/2}}{e^{x/2} + e^{-x/2}}) \\ &= \frac{1 + tanh(x/2)}{2} \end{aligned} \tag{2.16}$$

$$\frac{1}{1+e^{-Nx}} = \frac{1 + tanh(Nx/2)}{2} \tag{2.17}$$

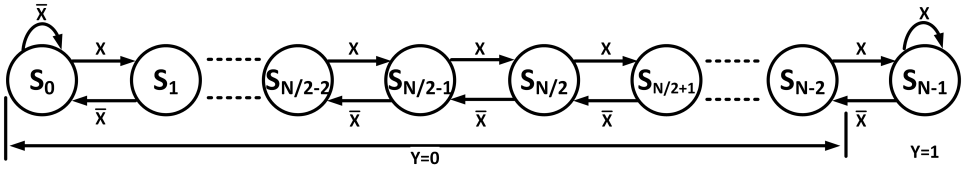


Figure 2.10: Finite state machine of stochastic ReLU function ($N > 2$).

2.2.4 ReLU Function

Rectified linear unit (ReLU) function [33] defined in Equation (2.18) is one type of activation functions in neural networks.

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.18}$$

The stochastic ReLU is proposed in this section which is approximately implemented with FSMs as seen in Fig. 2.10. According to the properties of the stochastic computing linear FSMs in [31], $P(Y)$ is mainly determined by the configuration of the states from $S_{N/2}$ to S_N when $P(X)$ is larger than 0.5. Compared the ReLU function and the absolute function, the difference is at the range of $x \leq 0$. Thus, we changed the states S_0 to $S_{N/2-1}$ of the stochastic absolute FSMs to zeros and approximately derived the ReLU stochastic FSMs as seen in Fig. 2.10. The stochastic ReLU has the bipolar inputs and unipolar outputs.

Fig. 2.11 gives the comparisons between the stochastic ReLU implementation with different numbers of FSMs and deterministic ReLU function. As increasing the number of FSMs, the curves of stochastic ReLU functions become closer to the deterministic ReLU function.

2.2.5 Neural Networks Stochastic Implementation

In this section, we give the introduction of three classical types of neural networks. Their structures in this paper are described. In addition, their stochastic implementation data-flows are provided. According to those data-flows, it helps to clearly see the connections of the stochastic components between different layers in the stochastic neural network implementations.

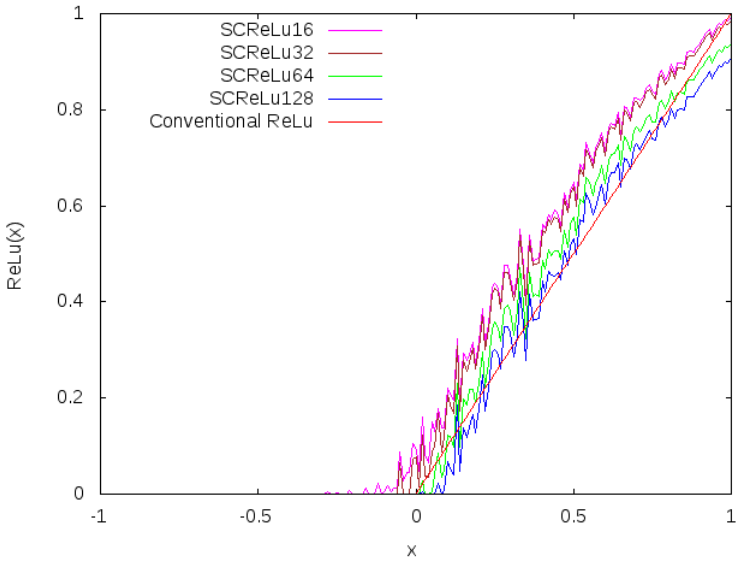
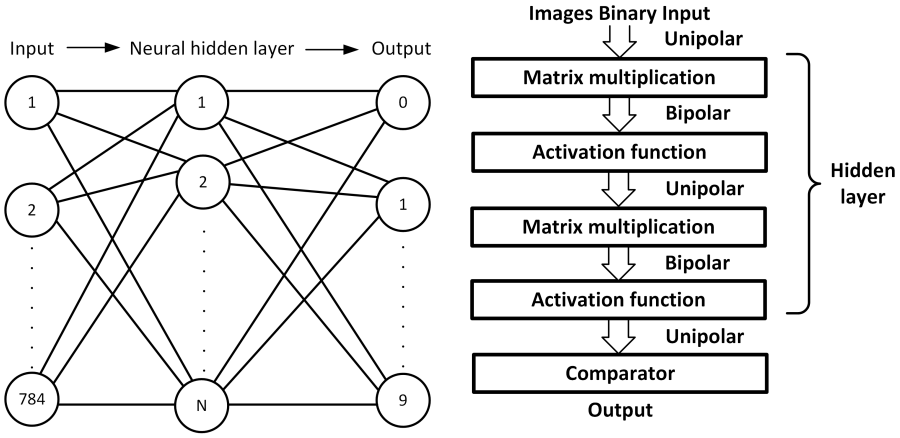


Figure 2.11: The comparisons between the stochastic ReLU implementation with different FSMs and deterministic ReLU function. The bit length of stochastic ReLU implementation is 1024.

The multilayer perceptron (MLP) [34] is a feed-forward neural network which consists of multiple layers in a directed form. The nodes of each layer are fully connected to the next layer. As seen in Fig. 2.12a, those multiple layers can be classified as three functional layers: input layer, hidden layer and output layer. For image recognition problem, the input layer corresponds to the feature of the input image. The hidden layer corresponds to the dependency of the image features. The output layer is the final decision function.

The stochastic implementation of the MLP is followed the Fig. 2.12b. The nonlinear activation function is used with sigmoid functions. The matrix multiplication function is implemented by the uni_pos_neg adder. In the experimental results, we also investigated effects of the recognition rates with varying the number of neurons.

The convolutional neural network (CNN) [35] normally contain three main types of layers, convolutional layer, pooling layer and fully-connected layer. Three functions, tanh(), ReLU function and sigmoid function, can be used as the activation function in the fully-connected layer. The pooling layer mainly focuses on down-sampling input



(a) Two layer MLP structure with N hidden neurons.

(b) The stochastic MLP classifier implementation data flow

Figure 2.12: MLP structure and its stochastic data flow

data. The major operation in the pooling layer is to average the neighbor data values or to obtain the maximum neighbor values.

In this paper, we used the CNN structure as seen in Fig. 2.14. The matrix convolutional operation is quite similar with the matrix multiplication. The only difference is the order of the operated values. So, we can use the uni_pos_neg adder to implement the matrix convolutional operation. Additionally, we use the mean pool to down-sample 2×2 pixel region to one pixel. The mean pool just averages the four pixels. Therefore, we can simply use AND gate to down-sample each pixel by 4 times and then sum them up. The circuit design is seen in Fig. 2.13.

Fig. 2.15 provides the data-flow of the CNN stochastic implementation. The output format of each stage can be perfectly matched to its next inputs. All operations have been introduced in Section 2.1. We investigated the effects of the different activation functions and different bit lengths with stochastic computing shown in Section 2.2.6.

A restricted Boltzmann machine (RBM) [36] is one type of artificial neural networks that is able to represent and solve complicated problems. In this section, we mainly focused on the stochastic RBM classifier implementation. The structure of the RBM implemented in this paper is shown in Fig. 2.16a which has two hidden layers.

We implemented the stochastic RBM classifier following the data-flow in Fig. 2.16b.

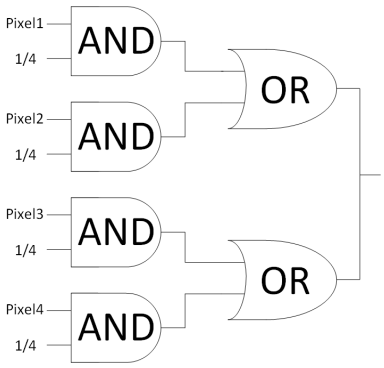


Figure 2.13: Stochastic mean pooling circuit implementation with four AND gates and two OR gates

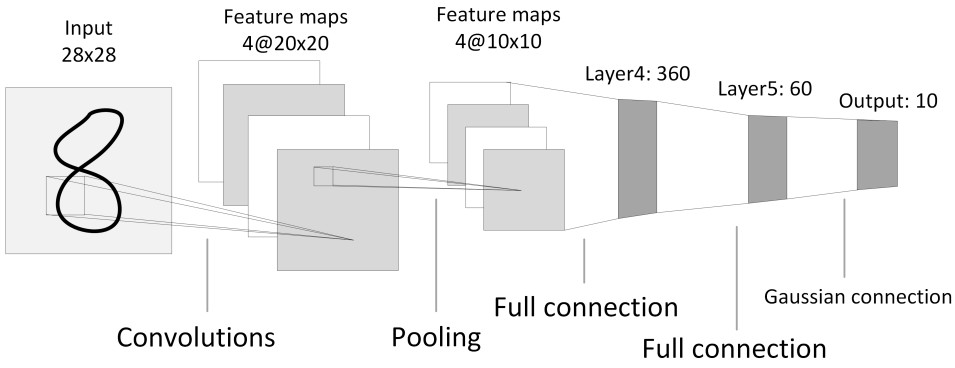


Figure 2.14: A six layers CNN structure

According to the stochastic arithmetic components, each stage of the stochastic RBM classifier is well adaptable to the bit streams formats of its previous stage and its next stage. Thus, it efficiently connects all stages and fully uses the formats transformations of each stochastic component.

2.2.6 Experimental results of Neural Network Comparison

In this section, all neural networks discussed in previous section are implemented as image classifiers. We used the MNIST handwritten digit image dataset [32], which consists of 60,000 training data and 10,000 testing data. All weights and coefficients of

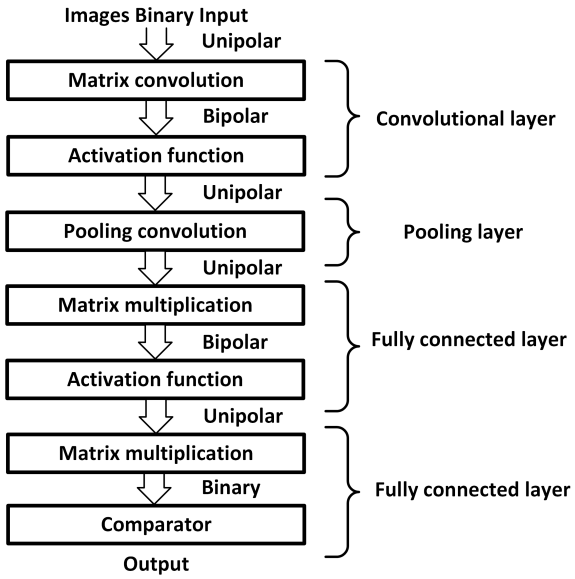
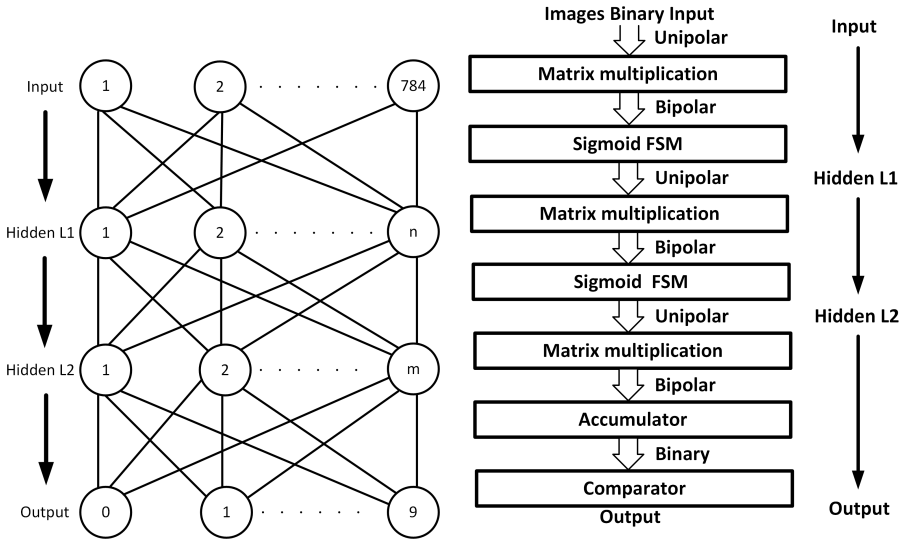


Figure 2.15: The stochastic CNN classifier implementation data flow



(a) Two layers restricted Boltzmann machine structure

(b) The stochastic RBM classifier implementation data flow

Figure 2.16: RBM structure and its stochastic data-flow

Table 2.1: Recognition error rates comparisons of MLP classifier with different bit-lengths and different numbers of neurons.

# of neurons	Conventional MLP	Stochastic MLP(with different bit-lengths)					
		128	256	512	1024	2048	4096
32	4.64%	33.53%	16.00%	9.12%	6.79%	5.92%	5.34%
64	3.76%	34.37%	14.96%	7.39%	5.28%	4.36%	4.12%
128	3.03%	35.51%	14.90%	7.21%	4.90%	4.07%	3.75%
256	2.58%	32.84%	13.03%	6.89%	4.44%	3.99%	3.74%
512	2.23%	31.98%	11.36%	5.36%	3.81%	3.15%	3.08%

Table 2.2: Recognition error rates comparisons of CNN classifier with different bit-lengths and different activation functions.

Activation function	Conventional CNN	Stochastic CNN (with different bit-lengths)			
		512	1024	2048	4096
Sigmoid	1.39%	6.79%	3.95%	2.95%	2.45%
ReLU	1.12%	55.06%	5.32%	3.18%	2.57%

neural networks are trained by the training data and are regarded as constant values for the classifiers. 10,000 testing images are used for testing the stochastic and conventional neural networks. The recognition error rates, fault tolerance and hardware cost of conventional and stochastic neural networks are discussed in following sections.

Recognition Error Rates

To calculate the recognition rates, we ran our simulations for all stochastic neural networks with different lengths of bit streams to obtain the trade-off between the output error rate and the required clock cycles.

We investigated the MLP with different numbers of neurons and with different bit lengths. Neurons number N shown in Fig. 2.12a is varied from 32 to 512. The activation function uses the sigmoid function.

By the observation in Table 2.1, with the number of neurons increasing, the error rates are decreased for the conventional MLP since more neurons mean stronger

Table 2.3: Recognition error rates comparisons of RBM classifier with different bit-lengths and different sizes.

Size	Conventional RBM	Stochastic RBM (with different bit-lengths)			
		512	1024	2048	4096
784-100-200-10	1.96%	10.06%	5.72%	3.91%	3.14%
784-200-400-10	1.35%	7.93%	4.24%	3.08%	2.35%
784-300-600-10	1.17%	6.33%	2.92%	1.80%	1.80%
784-400-800-10	1.10%	5.17%	2.37%	1.52%	1.34%
784-500-1000-10	0.98%	5.10%	2.32%	1.64%	1.32%

computation ability. For the stochastic MLP, the error rates are decreased with longer bit-stream lengths. Since computation time and accuracy of stochastic operations are directly proportional to the length of bit streams [37], the longer lengths mean the lower error rates, but longer computation time (clock cycles). Thus, the right bottom corner value in Table 2.1 is the smallest error rate because it has the longest bit-length and the largest neuron number. Compared to the conventional MLPs, the stochastic MLPs have a little worse recognition error rates. However, starting from 512 bit length, the recognition error rates of stochastic MLPs become less than 10%. With increasing the bit lengths, the recognition error rates of stochastic MLPs are quite close to the recognition error rates of the conventional MLPs.

As seen in Fig. 2.14 and Fig. 2.15, we use the CNN structure built with a 784-1600-400-360-60-10 configuration. Since the CNN contains different types of activation functions, two types of activation function combinations, sigmoid function and ReLU function are investigated in this section. Since the input and output formats of the stochastic tanh() function are not suitable to the stochastic matrix multiplication, we did not investigate the tanh() activation function in this work.

According to Table 2.2, two types of activation functions for the conventional CNN have similar error rates, which are 1.39% and 1.12% respectively. The stochastic CNN with sigmoid function has the better error rate among all bit lengths. The stochastic ReLU CNN with the bit lengths from 1024 to 4096 has just a little worse error rates than the sigmoid CNN. However, it faces an unacceptable error rate at 512 bit length,

which reaches to 55.06%. The reason of the performance cliff from bit-length 1024 to 512 is that the fault tolerance of the ReLU CNN is at the boundary between the bit lengths of 512 and 1024. The longer bit length can provide more precision than the shorter one. Therefore, when the bit length goes down to 512, the ReLU CNN has a big performance jump. In summary, since the bit-streams cannot precisely represent floating values as the conventional CNN, the stochastic CNN with both ReLU and sigmoid functions has only a little higher recognition error rate than the conventional binary CNN implementation.

We investigated the restricted Boltzmann machine with varying its size from 784-100-200-10 to 784-500-1000-10. As seen in Table 2.3, the error rates of the stochastic RBM classifier are decreased with increasing the bit-stream length or increasing size. Thus, the best performance of stochastic RBM is at the size of 784-500-1000-10 and bit-length of 4096. The trends of error rates of stochastic RBMs are quite similar to the trends of stochastic sigmoid CNN. Compared to the conventional RBM, the stochastic RBM obtains from 0.24% to 3.76% worse recognition error rates with varying the bit-lengths from 1024 to 4096.

Fault Tolerance

An attractive feature of stochastic architectures is their ability to tolerate faults, especially the faults caused by manufacturing process, variations, or environmental noise. The reason is that a single bit flip in a long bit-stream results in only a small change in the value of the stochastic number because all bits have the same significance in this type of data encoding [38].

We injected the faults in the same way as Qian *et al.* proposed in [39]. Soft errors are simulated by independently flipping a given fraction of the input and the output bits of each computing element. For example, a soft error rate of 10% means that 10% of the total number of the signal bits is randomly chosen and flipped. Note that, if the output of the computational element is connected to the input of another computational element, we do not inject the error twice on the intermediate line. Since all dataset input values are scaled down into much smaller corresponding values (in order to have more accurate results from performing add operation using OR gates) we inject the fault into the stochastic architecture in two different approaches, the internal computational

elements (ICE) only, and both bit-stream generator and internal computational elements (BG&ICE). In the first approach we do not inject the faults into the bit stream generator modules of the system whereas in the second approach the fault is injected in both internal computational elements and also bit stream generator modules. Injecting faults into the bit stream generator module can only cause to flip a lot of 0 bits to 1 bits because the input values are very small and so 0 bits have formed most of the streams. For both error injection approaches, we do not inject the error to the output of the comparators in the final layer. Since for those back-end devices, the bit-streams can be transfered via WiFi or blue-tooth to computation nodes and such accumulation and comparator in the final layer of neural networks can be handled by those computation nodes. (One reviewer mentioned: Both the RBM and CNN have a binary comparator, so why is their fault tolerance not affected like the APC based CNN). So, the this experiment mainly focuses on the fault tolerance of the internal computation. The authors believe that reporting fault injection results in these two different approaches can better show the fault tolerant capability of the proposed system.

First, we give the results of the ICE error injections for conventional neural networks since the conventional neural networks do not have bit stream generators. As seen in Fig. 2.17 and Fig. 2.18, the conventional neural networks will obtain more than 20% recognition error rates with injecting only 0.01% errors on their computation modules' inputs and outputs. These results provide a reference to demonstrate how well the stochastic neural networks can tolerate errors if the hardware design faces some faults in their computational circuits.

We investigate the fault tolerance with the error injections of the ICE and the BG&ICE varying the error injection rates from 1% to 4% and from 0.1% to 4%, respectively.

As seen in Fig. 2.19, the error injections have very little impacts on their error rates for the ICE error injection. When the error injection rate reaches to 4%, the recognition error rates only have little increase compared to the recognition error rates with 0% error injection. By observing the BG&ICE error injection, it has larger effects on the stochastic MLP than the ICE error injection. With the BG&ICE error injections smaller than 2%, the recognition error rates are close to the error rate with 0% error injection. In summary, the stochastic MLPs with error injections have much higher

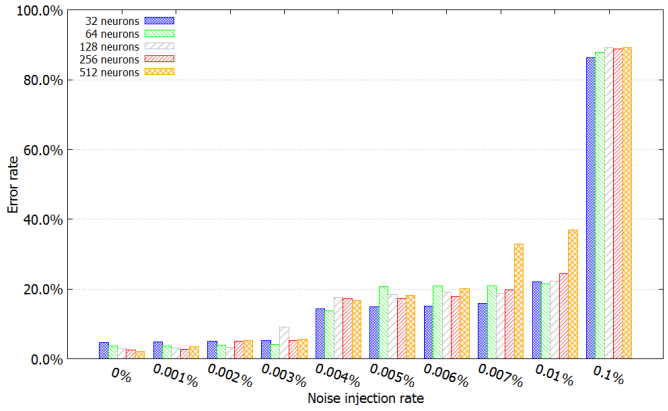


Figure 2.17: Recognition error rates of conventional MLPs with different neurons for different noise injections.

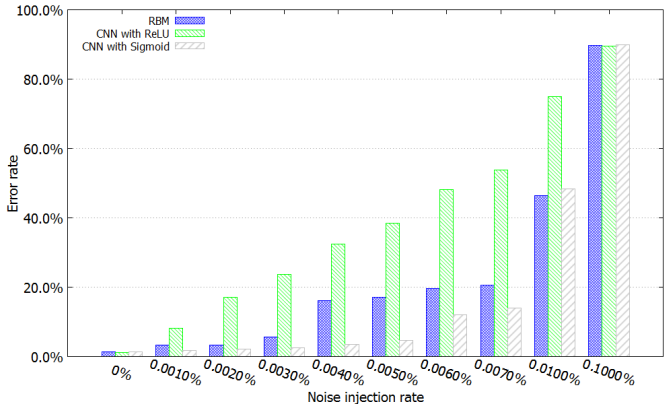


Figure 2.18: Recognition error rates of conventional RBM and CNN with different noise injections.

fault tolerance than the conventional MLPs.

First, we investigated the fault tolerance of the APC-based stochastic CNN [40]. We injected the ICE varying the error injection rate from 0.01% to 0.08%. As seen the results in Table 2.4, it reaches to more than 35% recognition error rate when the ICE error injection only has 0.04%. The reason is that it uses the binary parallel counter in middle layers to sum up the '1's in bit-streams. And the binary results need to be passed to next stochastic layers. The sum results in middle of computation which are the binary expression have very low fault tolerance as investigated by Qian *et al.* [39]. Thus,

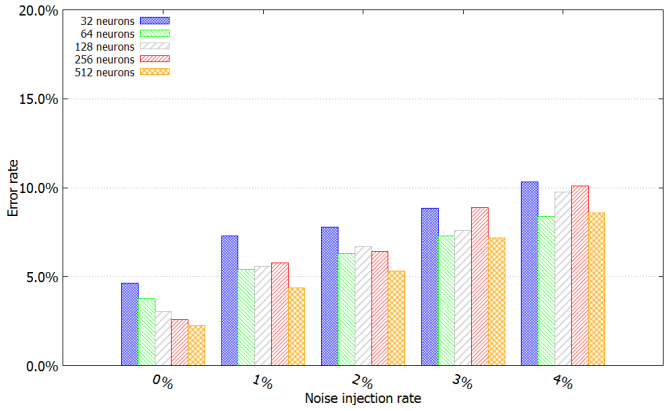


Figure 2.19: Recognition error rates of stochastic MLP with different noise injections for internal computation elements only (ICE).

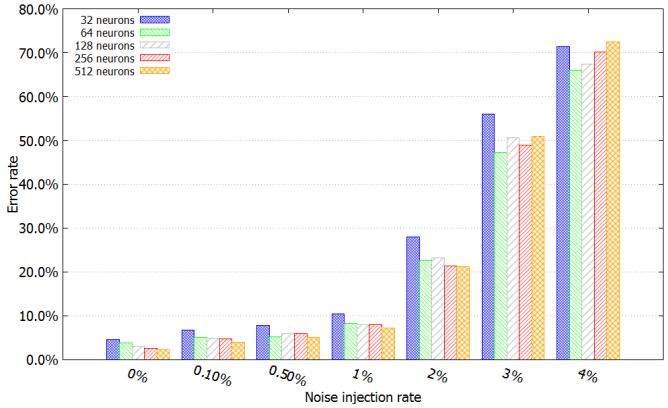


Figure 2.20: Recognition error rates of stochastic MLP with different noise injections for bit-stream generator and internal computation elements (BG&ICE).

Table 2.4: Recognition error rates of APC-based stochastic CNN with different error injection rates.

Error injection rate					
0%	0.01%	0.02%	0.04%	0.06%	0.08%
1.74%	6.08%	7.23%	35.62%	54.56%	75.12%

it causes the whole APC-based stocahstic CNN system has very low fault tolerance.

For our proposed stochastic RBM and CNN, we investigate both two types of error

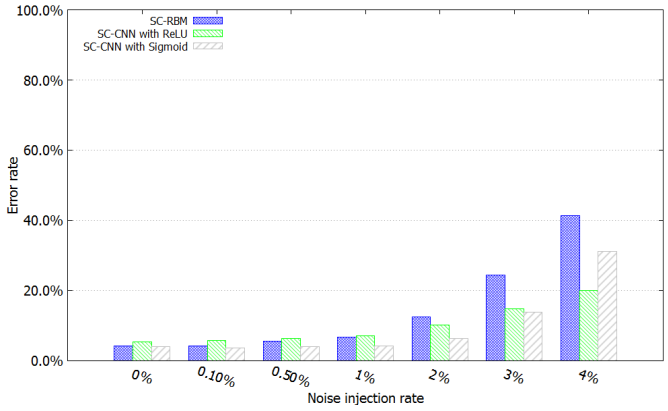


Figure 2.21: Recognition error rates of stochastic RBM and CNN with different noise injections for internal computation elements only (ICE).

injections. The error injection rates varies from 0.1% to 4%. The stochastic RBM uses the structure of 784-200-400-10. Additionally, we investigate the fault tolerance of the stochastic CNN with both ReLU function and sigmoid function.

As seen in Fig. 2.21 and Fig. 2.22, three stochastic implementations have similar increments with increasing the error injection rate. For both of the error injection cases, three stochastic implementations have less than 10% error rates when two error injection rates are increased to 0.5% and 1% respectively, which are much better than their conventional binary implementations and the APC-based stochastic CNN.

Hardware Cost

We used the design compiler to synthesis three typical neural networks with FreePDK 45nm library [41] for the binary and stochastic implementations. The random number generators [42] are included in stochastic hardware synthesis results. The random number generators [42] are included in stochastic hardware synthesis results. The binary implementation uses 9-bit fixed binary with pipelined implementation.

For the hardware comparison of MLPs, we studied two cases of MLPs with 32 and 512 neurons. As seen in Table 2.5, compared to the MLP binary implementation, the proposed stochastic MLPs can achieve about 73x - 78x less area and about 180x - 250x power reduction. Additionally, the results indicate that the stochastic MLP obtain

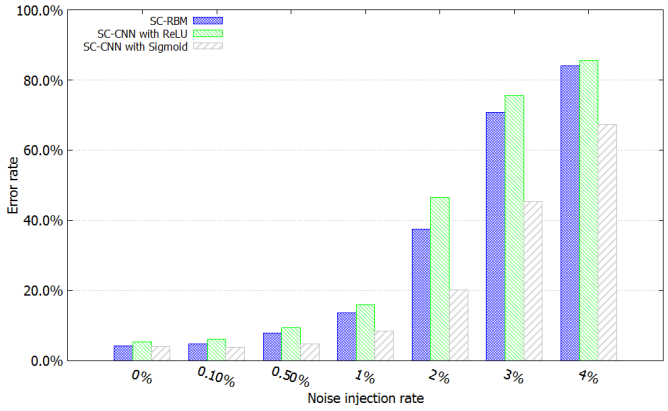


Figure 2.22: Recognition error rates of stochastic RBM and CNN with different noise injections for bit-stream generator and internal computation elements (BG&ICE).

1.75x - 2.3x less energy with 512-bit length compared to the binary MLPs.

Table 2.5: Hardware comparison between stochastic and conventional MLP implementation

MLP	Area (mm^2)	Power (mW)	Energy (nJ)
Binary N#32	16.07	1632	16.32
Binary N#64	32.09	3259	32.58
Binary N#128	64.17	6513	65.13
Binary N#256	128.28	13018	130.19
Binary N#512	256.55	26033	260.34
SC N#32 512-bit	0.34	26.29	134.6
SC N#64 512-bit	0.55	33.33	170.6
SC N#128 512-bit	0.96	47.42	242.8
SC N#256 512-bit	1.79	75.58	386.9
SC N#512 512-bit	3.45	131.96	675.6

Note: N#32 means the number of neurons is 32

For the RBM hardware comparison, we exploited the RBM sizes from 784-400-800-10 to 784-500-1000-10. In Table 2.6, our proposed RBM obtains about 70x and 236x less area and power than the binary RBM implementation, respectively. For the energy,

the stochastic RBM with 512-bit length can save 60% energy reduction compared to the binary RBM implementations. When the bit length increases to 1024, the stochastic RBM only has 15% higher energy consumption than the binary RBM implementation.

Table 2.6: Hardware comparison between stochastic and conventional RBM implementation

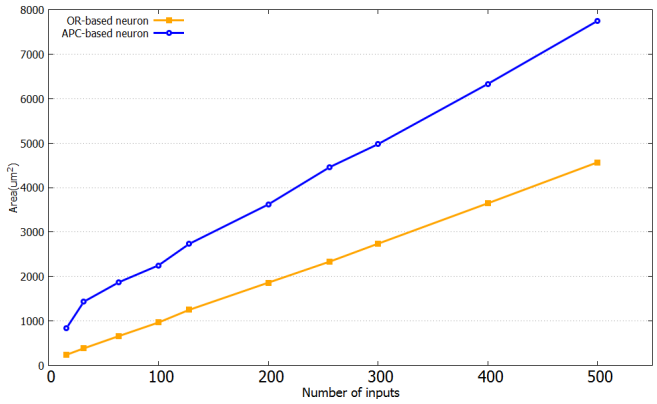
Neural networks	Area (mm^2)	Power (mW)	Energy (nJ)
RBM Binary 400-800	406.65	41365	413.66
RBM Binary 500-1000	570.54	58022	580.23
SC_RBM400-800 512-bit	5.80	175.26	897.31
SC_RBM400-800 1024-bit			1794.6
SC_RBM400-800 2048-bit			3589.2
SC_RBM500-1000 512-bit	8.14	247.19	1265.62
SC_RBM500-1000 1024-bit			2531.2
SC_RBM500-1000 2048-bit			5062.5

Table 2.7: Hardware comparison between stochastic and conventional CNN implementation

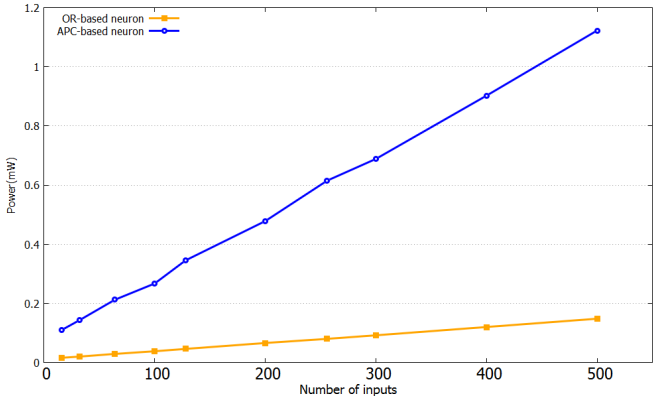
Neural networks	Area (mm^2)	Power (mW)	Energy (nJ)
CNN Binary	192.64	20013	200.13
APC-SC_CNN 64-bit	6.44	818.07	190.12
APC-SC_CNN 128-bit			380.24
OR-SC_CNN 512-bit	2.95	166.55	852.73
OR-SC_CNN 1024-bit			1705.46

For the CNN hardware comparison, the CNN structure uses the 784-1600-400-360-60-10 configuration. We investigated the three types of implementation, binary implementation, APC-based implementation [40] and our proposed implementation.

First we make the hardware comparison between APC-based neuron and our neuron



(a) Area



(b) Power

Figure 2.23: Hardware cost comparison between APC-based and OR-based neuron.

with different numbers of inputs. As seen in Fig. 2.23, with increasing the number of neuron inputs, the difference of area and power between the APC-based neuron and our neuron (OR-based) becomes larger. For one neuron unit, the OR-based neuron obtains at least 2x less area and over 7x less power consumption than the APC-based neuron with 256 inputs. This is because the OR gates cost much less power and area than the binary parallel counter.

To investigate the whole CNN hardware cost as seen in Table 2.7, in terms of area and power our stochastic CNN implementation consumes about 2.4x and 8.8x less cost compared to the APC-based CNN implementation, and derives about 77x and 225x less cost compared to the binary CNN implementation, respectively. For the energy

consumption, with the recognition error rate lower than 4%, the OR-based SC_CNN with 1024-bit length saves about 53% energy than APC-based SC_CNN with 64-bit length, and only has 22% higher energy consumption than the binary CNN. If pursuing lower energy consumption with acceptable recognition error rate, the OR-based stochastic CNN with 512-bit can achieve about 63% and 3x energy reduction compared to the binary implementation and APC-based stochastic CNN, respectively.

2.3 Neural Network Classifiers with a Hardware-Oriented Approximate Activation Function

Neural networks are becoming prevalent in many areas, such as pattern recognition and medical diagnosis. Stochastic computing is one potential solution for neural networks implemented in low-power back-end devices such as solar-powered devices and Internet-of-things (IoT) devices. In this section, we investigate a new architecture of stochastic neural networks with a hardware-oriented approximate activation function. The new proposed approximate activation function can be omitted while keeping the functionality well. Thus, it reduces the stochastic implementation complexity and hardware costs. Moreover, the new architecture significantly improves recognition error rates compared to previous stochastic neural networks with sigmoid function. Three classical types of neural networks are explored, multiple layer perceptron (MLP), restricted Boltzmann machine (RBM) and convolutional neural networks (CNN). The experimental results indicate the new proposed architecture achieves more than 25%, 60% and 3x reduction than previous stochastic neural networks, and more than 30x, 30x and 52% reduction than conventional binary neural networks, in terms of area, power and energy, respectively, while maintaining the similar error rates compared to the conventional neural networks.

2.3.1 Introduction

Neural networks, which are designed as a computational model based on neurons, are becoming prevalent in many areas. With growing interests in neural networks, people are beginning to focus on hardware implementations to achieve low-power back-end applications rather than only being limited to software implementations. Some prior

works [2][3] are implemented by FPGA. However, those implementations are restricted to implement on mobile devices or Internet-of-things (IoT) due to their high power and large area. Additionally, the power and area of neural networks will increase tremendously with growth of application complexity. Therefore, the requirement of low power and small area will become more urgent in such applications.

A new method called stochastic computing (SC) is investigated by some prior works [17][43][38][44], primarily targeting for hardware systems with low power, small area and high fault tolerance. First, in stochastic computing domain, simple logics can achieve many complex arithmetic operations in binary domain. For example, an AND gate can achieve multiplication; Several finite state machines achieved by flip-flops can accomplish the tanh(), exponential function, etc. [31]. Secondly, stochastic computing using bit-streams to represent floating values increases fault tolerance abilities in hardware implementations. Therefore, this method provides a good opportunity to significantly reduce hardware cost of current neural network implementations.

Prior works [45][46][47][48][24][49] have explored stochastic neural networks using different designs. For example, the stochastic RBM implementation is proposed by Li *et al.* [46]. Another work [24] derived an accurate tanh based neurons and improve error rates of stochastic neural networks substantially. Two other works [45][48] implemented a deep stochastic CNN with approximate parallel counters using bipolar and unipolar encoding formats, respectively. However, all of them need to either improve recognition error rates or reduce hardware cost. Moreover, they use stochastic components to implement traditional activation functions, which limits further hardware optimization.

This section proposed a new stochastic neural network architecture with a hardware-oriented approximate activation function. The format of stochastic arithmetic components uses only unipolar encoding, which reduces hardware cost compared to previous bipolar format methods. In addition, the hardware-oriented approximate activation function can be totally omitted in hardware implementations. Three classical neural networks, multiple layer perceptron (MLP), restricted Boltzmann machine (RBM) and convolutional neural networks (CNN), are implemented with the new stochastic structure. The results indicate that the new proposed stochastic architecture with the approximate activation function reduces hardware cost significantly, while remaining similar recognition error rates compared to conventional deterministic neural network

implementations. The main contributions of this section are summarized as follows:

- A hardware-oriented approximate stochastic function is proposed, which can reduce hardware cost with hiding activation function layers.
- A new stochastic neuron with the hardware-oriented activation function is proposed. It not only well adapts the new hardware-oriented activation function, but also further reduces hardware cost compared to previous stochastic neuron architectures.
- The stochastic MLP, RBM and CNN with the new proposed stochastic architecture achieve much less hardware cost than previous stochastic neural network implementations, and also have similar recognition error rates compared to deterministic neural networks.

2.3.2 Motivation

Two major goals are pursued by stochastic neural networks, lower hardware cost and higher recognition error rate.

Previous works [45][46][48][24][49] focused on how to make stochastic implementations matched to original activation functions such as ReLU, tanh and sigmoid in stochastic domain. However, those hardware implementations limit further hardware optimization. Thus, reserver thinking: is there any activation function with an approximate format that can optimize stochastic hardware further?

Secondly, in stochastic computing, two encoding schemes are investigated for different ranges of [0, 1] and [-1, 1], respectively. Therefore, two formats are associated with different arithmetic computing components. For example, multiplication for unipolar format uses an AND gate and multiplication for bipolar uses a XNOR gate which normally consists of three gates including AND and OR gates. As a result, AND gate multiplications achieve smaller hardware cost than XNOR gate multiplications. Furthermore, in large neural networks, the difference of two types of multiplications will be amplified by increasing neuron numbers and has significant impacts on total hardware cost of neural networks. Therefore, in this work we only use unipolar format bit-streams in stochastic neural network implementations which significantly reduce hardware cost of neural network implementations compared to prior works.

Additionally, three most popular activation functions in neural networks are rectified linear unit (ReLU), tanh and sigmoid. Prior works [24][49] have well investigated the error rates of stochastic neural networks with ReLU and tanh. Their implementations had similar recognition error rates to the deterministic neural networks. However, the sigmoid function has not been fully explored. Previous works [47][48] implemented with the stochastic sigmoid function are still substantially worse than conventional implementations. Therefore, above discussions motivate us to design a new stochastic neuron architecture with lower hardware cost and higher recognition error rates.

2.3.3 Approximate Sigmoid Function

We propose a hardware-oriented approximate activation function, which can well match stochastic neural networks. First, we extend the sigmoid function using Taylor series as seen in Eq. 2.19. Since the most neural network coefficients are operated in the range of $[-1, 1]$, the terms with high degrees will be around zero and can be ignored. Therefore, if we get rid of the terms with degrees higher than 2, first two terms are left and the equation becomes $\frac{1+x/2}{2}$. Then, the proposed approximate sigmoid function is defined in Eq. 2.20.

$$\frac{1}{1+exp(-x)} \approx 1/2 + x/4 - x^3/48 + x^5/480 \ldots \tag{2.19}$$

$$y = \begin{cases} 1, & x > 2 \\ 1/4 * x + 1/2, & -2 \leq x \leq 2 \\ 0, & x < -2 \end{cases} \tag{2.20}$$

By observation, $\frac{1+x/2}{2}$ is the bipolar format of $x/2$ in stochastic computing. Suppose a stochastic function can convert bit-streams from the unipolar format to bipolar format, the unipolar format of $x/2$ after going through such function becomes the bipolar format of $x/2$, which is the unipolar format of the approximate activation function in Eq. 2.20. Therefore, activation function layers can be totally omitted if a stochastic neuron has a output format like $\frac{1+x/2}{2}$. In other words, with a output style of $\frac{1+x/2}{2}$ in a neuron, we can totally ignore the activation function layers by keeping bit-stream formats as the unipolar format throughout the whole neural network. Such function is handled in our stochastic matrix multiplication without extra hardware implementation as discussed

in Section 2.3.4. Thus, the omitted activation function layers reduce hardware cost of stochastic neural networks. Moreover, for very deep neural networks like 19 layers VGG Net [50] and 22 layers GoogleNet [51], our implementation can obtain more benefits. Additionally, though the training issue is out of scope of this work, the new proposed activation function provides an opportunity to solve the vanishing gradient issue of sigmoid function because its gradient has a constant value in the range of $[-2, 2]$. As the results shown in Section 2.4.5, the deterministic neural networks keep almost the same recognition error rates by replacing the conventional sigmoid function with the approximate activation function.

2.3.4 Stochastic Neuron Implementation

A single neuron is a basic unit in neural networks, which consists of a matrix multiplication and an activation function. In stochastic neural network implementations, matrix multiplication is a kernel component. On the one hand, generally it has large number of additions that may cause large errors in the whole structure. On the other hand, it ensures correct transformations between two layers. Thus, how to implement a stochastic neuron becomes significantly important for stochastic neural networks.

As discussed in [24], multiplexer adder sacrifices precisions because of scaling down. OR adders only can address input ranges located in $[0, 1]$. Prior work [47] proposed an uni_pos_neg adder to implement such large size of matrix multiplications based on OR adders. However, because OR adders are sensitive to correlation, implementations with OR gates have much worse recognition error rates than deterministic implementations. Other previous works [45][24] implemented matrix multiplications by parallel counters with bipolar inputs and outputs. However, one weakness of their implementations is that the bipolar multiplication is implemented with more hardware cost than the unipolar multiplications as discussed in Section 2.3.2.

In this work, we proposed a new stochastic neuron with matrix multiplications and an approximate activation function as seen in Fig. 2.24. Generally, inputs of matrix multiplication in neural networks are from trained constant weights and image pixels (or activation function outputs). Since image pixel inputs and outputs of the sigmoid function are always in the range of $[0, 1]$, the sign of each input bit-stream in Fig. 2.24 only depends on signs of trained weights, which are constant values in neural network

classifiers. Therefore, numbers of inputs at the negative part and at the positive part are clearly known as N and M in Fig. 2.24 once neural networks have been trained. First, we scaled down constant weights 4 times before they passed through pseudo random bit-stream generators, linear-feedback shifted registers (LFSRs) [42]. Then, according the signs of constant weights, we separate products of inputs and weights into positive and negative parts. The positive and negative parts have extra constant inputs, $1/2 + bias^{+}/4$ and $bias^{-}/4$, respectively, where $bias^{+}$ and $bias^{-}$ are obtained from training process as constant values here. Then, parallel counters sum positive and negative output values together. Finally, the output becomes a bit-stream going through a comparator.

As seen in Eq. 2.21, Eq. 2.22 and Eq. 2.23, P_{is} and Q_{sj} are inputs of matrix multiplication. The product of $P_{is} \cdot Q_{sj}$ is implemented by an AND gate. According to signs of trained weights, products of bit-streams are assigned to the positive or negative part. A_{ij} and B_{ij} are the results of parallel counters from the positive part and the negative part, respectively. By subtracting A_{ij} from B_{ij}, the result in Eq. 2.23 equals to the unipolar format of our proposed approximate sigmoid function in Eq. 2.20.

$$A_{ij} = 1/4 * \sum_{pos} P_{is} \cdot Q_{sj} + 1/2 + bias^{+}/4 \tag{2.21}$$

$$B_{ij} = 1/4 * \sum_{neg} P_{is} \cdot Q_{sj} + bias^{-}/4 \tag{2.22}$$

$$\begin{aligned} A_{ij} - B_{ij} &= \frac{1 + 1/2 * (\sum_{pos} - \sum_{neg} + bias^{+} - bias^{-})}{2} \\ &= \frac{1 + 1/2 * (\sum P_{is} \cdot Q_{sj} + bias)}{2} \end{aligned} \tag{2.23}$$

Therefore, only pre-operations are needed in this work like 4 times scaling down for trained weights and adding 1/2 into $bias^{+}$ and the new proposed neuron does not have any specific hardware implementation for the activation function. Fig. 2.24 indicates a single neuron structure with the new proposed activation function. The format of inputs and outputs in this structure is unipolar encoding.

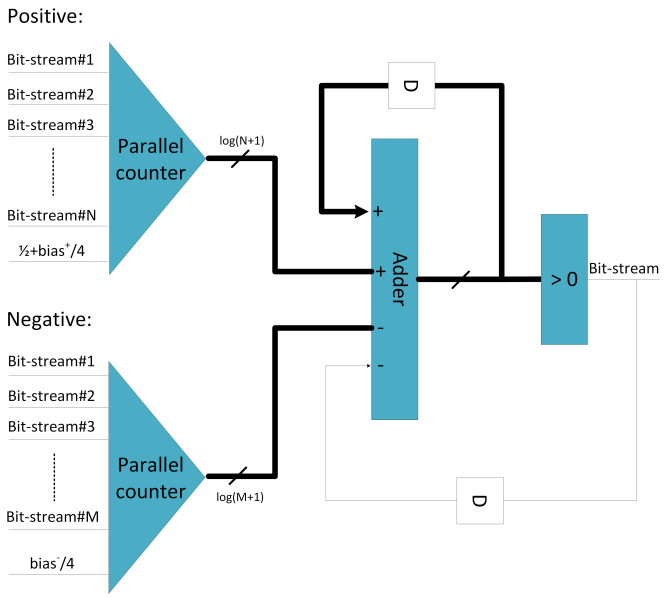


Figure 2.24: The structure of a single neuron with matrix multiplication and approximate sigmoid function. Each bit-stream input is derived from an AND gate multiplication as seen in Fig. 2.4. D is a delay element.

Multilayer Perceptron and Restricted Boltzmann Machine

The multilayer perceptron [34] and restricted Boltzmann machine [36] are two types of neural networks. They share a common feature that nodes in one layer are fully connected to their next layers. In classifiers, the input layer corresponds to features of input images. Hidden layers correspond to dependencies of the image features. The output layer is a final decision function.

The stochastic implementations of the MLP and RBM are shown in Fig. 2.25 and Fig. 2.26. According to stochastic arithmetic components, each stage of stochastic classifiers is well adaptable to bit streams formats of its previous stage and its next stage. The nonlinear activation function is used with the approximate sigmoid function. Since overall formats of bit-streams use the unipolar format, layers of the activation function can be ignored as discussed in Section 2.3.3. Matrix multiplications are implemented by parallel counters with the unipolar format as seen in Fig. 2.24.

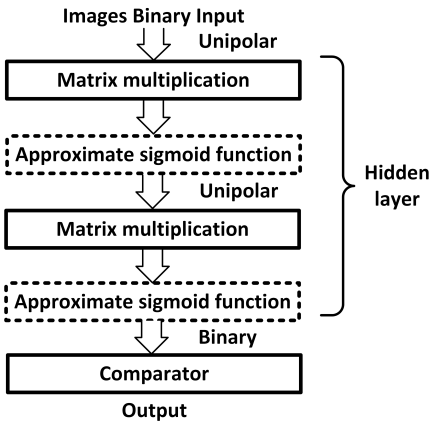


Figure 2.25: The stochastic MLP classifier implementation data flow

Convolutional Neural Network

The convolutional neural network (CNN) [35] normally contains three main types of layers, convolutional layer, pooling layer and fully-connected layer. The pooling layer mainly focuses on down-sampling input data. A major operation in the pooling layer is to average neighbor data values or to obtain the maximum neighbor values. In this work, we use the average pooling to down-sample 2×2 pixel region to one pixel. Therefore, we can use AND gates to multiply each input by 1/4 and then sum them up to obtain their average value as seen in Fig. 2.13. Matrix convolutional operation is quite similar with matrix multiplication. The only difference is from their orders of operated values. Therefore, we use a similar circuit shown in Fig. 2.24 to implement the matrix convolutional operation.

As seen in Fig. 2.14, the CNN is built with a 784-1600-400-360-60-10 configuration. Fig. 2.27 provides a data-flow of the CNN stochastic implementation. Output formats are perfectly matched to their next inputs. The activation function layers are hidden as discussed in Section 2.3.2.

2.3.5 Experimental results of Neural Network Comparison

In this section, all neural networks discussed in the previous sections are implemented as image classifiers. We use the MNIST handwritten digit image dataset [52], which

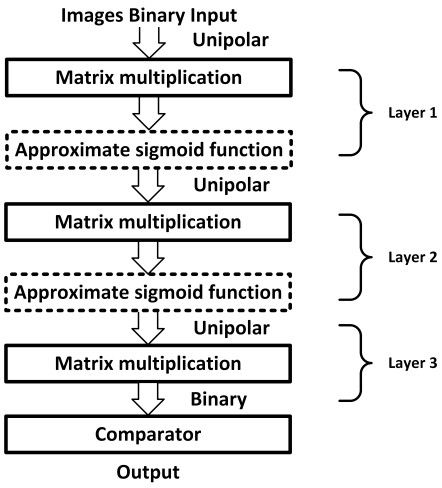


Figure 2.26: The stochastic RBM classifier implementation data flow

consists of 60,000 training data and 10,000 testing data. All weights and coefficients of neural networks are trained and are regarded as constant values for the classifiers. 10,000 testing data are used for testing stochastic and conventional neural networks. Recognition error rates and hardware cost of conventional and stochastic neural networks are discussed in following sections.

Recognition Error Rates

To calculate recognition error rates, we ran our simulations for all stochastic neural networks with different bit-stream lengths to obtain a trade-off between output error rates and required clock cycles.

MLP:

We first investigated the MLP with different numbers of neurons with different bit lengths. Numbers of neurons N shown in Fig. 2.12a are varied from 32 to 512. We compared error rates of stochastic and deterministic MLP implementations. In the deterministic implementation, MLPs with the sigmoid function and the approximate sigmoid function are provided. In the stochastic implementation, we compared our work with the MLP implementation in [46]. The bit-stream lengths are varied from 64

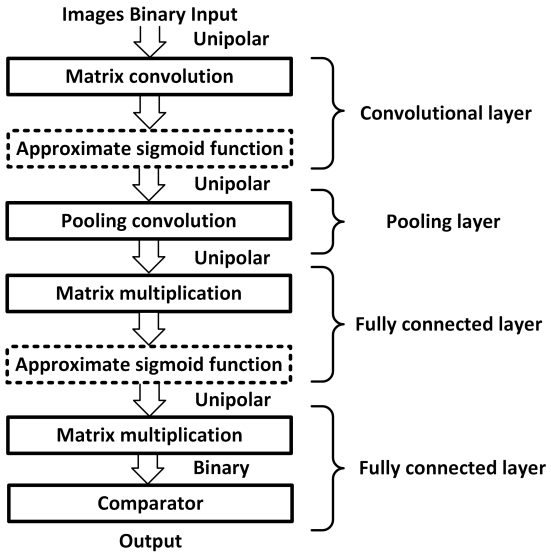


Figure 2.27: The stochastic CNN classifier implementation data flow

to 1024 bits.

As seen in Table 2.8, two deterministic MLP implementations obtain similar error rates with different numbers of neurons. For the stochastic MLP, error rates are decreased with increasing bit-stream lengths. Since computation time and accuracy of stochastic operations are directly proportional to the length of bit streams [37], longer lengths means lower error rates, but longer computation time (clock cycles). Thus, the MLPs with the longest bit-length in Table 2.8 achieve the smallest error rates. Compared to previous stochastic implementations, our implementation derives much better error rates. The OR-tree based stochastic MLP loses recognition ability when bit-stream lengths are smaller than 512. While, our implementation keeps less than 8% recognition error rates with the bit-stream length of 64. For the bit-length of 1024-bit, in our experiments, our method derives about 2.46% - 3.91% recognition error rates which are quite similar to the deterministic implementations. However, the OR-tree based implementation only obtains 3.81% - 6.24% error rates. In summary, our stochastic MLP implementation achieves similar recognition error rates with the deterministic MLP implementation for all cases of neuron numbers.

Table 2.8: Recognition error rates comparisons of MLP classifiers with different bit-lengths and different numbers of neurons.

# of neurons	Deterministic MLP with sigmoid function	OR-tree based MLP [46]				
		64	128	256	512	1024
32	4.04%	N/A	33.53%	17.15%	8.75%	6.24%
64	3.07%	N/A	34.37%	19.00%	8.64%	5.60%
128	2.75%	N/A	35.51%	19.92%	9.12%	5.33%
256	2.43%	N/A	32.84%	10.56%	8.41%	5.35%
512	2.23%	N/A	31.98%	11.36%	5.36%	3.81%
# of neurons	Deterministic MLP with approximate sigmoid function	This work				
		64	128	256	512	1024
32	4.05%	7.02%	5.67%	5.27%	4.13%	3.91%
64	3.07%	4.11%	3.41%	3.23%	3.06%	3.06%
128	2.51%	3.83%	2.98%	2.64%	2.51%	2.43%
256	2.46%	5.30%	2.93%	2.63%	2.54%	2.54%
512	2.26%	7.43%	3.29%	2.69%	2.46%	2.46%

RBM:

We investigated restricted Boltzmann machines with varying sizes from 784-100-200-10 to 784-500-1000-10. We compared error rates of stochastic and deterministic RBM implementations. In the deterministic implementation, the RBM with the sigmoid function and the approximate sigmoid function are provided. In stochastic implementations, we compared our work with the RBM implementation in [46]. The bit-stream lengths are varied from 64 bits to 1024 bits.

As seen in Table 2.9, the deterministic RBM implementation with the approximate sigmoid function has little difference of error rates compared to the RBM implementation with the conventional sigmoid function. However, in stochastic implementations, our proposed method improves error rates significantly compared to the OR-tree based stochastic RBM implementation. The OR-tree based stochastic RBM loses the recognition ability when the bit-stream lengths are smaller than 256. While, our implementation remains less than 3.10% recognition error rates with bit-stream lengths varying

Table 2.9: Recognition error rates comparisons of RBM classifiers with different bit-lengths.

RBM structure	Deterministic RBM with sigmoid function	OR-tree based RBM [46]				
		64	128	256	512	1024
784-100-200-10	1.96%	N/A	N/A	23.38%	10.06%	5.72%
784-200-400-10	1.35%	N/A	N/A	18.83%	7.93%	4.24%
784-300-600-10	1.17%	N/A	N/A	17.53%	6.33%	2.92%
784-400-800-10	1.10%	N/A	N/A	16.69%	5.17%	2.37%
784-500-1000-10	0.98%	N/A	N/A	16.53%	5.10%	2.32%
RBM structure	Deterministic RBM with approximate sigmoid function	This work				
		64	128	256	512	1024
784-100-200-10	2.05%	3.09%	3.01%	3.01%	3.08%	3.04%
784-200-400-10	1.34%	1.88%	1.90%	2.07%	2.04%	1.88%
784-300-600-10	1.15%	1.99%	2.06%	2.16%	1.53%	2.11%
784-400-800-10	1.07%	1.59%	1.58%	1.69%	1.73%	1.88%
784-500-1000-10	1.01%	1.49%	1.53%	1.53%	1.45%	1.46%

from 64 to 1024. For all sizes of RBM, our method derives about 1.45% to 3.09% recognition error rates which are quite close to the deterministic implementations.

CNN:

We investigated three types of stochastic CNN implementations with sigmoid function. As seen in Table 2.10, our stochastic recognition error rates derive a quite close error rates to the corresponding deterministic implementation. However, prior stochastic implementations obtain much worse error rates than their deterministic CNN implementations. Especially for the best case of our work, our stochastic recognition error rates are almost the same as the deterministic implementations. For instance, our 1024-bit implementation obtains 1.42% error rate compared to 1.41% error rate of the deterministic implementation.

Table 2.10: Recognition error rates comparisons of CNN classifiers with different bit-lengths.

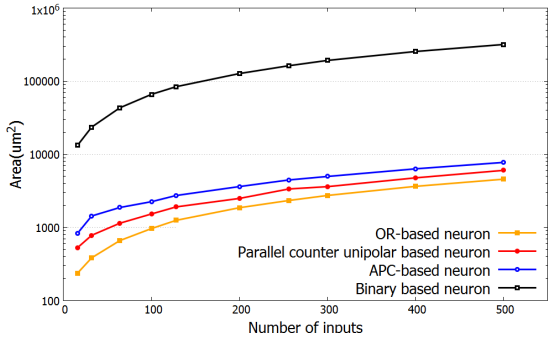
CNN method	Deterministic CNN	Bit-length				
		64	128	256	512	1024
OR-based [47]	1.39%	N/A	N/A	19.35%	6.79%	3.95%
Bipolar-based [48]	2.99%	6.06%	5.58%	4.34%	4.47%	4.49%
Our work	1.41%	2.12%	1.57%	1.49%	1.44%	1.42%

2.3.6 Hardware cost

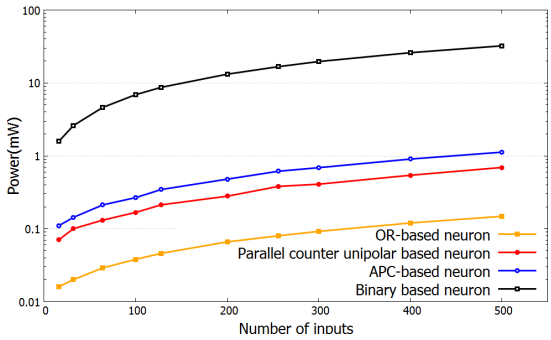
We used the design compiler to synthesis three typical neural networks with FreePDK 45nm library [41] for the binary and stochastic implementations. The random number generators [42] are included in stochastic hardware synthesis results. The binary implementation uses 9-bit fixed binary.

To investigate hardware cost of different approaches in neural networks, we first explored hardware cost of a single neuron because neurons are constitutive units in neural networks. Therefore, the hardware cost of a single neuron can directly reflect total hardware cost of neural networks. In stochastic technologies of neural networks, we can simply categorize the methods into three groups, OR-based neuron [46][47], parallel counter bipolar (PC bipolar) based neuron [45][24] and parallel counter unipolar based neuron (our work). With considering the binary based neuron, we compared four types of methods with different numbers of inputs in terms of area and power as seen in Fig. 2.28a and Fig. 2.28b. The OR-based neuron derives the least power consumption and area due to matrix multiplication only using OR adders. However, as discussed previously, the stochastic neural networks with OR-based neurons have much worse error rates compared to the neural networks with binary implementation and parallel counter implementation. Thus, we will not compare the OR-based neurons in following hardware comparisons. Compared with other two implementations, our work obtains about 47% area and 60% power reduction than the parallel counter bipolar based neuron, and 43x area and 40x power reduction than the binary based neuron on average.

For hardware comparisons of MLPs, we vary number of neurons from 32 to 512.



(a) Area



(b) Power

Figure 2.28: Hardware cost comparison of a single neuron with a sigmoid function varying number of inputs.

Three methods are explored, binary, parallel counter bipolar and parallel counter unipolar. As seen in Table 2.11, our work obtains about 50x area reduction and about 45x power consumption reduction than the binary implementation. Compared to the parallel counter bipolar implementation, we reduce about 25% area and about 60% power consumption. Additionally, with the bit-length of 64, our work achieves about 60% and 3x energy reduction compared to the binary implementation and parallel counter bipolar implementation, respectively.

For the RBM, we explored different RBM sizes from 784-100-200-10 to 784-500-1000-10. In Table 2.12, our proposed stochastic RBM obtains about 50x, 45x and 52% reduction compared to the binary RBM implementation in terms of area, power and energy. In addition, our work reduces about 30% area, saves power consumption about 63% and achieves about 2.5x energy reduction compared to the parallel counter bipolar

Table 2.11: MLP hardware comparison between stochastic with 64 bit-length and conventional MLP implementation

MLP	# neurons	Area (mm^2)	Power (mW)	Energy (nJ)
Binary	32	16.07	1632.4	7.42
	64	32.09	3258.9	14.83
	128	64.17	6512.7	29.63
	256	128.28	13018	59.23
	512	256.55	26033	118.45
Stochastic (PC bipolar)	32	0.40	59.04	13.99
	64	0.78	115.17	27.29
	128	1.53	227.37	53.88
	256	3.03	451.79	107.07
	512	6.04	900.54	213.43
Stochastic (Our work)	32	0.32	36.71	4.70
	64	0.62	70.54	9.03
	128	1.20	138.41	17.72
	256	2.39	274.19	35.11
	512	4.76	545.67	69.86

implementation.

For the CNN hardware comparison, we compared with different methodologies. As seen in Table 2.13, our implementation obtains 31x, 30x, 82% reduction compared to the binary CNN implementation and saves about 53.5%, 58.8% and 3.4x compared to the prior stochastic CNN with sigmoid function in terms of area, power and energy, respectively.

In summary, for stochastic implementations, there is a trade-off between energy and recognition error rates. Generally, higher recognition error rates need longer bit-length and then cause larger energy consumption. Thus, by increasing bit lengths, our implementation loses energy advantages compared to conventional binary implementations. When the bit length is increased to 128-bit, our works achieve a little bit worse energy

Table 2.12: Hardware comparison between stochastic RBM with 64 bit-length and conventional RBM implementation

Neural networks	Area (mm^2)	Power (mW)	$Energy$ (nJ)
RBM Binary 100-200	63.96	6530.11	29.71
RBM Binary 200-400	152.40	15569.3	72.86
RBM Binary 300-600	267.69	2723.6	122.96
RBM Binary 400-800	406.65	41365.7	191.55
RBM Binary 500-1000	570.54	58022.7	274.76
Parallel counter bipolar implementation			
SC_RBM 100-200	1.67	234.15	55.49
SC_RBM 200-400	3.87	550.09	127.84
SC_RBM 300-600	6.59	949.87	204.58
SC_RBM 400-800	9.86	1436.77	332.24
SC_RBM 500-1000	13.74	2015.55	451.07
Parallel counter unipolar implementation (our work)			
SC_RBM 100-200	1.27	143.46	18.41
SC_RBM 200-400	2.91	329.92	37.69
SC_RBM 300-600	5.01	571.25	62.20
SC_RBM 400-800	7.60	867.09	111.02
SC_RBM 500-1000	10.76	1234.7	180.26

Table 2.13: Hardware comparison between stochastic CNNs with 64 bit-length and conventional CNN implementation

Neural networks	Area (mm^2)	Power (mW)	$Energy$ (nJ)
Binary CNN	122.25	13479	91.59
Bipolar-based CNN	5.91	740.94	172.20
Our work	3.85	439.53	50.21

than the conventional binary implementations. However, our work will keep the same energy reduction compared to previous stochastic neural networks if stochastic neural networks use the same bit lengths.

2.3.7 Conclusion

In this section, we explored three classical neural networks with a new activation function. First, the approximate sigmoid function can make neural networks well fit to their stochastic implementation. Secondly, a new proposed neuron with the hidden approximate activation function improves recognition error rates of stochastic neural networks further. Therefore, the omitted activation function and unipolar based architecture significantly reduce hardware cost of neural networks. According to our experimental results, our stochastic neural networks obtain similar recognition error rates with their binary implementation and better error rates than previous stochastic neural networks with the sigmoid activation function. Moreover, our stochastic implementation achieves more than 25% area reduction, 60% power saving and 3x energy reduction than previous stochastic neural networks, and more than 30x, 30x and 52% reduction than the conventional binary neural networks in terms of area, power and energy, respectively.

2.4 Quantized Neural Network for Low-cost Hardware Design

With increased interests of neural networks, hardware implementations of neural networks have been investigated. Researchers pursue low hardware cost by using different technologies such as stochastic computing and quantization. For example, the quantization is able to reduce total number of trained weights and results in low hardware cost. Stochastic computing aims to lower hardware costs substantially by using simple gates instead of complex arithmetic operations. In this section, we propose a new stochastic multiplier with shifted unary code adders (SUC-Adder) for quantized neural networks. The new design uses the characteristic of quantized weights and tremendously reduces the hardware cost of neural networks. Experimental results indicate that our stochastic design achieves about 10x energy reduction compared to its counterpart binary implementation while maintaining slightly higher recognition error rates than the binary

implementation.

2.4.1 Introduction

Neural networks as a computational model based on neurons, are becoming a prevalent method in many areas. Researchers start to investigate neural networks with hardware implementation, rather than restricting to software implementations. In the hardware implementations, two major directions are explored. One aims to accelerate neural network performance by using FPGA or VLSI designs [2][3]. The other targets to low-power applications such as mobile devices and Internet-of-things (IoT).

To achieve low hardware cost, quantization and stochastic computing have been investigated, respectively. First, stochastic computing [1] is known as a low-cost and fault-tolerate technology used in approximate computation areas. By using simple gates to implement complex arithmetic operations, stochastic computing achieves extremely low hardware cost. For example, several finite state machines are able to implement exponential function or tanh function. Thus, stochastic computing is a promising approach to reduce hardware cost in many applications. Second, the quantization is an approximate method in digital designs. In terms of smaller quantization levels, quantization introduces larger quantization errors while reducing complexity of applications significantly. Therefore, to investigate low-cost neural networks, people applied those two approaches into neural network implementations, respectively.

On one hand, previous works have investigated neural networks by using stochastic computing. For example, the stochastic RBM implementation was proposed by Li *et al.* [46][47]. Another work [24] implemented a tanh based neuron and improved error rates of stochastic neural network substantially. Two works [45][48] implemented deep stochastic CNNs with approximate parallel counters using bipolar and unipolar encoding formats, respectively. On the other hand, some researchers studied efficient hardware implementations of neural networks with the quantization technology. For example, Hwang *et al.* [53] studied fixed-point feedforward neural networks with different quantization levels. According to their results, they achieved similar recognition error rates compared to floating-point neural networks. Another work [54] exploited sparseness in vocabulary speech recognition to reduce the model size and execution time. However, the advantages of both quantization and stochastic computing in neural networks are

not well investigated. Therefore, there is an opportunity to design a quantized neural network by using stochastic computing in order to reduce hardware cost of neural networks further.

In this section, we proposed a new stochastic neural network architecture for quantized neural networks. The goal of this work is to combine two technologies, quantization and stochastic computing in neural networks and then to reduce hardware cost of neural networks further. Regarding neural network implementations, first we retrain a fully trained neural network with the back-propagation algorithm in order to obtain neural networks with quantized weights. Second, according to the quantized weights, a stochastic matrix multiplication is implemented with a new component called shifted unary code adder (SUC-Adder). The SUC-Adder is capable of efficiently summing up products of matrix elements and quantized weights. As a result, the SUC-Adder can decrease the number of inputs of parallel counters and then reduce hardware cost of the whole neural network.

The main contributions of this section are summarized as followed:

- Retrained neural networks with different quantization levels are implemented by stochastic computing.
- A stochastic quantized matrix multiplier is proposed with SUC-Adders for the quantized neural networks. The method can efficiently and accurately achieve high accuracy of partial matrix multiplication by only using several AND and OR gates.
- The stochastic neural networks with proposed multipliers achieve much lower hardware costs compared to previous works and binary implementations while maintaining very close recognition error rates to binary neural networks.

2.4.2 Motivation

A deep neural network normally has thousands of weights, which result in long execution time within CPU/GPUs or large hardware cost for neural network accelerators. Therefore, previous researchers reduced the model size of deep neural networks by quantizing all trained weights. Consequently, the quantized weights decreased complexity of neural networks and finally lowered execution time and hardware cost in both conventional

hardware and CPU/GPU implementations.

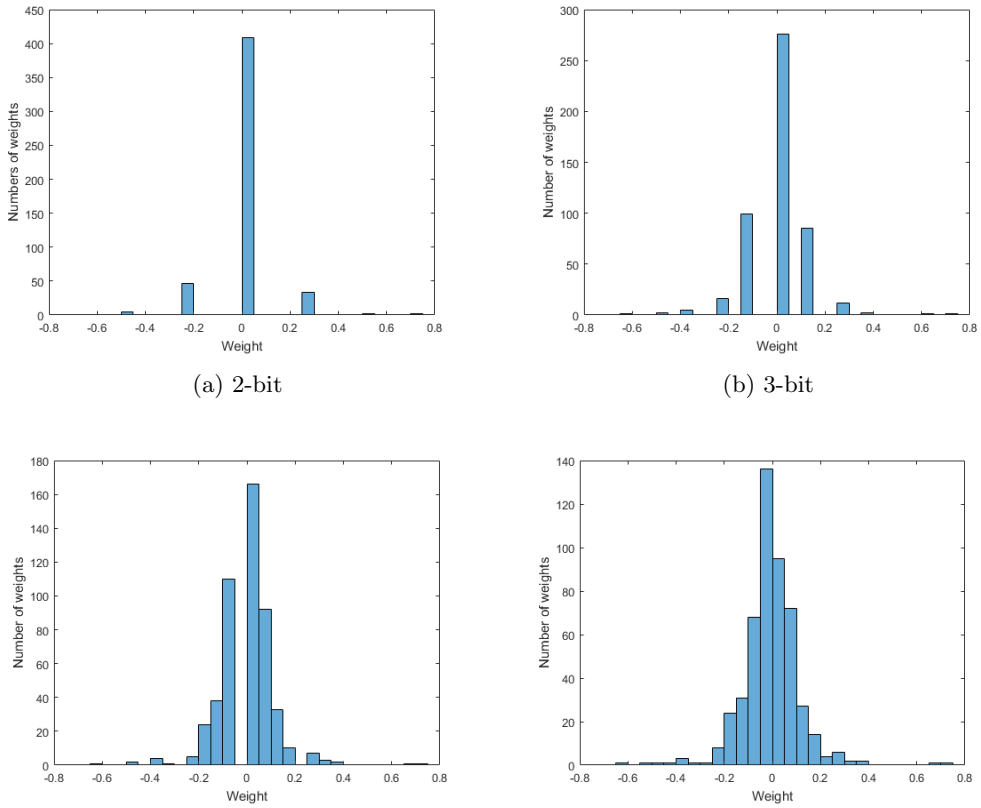
Similar to binary implementations, one advantage of quantization in stochastic computing is that the number of non-zero weights is reduced because because near-zero weights are quantized to zeros. As seen in Fig. 2.29, a large portion of weights in neural networks are around zero. Therefore, removing near-zero weights can reduce a large number of operations. Another advantage is that original weights in binary implementations are quantized to their nearby levels and thus thousands of weights are replaced by several constant values. As a result, the quantized weights will further reduce resource utilization.

Nevertheless, directly applying stochastic computing to quantized neural network may only obtain limited benefit. As the second advantage mentioned above, the benefit of quantized neural networks for stochastic computing is reducing the number of stochastic number generators (SNG). However, because of correlation between bit-streams, for the same value, we cannot encode them by the same SNG when they are operated by each other in conventional stochastic implementations. In addition, some researchers [42][55] have investigated low-cost SNGs. As a result, the benefit of reducing number of SNGs will become much smaller in future. Therefore, how to efficiently apply stochastic computing to the quantized neural networks motivates us to design a new stochastic neuron architecture.

2.4.3 Neural Network Retraining

In general, a neuron implements a function of $p = \sigma(\mathbf{A} * x + \mathbf{B})$, where σ is the activation function, x is input of the neuron, $\mathbf{A}$ is the coefficient for matrix multiplication, and $\mathbf{B}$ is the bias. Binary neural networks quantize all weights ($\mathbf{A}$ and $\mathbf{B}$) and inputs (x) in retraining process [53] in order to decrease sizes of neural networks and then reduce resource utilization. However, the trade-off is that the quantized neural networks increase recognition error rates.

For stochastic quantized neural networks, we only quantize the coefficient $\mathbf{A}$. The reasons are given as follows: First, our new proposed architecture obtains benefits from the weight $\mathbf{A}$ which is used for multiplications. Thus, it is not necessary to quantize inputs and other weights $\mathbf{B}$ in our implementations. Second, we use the same unquantized input images as previous works in order that our work is designed for the



(a) 2-bit (b) 3-bit

(c) 4-bit (d) Floating-point

Figure 2.29: Retraining with quantization levels of 2-bit, 3-bit, 4-bit and original floating-point (no retraining).

same input datasets as the previous works. Third, since quantization reduces precision of neural networks, quantizing unnecessary weights results in increasing error rates. As seen in Fig. 2.30, the results indicate the neural network with partially quantized weights has better recognition error rates than the one with fully quantized weights. Therefore, to obtain lower recognition error rates, we only quantized coefficient **A**.

In our training process, we use the following steps retraining neural networks in order to obtain the quantized weights:

1. Fully train neural networks with floating-point weights.

2. Choose number of iterations to minimize the output error.

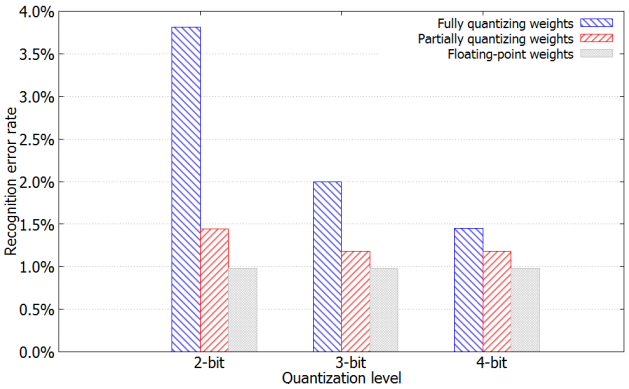


Figure 2.30: Error rate comparisons between fully quantizing weights, partially quantizing weights and floating-point weights.

3. Quantize weights used for multiplications in hidden layers.

4. Remove near-zero biases of hidden layers.

5. Perform back-propagation algorithm [56] to update weights.

6. Repeat Step 3 to Step 5 to minimize the output error and quantize partial weights.

After retraining the neural networks, we obtain the quantized neural networks and all weights are regarded as constant values for stochastic neural network classifiers. In addition, the recognition error rates can be found in Table **??**. Compared to the floating-point neural network, the quantized neural network has slightly higher recognition error rates.

2.4.4 Stochastic Neural Network Implementations

In this section, we introduce stochastic implementations for quantized neural networks. We use 2-bit quantization as an example and the architecture of the stochastic implementation are quite similar for other quantization levels.

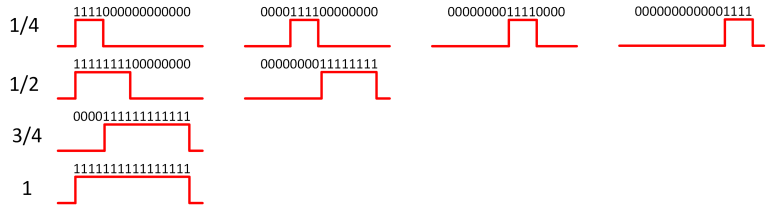


Figure 2.31: Sequences of 2-bit quantized weights.

Stochastic Quantized Bit-streams

In our stochastic implementation, multiplication for unipolar format bit-streams is achieved by ANDing two inputs. One of the inputs is quantized weight as a constant value. For 2-bit quantized weights with unipolar format, the quantized weights only have five values, 0, 0.25, 0.5, 0.75, and 1 in stochastic computing domain. Thus, the multiplication is changed to select 0%, 25%, 50%, 75% and 100% bits of the other input bit-stream and then pad un-selected bits with zeros. As a result, only selected bits contain useful information for future addition operations. Therefore, considering how to efficiently use the unselected bits, we propose a new type of bit-stream for the quantized weights.

The quantized weight bit-streams are similar to the time-based unary streams in previous works [21][20]. The difference is that our quantized weight bit-streams have contiguous '1's located in different phases. As seen in Fig. 2.31, it is an example for 2-bit quantized weights and the quantized bit-streams are generated from unary bit-streams by shifting different bits. By using the quantized weight bit-streams, the multiplication becomes to select a sequential part of a bit-stream as seen an example in Fig. 2.32. Finally, the output of quantized multiplication consists of a sequential part of one input and a sequence of zeros. The advantage of such encoding method is introduced in the following sections.

Stochastic Quantized Addition

As discussed above, the multiplication with 2-bit quantized weights truncates inputs with a number of sequential bits and pads them with zeros. Therefore, if the output bit-stream of the multiplication is padded with another product of two bit-streams

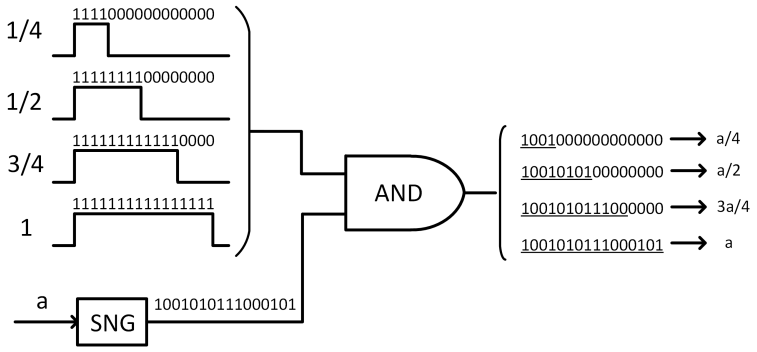


Figure 2.32: An example of multiplication with 2-bit quantized weights.

instead of padding zeros, the output bit-stream will become the sum of two products (suppose two product bit-streams have enough number of padded zeros). For example, suppose an operation is to compute $a/4 + b/4 + c/4 + d/4$. As seen in Fig. 2.33, four bit-streams (a, b, c and d) generated by four stochastic number generators (SNGs) are ANDed with four shifted unary bit-streams of 1/4 whose phases of '1's are interleaved. That is, those four bit-streams of 1/4 select different part of a, b, c and d. Consequently, after going through OR gates, the output becomes $(a + b + c + d)/4$. Therefore, the four products are summed up by using only seven simple gates in this example. We call the adder as **S**hifted **U**nary **C**ode **Adder** (SUC-Adder), whose inputs are products of shifted unary bit-streams and conventional stochastic bit-streams.

Moreover, for different combinations of quantized weights, the SUC-Adders are implemented by different circuits. Fig. 2.34 lists four types of SUC-Adders for 2-bit quantized weights. For stochastic quantized neural network implementation, since 2-bit quantized weights are constant values and we know the total number of each quantized value, the number and types of SUC-Adders are determined after training neural networks.

Compared to traditional unipolar adder (only OR gate), as seen in Table 2.14, our SUC-Adder has much better accuracy than the OR based adders. In addition, compared to previous stochastic neural network implementations [24][45][57], the SUC-Adder can reduce the number of inputs of parallel counters because it simply implements multi-input addition with several AND and OR gates. Therefore, the stochastic quantized neural networks with SUC-Adders will reduce hardware cost a lot compared to previous

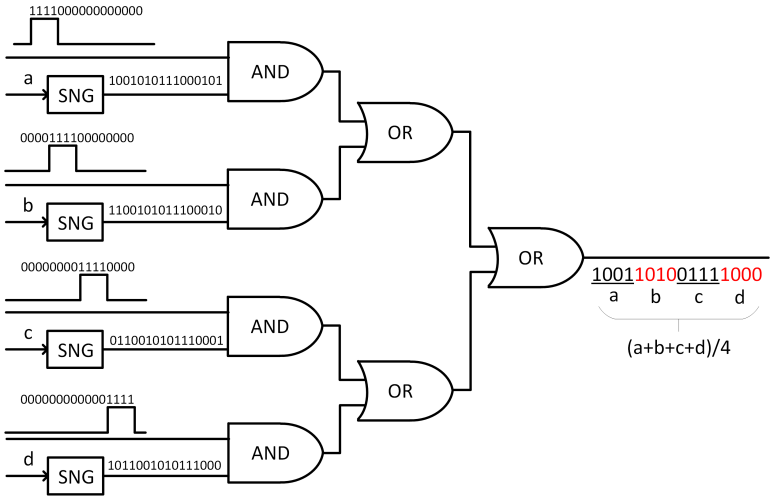


Figure 2.33: An example of a four-input SUC-Adder with 2-bit quantized weights.

works.

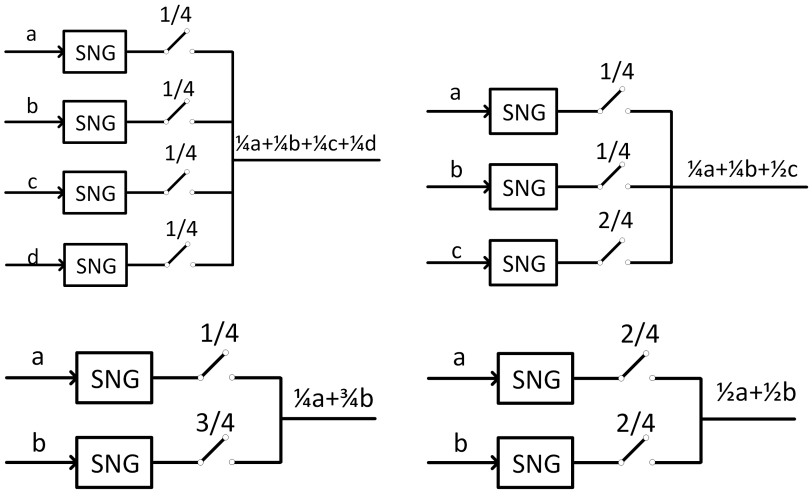


Figure 2.34: SUC-Adders with different combinations of weights. All quantized weight bit-streams have different phases of '1's in each circuit.

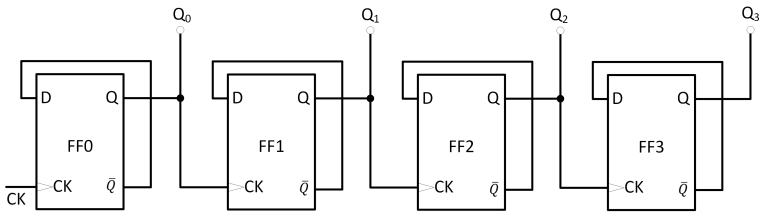


Figure 2.35: An example of quantized weight generator for bit-streams with a period of 16 bits. A binary value and $D_3D_2D_1D_0$ go through a comparator to generate a bit-stream of the binary value.

Table 2.14: Mean absolute error (MAE) comparison between quantized weight generator and LFSRs

Bit-length	16	32	64	128
Operation	$(a+b+c+d)/4$			
(i) Quantized weight generator	7.74%	5.17%	3.61%	2.50%
(ii) Same LFSR	9.30%	6.12%	4.00%	2.75%
(iii) Different LFSRs	7.82%	5.33%	3.64%	2.53%
(iv) OR with same LFSR	26.9%	26.8%	26.7%	26.7%
(v) OR with different LFSRs	11.8%	10.2%	9.40%	8.97%
Operation	$(a+b+c+d+e+f+g+h)/8$			
(i) Quantized weight generator	8.16%	5.60%	3.88%	2.69%
(ii) Same LFSR	11.88%	7.80%	5.03%	3.28%
(iii) Different LFSRs	8.50%	5.63%	3.94%	2.75%
(iv) OR with same LFSR	37.6%	37.7%	37.4%	37.5%
(v) OR with different LFSRs	12.5%	11.0%	10.3%	9.91%

Stochastic Quantized Weight Generator

In this section, we mainly focus on the stochastic quantized weights generator. As discussed previously, quantized weights have 2^n non-zero values (n is the quantization level) in stochastic computing. Since each non-zero value needs to be encoded to interleaved bit-streams for SUC-Adders, for one non-zero value $i/2^n$ ($1 \leq i \leq 2^n$), there are $\lfloor 2^n/i \rfloor$ types of shifted unary bit-streams. Therefore, it totally needs $\sum_{i=1}^{n} \lfloor 2^n/i \rfloor$

quantized weight generators used in the whole neural networks.

The design of the quantized weight generator is shown in Fig. 2.35, which is an example for producing bit-streams with a period of 16 bits. To generate bit-streams of different levels of quantized weights and different phases of '1's, we only need to initialize D Flip-flops by loading desired values.

To investigate accuracy of the shifted unary bit-streams, we use the operations of $(a+b+c+d)/4$ and $(a+b+c+d+e+f+g+h)/8$ with 3-bit quantized weights. Then, we compare our generator with linear-feedback shift registers (LFSR) by the mean absolute error (MAE). Moreover, we compare conventional OR adders with our design as well. In the implementation, all inputs are encoded by LFSRs and the coefficients (1/4 and 1/8) are produced by different methods. Five configurations are compared: (i) SUC-Adder: the same coefficient values are generated by the same quantized weight generator. (ii) Parallel adder: the same coefficient values are generated by the same LFSR. (iii) Parallel adder: the same coefficient values are generated by different LFSRs. (iv) OR adder: the same coefficient values are generated by the same LFSR. (v) OR adder: the same coefficient values are generated by different LFSRs.

As seen in Table 2.14, because of correlation the OR adder with same LFSR gets the worst MAE results and the OR adder with different LFSRs are worse than our design. In addition, our quantized weight generator has even better MAE results with different bit lengths than the parallel counter implementations. Therefore, it proves that bit-streams generated by our quantized weight generators can obtain similar or even better accurate results compared to the conventional LFSR implementations. The details of hardware comparison of generators are shown in Section 2.4.5.

Stochastic Neuron Implementation

A single neuron is the basic unit in neural networks, which consists of a matrix multiplication and an activation function. For the structure of a stochastic neuron shown in Fig. 2.36, the inputs of neurons come from image bit-streams generated by stochastic number generators or outputs of previous layers. Since image pixel inputs and outputs of previous layers (outputs of sigmoid function) are always in the range of $[0, 1]$, the signs of products of inputs and quantized weights are determined by the quantized weights. Therefore, the number of positive and negative products are clearly known when the

Table 2.15: Recognition error rates comparisons of RBM classifiers with different bit-lengths.

	Bit-length	2-bit	3-bit	4-bit
Conventional Binary	0.98%			
Quantized Binary		1.44%	1.18%	0.99%
Prior work [57]	16	4.29%	7.68%	15.59%
	32	2.86%	2.80%	3.36%
	64	2.52%	2.24%	2.08%
	128	1.92%	2.03%	1.80%
Our method [59]	16	2.73%	2.59%	2.39%
	32	2.71%	2.42%	2.05%
	64	2.56%	2.49%	1.85%
	128	2.37%	1.82%	1.79%

classifiers have been trained.

In our proposed neuron, inputs are separated into positive and negative groups based on their quantized weights and are ANDed with the quantized weights to compute their products. Then, we group some of products going to the same SUC-Adder. Since number of quantized weights are determined, we can minimize the number of SUC-Adders by adding as many products as possible in one SUC-Adder. For example, we can group two products with weights of 1/4 and 3/4, or three products with weights of 1/4, 1/4 and 2/4. After that, partial addition results and bias are assigned to parallel counter to sum bit-streams up. $bias^+$ and $bias^-$ in the positive and negative parts are extra constant inputs, which are obtained from training process as constant values here. Finally, the full adder will compute the subtraction of positive and negative part and go through a comparator to generate a bit-stream which is the input of next layer. In addition, the activation function is sigmoid function in restricted Boltzmann machine and because the Taylor expansion of the sigmoid function is $sigmoid(x) = 1/2 + x/4 - x^3/48...$, we use the approximate the sigmoid function [58] of $\frac{x+2}{4}$ and it is automatically achieved in the proposed neuron.

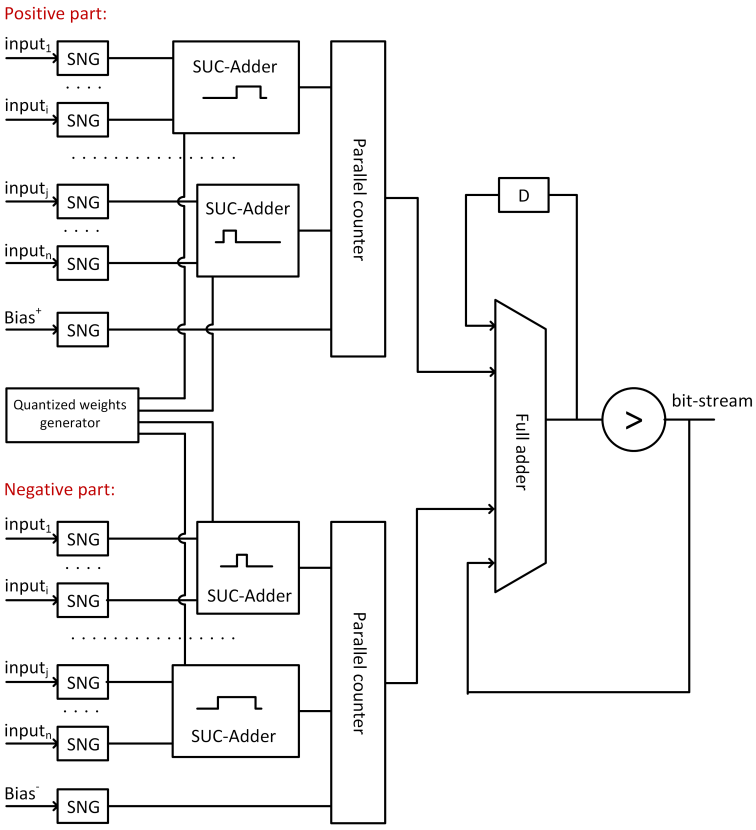


Figure 2.36: The stochastic structure of a neuron with matrix multiplication.

2.4.5 Experimental results of Neural Network Comparison

In this section, we use RBM classifier implemented by stochastic computing with a configuration of 784-500-1000-10. The MNIST handwritten digit image dataset [52] is used as input images, which consists of 70,000 data. Among the images, 60,000 images are training data and the rest 10,000 data are used for testing stochastic and conventional neural networks. All weights and coefficients of neural networks are first fully trained and then retrained.

Recognition Error Rates

First, we compared error rates of our stochastic implementation with previous stochastic methods and deterministic RBM implementation. In the stochastic implementation, we

compared our work with the stochastic method in [57]. The bit-stream lengths are varied from 16 bits to 128 bits.

As seen in Table 2.15, the quantized binary RBM implementation obtains recognition error rates quite similar to the conventional binary RBM. For the stochastic RBM comparisons, our method obtains slightly higher recognition error rates than the binary implementation. However, compared to the previous work, our implementation achieves much lower recognition error rates at shorter bit lengths.

Hardware Cost

In this section, we focus on hardware cost of different neural network implementations. We use the design compiler to synthesis neural networks with FreePDK 45nm library [41] for the binary and stochastic implementations. The binary implementation uses 8-bit fixed binary.

First, we compare hardware cost of the conventional LFSR and our proposed quantized weight generator. As seen in Table 2.16, our proposed generator has a little lower area and power compared to the conventional LFSR for producing different lengths of bit-streams. Thus, our generator does not introduce any extra overhead in the hardware implementations.

There are mainly two types of implementations in current stochastic neural networks. One type of stochastic implementations [45][24] is based on bipolar format. They normally encode input images and weights as bipolar format bit-streams. Then, to achieve a neuron including matrix multiplication and activation function, the bit-streams go into parallel counters and then go to finite state machines. Finally, outputs of a neuron keep bipolar format and are matched to next layers. Another type of stochastic implementations [57] is unipolar based. Its structure is quite similar to ours. The differences between our method and other unipolar based implementations are stochastic matrix multiplication and quantized weight generator.

In hardware implementation, we compare our design with two previous stochastic methods and binary implementations. As seen in Table 2.17, compared to previous stochastic implementations, our architecture obtains about 2.18x, 5.94x, and 9.58x reduction in terms of area, power and energy, respectively. This is because our design significantly decreases the sizes of parallel counters and finally obtains reduction of

Table 2.16: Hardware comparison between LFSR and quantized weight generator

Types of SNGs	Area (um^2)	Power (uW)
4-bit LFSR	81.7	15.2
5-bit LFSR	100.4	18.7
4-bit quantized weight generator	74.6	14.4
5-bit quantized weight generator	99	17.2

Table 2.17: Hardware comparison between stochastic RBM with 32-bit length and conventional RBM implementation with SNGs

Neural networks	Area (mm^2)	Power (mW)	*Energy* (nJ)
Binary	188.6	14668	73.6
Stochastic Unipolar [57]	5.01	558.8	34.89
Stochastic Bipolar [24][45]	6.86	870.0	107.7
Our work [59]	**2.72**	**123.2**	**7.44**

hardware cost. Moreover, compared to the binary implementation, our approach with 32-bit length derives about 69x, 119x and 10x less area, power and energy, respectively. Furthermore, even we increase the bit-length to 128, our design still has about 2.5x less energy compared to the binary implementation.

2.4.6 Conclusion

In this section, we propose a new stochastic architecture for quantized neural networks. The quantized neural networks are retrained by different quantization levels and obtain recognition error rates similar to floating point neural networks. Then, we propose a new adder called SUC-Adder, which is capable of summing several product bit streams without losing precision. The SUC-Adder can reduce the number of inputs of parallel counters and hence decrease hardware cost of the whole neural networks. In experimental results, in terms of area, power and energy, our approach with 32-bit length derives

2.2x, 6x and 10x reduction compared to previous stochastic implementations, and obtains 69x, 119x and 10x less hardware cost compared to the binary implementation while maintaining slightly higher recognition error rates than the binary implementation.

Chapter 3

System Performance Evaluation in Storage and Network Environment

In this chapter, we start to investigate the performance evaluation of the distributed edge computing environment.

3.1 Introduction

The performance gap between computer networks and storage systems has decreased substantially in recent years. Networks have gotten fast enough that the computational load the network places on the system rivals the load of the storage subsystem. This change makes it more difficult to evaluate and understand the performance of the overall system, ranging from large data centers with file servers and cloud servers to individually connected personal computers [60][61][62][63]. As a result, new performance evaluation tools are needed to understand the impact of the network and storage systems simultaneously. The most accurate way to evaluate or debug these systems is to execute a real application program that exercises both the network and storage concurrently. However, setting up an application can be complicated and time-consuming. Thus, the trace-driven replayer becomes a promising alternative to evaluate the performance of

target systems in the storage and network environment.

Trace-driven replayers are used for debugging or mimicking the behavior of real applications on difference environments including large scale-out systems. Previous trace-driven replayers [4][5][6][7][8] are only investigated for either the network or the storage system. In the past, people could ignore the impacts of either storage or network while studing the other since there was a big performance gap between them. However, the scenario has changed as fast storage devices (e.g., NAND flash) become prevalent. Some file system based replayers [9] focused on both network and storage impacts. They only replayed network traces to the target system and used the storage receiver (e.g., iSCSI target) to generate the corresponding storage I/O requests. This method is applicable to the case when all storage I/O requests are generated by the requests delivered by network. However, some storage requests may be independent from network traffic and they are generated by the storage system internally (e.g., log I/Os triggered by I/Os from clients) in the storage and network environment. Therefore, to generate the same workloads as the original applications and to accurately evaluate the performance of target network-storage systems, it is necessary to replay storage and network traffic separately while maintaining some appropriate connection.

Replaying both storage and network traffic requires a change in perspective. For a single replayer, people mostly focus on the replay accuracy, which indicates how well the replayers can issue requests based on timestamps in order to generate similar workloads as original applications during a specific time period. For multiple replayers in the network and storage environments, more devices are involved into the replaying process. Therefore, it is necessary and challenging for replayers to maintain relative issue times between storage and network requests so that the similar network and storage workloads are generated on the target system as the original application is running on it.

This section introduces a replayer called NetStorage, which aims to synchronously replay storage and network traces on target systems. The goal of NetStorage is to regenerate the same network and storage workloads as original applications in network-storage systems during specific time periods. In the NetStorage, there are three major components, network replayer, storage replayer and manager component. The network replayers and storage replayer are used for replaying network traces and storage traces, respectively. The manager component is responsible for synchronizing between the

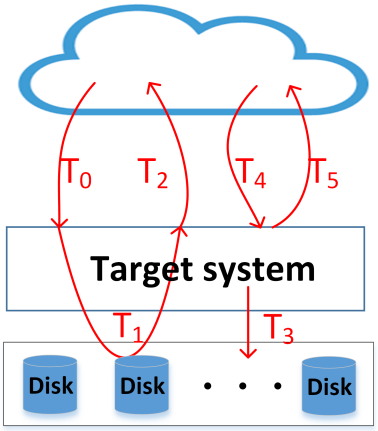
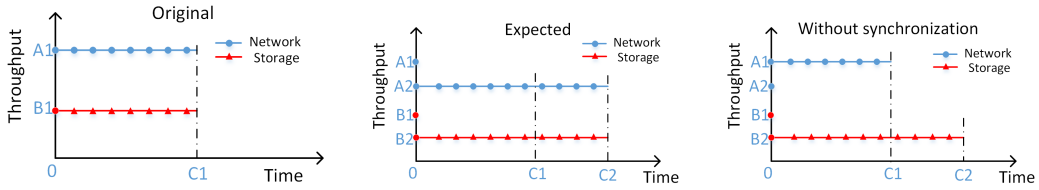


Figure 3.1: Example of a network-storage environment.



(a) (b) (c)

Figure 3.2: An example of scenarios of running the original application and replaying traces without synchronization.

network and storage replayers. This tool can also provide the performance evaluation of network-storage systems with different metrics such as CPU utilization, network throughput and storage throughput.

3.2 Motivation and Challenge

3.2.1 Motivation

With more data being generated by cloud and the performance gap between storage and network decreasing substantially, the need to benchmark network-storage systems has been becoming urgent. As discussed in Section 3.1, more research needs to be done on benchmarking systems in network and storage environment simultaneously.

To generate both network and storage traffic on the target system, one method is to replay network traces against the target system and make the system software such as the iSCSI target and TBBT [9] generate the corresponding storage I/O requests. Though this method is capable of replaying the network traffic and its corresponding storage I/O requests accurately, it is not sufficient and not convenient to test and evaluate all different scenarios in storage and network environment. For example, some network packets do not generate storage I/O requests. Some storage requests are generated from system log files, which are independent from the network packets. Thus, this method is not sufficient for those scenarios which contain independent network and storage traffic. When testing whether the network and storage components of the target systems have enough resources to run specific workloads separately, it is necessary to replay network and storage traffic separately. In addition, when testing how much network and storage throughput a system is able to sustain, replaying network traces to generate storage I/Os is really hard to accomplish due to not easily controlling storage traffic. Moreover, installing proper protocols such as the iSCSI target that properly parse and extract out storage I/O requests encapsulated in network packets, becomes really complicated, especially when there is no knowledge about network traces. That means that all possible protocols should be installed on the target system. Therefore, compared to the method mentioned above, replaying network and storage traces separately is a more efficient and flexible way to generate similar network and storage workloads on the network-storage systems as original applications.

Separately replaying network and storage traffic needs to involve multiple replayers, which is quite different from the scenario of only having one replayer. For a single replayer, the most important factor is the replay accuracy. If a replayer can exactly follow the timestamps of the requests as they appear in the traces, we say that the replayer has very high replay accuracy. However, the scenario is changed when replaying both storage and network traces due to involving several replayers. For example, as seen the processes of T_0, T_1 and T_2 in Fig. 3.1, replayers need to replay the network requests and their corresponding storage requests properly. In networked storage environments, clients send network requests to a target system. The target system receives requests and parses them to generate corresponding storage I/O requests. After that, the target system catches the contents from storage systems and sends them back to clients for

read requests or the target system writes the requests to the storage systems and sends back the success/failure message to clients for write requests. Moreover, for some other network-storage environments, even though some network packets and storage requests are independent, such as the processes of T_3, T_4 and T_5, those storage requests and network packets should be replayed with a time relationship because it is necessary to generate the same instantaneous network and storage traffic as original applications. Therefore, a network and storage replayer basically needs to replay the processes of T_0, T_1, T_2, T_3, T_4 and T_5 as shown in Fig. 3.1. As a result, the network and storage replayer needs to consider more factors like synchronization between replayers than a single replayer, and the management for those replayers will become much more complicated. In the following, we introduce major issues and challenges of the network-storage system replaying and provide our solutions for those issues.

3.2.2 Replay Accuracy

Replay accuracy is the most important feature for each individual replayer, indicating how well replayers issue requests following timestamps. Our network-storage replayer contains network replayers and a storage replayer. The storage replayer is based on the hfplayer [64]. Haghdoost *et al.* [64] solves the replay accuracy issue by exploring four different scenarios, I/O stack queue sizing, system call isolation, in-flight I/O control and I/O request bundling. Thus, the storage replayer promises the high replay accuracy of storage replaying in network and storage environments. For the network replayers, as discussed in Section 3.8, all previous works do not satisfy our requirements. The network replayers not only need to send and receive network requests, but also need to replay traces based on timestamps. On one hand, with high replay accuracy, it cannot have large overhead like previous works. On the other hand, with real bi-directional network traffic on the target system, it must keep the functionality to send and receive network requests like T_0 and T_2 processes in Fig. 3.1. The replay accuracy comparison with other replayers is investigated in Section 3.5.

3.2.3 Synchronized Replaying

Synchronized replaying is another important function in network-storage replayers. The reason is that to generate the same instantaneous throughput on the target system as running original applications, the replaying process should keep the time relationship between network and storage as the original running process. Therefore, replayer needs to issue storage and network requests at the same pace. For example, as seen in Fig. 3.2, assume that a workload is captured on system A with a 10 Gbps Ethernet port and 700MB/s SSD and the performance of the ssystem A with running the workload is shown in Fig. 3.2a. If we want to replay the same workload on the system B which has a 10 Gbps Ethernet port and 100MB/s HDD, the expected performance of replaying on the system B may be like the results in Fig. 3.2b because the lower storage bandwidth on the system B might slow down the whole application process. However, if the replayers do not have synchronization function, two replayers will independently replay their requests and the results will be like in Fig. 3.2c since the network workload is replayed on the port which has the same bandwidth as the original system while the storage workload is replayed on the much slower storage devices which cause the delayed replaying. Obviously, the results in Fig. 3.2c cannot represent the behavior in Fig. 3.2b and lead to wrong conclusion. In real cases, we rarely know the configurations of trace-captured systems. So, it is difficult to distinguish whether the target system has the higher or lower bandwidth of network and storage than the source system. Moreover, due to the unexpected workloads and system resource usage, it is highly possible that the replaying mismatch happens in the middle of network and storage replaying without synchronization. For example, a more general scenario might be that the network replaying keeps the same pace but the storage replaying becomes slow because of the high storage requests ratio in a specific duration. Therefore, to obtain the correct replaying process and conclusion, the replayer needs to have a synchronization function which can make replayers replaying at the same pace. The manager component periodically monitors all replayers and keeps them replaying with the same pace as discussed in Section 3.3.4.

3.2.4 Start Time of Replaying

During network and storage replaying, the ideal case is that all replayers start replaying simultaneously because the replaying process should generate the same instantaneous network and storage workloads as original applications. If a delay happens on either network or storage replayer at the start of replaying, there will be a mismatch over the whole replaying process. For example, assume both network and storage bursts occur at time A with running an original application. However, a delay x happens when starting the network replayer. Then, the storage and network bursts are replayed at time A and $A + x$, separately. Obviously the replaying with mismatched bursts cannot mimic the real behavior of the original application. Therefore, it is important to start replaying of all replayers at the same time. How that is accomplished is discussed in Section 3.3.4.

3.2.5 Trace Capture

The most straight-forward method (method #1) to capture network and storage traces is to use network-storage systems running with real applications. On the clients' side, run applications to send requests to the target system, and on the target system side use tcpdump and blktrace to capture both network and storage traces. By doing so, traces of network and storage contain both correlated and uncorrelated requests. So those directly captured storage and network traces have the same characteristic as an application running on the real systems.

In some cases, the method of direct trace capture is not practical. In this work, we use available server storage traces to generate their corresponding network traces (method #2). Those available server storage I/O traces are real-world traces from Microsoft. Also, it is possible to generate synthesized traces of storage and network like sequential or random workloads (method #3).

Method #1 is the most accurate way to evaluate target systems but usually it is not practical. Method #3 provides the most flexibility to adjust the trace characteristic while it cannot really reflect real applications. Method #2 provides some flexibility and can also reflect the characteristic of real applications. In this paper, we use method #2 to capture correlated storage and network traces discussed in Section 3.4. Compared to all previous replayers, the NetStorage tool is capable of replaying both storage and

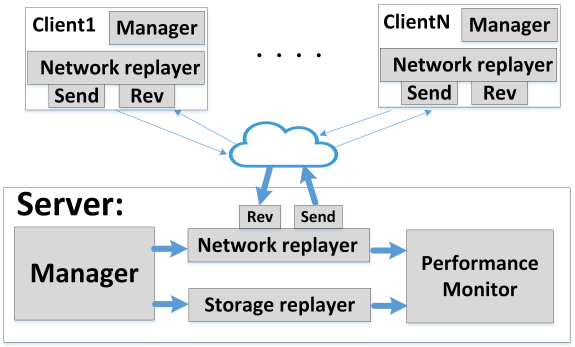


Figure 3.3: NetStorage replayer is used in the network-storage system system.

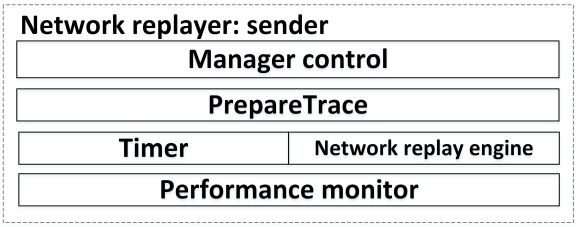


Figure 3.4: Sender component in the network replayer.

network traces captured by above three methods in order to achieve more accurate or more flexible system evaluation.

3.3 Architecture of NetStorage Replayer

The NetStorage is implemented with three components: network replayer, storage replayer and management component as seen in Fig. 3.3. The details of each function are introduced in following subsections.

3.3.1 Terms and Notations

We first summarize the terms and notations that are used frequently throughout this paper.

- **Start time** is recorded at the beginning of replaying and is used to compute issue time.

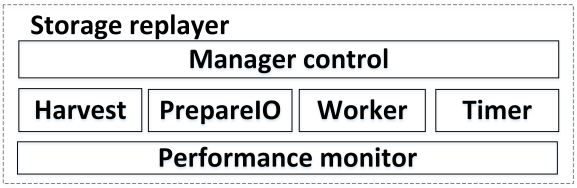


Figure 3.5: Storage replayer consists of four major functions: Timer, Worker, Harvest and PrepareIO.

- **Timestamp** is the time that a request is supposed to be issued. It is recorded when a request is captured with running real applications.
- **Issue time** is the difference between current time and start time. If issue time is larger than the timestamp of a request, replayers will issue the request. It has $issue\_time = current\_time - start\_time$, where **current time** is the current CPU time read by a timer of a replayer.
- **Issue error** indicates how well a replayer replays requests based on their timestamps. It is calculated from absolute difference between the timestamp and issue time. Thus, $issue\_error = \sum |issue\_time - timestamp|$.
- **Timer slowdown ratio** (≥ 1) is used to mimic the behaviors of having a slower network or storage device. With slowing down a timer, the issue time is changed to $issue\_time/slowdown\_ratio$. For example, suppose a request's timestamp is A. When $issue\_time \geq A$, the replayer issues the request. When $issue\_time$ is changed to $issue\_time/slowdown\_ratio$, the replayer needs to wait longer time to issue the request. Thus, the timer is slowed down by $slowdown\_ratio$.

3.3.2 Network Replayer

The network replayer is used to replay and receive network requests. It can be split into two parts, receiver and sender. The receiver is a simple tool to build the connection and to receive requests sent from a sender. The sender consists of five major functions, manager control, PrepareTrace, timer, network replay engine and performance monitor as seen in Fig. 3.4.

In the sender, there are two functions. One is the timer function which periodically monitors the CPU timer register. The timer is the same as the storage replayer timer

because authors found different timers even used in the same system may cause inaccurate issue time. For example, in our experiment, the issue time computed by the timer using inline assembly codes to read CPU register is much different with the issue time computed by the timer using clock() functions in the same system. As a result, using different timers will cause asynchronous replaying. Another function is the network replay engine function which is used to issue packets to the network.

Another function in the sender is the manager control. Before network replaying, the manager control function waits for the signal sent from the manager. It helps all components start mostly at the same time. Additionally, the PrepareTrace function is developed for reading packets from a network trace file. When a packet is read into the program, PrepareTrace creates dynamic memory for the packet. At the same time, a class called *Packet* is built to store packet information, such as the packet size, IP source address, destination IP address and timestamp. Next, the request is transferred to the network replay engine. Once PrepareTrace function collects the packet information from the trace file, the replay engine re-wraps the packet and starts to issue the packet based on packet length, destination IP address and packet's timestamp. After issuing the packet, the replay engine releases memory of the packet. Finally, after replaying all packets, the performance monitor function reports the sending throughput including KB/s and packets/s.

3.3.3 Storage Replayer

The storage replayer is designed to replay existing storage block I/O traces. As seen in Fig. 3.5, the storage replayer consists of four major functions, PrepareIO, worker,harvest and timer function. The timer function is used to record the CPU time periodically. The worker function is designed for issuing the requests based on the timestamps from traces. By comparing the timestamps and the time captured by the timer function, the worker function decides whether to issue requests or to wait. For example, if the issue time computed by the timer function is equal to or larger than the timestamps of requests, the worker function will issue them. Otherwise, the worker function waits for issuing requests until the issue time is equal to or larger than the timestamps. The worker function also records the issue errors of requests and computes the average issue errors after replaying. The average issue error indicates how accurately replayers issue

requests. More details are discussed in following sections. PrepareIO is used to load requests into the storage replayer. The harvest function is used to collect the finished requests and release their dynamic memory. The manager control is on the top of four functions and is responsible to communicate with the manager component and trigger or synchronize storage replaying. Additionally, when the number of disks on source systems is larger than the number of disks in target system, extra disks' requests are distributed to the disks in a round-robin order. For example, if the system has disks from disk#0 to disk#6, the request from disk#8 will be submitted to disk#1 because $8\ modulo\ 7 = 1$.

3.3.4 Manager

The manager component is the control engine. Two major tasks are addressed by the component discussed in the following paragraphs.

For the first task, the manager is responsible for triggering network and storage replayers together on the server side. In addition, it needs to control the starting time of network replayers between the server and clients. Since both storage and network replaying are based on timestamps, it is very important that the replayers start to replay at the same time as discussed in Section 3.2. As seen in Fig. 3.6, at the beginning replayers start individually to execute their preparation codes like building connection between the sender and receivers and reading traces. Then, all replayers wait for triggering signals to issue their requests. At the same time, managers build their connections between the server and clients. Then, the managers simultaneously trigger components on each side and the network replayers and storage replayer start to replay their traces. Meanwhile, the replayers record their *start_times* simultaneously in order that all replayers have synchronous issue times. According to different network environments and their latencies, people may want to add a bias to the *start_times* on client side. In our experiments, the client and target systems are local machines and thus assume the network latency can be ignored.

For the second task, the manager needs to periodically synchronize replayers because the storage or network replayers may not issue requests following their timestamps due to hardware limitations and unexpected workloads and system usage. Since timestamps

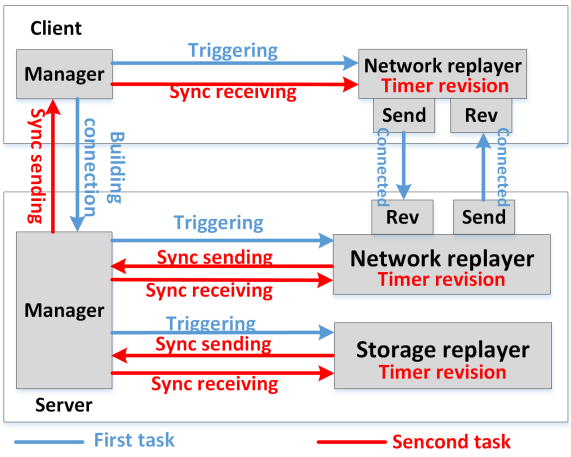
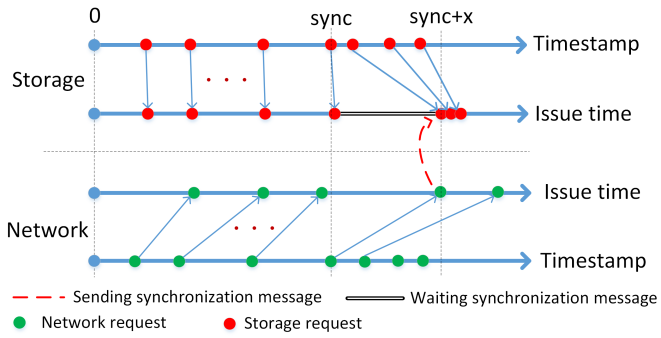


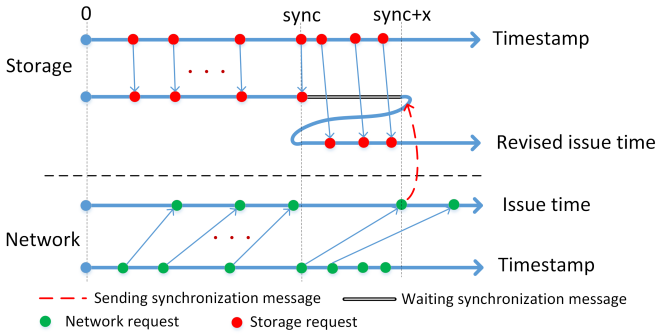
Figure 3.6: The manager component controls storage and network replayers on both the server and client sides.

of storage and network traces contain their time relationship as running original applications, the manager uses their timestamps to synchronize the replayers. To periodically synchronize replayers, a parameter called *sync* is set to indicate how frequently the manager synchronizes replayers. For instance, for each *sync* seconds, at timestamp $sync+1$, the network replayer sends synchronization messages via pipe to the storage replayer, and the storage replayer waits for receiving synchronization message without issuing any requests. At timestamp $sync + sync/2$, the storage replayer sends synchronization messages and the network replayer waits for receiving synchronization messages. The reason for sending and receiving synchronization messages from both replayers is that either storage or network might be the slower one. Additionally, the network replayer also needs to send synchronization messages to network replayers at clients via the Internet. Because this tool only considers the performance of the server rather than clients, we assume replayers at clients can replay network traces following their timestamps. Thus, it is not necessary for the network replayers at clients to send synchronization messages to the server.

In addition, since the manager synchronizes replayers based on timestamps of traces, the timers of replayers need to be synchronized periodically as well. For example, as seen in Fig. 3.7, assume network replaying cannot catch up with its request rate while storage



(a) Timer without synchronization



(b) Timer with synchronization

Figure 3.7: An example of the timer synchronization.

replaying can follow its. Therefore, it might happen that the storage replayer issues a request with the timestamp of *sync* and then waits for a synchronization message. At the issue time of *sync+x*, the network replayer issues the request with the timestamp of *sync* and then sends a synchronization message to the storage replayer. After receiving the synchronization message, the storage replayer recovers to issue requests. If there is no timer synchronization as seen in Fig. 3.7a, the storage replayer will issue requests with very high rate starting from the issue time *sync+x* because current issue time is much larger than timestamps of following requests and it will replay requests with full speed for a while until the timestamp of one request is larger than the issue time. Additionally, the timestamp difference of current replayed requests between network and storage will keep increasing with time passing. However, with timer synchronization as seen in Fig. 3.7b, we revise the storage issue time back to *sync*. Consequently, replaying

with the revised issue time not only is able to avoid the unexpected high replaying rate, but also decreases the timing difference between the storage and network replayers. The timer revision always happens right after receiving synchronization message for network and storage replayers.

In summary, starting replaying simultaneously promises that the timestamp based replaying process enables the correlation between the storage and network traces. The synchronization of replayers and their timers makes replayers replay at the same pace. In the following sections, the synchronization influence on the replay accuracy is investigated.

3.3.5 Trace Format and Performance Metrics

Two types of traces are used with this tool, network trace and its corresponding storage trace. First, the network replayer uses *PCAP* format [65], which is a very popular format used in network traces. The *PCAP* format contains packet information including timestamp, packet size, destination IP, source IP, etc. Second, the storage I/O replayer has five fields in traces, timestamp, I/O type, request size, offset and disk name. By converting existing storage traces into its storage replayer input format, it is quite easy to replay them by the storage replayer.

There are six metrics used to evaluate the network-storage system performance, CPU utilization, execution time, network throughput, network issue error, storage throughput and storage issue error. CPU utilization encompasses all of the workloads executed on the network-storage system and it indicates how much CPU capacity remains to run other applications. The execution time indicates the time to finish the whole replaying process. Network and storage throughputs indicate the network and storage performance of network-storage system separately. Normally, both throughputs should be the same as the original traces. The network and storage issue errors demonstrate the capacity of network and storage to tolerate request rates.

3.4 System Environments and Trace Capture

In this section, we describe system environments, trace configurations and how the network and storage traces are captured.

Table 3.1: System configurations

System	CPU	Other configurations
Sys1	Intel 3.6G	400GB disk, 5.8GB memory
Sys2	Intel 2.9G	150GB disk, 2GB memory
Sys3	Intel 3.724G	16 cores
Sys4	Power8E 2G-3.6G	20 cores, 160 logical CPUs, 2.9 TB NVMe
Sys5	Intel 2.4G	1TB disk, 24 processors
Sys6	Intel 2.4G	1TB disk, 24 processors

3.4.1 System Environments

The system configurations are shown in Table 3.1. ***Sys1*** and ***Sys2*** are connected through a 1GbE private network. The connection between ***Sys3*** and ***Sys4*** is through a 10GbE private network. ***Sys5*** and ***Sys6*** have the same system configuration and they are connected by a 1GbE private network. They have the Intel Xeon(R) E5-2620@2.4GHz CPU with 1TB HDD. ***Sys5*** and ***Sys6*** have 24 processors.

3.4.2 Trace Description

Eight sets of traces are given in Table 3.2 from MSR Cambridge Traces [66]. There are two types of traces used with the NetStorage tool, network trace and storage trace. The storage traces are truncated with one hour period for each one. Simple C code was used to change the format of MSR Cambridge trace to the NetStorage format. The network traces were captured based on the storage trace. How they were captured is described in Section 3.4.3. The method correlates most of captured network and storage requests. It is also possible to run some applications like surfing the Internet and watching videos for generating more independent storage and network requests in the experiments.

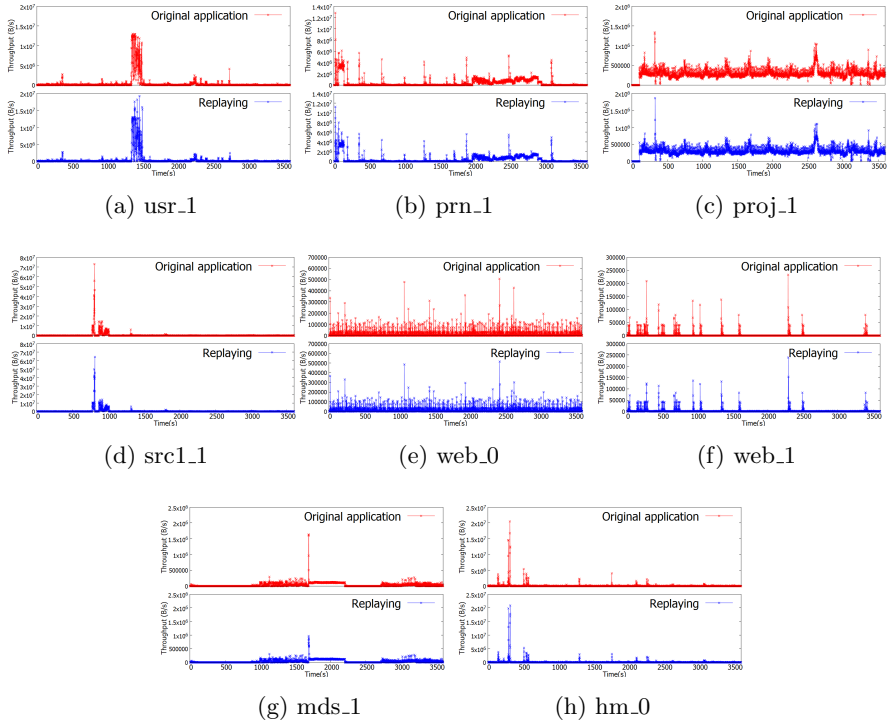
3.4.3 Capturing Environment

To transmit storage I/Os on network, the storage must be on the remote side and clients access storage through network. Although there are many storage network protocols e.g., Fibre Channel, InfiniBand, iSCSI, etc., we utilize iSCSI protocol to build a storage

Table 3.2: Trace configurations

Traces	Function	Type	# of requests
usr_1	User home directories	Network	736002
		Storage	54864
proj_1	Project directories	Network	736003
		Storage	38725
prn_1	Print server	Network	813109
		Storage	71206
hm_0	Hardware monitoring	Network	248051
		Storage	37397
src1_1	Source control	Network	744456
		Storage	73026
web_0	Web/SQL server	Network	45846
		Storage	7296
web_1	Web/SQL server	Network	7691
		Storage	552
mds_1	Media server	Network	130132
		Storage	27134

network considering the convenience of TCP/IP network. We installed iSCSI initiators on client machines so that I/Os can be transmitted via the network. We use one machine as a storage server to provide storage to clients. On this machine, we installed an iSCSI target so that incoming SCSI commands will be extracted and sent to the block device for processing. We sniffed at the NIC where the iSCSI target listened on, to capture network traces. When capturing traces, we ran workloads from MSR Cambridge Traces on client and then network packets and I/O requests were automatically generated by our system. Since the I/O trace format of the storage replayer follows the hfplayer [64], we converted the format of MSR Cambridge Traces to the format described in Section 3.3.5.



(a) usr_1 (b) prn_1 (c) proj_1

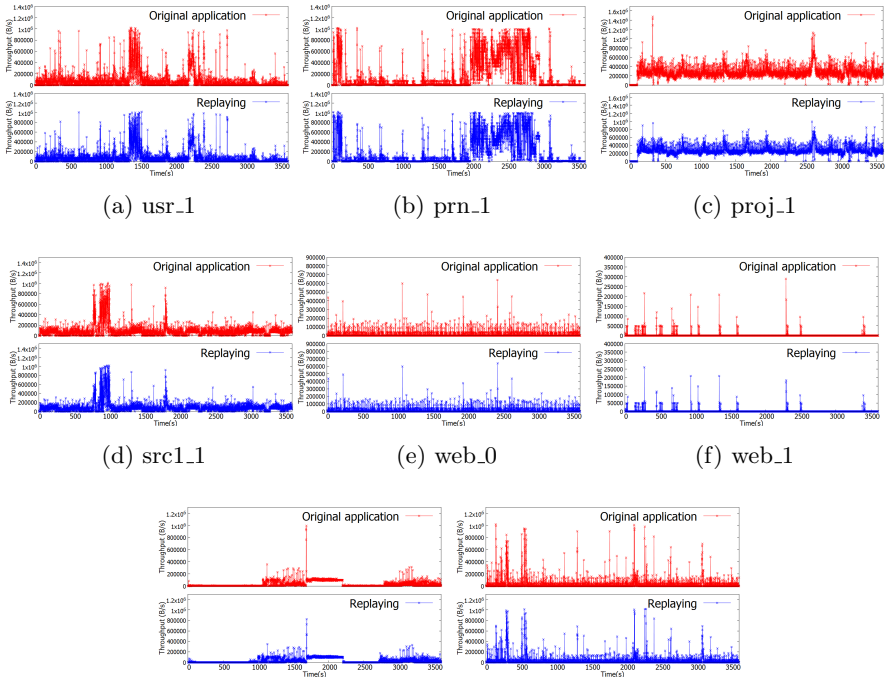
(d) src1_1 (e) web_0 (f) web_1

(g) mds_1 (h) hm_0

Figure 3.8: Instantaneous network throughput comparison between replaying and original application.

3.4.4 Comparison between replaying and capturing

To validate our NetStorage replayer, we compared the network and storage instantaneous throughputs per second for the capture process and the replaying process. In the capture process, the client issued I/O traces via iSCSI to the server over TCP/IP and we simultaneously recorded both network and storage throughputs per second on the server side. In the replaying process, the captured traces were replayed to the storage and network by the NetStorage replayer and we simultaneously recorded the network and storage throughputs on the server side as well. One thing needs to be mentioned is one of the network traces is recorded with the times when requests arrive to the server (the server receives the requests) as seen the processes of T_0 and T_4 in Fig. 3.1. Therefore, to simplify the experiments, we assume the request issue times from the client's side are same as the times when the server receives those requests since the client and



(a) usr_1 (b) prn_1 (c) proj_1

(d) src1_1 (e) web_0 (f) web_1

(g) mds_1 (h) hm_0

Figure 3.9: Instantaneous storage I/O throughput comparison between replaying and original application.

server are local machines and the network latency is really small and can be ignored in the experiments. In future works, the NetStorage will consider the network latency as a function of the synchronization for the cloud environments.

For validation experiments, ***Sys5*** was used as a client and ***Sys6*** was used as a server. As seen in Fig. 3.8 and Fig. 3.9, both instantaneous network and storage throughputs are plotted. To quantitatively evaluate the correlation between those plots, we use the Pearson correlation coefficient [67] which indicates the linear correlation relationship between two variables. It has the value between -1 and 1. 0 is no correlation and 1 is perfect correlation between two variables. As seen in Table 3.3, the Pearson correlation coefficients for eights traces of network and storage are more than 0.9. That means the capture and replayed processes have almost the same trend of instantaneous throughputs. The instantaneous throughputs in Fig. 3.8 and Fig. 3.9 with the Pearson correlation results prove that the NetStorage replayer can re-generate similar network

Table 3.3: Pearson correlation coefficients between the capture and replayed processes for storage and network.

	Network	Storage
usr_1	0.9769	0.9814
proj_1	0.9658	0.9095
prn_1	0.9612	0.9828
src1_1	0.9663	0.9310
web_0	0.9322	0.9464
web_1	0.9550	0.9561
mds_1	0.9539	0.9237
hm_0	0.9937	0.9753

and storage workloads to the server as the original application running on the sever.

3.5 High Fidelity Replaying

In this section, we investigate the timing fidelity of network and storage replayers compared to previous works separately in order to validate the NetStorage replayer ability to replay network packets and storage I/O request accurately.

3.5.1 Network Replaying

To compare NetStorage with TCPreplay [68], eight network traces in Table 3.2 are replayed on ***Sys6*** by seven times for both replayers. As seen in Fig. 3.10, the NetStorage replayer has much lower issue error than TCPreplay when replaying all eight traces because our replayer has more accurate timer and lower overhead during replaying. On average the NetStorage replayer has 74x less average issue errors compared to TCPreplay. Therefore, our network replayer has better replay accuracy than TCPreplay among all eight traces.

Moreover, TCPreplay cannot generate real network traffic to servers. This means that TCPreplay cannot simulate the behavior of bi-direction network traffic. In the NetStorage design, the tool replays bi-direction traffic which is sent and received by the

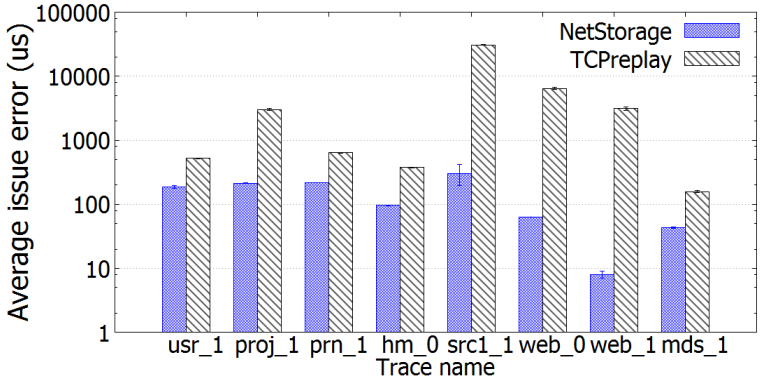


Figure 3.10: Average issue error comparison between NetStorage network replayer and TCPreplay with 95% confidence intervals. (Y-axis is log scale.)

target system. Thus, the replay process of the NetStorage is quite close to the scenarios of running real applications.

3.5.2 Storage Replaying

For the storage replayers, we compared our storage replayer with Haghdoost *et al.* [64]. Eight traces are replayed by NetStorage replayer and hfplayer on ***Sys6*** for seven times. As seen in Fig. 3.11, they have similar average issue errors and 95% confidence intervals for all eight traces.

In addition, another difference between the hfplayer and the NetStorage storage replayer is the number of requests which replayers can replay. The hfplayer [64] loads all requests into memory first and replays them. Hence, the memory size limits the number of requests that the hfplayer can replay. Based on our experiments, hfplayer can only replay around 700,000 requests with 4GB memory size. However, the NetStorage read requests alongside with issuing requests by the PrepareIO function and also has the harvest function which can release the memory of accomplished requests. Thus, the NetStorage achieves unlimited replaying. In summary, our storage replayer with adding new control functions and unlimited replaying feature has high timing fidelity similar to the hfplayer.

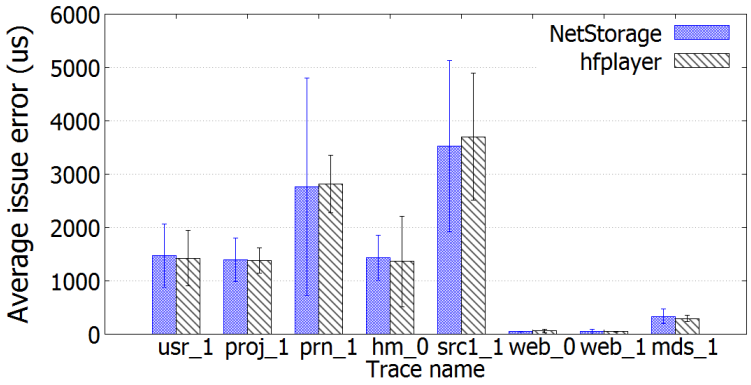


Figure 3.11: Average issue error comparison between NetStorage storage replayer and hfplayer with 95% confidence intervals.

3.6 Synchronized Replaying

To mimic real networked storage system behaviors, NetStorage not only needs to replay all traces from network and storage, but also needs to keep all replayers at the same pace. In other words, synchronized replaying can regenerate the same instantaneous workloads of network and storage as the original application during specific time periods. In following experiments, ***Sys1*** was used as the server and ***Sys2*** was used as a client. Additionally, to simplify complex relationship between replayers and make it easier to understand synchronized replaying, we analyzed different scenarios only for the server side. The conclusion will be the same for replayers between clients and the server.

Two scenarios are investigated under synchronized replaying. **The first scenario** assumes those replayers are able to replay at the same pace and the variance may happen in the middle of replaying. One common case of the first scenario is that the target systems are fast enough to replay both storage and network traces with high replay accuracy. It is also possible that the target system has slow storage and network devices and replayers cannot replay traces based on the timestamps, while the replayers can replay traces with the same pace. However, the second case rarely happens. Thus, we only consider the first common case in following experiments. **The second scenario** is that storage and network replayers replay traces with different paces. Either storage or network, or both devices might be slow. As a result, one replayer will run faster than

the others if no synchronization happens.

To indicate how well two replayers synchronously replay traces, a metric called synchronization delta average (Δ) is proposed. The definition is shown in Eq. 3.1.

$$\Delta = \sum_{t=1}^{N} |T_t^{sto} - T_t^{net}|/N \tag{3.1}$$

where, T_t^{sto} is the actual issue time of the storage request whose timestamp is t. T_t^{net} is the actual issue time of the network request whose timestamp is t. N is the total number of samples. To make sure there is a 1:1 correspondence between storage and network requests, the requests with T_t^{sto} and T_t^{net} are dummy requests. Those dummy requests are only responsible for recording the corresponding times and will not be issued. Thus it will not affect the performance. In our experiments, the samples are captured every one second.

For the first scenario, because both network and storage replayers can replay their requests with high replay accuracy on target systems, it has $T_t^{sto} \approx T_t^{net} \approx t$. Therefore, we simply use the average issue error to indicate the synchronization between two replayers and also to investigate synchronization impacts on replay accuracy for the first scenario.

For the second scenario, the replaying process may obtain $T_t^{sto} > T_t^{net} \approx t$ or $T_t^{net} > T_t^{sto} \approx t$ because one of the replayers cannot catch up with request rates. However, the real applications running on slower systems should have $T_t^{sto} \approx T_t^{net} > t$ because the network and storage traces are dependent to each other. Thus, the issue error cannot directly reflect the synchronization between two replayers any more and we use synchronization delta average (Δ) to reflect the synchronization between two replayers for the second scenario. The smaller Δ means the better synchronization between two replayers.

3.6.1 Replaying with Same Pace

For the first scenario, we investigated the average issue errors for both network and storage replayers with varying synchronization period *sync*. It helps to understand how much influence synchronized replaying has on replay accuracy. Eight traces are tested for this scenario.

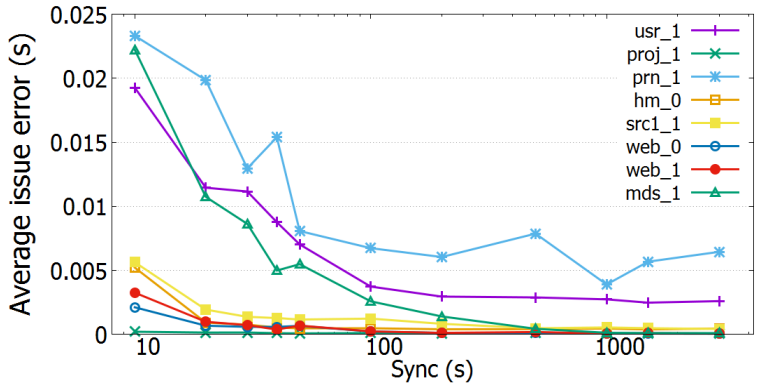


Figure 3.12: The relationship between average issue error and *sync* for the network replaying (the first scenario).

As seen in Fig. 3.12, the average issue errors of network replaying are decreased with increasing the value *sync*. Especially, sharp drops occur at the smaller synchronization periods. This is because the smaller synchronization periods introduce more overhead and slow down the network replayer. In contrast, as seen in Fig. 3.13, the storage replayer has the stable average issue errors for different *sync* values. This is because the storage traces have much lower throughput rates than the network traces shown in Table 3.2. Replaying with lower throughput rates is not affected much by the synchronization. Therefore, the network replayer suffers much more overhead than the storage replayer under synchronized replaying for those eight traces.

3.6.2 Replaying with Different Paces

For the second scenario, the synchronization function will force the faster replayer to slow down and try to make those replayers run at the same pace. We slow down the timer of network replayer of the server to mimic the behavior that the network replayer issues its requests slower than the requests' timestamps. We set $slowdown\_ratio = 2$ for network replayer and $slowdown\_ratio = 1$ for storage replayer. Therefore, the storage replayer waits for the network replayer during synchronized replaying.

Fig. 3.14 indicates the relationship between the synchronization delta average (Δ) and the synchronization period (*sync*) for the second scenario. It clearly indicates that Δ becomes much smaller with the smaller synchronization periods. In addition, compared

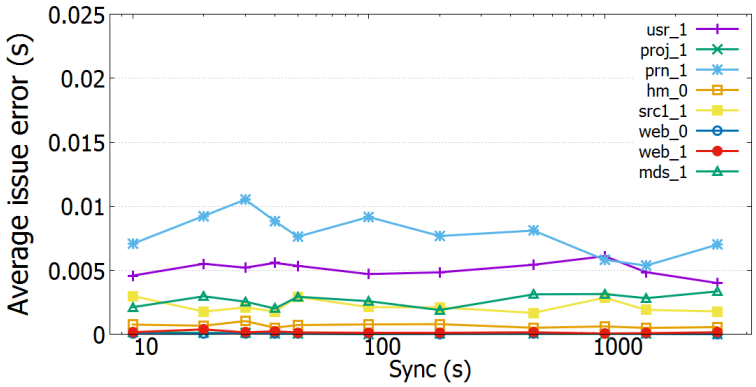


Figure 3.13: The relationship between average issue error and *sync* for the storage replaying (the first scenario).

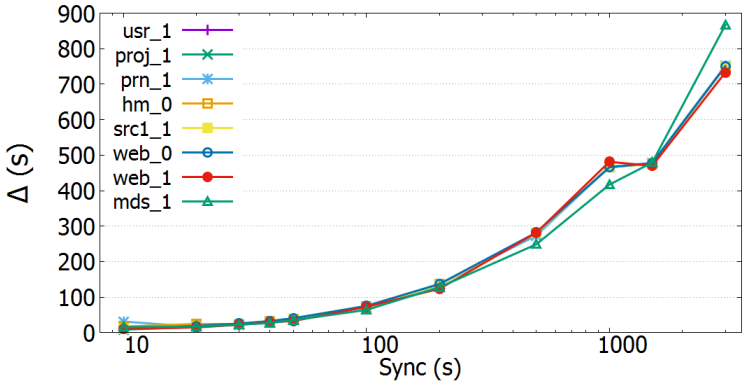


Figure 3.14: The relationship between Δ and *sync* for the network replaying with slower network (the second scenario).

to replaying without synchronization (the most right points when *syn* is larger than trace length), replaying with $sync = 10$ can reduce Δ about 70x. That means the synchronization management is successful at making two replayers run synchronously and higher synchronization frequencies are capable of improving that correlation.

Additionally, the faster replayer needs to wait for the slower replayer during synchronized replaying. We investigate the relationship between the synchronization period (*sync*) and the average issue error of the storage replayer. As seen in Fig. 3.15, with $sync \leq 3600$, the issue errors of the storage replayer are increased with increasing *sync*

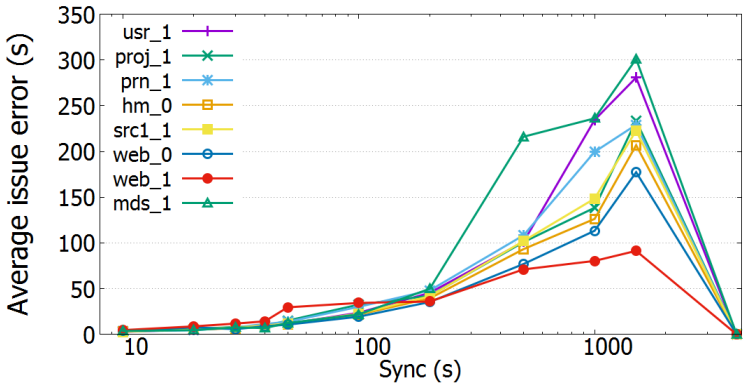


Figure 3.15: The relationship between average issue error and *sync* for the storage replaying with slower network(the second scenario).

values. This is because the lower *sync* values mean the storage replayer waits shorter time to recover issuing its requests. Moreover, more frequent timer revisions also help to decrease the average issue errors.

However, there is an exception for the average issue error when *sync* values are larger than the trace length (3600s). As seen in Fig. 3.15, the most right storage issue errors are dropped to the values around zero. This is because when *sync* is 4000 (larger than trace length), there is no synchronization happened between replayers during the whole replaying process. Therefore, the storage replayer independently issues requests only based on their timestamps.

3.6.3 Timer Slowdown Ratio

In this subsection, we investigate the scenario with varying the network timer slowdown ratio from 1x to 3x. The replayed trace is usr_1.

As seen in Fig. 3.16, for each line with the same *sync* value, Δ is increased with increasing the slowdown ratio. This is because the larger slowdown ratios increase timing difference between the storage and network replayers. Comparing the lines with different *sync* values, Δ is increased with larger *sync* values because replaying with smaller *sync* has more frequent synchronization between the storage and network replayers and it decreases the timing difference between those replayers. Similar conclusions are also drawn in Fig. 3.17. The storage issue error is increased with increasing the

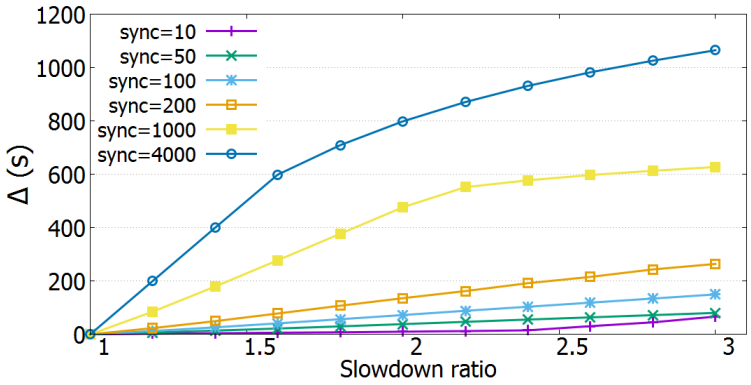


Figure 3.16: The relationship between Δ and slowdown ratio with replaying usr_1.

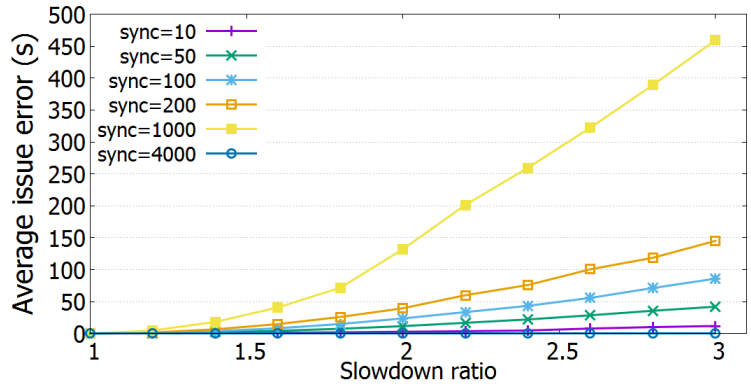


Figure 3.17: The relationship between storage issue error and slowdown ratio with replaying usr_1.

slowdown ratio or increasing the *sync* values. Additionally, similar to the discussion in Section 3.6.2, for $sync = 4000$, the issue errors of the storage replay are close to zero.

In summary, the NetStorage tool is able to synchronously replay storage and network traces even facing slow devices. The synchronization function actually reduces timing difference between those replayers and keeps them having the same instantaneous throughputs as running applications on slower systems. The synchronized replaying also slows down the faster replayer and increases its average issue error. Thus, the synchronization makes the replayers loose their replay accuracy when there is slow hardware in systems. For the normal scenarios like the first scenario, the synchronization feature

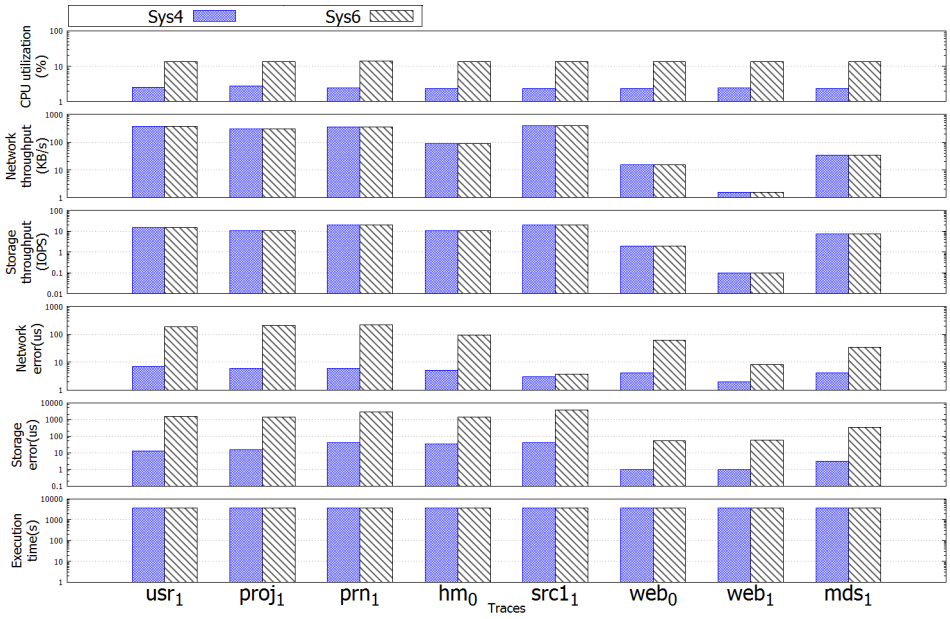


Figure 3.18: *Sys*4 and *Sys*6 comparisons with eight traces.

has limited influence on replayers and they are able to replay with high replay accuracy.

3.7 Systems Performance Evaluation

In this section, we present the case study of using the NetStorage replayer to evaluate performance of two servers, ***Sys4*** and ***Sys6***. ***Sys3*** as a client that communicates with ***Sys4***. ***Sys5*** as a client of ***Sys6***.

In our experiments, eight traces are used to compare two servers. As seen in Fig. 3.18, two servers produce similar results for network and storage throughputs and total execution time. This is because they can replay the traces based on timestamps with high replay accuracy. It means by replaying the same traces, they can finish replaying traces at almost the same time. The tiny throughput difference might come from replaying the last request. The issue errors of network and storage indicate how accurately replayers can issue requests on target systems. Thus, the system with lower issue errors might tolerate higher request rate than the other. Compared to ***Sys6***, ***Sys4*** attached to NVMe SSD and 10Gb Ethernet, has better storage system and higher network bandwidths

than ***Sys6***. As a result, ***Sys4*** achieves lower network and storage issue errors than ***Sys6*** for all eight traces. The CPU utilization can tell us how much capacity of servers remains to run other applications. ***Sys4*** derives about 4.77x less CPU utilization than ***Sys6*** because ***Sys4*** has higher CPU frequency and larger number of logical CPUs than ***Sys6***. In summary, ***Sys4*** is preferred with the original applications of the eights traces or the applications which have characteristics similar to the eight traces.

3.8 Related Work

For the network, some network performance measurement tools like iPerf [69] and NetPerf [70] can only monitor the bandwidth while running real applications on both server and clients sides. Another type of network tool called network replayer is able to regenerate network traffic based on network traces. TCPreplay [68] is a network replayer which issues requests based on timestamps. It is designed to detect the network intrusion. Also, it provides traffic for passive sniffer devices and in-line devices. The trace format is based on PCAP format. Additionally, it can replay the requests at arbitrary speeds. TCPivo [71] is designed to replay the traces as a cost-effective tool which is able to be implemented on x86 based systems. Monkey [72] consists of two components, Monkey see and Monkey do. The Monkey see is used for capturing TCP packets and the Monkey do replays TCP packets. It helps to evaluate the effects of different network implementations or protocol optimization. Some other works focus on debugging glitches of network by pervasively instrumenting the whole network ecosystem like OFRewind [4] and Netreplay [73], XTrace [6]. The works like [74][75] are designed to implement recording and replaying the full network with the context of distributed applications.

The storage performance evaluation tools such as IOzone [76], Iometer [77] and Bonnie/Bonnie++ [78][79] generating those synthetic workloads to I/O system are not based on real applications. Another type of storage tool can replay storage I/O requests. As discussed in [64], hfplayer, the state-of-art replayer, has better replay accuracy than other storage I/O replayers like FIO (Flexible I/O Tester) [80], blkreplay [7] and Buttress [8]. Moreover, TBBT [9] is the first comprehensive NFS trace replayer which is able to detect and replay missed requests in the traces. However, as discussed in Section 3.2,

none of above works satisfy the requirements of issuing high-throughput network and storage traces.

3.9 Conclusion

In this paper, we built a network-storage system level replayer called NetStorage that consists of a storage replayer, network replayers and manager components. The replayers are used to replay storage and network traces on target systems. The manager components are proposed to manage synchronization between replayers. During replaying processes, the manager periodically monitors replayers and synchronizes them with the synchronization period *sync*. Additionally, the manager revises timers of replayers in order to keep the same request rates as original traces. In our experimental results, both storage and network replayers are able to replay trace with high replay accuracy compared to previous replayers, respectively. Moreover, by investigating different scenarios, the synchronization has limited influence on the performance of NetStorage. Finally, the NetStorage replayer evaluates two systems by replaying eight traces. According to the metrics like CPU utilization and issue error, people can easily distinguish which systems have the better performance when running the applications of the eight traces.

Chapter 4

Erasure code for Reliability

In this work, we mainly introduce our work [81] to focus on the reliability of the storage systems.

4.1 Introduction

The new storage systems often have to deal with multiple disk failures. With steadily increasing the capacity of individual disks such as HDD, Kinetic drive [82] and interlaced magnetic recording (IMR) drive [83], disk failures have remained a significant concern [11][12][84]. Therefore, as demand for storage continues to grow, large storage systems often have to deal with multiple simultaneous disk failures. In order to tolerate failures, people use the Redundant Array of Independent Disks [10] (RAID) configuration to build fast and reliable disk systems.

RAID (Redundant Array of Independent Disk) [1] is becoming increasingly important because of two primary benefits: (1) high read performance and (2) robust fault tolerance. However, RAID system suffers a significant write penalty. Two major factors affect the write performance for an I/O request in a RAID system, the computing complexity and the number of parity updates. In RAID systems, one of the most common types, RAID-6, is commonly used in large disk arrays since RAID-6 can tolerate up to two simultaneous disk failures. Compared to RAID-5, though RAID-6 systems provide much safer systems, RAID-6 systems suffer more write penalties because they involve higher parity computation complexity and more parity updates.

Previously, the most popular RAID-6 uses the Reed-Solomon code [85]. However, it encounters very high overhead from the parity computation. Later, an new code called Cauchy Reed-Solomon (CRS) [86] improved the parity computation by converting multiplication to XOR operations. However, there still has been significant opportunity to reach the optimal computation complexity that has the minimum number of XORs for parity updates. Regarding the reduction of the overhead from the parity computation, several previous codes achieve the optimal computation complexity including X-code [13], P-code [14], B-code [87], EVENODD code [88], and RDP-code [15]. All of these codes use the minimum number of XORs in their encoding processes. However, those codes need to update more parities to achieve the optimal computation complexity because they use scattered layouts.

In addition, a degraded mode read operation can be regarded as the inverse operation of a write operation, whose performance is related to the decoding complexity and the number of read operations for recovering failed data blocks. Similar to the write operation, none of the previous codes can keep both the optimal decoding computation complexity and the minimum number of extra reads. Therefore, there is an opportunity to improve the write and degraded mode read performance while maintaining the optimal computation complexity and the minimum number of operations.

In this paper, we propose a new XOR-based RAID-6 code called Tier-code, which has optimal computation complexity and the minimum number of operations compared to the previous codes. The main characteristic of this code is to use two tiers of labeling, block-level labeling and chunk-level labeling, to compute parities. By using tier-labeling, the code produces better write performance and degraded mode read performance than previous codes while still maintaining fault-free read performance [13][14][89][15][90][16][91]. To evaluate the write and degraded mode read performance, we compare those codes from two aspects, parity computation complexity and I/O performance. Real system testing and VLSI implementation are employed to indicate the parity computation complexity. The DiskSim simulator is used to demonstrate the I/O performance of systems implemented with different codes.

4.2 Background and Related Work

4.2.1 Terms and Notations

To give a better understanding of the whole paper, we first summarize the terms and notations that will be used frequently throughout this paper.

- **Update Complexity [92]** is the number of parity blocks that need to be accessed when one data block is updated.
- **Computational Complexity [92]** corresponds to the operations that calculate the parities in a RAID systems. If a RAID code uses minimum number of XORs to compute parities, it is regarded as having the optimal computational complexity. Compared to XOR operations, the multiplication in the Galois field (GF) is much more expensive.
- **Storage Efficiency** is the ratio between the total useful data size and the raw size. In RAID-6 systems, the optimal storage efficiency is (M-2)/M, where M is the number of disks.
- **Multi-block Access Complexity [90]** indicates the average number of parity updates and read operations when a request accesses multiple data blocks.
- **Degraded Mode Read.** A RAID system enters degraded mode when a failure occurs in one or more disks. If one request wants to read data blocks including failed blocks, the system not only needs to read non-failed data blocks, but also needs to reconstruct the failed data blocks by reading other uncorrupted data and parity blocks. Thus, the system can provide the host with the requested data blocks correctly even with failed disks. However, the performance of degraded mode reads may be much lower than the performance of normal read operations due to the time required to access the additional blocks and compute the desired block.
- **Degraded Read Complexity** is defined in terms of the average number of extra read operations for recovering the failed data blocks in the RAID system during degraded mode.
- **Data Block Labeling and Parity Block Labeling.** The labeling shows the dependency of parity and data blocks. Parity is computed from those data blocks that share the same label as the parity's label. As seen in Fig. 4.1, the two parity

blocks labeled "1" are computed from the four data blocks labeled "1". **Block-level labeling** and **chunk-level labeling** are two-tier labellings. Each block consists of several chunks.

- A **Stripe** is a set of data and parity blocks that have the same label value.
- **Read-modify-write [93].** If a write request does not cover a whole stripe array, RAID systems cannot simply write the small portion of data to disks. The systems first must read old data blocks in this stripe, then obtain new data from the host, and then compute new parities using those data. After computing the parities, it can write the new data to target locations and update the new parities.

4.2.2 Background and Related Work

Erasure coding is a popular technique which has been used in many fields such as network and storage systems. Recently, new erasure codes are designed for storage systems to tolerate up to two disk failures. However, those erasure codes designed for RAID-6 storage systems suffered parity write and degraded read overhead. Some of RAID-6 codes are introduced in the following:

Reed-Solomon code [85] is a popular erasure coding technique using Galois Field arithmetic ($GF(2^w)$) (where w is the number of columns for the encoding matrix) during coding and decoding. However, the computation overhead of Galois Field arithmetic is very expensive.

PS-code [90] is a vertical code that attains optimal multi-block access complexity based on its labeling algorithm. It improves write performance when the write operations access multiple contiguous blocks. However, even though it uses the Cauchy Reed-Solomon code to compute its parities, PS-code still suffers low performance on the parity computation.

RDP-code [15] is constructed using $prime + 1$ columns and $prime - 1$ rows, which is similar to EVENODD code in terms of using the diagonal data blocks to calculate the second parities. The difference lies in the RDP-code's ability to obtain the optimal computational complexity because it directly XORs all diagonal data blocks instead of

introducing an extra intermediate parameter as used in EVENODD code.

X-code [13] has the $p * p$ structure, where p is a prime number. Data blocks are stored in the first $p-2$ rows and parities are stored at last two rows. It uses two diagonals with slope 1 and -1 to compute the first and second row parity. X-code has the optimal update complexity and computational complexity.

Liberation codes [94] is proposed with $prime + 2$ disks based on a bit matrix-vector product. A specified binary distribution matrix and a data vector are used in coding process. They propose a new class of X_i matrices for the liberation codes and use the notion of "bit matrix scheduling" to improve the decoding performance dramatically. However, they do not reach the optimal computational complexity.

P-code [14] is constructed with a $(p-1)/2 * (p-1)$ matrix. The parity blocks are located at the first row and the data blocks are located at the remaining $(p-3)/2$ rows. A pair of tuple values (m, n) and integer i are assigned to each data block and parity, respectively. By following the labeling rules in P-code, it can achieve the optimal computational complexity and update complexity.

HV-code [16] is a type of hybrid vertical and horizontal code that has optimal computational and update complexity. By using the horizontal chain to compute first parties and the diagonal chain to compute second parties, it improves I/O balancing and optimizes the operation of partial stripe writes to contiguous data elements.

The above codes share a common problem: they are not able to achieve optimal computation complexity, low multi-block access complexity, and low degraded read complexity at the same time. As shown in the following, our new Tier-code achieves all of these goals.

4.3 Tier-code

4.3.1 Tier-code Description

In RAID-6 systems, for each data block write, at least two parity block updates have to take place [13][14][94][88][92][16][95]. For such case, the update complexity [92] indicates the number of parities when one data block is updated. Additionally, in real systems, multiple blocks are updated together for each write I/O request. This scenario is investigated by the multi-block access complexity [90], which demonstrates one request that requires multiple block writes. Therefore, two above complexities can reflect the overhead for write requests in RAID systems.

Tier-code is a RAID-6 code constructed using $M = p + 1$ disks, where p is prime number. The main purpose of Tier-code is to improve write and degraded mode read performance. To achieve the best update complexity and multi-block access complexity, Tier-code uses the contiguous block-level labeling layout. Moreover, a revised code labeling algorithm is applied into Tier-code's chunk-level labeling so that Tier-code has the optimal computation complexity by using the minimum number of XORs to compute the parities. The chunk-level labeling may use the codes such as RDP-code, P-code and other and then most of the array codes, especially for the strong systematic ones, will have similar performance. In this paper, we use revised RDP-code as an example for chunk-level labeling. Compared to XORs, some previous works like RS-code using multiplication over GF to compute parities faces significant overhead on the parity computation time. Therefore, by using the two-tiered labeling algorithms, Tier-code can achieve better write and degraded mode read performance. In addition, the storage efficiency of Tier-code is the same as other codes, which is $(M-2)/M$. The labeling and encoding algorithms are described in the following paragraphs.

Tier-code's encoding algorithm can be divided into three primary steps – block-level labeling, chunk-level labeling and construction. Each block consists of $(M-2)$ chunks. Fig. 4.1 and Fig. 4.2 give two examples of block-level labeling and chunk-level labeling.

Block-level Labeling

In the block-level labeling, we assume that the block matrix has the size of $M * M/2$ including both data blocks and parity blocks as seen in Fig. 4.1. i $(0 < i \leq M/2)$ and j

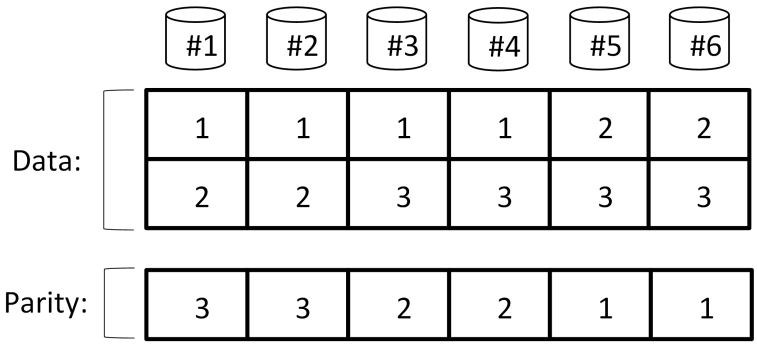


Figure 4.1: An example of the Tier-code with number of disks M=6. The labels of the parity row are calculated in Eq. 4.2. As an illustration, the two parity blocks P_5 and P_6 labeled "1" are computed from four data blocks labeled "1".

$(0 < j \leq M)$ are the row number and the disk number, respectively. D_{ij} is the value of the data block located at the i^{th} row and the j^{th} disk, and C_{ij} is its corresponding data block's label value. First, we assign integer labels to each data block where the label C_{ij} can be obtained from Eq. 4.1.

$$C_{ij} = \lfloor \frac{i * M + j - 3}{M - 2} \rfloor \tag{4.1}$$

To compute the parity in the block level, the parity row is labeled using Eq. 4.2. P_j is the parity's label value $(0 < P_j \leq M/2)$, j $(0 < j \leq M)$ is the disk number (or column index).

$$P_j = \lfloor \frac{M - j + 2}{2} \rfloor \tag{4.2}$$

As shown in Fig. 4.1, with an example using $M = 6$ disks, label values of the data and parity blocks are distributed across all disks, which are computed from Eq. 4.2.

Chunk-level labeling

After the block-level labeling, each block-level stripe needs to be addressed by the chunk-level labeling. First, we focus on one block-level stripe because all block-level stripes share the same chunk-level labeling algorithm.

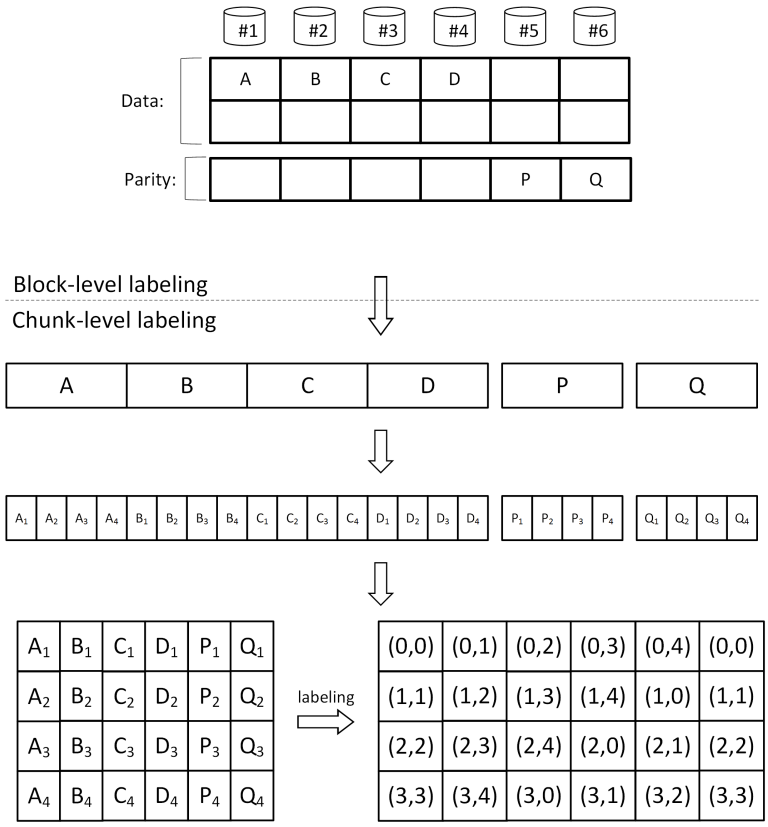


Figure 4.2: An example of Tier-code with the number of disks $M = 6$. Suppose the blocks 'A', 'B', 'C', 'D', 'P' and 'Q' are from the same stripe with the label value '1'. The first four blocks are data blocks. The blocks 'P' and 'Q' are parities. The labeling tuples are assigned to the blocks. The first element of the tuple is used for the parity block 'P' computation. The second element of the tuple is used for the parity block 'Q' computation. The other block-level stripes need to follow the same process of the tuple labeling to compute the parity values.

The chunk-level labeling has two steps. The first step of the chunk-level labeling is to split each block of the selected block-level stripe into $(M-2)$ chunks. Then, the second step is to assign a label tuple to the $(M-2)$ chunks. The labeling algorithm is shown in Eq. 4.3.

$$T_{ij} = (i, j\%(M-1)+i)) \tag{4.3}$$

where, T_{ij} is the labeling tuple of the i^{th} chunk of the j^{th} block and M is the number of disks. The first element of T_{ij} is the labeling for computing parity P. The second element is the labeling for computing parity Q.

Fig. 4.2 shows a tuple labeling example with $M = 6$. The blocks from the same block-level stripe with the label value '1' in Fig. 4.1 are split into smaller size of chunks and then they are given chunk-level tuple labels. Finally, the rest of the block-level stripes in one matrix need to be labeled using the chunk-level labeling by following the above steps.

4.3.2 Construction

After the chunk labeling process, the algorithm starts to compute parities. For the parity block P, each chunk of the parity P is computed by XORing all the data chunks whose first values of the tuple are equal to the first element of the parity P's chunk (We call the stripe as P stripe). After computing the parity Ps of all chunks, the parity P computation is done for this chunk-level stripe. The parity Q that are computed from second elements of tuples follows a similar process (We call the stripe as Q stripe). Two computation examples for the parities P and Q are given in Eq. 4.4 and Eq. 4.5 based on the layout shown in Fig. 4.2.

$$\begin{cases} P_1 = A_1 \oplus B_1 \oplus C_1 \oplus D_1 \\ P_2 = A_2 \oplus B_2 \oplus C_2 \oplus D_2 \\ P_3 = A_3 \oplus B_3 \oplus C_3 \oplus D_3 \\ P_4 = A_4 \oplus B_4 \oplus C_4 \oplus D_4 \end{cases} \tag{4.4}$$

$$\begin{cases} Q_1 = A_1 \oplus C_4 \oplus D_3 \oplus P_2 \\ Q_2 = A_2 \oplus B_1 \oplus D_4 \oplus P_3 \\ Q_3 = A_3 \oplus B_2 \oplus C_1 \oplus P_4 \\ Q_4 = A_4 \oplus B_3 \oplus C_2 \oplus D_1 \end{cases} \tag{4.5}$$

The construction for the selected block-level stripe is done after finishing computation of the parities. Then, another block-level stripe is selected for its construction following exactly the same process above until all of the block-level stripes have been constructed.

4.3.3 Reconstruction of Tier-code

In Fig. 4.1, we can see that, in the block-level labeling, the parity P and parity Q are computed from the same data blocks because both of them share the same labeling algorithm. However, in the chunk-level labeling, the parity P and parity Q are calculated from different chunks based on the first element and second element of the chunk-level labeling tuples, respectively.

In the block-level labeling, each block label value is unique in one disk. Also, each block label value is distributed across all the disks. Consequently, every stripe has M blocks covers M disks. In other words, each disk contains only one block from each stripe. Moreover, each stripe has the same construction process. Therefore, we only need to consider one block-level stripe reconstruction process and the rest of the stripes use exactly same process for the reconstruction process.

For a block-level stripe, according to the chunk labeling algorithm shown in Fig. 4.2, the first elements of the chunks' tuples are the same for the chunks located in the same row. For the second element of tuples, each block contains $M-2$ chunks with $M-2$ different values. Additionally, any two blocks that keep $M-2$ chunks with $M-2$ different second labeling values have at least two different second labeling values. In other words, there are two different second labeling values in two blocks. The two unique second labeling values are the starting points for recovering failed blocks. An example of two failed blocks is given in Fig. 4.3. Block B contains the second labeling '1', '2', '3' and '4' in its chunks' tuples. Block C contains the second labeling '0', '2', '3' and '4' in its chunks' tuples. Therefore, the labeling '0' and '1' are two unique values in the block B and C labellings. So, the two chunks holding the unique second values are the starting points of the reconstruction process. As seen in Fig. 4.3, the two chunks are marked with ① as the first reconstruction step. Then, the chunks locating at the same row with the starting point chunks can be recovered from the first labeling values. After that, we always can find one or two failed chunks that have the unique second labeling value among the rest of the failed chunks. The reconstruction follows the above process until all of the chunks are recovered. Fig. 4.3 gives an example of the recovery process with $M = 6$ for one block-level stripe. Once one block-level stripe is recovered, the failed data blocks in other block-level stripes can be recovered using the same steps.

The chunk-level labeling algorithm is similar to the RDP-code and the correctness

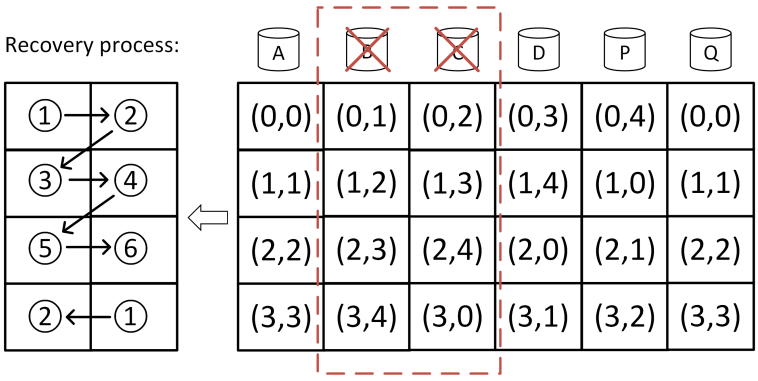


Figure 4.3: An example with block B and C failed for $M = 6$. The left part of the figure shows the recovery steps following the ordinal number. The first step is to recover the left top chunk and right bottom chunk because both chunks are the only failed chunks in their Q stripes. Once they are recovered, the second step is to recover the failed chunks from their P stripe. Then, the next step is to recover the Q stripe again and then P stripe until all chunks are recovered.

of RDP-code has previously been proven [15]. Thus, the Tier-code can recover at most two failed blocks in each block-level stripe. The block-level labeling ensures that each block-level stripe gets distributed across all disks. According to Eq. 4.1 and Eq. 4.2, one label value is evenly distributed across all disks for data blocks and parity blocks. Therefore, each block-level stripe can have at most two failed blocks when two disks fail. So, the Tier-code algorithm can tolerate up to two disk failures.

However, there are two limitations in Tier-code. The first one is that the number of disks must be one plus a prime number, greater than three. Second, since the chunk size is computed by Eq. 4.6, the block size must be divisible by the number of disks. Table 4.1 provides one solution for the block and chunk sizes for different numbers of disks.

$$Size_{block} = (M - 2) * Size_{chunk} \tag{4.6}$$

Table 4.1: One possible block size list for the different numbers of disks

Disk#	Block size(KB)	Chunk size(KB)
6	64	16
8	48	8
12	40	4
14	72	6
18	64	4
20	72	4
24	88	4
30	112	4
32	120	4
38	144	4
42	160	4
...	...	...

4.4 Write and Degraded Mode Read Performance Analysis

In this section, we analyze the performance of the RAID system to determine major factors which have influence on a complete RAID write operation and a degraded mode read operation. According to the major factors, we explain the reason why Tier-code has better performance on those factors than the previous codes.

4.4.1 Performance Analysis for RAID Systems

The write operation in a RAID-6 system consists of the following four major operations:

1. Reading the existing blocks (data blocks or parity blocks).
2. Computing parities with new data.
3. Writing data blocks.
4. Writing parity blocks.

First, the cost of operation #3 above is the same for all RAID-6 systems because the number of the data blocks written is the same for all systems employing RAID-6

codes with a same write request. Second, the overheads of operations #1 and #4 are determined by the number of parities written and the number of existing block reads, respectively. Thus, RAID-6 codes have different overheads of operations #1 and #4 caused by their labeling algorithms. Third, the overhead of #2 indicates the parity computation complexity. The computation complexity is varied in RAID-6 codes based on the number and types of operations for computing parties. Therefore, it is important to note that the performance of a complete write is primarily dependent on operations #1, #2 and #4. The effect of these operations is discussed in depth below.

For the operation #2, there is no doubt that the optimal operation for computing parities is the XOR operation and the minimum number of XOR operations for one parity computation with M disks is $M-3$ for RAID-6 systems. Therefore, to pursue the optimal computation complexity, some previous codes [13][14][88][92][95] use different data layouts.

For the operations #1 and #4, two metrics can be used to evaluate their overhead. The first one, update complexity [14], is the average number of updated parity blocks that are associated with a data block. The second one, multi-block access complexity [90], is defined in terms of the number of parity updates and read operations when a request needs to access multiple data blocks. The first metric applies to the scenario with only one data block update. The second metric reflects real life applications because it considers multi-block writes.

In the degraded mode read, if the data blocks are accessed by a request without containing any failed blocks, the degraded mode read request is the same as a normal read request. Otherwise, extra reads are required for recovering failed blocks. These extra reads are similar to the reads generated from a write operation for parity updates. Both of them need to read all of the available blocks in the same stripe. Thus, two operations primarily affect the degraded mode read performance – the extra non-failed block reads aiming to recover the failed blocks, and the decoding algorithm. The optimal decoding complexity is $M-3$ XOR operations for one block recovery with M disks, which is exactly the same as the encoding algorithm. For the extra non-failed block reads, we proposed a metric called degraded read complexity to indicate the overhead of reading non-failed blocks. Fig. 4.4 provides an example to show the comparison of number of extra reads with two failed disks between different codes. In this example,

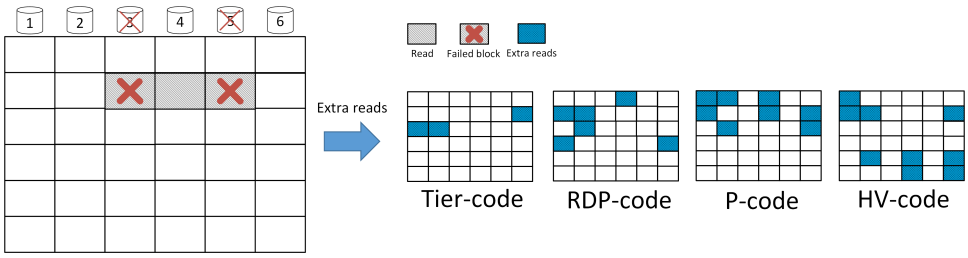


Figure 4.4: An example of a degraded mode read request of three blocks for different codes with failed disk#3 and disk#5. RDP-code, P-code, and HV-code need 6, 8 and 9 extra reads, respectively. However, Tier-code only needs 3 extra reads.

Tier-code needs the minimum number of reads to recover the blocks from disk#3 and disk#5 compared to previous works. In the following subsections, we use analytical models to conclude that Tier-code has better performance than previous codes over all above factors. The results in Section 4.5 validate the conclusions according to the software and hardware implementation, and simulation results.

4.4.2 Encoding and Decoding Complexity

To compute parities (operation #2), there are three major types of codes used in RAID-6 systems, Reed-Solomon codes, optimized Cauchy Reed-Solomon code (CRS) [86][96][90] and XOR-based codes. First, the Reed-Solomon code is the most popular code used in practical RAID-6 systems like mdadm [97]. It uses Galois Field arithmetic to compute the second parities. Second, the CRS codes like PS-code optimized Reed-Solomon codes by reducing the encoding and decoding complexity. They convert the operations over $GF(2^w)$ to XORs by converting distribution matrices to binary matrices. Third, the XOR-based codes do not have distribution matrices and directly use XORs to compute parities according to labeling algorithms. The XOR-based codes includes P-code, X-code, RDP-code, HV-code, etc. and our Tier-code belongs to the XOR-based codes as well. To investigate their encoding and decoding complexity, we calculated the number of operations and the type of operations during encoding and decoding processes as seen in Table 4.2 and Table 4.3.

In terms of the encoding process, the number of XOR operations is $(M-3)*2$ for the XOR-based codes. The number of XORs needed for the Cauchy Reed-Solomon codes are

obtained from the Jerasure simulator [98]. RS-code faces extra multiplication operations during its encoding process. Therefore, Tier-code (XOR-based codes) require the least XORs in Table 4.2. Tier-code requires about 1.5 times fewer XORs than PS-code. Compared to RS-code, Tier-code has no multiplications though they share the same number of XORs. For the decoding process, Tier-code has exactly the same number of XORs as its encoding process. For the decoding process of the CRS codes, it needs to calculate the inverse distribution matrix and then convert multiplications to XORs according to the inverse distribution matrix [99]. Since the inverse distribution matrices are different for different failed disk cases, Table 4.3 shows the average number of XORs for all possible failed cases. The RS-code has the normal decoding process as the common RAID-6 [100]. As the results shown in Table 4.3, Tier-code has about 2.4 times fewer XORs than PS-code and has substantially fewer XORs and look-up tables (LUTs) than RS-code. In summary, Tier-code theoretically has much better performance than Cauchy RS-code and RS-code on the encoding and decoding processes.

4.4.3 Complexity Evaluation for Write

For the operations #1 and #4 for writes, we use the multi-block access complexity to evaluate the write performance. The complexity indicates the average number of operations with varying the write request sizes. It is defined in Eq. 4.7.

$$\begin{cases} Ave_{write} = (\sum_{i=1}^{M*(M-2)} ReqW_i)/(M*(M-2)) \\ Ave_{read} = (\sum_{i=1}^{M*(M-2)} ReqR_i)/(M*(M-2)) \\ Ave = Ave_{write} + Ave_{read} \end{cases} \tag{4.7}$$

where $ReqW_i$ is the number of parity updates with I/O request size of i, $ReqR_i$ is the number of reads with I/O request size of i, and Ave is the average number of operations for M disks.

Different numbers of disks are investigated for the degraded read complexity. Fig. 4.5 compares Tier-code with previous codes in terms of the multi-block access complexity and Tier-code achieves the minimum number of parity updates and reads when varying the I/O request sizes for 6 disks. Moreover, Tier-code, RS-code and PS-code obtain the similar results because they use contiguous layouts. In addition, more investigations have been done with varying the number of disks. Fig. 4.6a shows the average number

Table 4.2: Comparison of the average number of operations between different codes during encoding.

disk#	Operation	Tier-code (XOR)	PS-code (CRS)	RS-code (RS)
6	$\oplus$	6	9.5	6
	$\times$	0	0	4
8	$\oplus$	10	15	10
	$\times$	0	0	6
12	$\oplus$	18	27.25	18
	$\times$	0	0	10
14	$\oplus$	22	32.75	22
	$\times$	0	0	12
18	$\oplus$	30	52.8	30
	$\times$	0	0	16
20	$\oplus$	34	58.8	34
	$\times$	0	0	18
30	$\oplus$	54	96.4	54
	$\times$	0	0	28
32	$\oplus$	58	104.2	58
	$\times$	0	0	30

of parity updates for different numbers of disks. Tier-code requires the fewest parity updates with the different number of disks. Additionally, as seen in Fig. 4.6b, Tier-code requires much fewer reads than the others. Furthermore, the difference between Tier-code and the other codes becomes even larger when increasing the number of disks. In summary, Tier-code has the best multi-block access complexity and comes out ahead in operations #1 and #4 described in Section 4.4.1 compared to the other codes.

4.4.4 Complexity Evaluation for Degraded Mode Read

As discussed in Section 4.4.1, the two major operations determine the performance of degraded mode read, the decoding algorithm and the extra non-failed block reads to

Table 4.3: Comparison of the average number of operations between different codes during decoding.

disk#	Operation	Tier-code (XOR)	PS-code (CRS)	RS-code (RS)
6	$\oplus$	6	16.5833	12
	LUTs	0	0	11.67
8	$\oplus$	10	24.6667	15.6
	LUTs	0	0	13.40
12	$\oplus$	18	42.1889	23.33
	LUTs	0	0	17.22
14	$\oplus$	22	51.0379	27.27
	LUTs	0	0	19.19
18	$\oplus$	30	73.7667	35.20
	LUTs	0	0	23.13
20	$\oplus$	34	83.2484	39.18
	LUTs	0	0	25.12
30	$\oplus$	54	130.4378	59.11
	LUTs	0	0	35.07
32	$\oplus$	58	139.9115	63.10
	LUTs	0	0	37.07

Note: Look-up tables (LUTs) are the operations to compute the inverse matrix multiplication. # of LUTs provides how many look-up-table operations are executed.

recover the failed blocks. We have discussed the decoding algorithm in Section 4.4.2. In this section, we focus on the number of extra reads to recover failed blocks.

A new metric called the *degraded read complexity* is proposed in this paper to compare the degraded mode read performance by counting extra read operations. In our evaluation of the degraded read complexity, we assume the location of the first block of each read operation follows a uniform distribution across all data blocks, 1 to $M*(M-2)$, in one matrix. Thus, the probability of each block being the starting point of a read is the same. For I/O request sizes, we only focus on the range from one block to

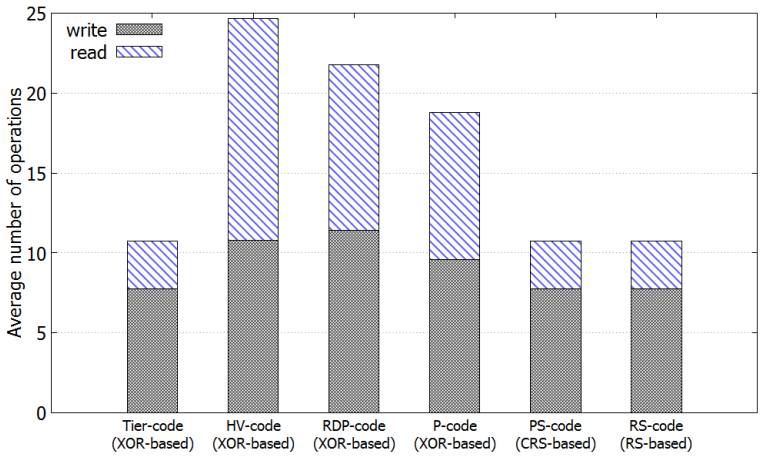


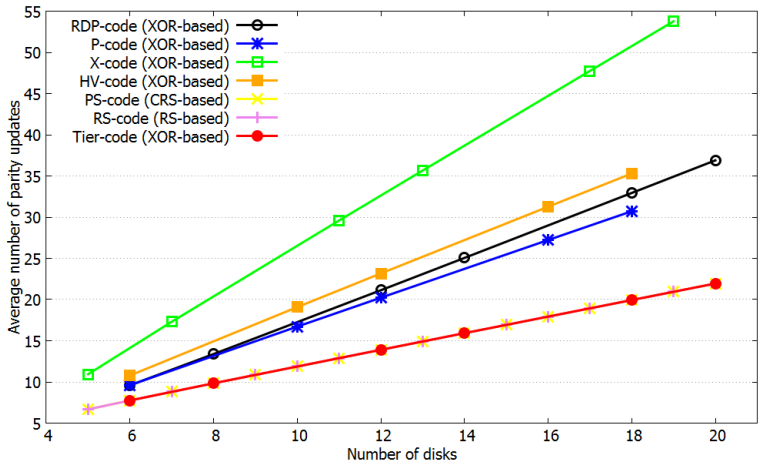
Figure 4.5: The average number of operations for the multi-block access complexity with M=6.

$M * (M-2)$ blocks because all matrix arrays have the same label layout and the degraded mode reads will be repeated after the I/O request size is larger than $M*(M-2)$. Moreover, we consider all possible combinations of two failed disks. For the degraded read complexity, we calculate the average number of the extra reads for recovering the failed data blocks. This ensures that the degraded read complexity reflects the code's degraded mode read performance accurately after considering all combinations of two failed disks, different I/O request sizes and different request start points. The degraded read complexity is calculated from Eq. 4.8.

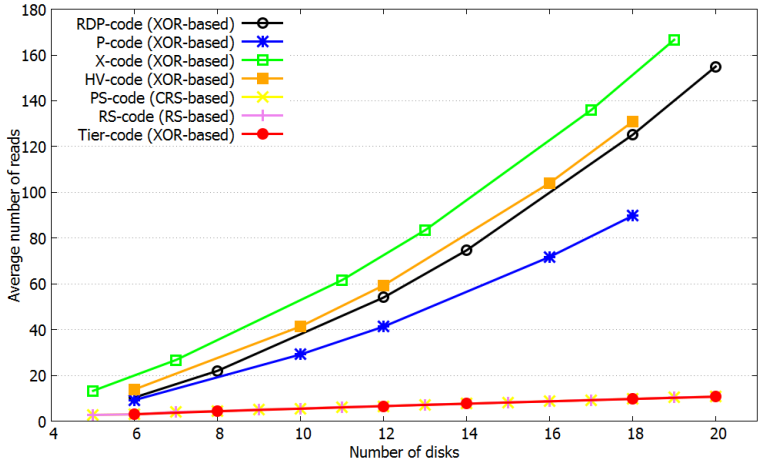
$$d = \frac{\sum_{f_1=1}^{M-1} \sum_{f_2=f_1+1}^{M} \sum_{i=1}^{M*(M-2)} \sum_{j=1}^{M*(M-2)} R_{f_1 f_2 ij}}{M^3 * (M-2)^2 * (M-1)/2} \tag{4.8}$$

where d is the degraded read complexity, f_1 is the first failed disk number, f_2 is the second failed disk number, i is the starting point of the requests varying from 1 to $M * (M-2)$, j is the I/O request size, and $R_{f_1 f_2 ij}$ is the total number of extra reads for recovering the failed data blocks when the I/O request size is j starting from the i^{th} block with failed disk $\#f_1$ and failed disk $\#f_2$.

The degraded read complexity metric in Eq. 4.8 is useful in comparing real systems since it analyzes the degraded mode read operation from several aspects, including number of disk failures, I/O request sizes, and request localities. Fig. 4.4 is an example



(a) Parity updates



(b) Reads

Figure 4.6: Multi-block access complexity for different numbers of disks.

of the degraded mode read of three blocks. In this example, $R_{f_1 f_2 ij} = 3$ for Tier-code with $f_1 = 3$, $f_2 = 5$, $i = 9$, and $j = 3$. This example shows that Tier-code needs three extra reads during the degraded mode read when the I/O request starts from the 9^{th} block with a size of 3 blocks when disks #3 and #5 are failed.

As seen in Fig. 4.7, Tier-code always has the lowest average number of extra reads compared to the previous RAID-6 codes. This is because Tier-code uses a contiguous labeling strategy on the block-level labeling, which is the same reason that Tier-code

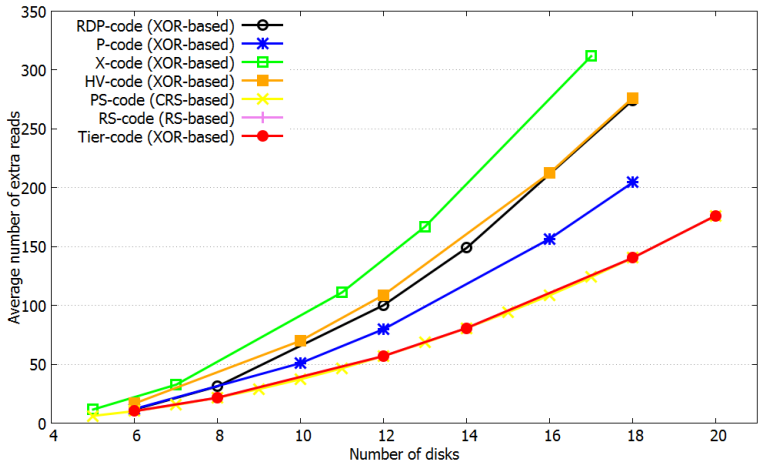


Figure 4.7: Degraded read complexity for different numbers of disks showing that Tier-code has the minimum average number of extra reads.

attains better write performance than previous codes. In terms of the degraded read complexity, Tier-code has 23.3% to 40.9% fewer extra reads on average than previous codes. Considering the conclusion from Section 4.4.2, Tier-code achieves the best performance for both factors in the degraded mode read.

In summary, Table 4.4 indicates the comparison between different RAID-6 codes. For the parity computation complexity, Tier-code achieves the minimum number of XORs for computing parities. For the write and degraded mode read complexities, Tier-code obtains the minimum average number of operations because of its contiguous layout. Therefore, it shows that Tier-code theoretically achieves the best results on all metrics. In next section, we validate the conclusion of those RAID-6 codes through real systems tests and simulations.

4.5 Experimental Results

In this section, we present the experimental results from two perspectives – computation complexity evaluation and I/O performance simulation – to validate that Tier-code has better performance than other codes. For the computation complexity, we used two methods, software testing and ASIC design, to measure the computation performance and hardware overhead of different encoding and decoding algorithms. For the overall

Table 4.4: Comparison of RAID-6 codes.

Codes	Parity update complexity	Multi-block access complexity	Degraded read complexity	Computation complexity	Disk number
Tier-code	2 updates	**Low cost**	**Low cost**	**Optimal**	p+1
RDP-code [15]	more than 2 updates	High cost	High cost	**Optimal**	p+1
PS-code [90]	2 updates	**Low cost**	**Low cost**	Not optimal	Any
RS-code [85]	2 updates	**Low cost**	**Low cost**	Not optimal	Any
HV-code [16]	2 updates	High cost	High cost	**Optimal**	p-1
X-code [13]	2 updates	High cost	High cost	**Optimal**	p
P-code [14]	2 updates	High cost	High cost	**Optimal**	p-1

I/O performance, we employed the DiskSim simulator [101] to measure the write and degraded mode read performance.

4.5.1 Computation Complexity Comparison

We begin by comparing the computation complexity amongst different codes using two methods, software testing and ASIC design synthesis. The software testing results provide a straightforward comparison between Tier-code (XOR-based code), PS-code (Cauchy Reed-Solomon code) and Reed-Solomon code in terms of the encoding and decoding throughputs. The ASIC design synthesis shows the hardware cost of different codes when performing those encoding and decoding in the hardware implementation such as RAID hardware controllers.

In the software testing, we performed encoding and decoding processes in the real system. The 64-bit system has 4 Intel Core i7-4790 CPUs with 3.60 GHZ. It has 4GB RAM size. For the test process, we used total 1GB data to perform encoding and decoding and the type of data is char. The PS-code and RS-code are performed in GF(16). For encoding, the testing results are shown in Fig. 4.8. Tier-code is always better than PS-code and RS-code with varying the number of disks. Tier-code has around 1.2GB/s encoding speed which is about 1.4 times better than PS-code and about 5 times better than RS-code. In Table 4.2, Tier-code has about 1.5 times fewer XOR operations than PS-code. Therefore, the software test results match the theoretical performance estimation well. For decoding, in the software test, suppose there are two

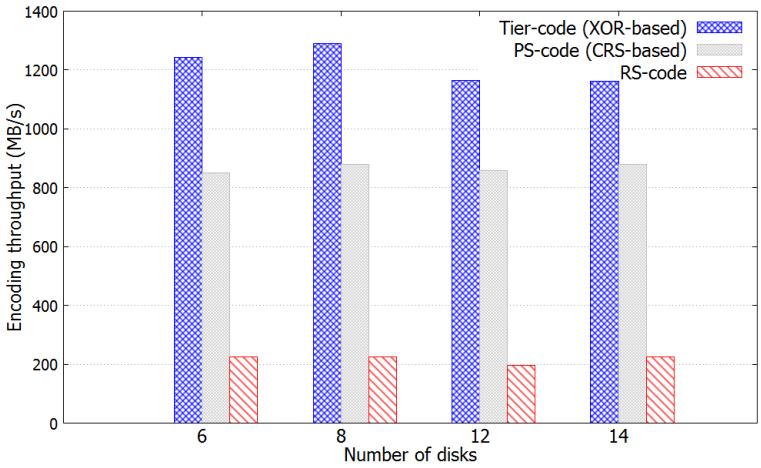


Figure 4.8: Software measurement results for the encoding process with different numbers of disks.

disks failed. Similar to encoding, the number of disks is varied from 6 to 14. As seen in Fig. 4.9, Tier-code has similar decoding speed as its encoding speed because their numbers of XORs are exactly the same with each other. However, PS-code only attains about 500MB/s decoding throughput and RS-code has only about 120MB/s decoding speed. Tier-code has about 2.5 times and 10 times faster decoding speed than PS-code and RS-code for all disk number cases, respectively. Considering the I/O performance of modern disks, current HDDs can reach over 150MB/s throughput and modern SSDs like Intel 3D Xpoint [102] even reach to over 2GB/s throughput. Therefore, the high speed of encoding and decoding in Tier-code will significantly improve performance of current RAID systems.

Additionally, we also explored the hardware implementation for those three types of codes. We used the design compiler to synthesize the three codes with FreePDK 45nm library [41]. The hardware ASIC design is different from the software implementation. For the hardware implementation, all operations can be performed in parallel, thus it becomes a trade-off between the hardware costs and encoding/decoding performance. In our implementation, we used 4 bits to express the data from each disk. To achieve a wider datapath or higher throughput, we can directly duplicate the circuits. As seen the encoding results in Table 4.5, Tier-code has a little better results than PS-code in

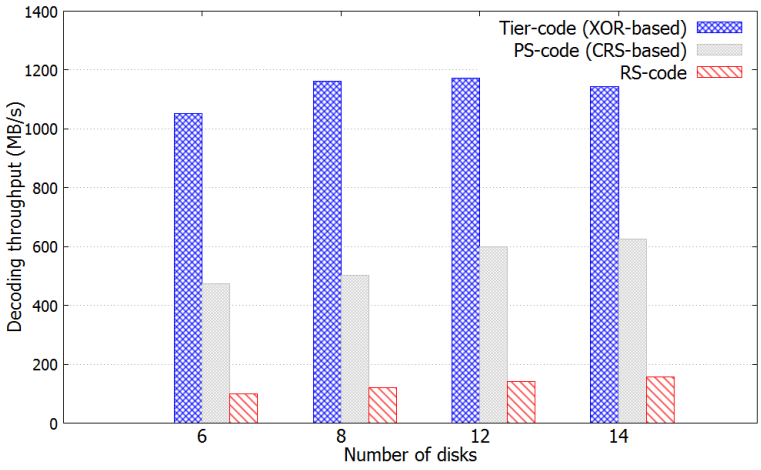


Figure 4.9: Software measurement results for the decoding process with different numbers of disks. The recovery process is only for the case when disks #1 and #3 have failed.

terms of area and power. The reason is that the parallelism in hardware implementation relieve the difference between Tier-code and PS-code. Additionally, the synthesis optimization in design compiler may narrow the computation gap between two codes. For example, in Eq 4.9, there are 6 XORs to compute P and Q. However, in the hardware implementation, it can reduce to 5 XOR gates by reusing $a_0 \oplus a_2$ in two equations.

$$\begin{aligned} P &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 = (a_0 \oplus a_2) \oplus a_1 \oplus a_3 \\ Q &= a_0 \oplus a_2 \oplus a_4 \oplus a_6 = (a_0 \oplus a_2) \oplus a_4 \oplus a_6 \end{aligned} \tag{4.9}$$

In addition, Tier-code is about 21%-47% faster on frequency and saves 25% - 60% energy compared to PS-code. Compared to RS-code, Tier-code has much better results in the hardware implementation because RS-code uses the look-up tables (LUTs) to implement the multiplication over GF(16). For the decoding results in Table 4.6, Tier-code derives much lower hardware cost than PS-code and RS-code. Tier-code attains 18%, 19%, 44% and 133% less hardware costs than PS-code and 11.78X, 5.66X, 2.14X and 20X less hardware cost than RS-code in terms of area, power, frequency and energy.

Table 4.5: Synthesis results for the encoding

	# of disks	Area (um^2)	Power (mW)	Frequency (GHz)	Energy (fJ)
Tier-code (XOR)	6	175.5	0.023	2.99	7.6
	8	369.3	0.046	2.41	19.1
	12	981.3	0.13	1.76	73.3
	14	1380.7	0.18	1.72	102.6
PS-code (CRS)	6	182.6	0.023	2.46	9.5
	8	380.8	0.049	1.77	27.8
	12	1033.6	0.14	1.30	105.2
	14	1396.6	0.19	1.15	163.8
RS-code (RS)	6	1796.5	0.11	1.06	99.3
	8	4022.4	0.23	0.97	240.4
	12	11143	0.58	0.82	709.1
	14	16035	0.92	0.82	1115.8

4.5.2 DiskSim Simulation Results

This section focuses on the I/O performance. We used DiskSim [101] simulations to compare the I/O performance of the different codes. The encoding and decoding computation time is integrated in the DiskSim simulation for write and degraded read performance. Since Tier-code is limited to (a prime number + 1) disks, and the previous RAID codes also have limitations on the number of disks they can use, we choose 6 disks for our experiments. The simulations use 64KB as the default block size and the total number of disks is 6. We use 10,000 I/O requests for sequential and random writes and the I/O request size is varied from 64KB to 2048KB. The reason for varying the request size is that, if one request size can completely cover the whole matrix data array (one matrix data array size is 24*64KB=1536KB for 6 disks), then these RAID-6 codes would have same the write performance since they have the same number of data and parity blocks to write. Thus, if the request is larger than 1563KB, we only need to focus on the incompletely covered parts of the blocks. So, we ignore the I/O request sizes larger than 2048KB.

Table 4.6: Synthesis results for the decoding

	# of disks	Area (um^2)	Power (mW)	Frequency (GHz)	Energy (fJ)
Tier-code (XOR)	6	173.6	0.029	3.20	9.0
	8	370.3	0.056	2.67	20.9
	12	980.4	0.14	1.74	81.8
	14	138.2	0.19	1.63	117.3
PS-code (CRS)	6	214.0	0.033	1.27	26.4
	8	455.5	0.072	1.11	64.6
	12	1132.2	0.16	1.14	143.4
	14	1554.3	0.22	1.21	182.8
RS-code (RS)	6	2945.3	0.25	0.38	669.1
	8	5741.4	0.47	0.38	1238.7
	12	13987	1.04	0.38	2762.2
	14	19268	1.39	0.38	3707.3

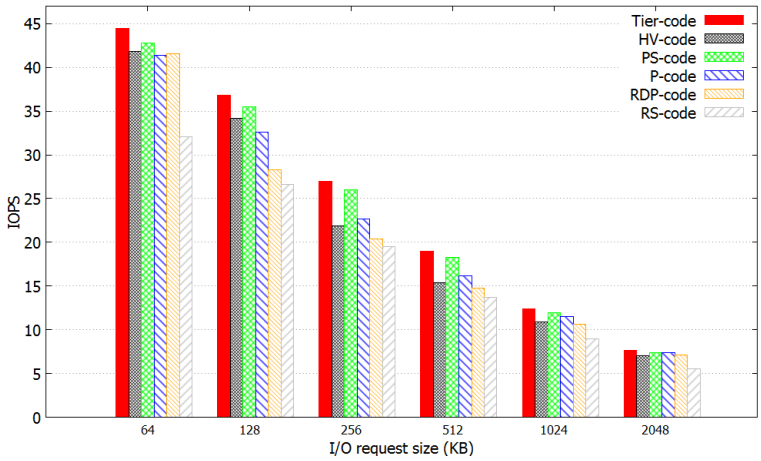


Figure 4.10: Sequential write performance between RAID-6 codes with M=6 while varying I/O request sizes.

As shown in Fig. 4.10 and Fig. 4.11, Tier-code has the best write performance compared to the other codes. Tier-code improves the sequential write performance

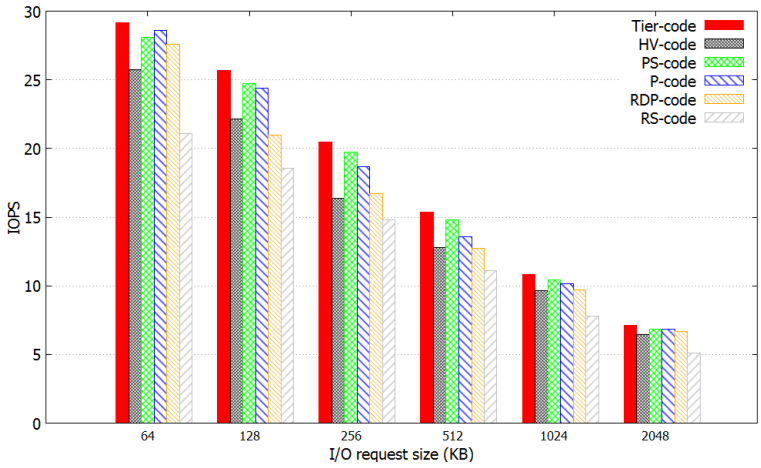


Figure 4.11: Random write performance between RAID-6 codes with M=6 while varying I/O request sizes.

20.5%, 14% and 11.5%, and random write performance 15%, 16.3% and 6.8% on average compared to RDP-code, HV-code and P-code, respectively. The primary reason for the improved performance is the smaller number of parity updates and the smaller number of reads for computing new parities. Compared to PS-code and RS-code, Tier-code achieves about 3.8% and 38.4% better write performance. This is because Tier-code has much better encoding speed than PS-code and RS-code.

To compare the degraded read mode operation of Tier-code to the other codes, we consider the cases of one and two disk failures. Even with two disk failures, a RAID-6 system can correctly continue to read data since the system can use a combination of uncorrupted data and parity blocks to recover the failed data blocks in the same stripe. For the one disk failure case, as shown in Fig. 4.12 and 4.13, in the degraded mode, Tier-code on average derives 86.4%, 12.8%, 12.3%, 20.7%, and 84.6% higher performance than HV-code, PS-code, P-code, RDP-code, and RS-code, respectively. For the two disk failure case, as shown in Fig. 4.14 and 4.15, in the degraded mode, Tier-code on average obtains 117.5%, 23.8%, 15.7%, 46.9%, and 157% performance improvement compared to HV-code, PS-code, P-code, RDP-code, and RS-code, respectively. The main reason is that Tier-code reads fewer data and parity blocks, and takes less decoding time when recovering the failed blocks. Compared two cases between one disk and two disk failure,

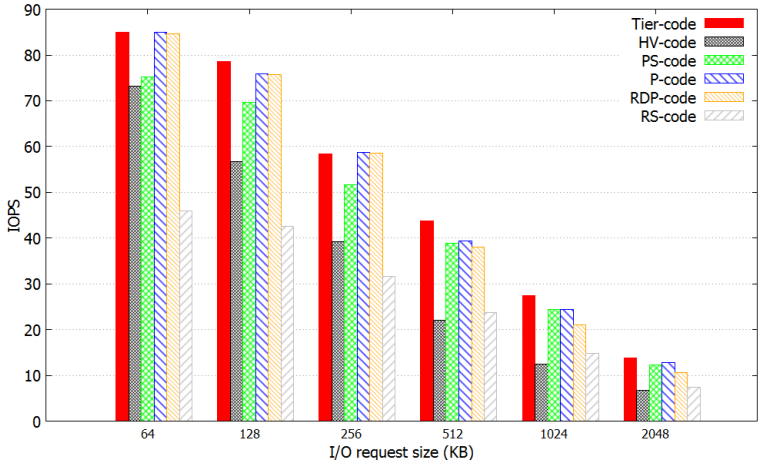


Figure 4.12: Sequential read performance in the degraded mode with one disk failure for RAID-6 with M=6.

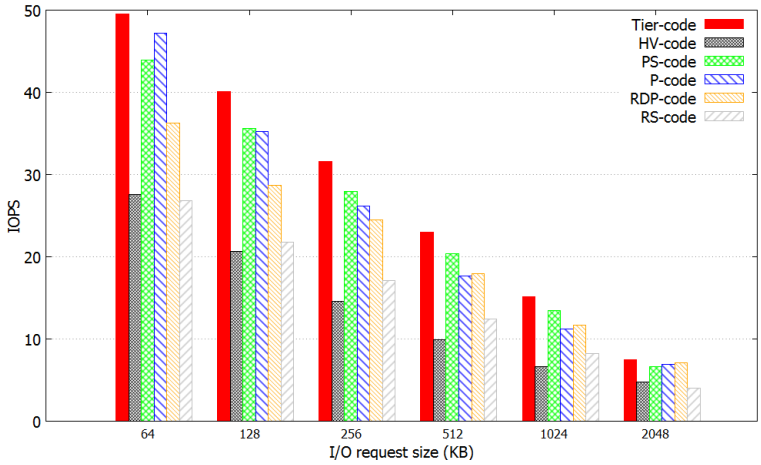


Figure 4.13: Random read performance in the degraded mode with one disk failure for RAID-6 with M=6.

the two disk failure has larger performance difference between those codes because two disk failure has higher probabilities to read failed blocks and also need to read more available blocks to recover them. Therefore, it enlarges the performance difference and Tier-code attains larger performance gains on two disk failure conditions than on the one disk failure conditions compared to previous RAID-6 codes.

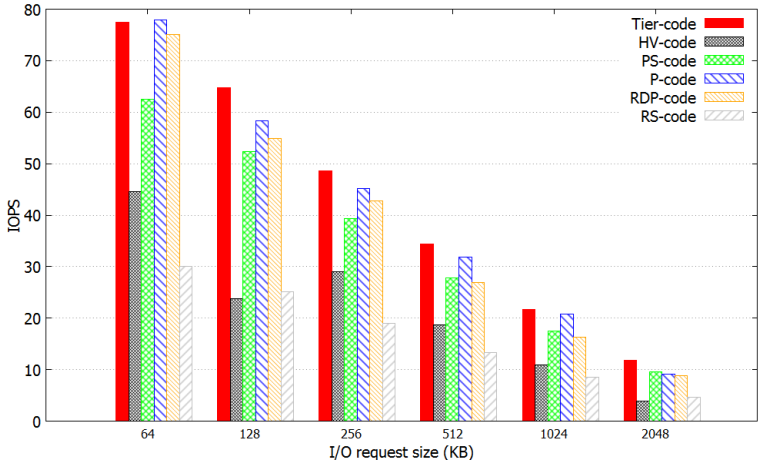


Figure 4.14: Sequential read performance in the degraded mode with two disk failures for RAID-6 with M=6.

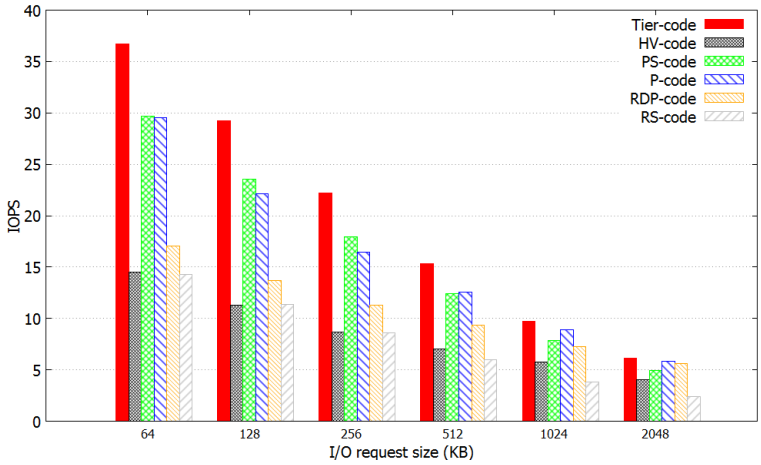


Figure 4.15: Random read performance in the degraded mode with two disk failures for RAID-6 with M=6.

4.6 Conclusion

This work proposed a new RAID-6 code called Tier-code that focuses primarily on improving the write performance and the degraded mode read performance of redundant arrays of independent disks. Our new labeling approach reduces the number of parity updates that need to be written to the disks and results in optimal parity computation

complexity. A new metric called degraded read complexity demonstrates the extra number of reads in the degraded-mode read. Different complexity metrics show that Tier-code has the best computation time, fewer write updates and fewer reads in terms of computation complexity, write performance and degraded-mode read performance, respectively. In the experimental results, the analytical model conclusions are validated for computation time and I/O performance. Computation time is compared using real system measurement and ASIC synthesis results. The Disksim simulator is used to validates the overall I/O performance conclusion. These results show that Tier-code improves the write and the degraded mode read performance compared to the previous codes.

Chapter 5

Conclusion and Future Work

Edge computing is one method of pushing the applications, data and computation components away from the centralized system. Each edge computing component has its own computation ability and storage capacity. And they also can communicate with the center server through the cloud and Internet to send the pre-computed data to the center. With increasing the interests of Internet of Things (IoT), more and more edge computing nodes will be attached to the cloud and will build a distributed edge computing infrastructure.

In Chapter 2, stochastic computing is introduced into the edge computing node since stochastic computing is a promising technology for low-cost design. Neural networks are used in edge computing nodes as their computing applications. Three main novel stochastic architectures are proposed in this chapter. First, we proposed a OR-based matrix multiplication adder, which can tremendously reduce the power and area consumptions. Second, an hardware oriented activation function is proposed for stochastic neural networks. By using that, we do not need any hardware cost for the activation function in neural networks, which is integrated in the other computation hardware. Therefore, it will further save the hardware cost for the stochastic neural network. Finally, quantized neural networks are investigated in stochastic neural networks because the quantized neural network is one method to reduce training time or hardware cost in conventional binary implementation. In this method, we proposed a new stochastic adder combining the features of stochastic computing and quantized values. With using the proposed stochastic adder, the quantized stochastic neural networks obtain

significant decrement of hardware cost.

In Chapter 3, we built a network-storage system level replayer called NetStorage that consists of a storage replayer, network replayers and manager components. The replayers are used to replay storage and network traces on target systems. The manager components are proposed to manage synchronization between replayers. During replaying processes, the manager periodically monitors replayers and synchronizes them with the synchronization period *sync*. Additionally, the manager revises timers of replayers in order to keep the same request rates as original traces. In our experimental results, both storage and network replayers are able to replay trace with high replay accuracy compared to previous replayers, respectively. Moreover, by investigating different scenarios, the synchronization has limited influence on the performance of NetStorage. Finally, the NetStorage replayer evaluates two systems by replaying eight traces. According to the metrics like CPU utilization and issue error, people can easily distinguish which systems have the better performance when running the applications of the eight traces.

In Chapter 4, we proposed a new RAID-6 code called Tier-code that focuses primarily on improving the write performance and the degraded mode read performance of redundant arrays of independent disks. Our new labeling approach reduces the number of parity updates that need to be written to the disks and results in optimal parity computation complexity. A new metric called degraded read complexity demonstrates the extra number of reads in the degraded-mode read. Different complexity metrics show that Tier-code has the best computation time, fewer write updates and fewer reads in terms of computation complexity, write performance and degraded-mode read performance, respectively. In the experimental results, the analytical model conclusions are validated for computation time and I/O performance. Computation time is compared using real system measurement and ASIC synthesis results. The Disksim simulator is used to validates the overall I/O performance conclusion. These results show that Tier-code improves the write and the degraded mode read performance compared to the previous codes.

In summary, in this thesis, we mainly focus on a edge computing distributed infrastructure including three aspects, edge computing low-cost design, reliability and performance evaluation. The stochastic computing implementation promises that the

edge computing node achieves low-cost design. The trace replayer is capable of evaluating and debugging the edge computing distributed infrastructure. Finally, the proposed codes maintain the reliability of the infrastructure.

Future work will be directed towards two aspects. For the edge computing, the new technology can be applied into stochastic neural network implementations such as spintronic devices in order to further decrease the hardware cost. Second, for the reliability of the distributed environment, new erasure codes may be proposed to improve the performance when some nodes are failed.

References

[1] Brian R Gaines et al. Stochastic computing systems. *Advances in information systems science*, 2(2):37–172, 1969.

[2] Amos R Omondi and Jagath Chandana Rajapakse. *FPGA implementations of neural networks*, volume 365. Springer, 2006.

[3] Seul Jung and Sung su Kim. Hardware implementation of a real-time neural network controller with a dsp and an fpga for nonlinear systems. *IEEE Transactions on Industrial Electronics*, 54(1):265–271, 2007.

[4] Andreas Wundsam, Dan Levin, Srini Seetharaman, Anja Feldmann, et al. Ofrewind: Enabling record and replay troubleshooting for networks. In *USENIX Annual Technical Conference*, 2011.

[5] Bingzhe Li, Farnaz Toussi, Clark Anderson, David Lilja, and David Du. Tracerar: An i/o performance evaluation tool for replaying, analyzing, and regenerating traces. In *Networking, Architecture and Storage (NAS), 2017 IEEE International Conference on*. IEEE, 2017.

[6] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[7] Thomas Schbel-Theuer. blkreplay and sonar diagrams. *https://github.com/schoebel/blkreplay/wiki*.

[8] Eric Anderson, Mahesh Kallahalla, Mustafa Uysal, and Ram Swaminathan. Buttress: A toolkit for flexible and high fidelity i/o benchmarking. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 4–4. USENIX Association, 2004.

[9] Ningning Zhu, Jiawu Chen, Tzi-Cker Chiueh, and Daniel Ellard. Tbbt: scalable and accurate trace replay for file server evaluation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 392–393. ACM, 2005.

[10] David A Patterson, Garth Gibson, and Randy H Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

[11] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *FAST*, volume 7, pages 17–23, 2007.

[12] Bianca Schroeder and Garth A Gibson. Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you? In *FAST*, volume 7, pages 1–16, 2007.

[13] Lihao Xu and Jehoshua Bruck. X-code: Mds array codes with optimal encoding. *Information Theory, IEEE Transactions on*, 45(1):272–276, 1999.

[14] Chao Jin, Hong Jiang, Dan Feng, and Lei Tian. P-code: A new raid-6 code with optimal properties. In *Proceedings of the 23rd international conference on Supercomputing*, pages 360–369. ACM, 2009.

[15] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14, 2004.

[16] Zhirong Shen and Jiwu Shu. Hv code: An all-around mds code to improve efficiency and reliability of raid-6 systems. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 550–561. IEEE, 2014.

[17] Bradley D Brown and Howard C Card. Stochastic neural computation. i. computational elements. *Computers, IEEE Transactions on*, 50(9):891–905, 2001.

[18] Peng Li, David J Lilja, Weikang Qian, Kia Bazargan, and Marc D Riedel. Computation on stochastic bit streams digital image processing case studies. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 22(3):449–462, 2014.

[19] Meng Yang, Bingzhe Li, David J. Lilja, Bo Yuan, and Weikang Qian. Towards theoretical cost limit of stochastic number generators for stochastic computing. In *IEEE Computer Society Annual Symposium on VLSI*, 2018.

[20] Devon Jenson and Marc Riedel. A deterministic approach to stochastic computation. In *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on*, pages 1–8. IEEE, 2016.

[21] M Hassan Najafi, Shiva Jamali-Zavareh, David J Lilja, Marc D Riedel, Kia Bazargan, and Ramesh Harjani. Time-encoded values for highly efficient stochastic circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(5):1644–1657, 2017.

[22] Siting Liu and Jie Han. Energy efficient stochastic computing with sobol sequences. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 650–653. IEEE, 2017.

[23] Ankit Mondal and Ankur Srivastava. Power optimizations in mtj-based neural networks through stochastic computing. In *Low Power Electronics and Design (ISLPED, 2017 IEEE/ACM International Symposium on*, pages 1–6. IEEE, 2017.

[24] Kyounghoon Kim, Jungki Kim, Joonsang Yu, Jungwoo Seo, Jongeun Lee, and Kiyoung Choi. Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks. In *Proceedings of the 53rd Annual Design Automation Conference*, page 124. ACM, 2016.

[25] M. H. Najafi, D. J. Lilja, M. D. Riedel, and K. Bazargan. Low-cost sorting network circuits using unary processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(8):1471–1480, Aug 2018.

[26] M. H. Najafi and D. Lilja. High quality down-sampling for deterministic approaches to stochastic computing. *IEEE Transactions on Emerging Topics in Computing*, PP(99):1–1, 2018.

[27] M. H. Najafi, S. Jamali-Zavareh, D. J. Lilja, M. D. Riedel, K. Bazargan, and R. Harjani. An overview of time-based computing with stochastic constructs. *IEEE Micro*, 37(6):62–71, November 2017.

[28] M. Hassan Najafi, David J. Lilja, and Marc Riedel. Fast-Converging, Scalable, Deterministic Bit-Stream Computing using Low-Discrepancy Sequences. In *2018 27th International Workshop on Logic and Synthesis (IWLS)*, pages 1–6, June 2018.

[29] Jeffery A Dickson, Robert D McLeod, and HC Card. Stochastic arithmetic implementations of neural networks with in situ learning. In *Neural Networks, 1993., IEEE International Conference on*, pages 711–716. IEEE, 1993.

[30] Weikang Qian and Marc D Riedel. Synthesizing logical computation on stochastic bit streams. *submitted to Communications of the ACM*, 2010.

[31] Peng Li, David J Lilja, Weikang Qian, Marc D Riedel, and Kia Bazargan. Logical computation on stochastic bit streams with linear finite state machines. *IEEE Transactions on Computers*, page 1, 2012.

[32] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[33] Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Mahowald, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947–951, 2000.

[34] Ronan Collobert and Samy Bengio. Links between perceptrons, mlps and svms. In *Proceedings of the twenty-first international conference on Machine learning*, page 23. ACM, 2004.

[35] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[36] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[37] Weikang Qian, Chen Wang, Peng Li, David J Lilja, Kia Bazargan, and Marc D Riedel. An efficient implementation of numerical integration using logical computation on stochastic bit streams. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pages 156–162. IEEE, 2012.

[38] Armin Alaghi and John P Hayes. Survey of stochastic computing. *ACM Transactions on Embedded computing systems (TECS)*, 12(2s):92, 2013.

[39] Weikang Qian, Xin Li, Marc D Riedel, Kia Bazargan, and David J Lilja. An architecture for fault-tolerant computation with stochastic logic. *Computers, IEEE Transactions on*, 60(1):93–105, 2011.

[40] Ji Li, Ao Ren, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, and Yanzhi Wang. Towards acceleration of deep convolutional neural networks using stochastic computing. In *The 22nd Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE*, 2017.

[41] James E Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W Rhett Davis, Paul D Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, et al. Freepdk: An open-source variation-aware design kit. In *Microelectronic Systems Education, 2007. MSE'07. IEEE International Conference on*, pages 173–174. IEEE, 2007.

[42] Kyounghoon Kim, Jongeun Lee, and Kiyoung Choi. An energy-efficient random number generator for stochastic circuits. In *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*, pages 256–261. IEEE, 2016.

[43] M Hassan Najafi, Peng Li, David J Lilja, Weikang Qian, Kia Bazargan, and Marc Riedel. A reconfigurable architecture with sequential logic-based stochastic computing. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(4):57, 2017.

[44] M Hassan Najafi, David J Lilja, Marc Riedel, and Kia Bazargan. Polysynchronous stochastic circuits. In *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*, pages 492–498. IEEE, 2016.

[45] Zhe Li, Ao Ren, Ji Li, Qinru Qiu, Yanzhi Wang, and Bo Yuan. Dscnn: Hardware-oriented optimization for stochastic computing based deep convolutional neural networks. In *Computer Design (ICCD), 2016 IEEE 34th International Conference on*, pages 678–681. IEEE, 2016.

[46] Bingzhe Li, M Hassan Najafi, and David J Lilja. An fpga implementation of a restricted boltzmann machine classifier using stochastic bit streams. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 68–69. IEEE, 2015.

[47] Bingzhe Li, M Hassan Najafi, and David J Lilja. Using stochastic computing to reduce the hardware requirements for a restricted boltzmann machine classifier. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 36–41. ACM, 2016.

[48] Ji Li, Zihao Yuan, Zhe Li, Caiwen Ding, Ao Ren, Qinru Qiu, Jeffrey Draper, and Yanzhi Wang. Hardware-driven nonlinear activation for stochastic computing based deep convolutional neural networks. *arXiv preprint arXiv:1703.04135*, 2017.

[49] V Lee, Armin Alaghi, J Hayes, Visvesh Sathe, and Luis Ceze. Energy-efficient hybrid stochastic-binary neural networks for near-sensor computing. *Proc. DATE17*.

[50] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[51] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[52] Yann LeCun and Corinna Cortes. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2010.

[53] Kyuyeon Hwang and Wonyong Sung. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pages 1–6. IEEE, 2014.

[54] Dong Yu, Frank Seide, Gang Li, and Li Deng. Exploiting sparseness in deep neural networks for large vocabulary speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 4409–4412. IEEE, 2012.

[55] Rangharajan Venkatesan, Swagath Venkataramani, Xuanyao Fong, Kaushik Roy, and Anand Raghunathan. Spintastic: spin-based stochastic logic for energy-efficient computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, pages 1575–1578. IEEE, 2015.

[56] Ruslan Salakhutdinov and Geoffrey Hinton. Deep boltzmann machines. In *Artificial Intelligence and Statistics*, pages 448–455, 2009.

[57] Vincent T Lee, Armin Alaghi, John P Hayes, Visvesh Sathe, and Luis Ceze. Energy-efficient hybrid stochastic-binary neural networks for near-sensor computing. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 13–18. IEEE, 2017.

[58] Bingzhe Li, Yaobin Qin, Bo Yuan, and David J Lilja. Neural network classifiers using stochastic computing with a hardware-oriented approximate activation function. In *2017 IEEE 35th International Conference on Computer Design (ICCD)*, pages 97–104. IEEE, 2017.

[59] B. Li, M. H. Najafi, B. Yuan, and D. J. Lilja. Quantized neural networks with new stochastic multipliers. In *2018 19th International Symposium on Quality Electronic Design (ISQED)*, pages 376–382, March 2018.

[60] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[61] Google. Google cloud platform. *https://cloud.google.com/docs/*.

[62] Amazon Elastic Compute Cloud. Amazon web services. *Retrieved November*, 9:2011, 2011.

[63] Alibaba. Alibaba cloud. *https://intl.aliyun.com/campaign/ecs-ssd-cloud-server?spm=a3c0i.7911826.709256.dcube4.U4oveP*.

[64] Alireza Haghdoost, Weiping He, Jerry Fredin, and David HC Du. On the accuracy and scalability of intensive i/o workload replay. In *FAST*, pages 315–328, 2017.

[65] Lawrence Berkeley National Labs. libcap. *http://www.tcpdump.org/*.

[66] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[67] Karl Pearson. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58:240–242, 1895.

[68] Tcpreplay tool. *http://tcpreplay.appneta.com/wiki/contributions.html*.

[69] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf: The tcp/udp bandwidth measurement tool. *htt p://dast. nlanr. net/Projects*, 2005.

[70] Rick Jones et al. Netperf: a network performance benchmark. *Information Networks Division, Hewlett-Packard Company*, 1996.

[71] Wu-chang Feng, Ashvin Goel, Abdelmajid Bezzaz, Wu-chi Feng, and Jonathan Walpole. Tcpivo: A high-performance packet replay engine. In *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 57–64. ACM, 2003.

[72] Yuchung Cheng, Urs Hölzle, Neal Cardwell, Stefan Savage, and Geoffrey M Voelker. Monkey see, monkey do: A tool for tcp tracing and replaying. In *USENIX Annual Technical Conference, General Track*, pages 87–98. Boston, MA, USA, 2004.

[73] Ashok Anand and Aditya Akella. Netreplay: a new network primitive. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):14–19, 2010.

[74] Samuel T King, George W Dunlap, and Peter M Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 1–1, 2005.

[75] Dennis Michael Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. 2006.

[76] Tom McNeal Don Capps. Iozone filesystem benchmark. *http://www.iozone.org*.

[77] David D Levine. Iometer users guide. *http://www.iometer.org/*.

[78] Tim Bray. Bonnie. *http://www.textuality.com/bonnie/*.

[79] Russell Coker. Bonnie++. *http://bonnie.sourceforge.net*.

[80] J. Axboe. Fio - flexible i/o tester. *http://freshmeat.net/projects/fio/*.

[81] Bingzhe Li, Meng Yang, Soheil Mohajer, Weikang Qian, and David Lilja. Tiercode: An xor-based raid-6 code with improved write and degraded-mode read performance. In *Networking, Architecture and Storage (NAS), 2018 IEEE International Conference on*. IEEE, 2018.

[82] M. Minglani, J. Diehl, X. Cao, B. Li, D. Park, D. J. Lilja, and D. H. C. Du. Kinetic action: Performance analysis of integrated key-value storage devices vs. leveldb servers. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 501–510, Dec 2017.

[83] Fenggang Wu, Baoquan Zhang, Zhichao Cao, Hao Wen, Bingzhe Li, Jim Diehl, Guohua Wang, and Du David HC. Data management design for interlaced magnetic recording. In *HotStorage*, 2018.

[84] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error correcting codes*. Elsevier, 1977.

[85] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial & Applied Mathematics*, 8(2):300–304, 1960.

[86] James S Plank and Lihao Xu. Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications. In *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on*, pages 173–180. IEEE, 2006.

[87] Lihao Xu, Vasken Bohossian, Jehoshua Bruck, and David G Wagner. Low-density mds codes and factors of complete graphs. *Information Theory, IEEE Transactions on*, 45(6):1817–1826, 1999.

[88] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. Evenodd: An efficient scheme for tolerating double disk failures in raid architectures. *Computers, IEEE Transactions on*, 44(2):192–202, 1995.

[89] Ping Xie, Jianzhong Huang, Qiang Cao, Xiao Qin, and Changsheng Xie. V 2-code: A new non-mds array code with optimal reconstruction performance for raid-6. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[90] Bingzhe Li, Manas Minglani, and David Lilja. Ps-code: A new code for improved degraded mode read and write performance of raid systems. In *Networking, Architecture and Storage (NAS), 2016 IEEE International Conference on*, pages 1–10. IEEE, 2016.

[91] James Lee Hafner. Weaver codes: Highly fault tolerant erasure codes for storage systems. In *FAST*, volume 5, pages 16–16, 2005.

[92] Chao Jin, Dan Feng, Hong Jiang, and Lei Tian. A comprehensive study on raid-6 codes: Horizontal vs. vertical. In *Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on*, pages 102–111. IEEE, 2011.

[93] Kumar Chinnaswamy, Michael A Gagliardo, Paul M Goodwin, John J Lynch, and James E Tessari. Read-modify-write operation, April 16 1991. US Patent 5,008,886.

[94] James S Plank. The raid-6 liber8tion code. *International Journal of High Performance Computing Applications*, 2009.

[95] Yingxun Fu and Jiwu Shu. D-code: An efficient raid-6 code to optimize i/o loads and read performance. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 603–612. IEEE, 2015.

[96] Xiaowen Chu, Chengjian Liu, Kai Ouyang, Ling Sing Yung, Hai Liu, and Yiu-Wing Leung. Perasure: a parallel cauchy reed-solomon coding library for gpus. In *Communications (ICC), 2015 IEEE International Conference on*, pages 436–441. IEEE, 2015.

[97] Linux raid. *https://raid.wiki.kernel.org/index.php/Linux_Raid*.

[98] James S Plank, Scott Simmerman, and Catherine D Schuman. Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2. Technical report, Technical Report CS-08-627, University of Tennessee, 2008.

[99] James S Plank, Jianqiang Luo, Catherine D Schuman, Lihao Xu, Zooko Wilcox-O'Hearn, et al. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST*, volume 9, pages 253–265, 2009.

[100] H Peter Anvin. The mathematics of raid-6. *http://www.dei.unipd.it/ capri/LDS/MATERIALE/raid6.pdf.gz*, 2007.

[101] John S Bucy, Jiri Schindler, Steven W Schlosser, and Gregory R Ganger. The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101). *Parallel Data Laboratory*, page 26, 2008.

[102] Intel. Revolutionizing memory and storage. *https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html*.