Algorithm/Architecture Co-Design for Low-Power Neuromorphic Computing

by

Nan Zheng

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering)
in The University of Michigan
2018

Doctoral Committee:

Professor Pinaki Mazumder, Chair
Assistant Professor Omar Ahmed
Associate Professor Clayton Scott
Professor Wayne Stark

Nan Zheng

zhengn@umich.edu

ORCID iD: 0000-0003-3261-2135

© Nan Zheng 2018

To my parents for their love, encouragement, and endless support

ACKNOWLEDGEMENTS

First, I would like to thank my research advisor, Prof. Pinaki Mazumder, who has provided me with much advice and guidance. I appeciate the opportunities he has offered during my PhD study to make me a good researcher. I would also like to thank my dissertation committee members, Prof. Omar Ahmed, Prof. Clayton Scott, and Prof. Wayne Stark. They provided me with valuable feedback, helping me improve this dissertation. I want to thank Prof. Satinder Singh and Prof. Todd Austin for the inspiring discussions with them. Special thanks to the National Science Foundation for supporting my doctoral research.

Many thanks to Prof. Michael Flynn who taught me the first integrated-circuit course at Michigan, Prof. David Wentzloff, Prof. Thomas Wenisch, Prof. Anthony Grbic, and many other professors who have had significant influence on me through their great and inspiring lectures. I am also very grateful to EECS staff Stephen Reger, Charlie Mattison, Beth Stalnaker, Karen Liska, Anne Rhoades, Steve Pejuan, Jose-Antonio Rubio. Special thanks to Joel VanLaven for his help with software tools.

I appreciate the support and help from my labmates and friends, Idong Effong Ebong, Zhao Xu, Mahmood Barangi, Yalcin Yilmaz, Jaeyoung Kim, Mahdi Aghadjani, Jiamin Huang, Zhongjun Jin, as well as all of my other dear friends here at the University of Michigan.

I would like to thank my parents, Qinhong Zheng and Jiafen Wang, for their support and help both emotionally and financially. It is them who supported me and encouraged me at the most difficult time during my doctoral study. I dedicate this

dissertation to them for their endless love and support.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES viii

LIST OF TABLES xii

ABSTRACT xiii

CHAPTER

I. Introduction 1

1.1 History of Neural Networks 1
1.2 Neural Networks in Software 3
1.2.1 ANN 3
1.2.2 SNN 4
1.3 Need for Low-Power Neuromorphic Hardware 5
1.4 Challenges 8
1.4.1 Challenges from Learning 8
1.4.2 Challenges from Memory 9
1.5 Dissertation Organization 9

II. On-Line Learning for Hardware-Based Multilayer Spiking Neural Networks 11

2.1 Introduction 11
2.2 Estimating Gradients from Spike Timings 13
2.3 Simulation Results 25
2.3.1 One-Hidden-Layer Neural Network 27
2.3.2 Two-Hidden-Layer Neural Network 33
2.3.3 MNIST Benchmark 34
2.3.4 Inference with a Progressive Precision 36

2.4 Chapter Summary 39

III. A Low-Power Hardware Architecture for On-Line Supervised Learning in Spiking Neural Networks 40

3.1 Introduction 40
3.2 Hardware Architecture 41
3.2.1 Algorithm Adaptations 41
3.2.2 Layer Unit 43
3.2.3 Leveraging Sparsity 45
3.2.4 Background ST Update 47
3.3 CMOS Implementation Results 49
3.4 Chapter Summary 52

IV. A Low-Power Accelerator for Action-Dependent Heuristic Dynamic Programming 53

4.1 Introduction 53
4.2 Action-Dependent Heuristic Dynamic Programming 55
4.2.1 Actor-Critic Networks 56
4.2.2 Learning Algorithm 57
4.3 Hardware Architecture 60
4.3.1 On-Chip Memory 60
4.3.2 Datapath 63
4.3.3 Controller 66
4.4 Virtual Update Algorithm 67
4.4.1 Algorithm 68
4.4.2 Hardware Implementation Considerations 70
4.5 Design Examples 72
4.6 Chapter Summary 78

V. Memory Reliability Enhancement Through On-Chip Compensation and Error Correction 79

5.1 Introduction 79
5.2 Counteract Variations in SRAM 82
5.2.1 Modeling of Variations in SNM 82
5.2.2 Adaptive Body Biasing 92
5.2.3 Overheads 94
5.2.4 Implementation Examples 98
5.3 Improved One-Pass Chase Decoding of BCH Codes 100
5.3.1 One-Pass Chase Decoding 101
5.3.2 Eligibility Verification Algorithm 103
5.3.3 VLSI Architecture 106
5.3.4 Design Example 109

5.4 Chapter Summary 111

VI. Conclusion 113

6.1 Summary and Contributions 113
6.2 Future Work 114
6.3 Related Publications 115

BIBLIOGRAPHY 117

LIST OF FIGURES

Figure

1.1 Illustration of the neural networks inspired by biological neural networks . . . 2
1.2 Illustration of one layer of an ANN. . . . 3
1.3 History of the development of computing devices . . . 6
2.1 Illustration of a multilayer neural network. A neuron located at the lth layer is denoted as x_i^l, where i represents the index of that neuron. 14
2.2 Illustration of the two regions divided by the spike timing of a presynaptic neuron. The causal and anti-causal regions are defined according to the causal relationship between presynaptic and postsynaptic spikes. . . . 15
2.3 Comparison of the gradients obtained from numerical simulations with the gradients obtained from STDP for (a) the first-layer synapses w_{ij}^1, (b) the second-layer synapses w_{ij}^2, and (c) the third-layer synapses w_{ij}^3 . . . 22
2.4 Correlations between the estimated gradients and the gradients obtained from the FD numerical method for (a) different window size WIN_{STDP} and (b) different evaluation duration D_L. The results obtained for all three layers of synaptic weights (w_{ij}^1, w_{ij}^2, and w_{ij}^3) are compared. Two different backpropagation methods (layer-by-layer and direct) are also compared. . . . 24
2.5 Comparison of the training and testing correct rates achieved with different levels of refractoriness and different initial weights. The refractory mechanism is helpful in avoiding dense spike trains, which helps improve the learning results. . . . 28
2.6 Comparison of the training and testing correct rates achieved with the LIF neuron model and the modified LIF model. The results obtained with the conventional LIF model with white noise residue injection are labeled as “LIF w/ white noise”, whereas the results obtained with the modified LIF model is labeled as “LIF w/ quantization noise.” . . . 29

2.7 Comparison of the training and testing correct rates achieved with different initial conditions. The case with pseudo-random initial membrane voltages outperforms the cases with fixed initial membrane voltages. A pseudo-random leakage technique is also employed to further improve the learning performance. 30
2.8 Comparison of the testing correct rates achieved with different learning and inference durations. The longer the learning or inference duration is, the higher the correct rate. 32
2.9 Comparison of the testing correct rates achieved with the two different backpropagation schemes. The two methods achieve similar performances. The two-hidden-layer neural network can yield better performance, but it requires a longer learning duration. 34
2.10 Comparison of a) the recognition rate and b) the effective inference durations needed for different levels of reduced margin. The results are obtained with the one-hidden-layer neural network. 37
2.11 The recognition accuracy and the corresponding effective inference duration needed when the first-to-spike-K-spikes readout is employed. The results are obtained with the one-hidden-layer neural network. . 38
3.1 Effect of shortening bitwidth of the synaptic weights on the classification results. 42
3.2 Comparison of the recognition rates achieved with the proposed simplifications on the original algorithm. 43
3.3 Schematic of the circuit implementing one layer of the neural network. 44
3.4 Illustration of the sparsity in the employed neural network over 10000 testing images. 45
3.5 Cache structure employed to store the active ST information. 46
3.6 Schematic of the circuit that implements the background ST information updating technique. 47
3.7 State diagram of the background ST updating scheduler. 48
3.8 Comparison of the number of clock cycles per forward iteration for different buffer depth. 49
3.9 Chip layout of the CMOS implementation. 50
3.10 Time and energy needed per inference as a function of the inference accuracy. 51
4.1 Illustration of the actor-critic configurations used in the ADHDP algorithm. Two neural networks, critic network and actor network, are employed in the algorithm to approximate functions need to be learned. 55
4.2 Pseudocode for the ADHDP algorithm 58
4.3 Hardware architecture for the proposed accelerators. Data-level parallelism is exploited through utilizing multiple datapath lanes. A reconfigurable five/six-stage pipeline is used for the datapaths. . . . 61
4.4 Illustration of data flow and memory access patterns in the proposed accelerators. Data buffers are employed to exploit the locality of the data. Synaptic weights needed in on tile operation is stored in one row. 61

4.5 Comparison of the learning performances achieved with different levels of data quantization for three classic ADP benchmarks. The obtained performances are normalized with respect to those obtained from double-precision floating-point computations. 64
4.6 Illustration of the instructions used in the accelerators. (a) Format of the instruction. (b) List of all operation codes and their corresponding operations. (c) A sample program for implementing the ADHDP algorithm shown in Fig. 4.2. 66
4.7 Pseudocode for the *while* loop corresponding to the critic update when the virtual update algorithm is employed 71
4.8 Chip layout and floorplan of the accelerator chip with the virtual update algorithm 72
4.9 Comparison of the learning performances achieved by the accelerators and the software approach for three commonly used benchmarks. The results obtained from the accelerators are normalized to those obtained from software. The error bars correspond to a confidence interval of 95%. 73
4.10 Typical waveforms obtained in the triple-link inverted pendulum task with the baseline accelerator. In the figure, the unit for distances and angles are meter and degree, respectively. 74
4.11 Comparison of (a) the numbers of clock cycles needed for every critic/actor update iteration, (b) the power consumption, and (c) the energy consumption for every critic/actor update iteration for the ADP accelerators with and without the virtual update technique. The first two groups of data are obtained from the cart-pole balancing task, whereas the third group of data is obtained from the triple-link inverted pendulum task. 77
5.1 Illustration of the energy per operation versus the supply voltage. . 80
5.2 Schematic of an 8T SRAM cell. 82
5.3 Transition voltages of an SRAM cell inverter as the threshold voltages of the NMOS and PMOS transistor vary. The solid line is obtained with (5.2), whereas the circles are obtained from the circuit simulation tool. 84
5.4 Illustration of (a) two VTCs in the nominal case and two VTCs under process variation, and (b) VTCs under variation that are shifted by the same amount such that VTC2 is moved back to its original location. 85
5.5 A typical mapping function. 87
5.6 Comparison of the sensitivities obtained from $g(s)$ and the circuit simulation tool. Two sets of results match well, demonstrating that $g(s)$ can be employed for estimating the SNM. 88

5.7 Comparison of the distributions of (a) the single-sided SNMs and (b) the double-sided SNMs obtained from the conventional linear superposition model, proposed model, and MC simulations. The results obtained with exaggerated (2x) variations are also plotted to show the trend as variations grow in advanced technologies. 90
5.8 (a) Conceptual diagram of the proposed adaptive body biasing circuit. (b) Simplified equivalent circuit of (a). 93
5.9 Configuration of the proposed adaptive body biasing circuit. (a) One example of generating body bias to mitigate the read SNM degradation when the global variation is present. V_T^* is the desired transition voltage. (b) Another example of generating body bias to mitigate the read SNM degradation. In the figure, $V_{DD2} > V_{DD1}$. 94
5.10 Configuration of the proposed adaptive body biasing circuit in an SRAM array. 95
5.11 (a) Transition voltage and (b) body bias voltage of an SRAM cell inverter at different process corners with the help of the circuit shown in Fig. 5.9(a). 97
5.12 (a) Transition voltage and (b) body bias voltage of an SRAM cell inverter at different process corners with the help of the circuit shown in Fig. 5.9(b). 97
5.13 Comparison of the read SNM obtained with and without the proposed adaptive body biasing circuit as the threshold voltages of the PMOS transistor and the NMOS transistor vary. (a) The results are obtained with the circuit shown in Fig. 5.9(a). (b) The results are obtained with the circuit shown in Fig. 5.9(b) 99
5.14 Illustration of (a) the conventional and (b) the one-pass Chase decoding of BCH code. 102
5.15 Comparison of the error-correction performances of an HDD, an SDD with exhaustive polynomial search, and an SDD with the proposed eligibility checking algorithm. The BCH code used here is a $(4200, 4096)$ code over $GF(2^{13})$. 107
5.16 A VLSI architecture of the proposed eligibility checking algorithm. Block I and II conduct multiply and modulo operations, and Block III conducts polynomial inversion operation. 108
5.17 Diagram of the proposed polynomial multiplication array. Same multiplicands b_i are shared along diagonals, and multipliers a_i are shared along columns. Multiplication results c_i are obtained by adding each row with XOR trees. 109

LIST OF TABLES

Table

2.1 Information of limiting operations used to obtain data in Fig. 2.3 . 23
2.2 Comparison of the classification accuracies for the MNIST benchmark task. 35
3.1 Power consumption breakdown 50
3.2 Summary and comparison with prior works 51
4.1 Comparison of the computational complexity of conventional update and virtual update 72
4.2 Specifications of the ADHDP accelerator 75
5.1 Summary of hardware complexity of the proposed eligibility verification circuit 110
5.2 Comparison of hardware complexity and critical path delay 110
5.3 Summary of the proposed design 111

ABSTRACT

The development of computing systems based on the conventional von Neumann architecture has slowed down in the past decade as complementary metal-oxide-semiconductor (CMOS) technology scaling becomes more and more difficult. To satisfy the ever-increasing demands in computing power, neuromorphic computing has emerged as an attractive alternative. This dissertation focuses on developing learning algorithm, hardware architecture, circuit components, and design methodologies for low-power neuromorphic computing that can be employed in various energy-constrained applications.

A top-down approach is adopted in this research. Starting from the algorithm-architecture co-design, a hardware-friendly learning algorithm is developed for spiking neural networks (SNNs). The possibility of estimating gradients from spike timings is explored. The learning algorithm is developed for the ease of hardware implementation, as well as the compatibility with many well-established learning techniques developed for classic artificial neural networks (ANNs). An SNN hardware equipped with the proposed on-chip learning algorithm is implemented in CMOS technology. In this design, two unique features of SNNs, the event-driven computation and the inferring with a progressive precision, are leveraged to reduce the energy consumption. In addition to low-power SNN hardware, accelerators for ANNs are also presented to accelerate the adaptive dynamic programing algorithm. An efficient and flexible single-instruction-multiple-data architecture is proposed to exploit the inherent data-level parallelism in the inference and learning of ANNs. In addition, the accelerator

is augmented with a virtual update technique, which helps improve the throughput and energy efficiency remarkably. Lastly, two techniques in the architecture-circuit level are introduced to mitigate the degraded reliability of the memory system in a neuromorphic hardware owing to the aggressively-scaled supply voltage and integration density. The first method uses on-chip feedback to compensate for the process variation and the second technique improves the throughput and energy efficiency of a conventional error-correction method.

CHAPTER I

Introduction

1.1 History of Neural Networks

Even though modern processors based on the von Neumann architecture are able to conduct logic and scientific computations at an extremely fast speed, they may still perform poorly in many tasks that are trivial to humans, such as image recognition and natural language processing, etc. With the desire to harness the power of human brains, neural networks were developed. The research on neural networks has continued for decades. Fig. 1.1 illustrates three basic types of neural networks. The simplest neural network is the perceptron, where hand-crafted features are employed as input to the perceptron. Outputs of perceptrons are binary numbers obtained through hard thresholding. Therefore, the perceptron can be conveniently used for classification problems where inputs are linearly separable. A more sophisticated type of neural network is sometimes called the multilayer perceptron (MLP). Nevertheless, the "perceptrons" in an MLP are different from simple perceptrons. In an MLP, a non-linear activation function is associated with each neuron. Popular choices for the non-linear activation function are sigmoid function, hyperbolic tangent function, and the rectified linear unit. The output of each neuron is a continuous variable instead of a binary number in perceptrons. The MLP is widely adopted in the machine-learning community because this type of neural network is powerful in solving many real-life

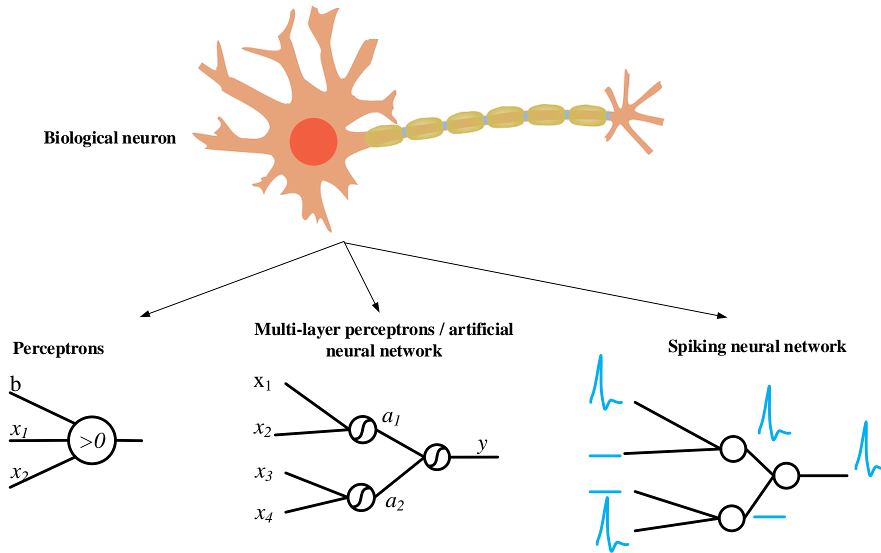


Figure 1.1: Illustration of the neural networks inspired by biological neural networks

problems and is also suitable for being implemented on general-purpose processors. The MLP is so popular that the word artificial neural network (ANN) is often used to specify it exclusively, even though ANN should have referred to any other neural networks besides biological neural networks. In this dissertation, we adopt this convention of referring an MLP as an ANN. A more complicated type of neural network is called a spiking neural network (SNN). Compared to the previous two types of neural networks, an SNN resembles more of a biological neural network in the sense that spikes are used to carry information. It is generally believed that SNNs are more powerful and more advanced than ANNs because the dynamics of an SNN is much more complicated.

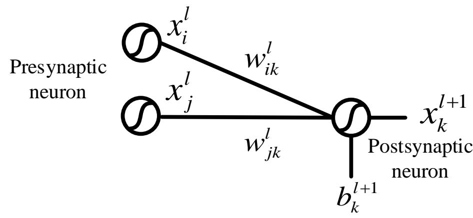


Figure 1.2: Illustration of one layer of an ANN.

1.2 Neural Networks in Software

1.2.1 ANN

Tremendous progress has been achieved in the late 1980s and the early 1990s for neural networks built in the form of software. One technique that truly boosted the development of ANNs was backpropagation [1]. It was shown that backpropagation is very effective in training multilayer neural networks. With the wide use of backpropagation, neural networks have been deployed to various real-life applications such as image recognition [2, 3], control[4, 5], prediction[6, 7], and so on.

In an ANN, information is encoded as real numbers. Fig. 1.2 illustrates one layer of an ANN. In the figure, the neural network consists of neurons that are represented by circles and synapses that are represented by wires. For each layer of a feedforward ANN, we call the neurons located at the input side of the synapse presynaptic neurons and the neurons located at the output side of the synapse postsynaptic neurons. During the process of evaluating the neural network, the activation levels from the presynaptic neurons x_i^l and x_j^l are multiplied with the synaptic weights w_{ik}^l and w_{jk}^l, respectively, where i and j indicate the index of the neuron and l specifies the index of the layer at which the neuron is located. The obtained products and the bias term b_k^{l+1} are added together and the sum is passed through an activation function in order to produce x_k^{l+1}, the activation level of the postsynaptic neuron. The computations associated with ANNs can be expressed as matrix operations, which can be

conveniently implemented on conventional processors.

In the late 1990s, it was found that other machine-learning tools such as support vector machines and even much simpler linear classifiers were able to achieve better or comparable performances in classification tasks, which were the most important applications of neural networks at that time. It was observed that the training of neural networks was often stuck at local minima, failing to converge to the true minimum point. Furthermore, it was generally believed that one hidden layer is enough for neural networks, as networks with more than one hidden layer are harder to train. Since then, research interests in neural networks started decaying.

The interest in neural networks was revived around 2006 as a few researchers demonstrated that a deep feedforward neural network was able to achieve outstanding classification accuracies with a proper unsupervised pre-training [8, 9]. Despite its success, the deep neural network was not fully recognized by the computer-vision and machine-learning community until astonishing results were achieved by the AlexNet, a deep convolutional neural network (CNN), in 2012 [10]. Since then, deep learning has emerged as the mainstream method in various tasks such as image recognition, audio recognition, etc.

1.2.2 SNN

SNNs did not receive much attention compared to the widely-used ANNs. Interests in SNNs mainly come from the neuroscience community [11, 12, 13]. Despite being less popular, it is generally believed that SNNs have more powerful computational capabilities compared to their ANN counterparts, thanks to the spatiotemporal patterns that are used to carry information in SNNs [14, 15, 16]. Even though SNNs are potentially more advanced, there are difficulties in harnessing the power of SNNs. The dynamics of an SNN is much more complicated compared to an ANN, making the purely analytical approach intractable. Furthermore, the event-triggered nature

of SNNs leads to an inefficient implementation on a conventional processor. This is one of the main reasons that SNNs are not as popular as ANNs in the artificial-intelligence community.

Similar to an ANN, an SNN also consists of neurons and synapses. In terms of the network topology and connections, SNNs and ANNs have much in common, as can be seen from Fig. 1.1. However, in an SNN, information is encoded on spikes instead of the real numbers used in an ANN. Even though the neural spikes observed in a biological neural network have a complex shape of excitatory postsynaptic potential, spikes in an SNN are often simplified to a short pulse or even the Dirac delta function for the ease of implementation and analysis. In addition, the dynamics of a spiking neuron can be much more complicated compared to that of an artificial neuron. Popular models for spiking neurons are the Hodgkin-Huxley model [17], the Izhikevich model [13], and the leaky integrate-and-fire model [18].

1.3 Need for Low-Power Neuromorphic Hardware

The development of hardware-based neural networks experienced similar phases that the software neural nework did. There was a period of time (late 1980s to early 1990s) in which many neuromorphic chips and hardware systems were introduced [19, 20, 21, 22]. Later on, after finding out that it is hard to improve the performance of neural networks, researchers in the hardware community also gradually steered their research direction toward the conventional computing based on von Neumann architecture as the CMOS technology kept advancing. Around 2006 when the breakthrough was made in the field of the deep learning, the research interests in hardware implementation of neural networks also revived. The possibilities of deploying neuromorphic hardware in real-life applications were explored after the scaling of transistors was slowed down.

Electronic computing devices have evolved for several decades, as shown in Fig.

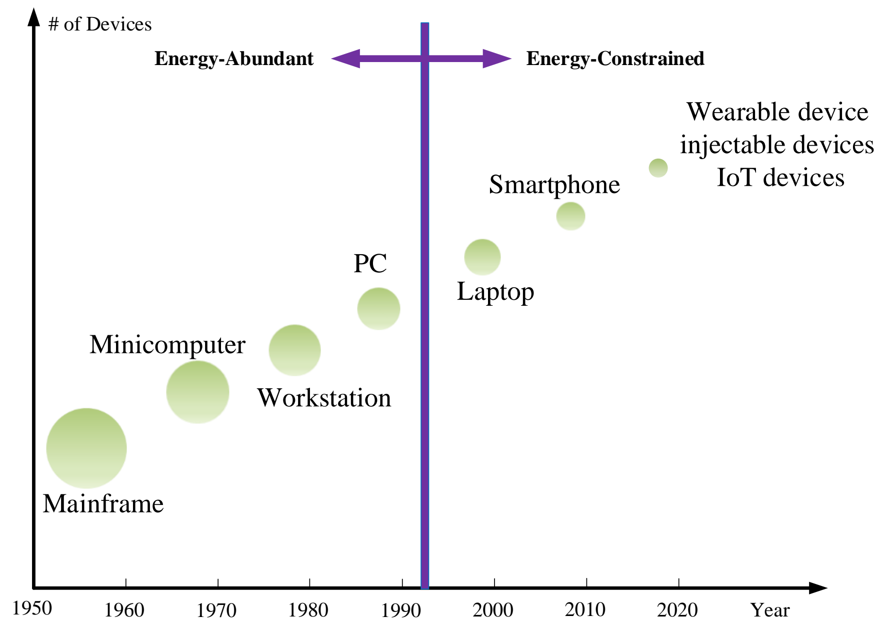


Figure 1.3: History of the development of computing devices

1.3. The sizes of the circles in the figure represent the relative sizes of the computing devices. Such an evolution is best described by Bell's law [23]. Two trends can be observed from Fig. 1.3. The first trend is that computing devices are becoming smaller and cheaper. Indeed, partially driven by Moore's law, the size and the price of the consumer electronics are decreasing continuously. In addition, many of the electronics in the recent generations are portable and, therefore, are powered by battery. As a consequence, energy efficiency has become more and more important for these modern computing devices. The second trend is that the data that the computing devices take are becoming less and less formatted. In the 1980s, work stations only took well-written computing scripts as input, yet nowadays there are many sensors (motion, temperature, pressure, etc.) in our smart phones and wearable devices that produce various forms of signals. Clearly, we are experiencing a transition from the conventional rule-based computing to the new data-driven computing.

With more and more low-power sensor devices and platforms being deployed in our everyday life, an enormous amount of data are collected continuously from these

ubiquitous sensors. One dilemma we often encounter is that despite the amount of data we gather, we still lack the capability to fully exploit the collected information. There is a strong need to provide these sensor platforms with built-in intelligence so that they can sense, organize, and utilize the data more wisely. Fortunately, deep learning has emerged as a powerful tool for solving this problem [9, 24, 25, 26, 27, 28, 29]. Despite its successes in small-scale tasks, a deep neural network can only be employed in a real-life application if hundreds of millions of synapses can be integrated in the system. Training of such a huge neural network usually takes weeks and excessive power consumption even when highly optimized hardware such as graphics processing units (GPUs) are employed and matrix solving are being largely parallelized [25]. In the near future, we will have more and more ultra-low-power sensor systems for health and environment monitoring [30, 31, 32], microrobots that chiefly rely on energy scavenging from the environment [33, 34, 35, 36], and over ten billion internet-of-things (IoT) devices [37]. For all of these applications where power consumption is of utmost importance, neither the power-hungry GPU nor sending raw data to the cloud for further analysis is a viable option. To tackle this difficulty, many efforts from both industry and academia have been made in order to develop low-power deep learning accelerators.

Over the past decade, an enormous amount of research effort has been made to build specialized neuromorphic computing hardware for real-life applications, while the development of the conventional von Neumann architecture-based computing approach has slowed down. In recent years, the research focus has gradually shifted from traditional rate-based ANNs, which were popular choices of hardware implementations in the 1990s, to SNNs. This trend is attributed to two unique advantages that SNNs have. The event-triggered nature of an SNN can lead to a very power-efficient computation. In addition, an SNN has better scalability because an address-event representation (AER) can conveniently interconnect sub-SNNs in a large network

[38, 39, 40]. For example, the TrueNorth from IBM [38] is a hardware spiking neural network that contains 1 million spiking neurons. It consists of 4096 cores and consumes merely 65 mW while running a real-time multi-object detection and classification task. Despite these advantages that an SNN provides, SNNs implemented on a general-purpose processor are not able to demonstrate superiority compared to ANNs owing to the lack of support of the event-based computation. Therefore, to better exploit the aforementioned advantages of an SNN, many specialized hardware systems have been built, such as the TrueNorth from IBM [38], CAVIAR in Europe [41], and neuromorphic chips from HRL [42].

1.4 Challenges

Building and utilizing specialized neuromorphic hardware is still in its early stage, especially for spike-based neural networks. There are many difficulties that need to be addressed before the hardware can become truly useful.

1.4.1 Challenges from Learning

The first challenge we are facing is how to properly train a spike-based neural network. It is the learning capability that empowers the neuromorphic system with the built-in intelligence. Unfortunately, in contrast to the ANN, which can be trained efficiently with a linearized model thanks to its relatively simple mathematic model, the complicated dynamics of an SNN impedes its learning.

Over the past decades, there were numerous efforts from both the artificial-intelligence community and the neuroscience community to develop learning algorithms for SNNs. Spike-timing-dependent plasticity (STDP), which was first observed in real biological experiments, was proposed as an empirically successful learning rule used for unsupervised learning [43, 44, 45, 46]. There are also various algorithms for supervised learning in SNNs, such as SpikeProp [47], ReSuMe [48], tempotron learn-

ing rule [49, 16, 14], and PSD rule [15, 50], yet they all have their own limitations, such as not being applicable to multilayer neural networks, restricting each neuron to fire once, etc. Therefore, in order to build spike-based intelligent computing devices, effective learning algorithms that are suitable for hardware implementation are needed.

1.4.2 Challenges from Memory

The second issue that needs to be solved is the demand for better memory in neuromorphic systems. Memory is the most essential building block for neural network hardware. It often consumes the most area and energy in a neuromorphic hardware. Therefore, it is desired to improve the energy efficiency and the density of the memory used in a neural network accelerator. One effective approach to lowering the power consumption of the chip is to reduce the supply voltage, as it has a quadratic effect on power consumption. Doing so, however, hurts the reliability of the memory significantly [51, 52, 53]. In addition to lowering the power consumption, the gigantic memory requirement forces the traditional memory technology to scale aggressively, which, in turn, also jeopardizes the reliability [54].

Clearly, both the requirements of lower power and higher density lead to less and less robust memory. unreliable memory has become the major threat to many low-power devices. Therefore, the reliability issue of the memory needs to be carefully addressed in this dissertation.

1.5 Dissertation Organization

The main objective of this dissertation is to advance low-power neuromorphic computing through innovations straddling across algorithm, architectures, and circuits. Chapter II - Chapter IV discuss the algorithm-architecture co-design that aims at building efficient learning algorithm and hardware architecture for next-generation

machine-learning accelerators. The challenge of lacking effective learning algorithm suitable for hardware implementation is tackled in these chapters. Chapter V studies architecture-circuit co-optimization in order to reduce power consumption and increase integration density of the neuromorphic system without sacrificing the reliability. This chapter strives to address the challenge from unreliable memory.

Chapter II introduces a learning algorithm that can be employed for multilayer spiking neural networks. The learning algorithm is formulated from the perspective of a circuit designer with the objective of providing an efficient hardware implementation. With the algorithm proposed in Chapter II, an efficient hardware architecture is proposed in Chapter III. The architecture is based on the event-triggered computational model, which is promising in saving energy in computing. Design methodologies for implementing the learning algorithm are also discussed.

Chapter IV presents accelerator designs for computing with conventional ANNs in order to leverage its advantage in high-precision computing. An efficient hardware architecture as well as adaptions in the learning algorithm are presented in order to improve the energy efficiency of the system.

Chapter V addresses the concern of the reliability for both volatile and non-volatile memory. Two methods are proposed in this chapter. The first method deals with the on-chip volatile static random-access memory (SRAM). The possibility of counteracting the variation through feedback compensation is explored. The second method strives to improve the error-correction algorithm used in a non-volatile memory, e.g. flash. Novel algorithms and architectures are presented to improve the energy efficiency of the error-correction circuit.

Chapter VI concludes this dissertation and provide outlooks for possible future work.

CHAPTER II

On-Line Learning for Hardware-Based Multilayer Spiking Neural Networks

2.1 Introduction

Many hardware implementations for spike-based neural networks have been demonstrated over the past few years with the objective of leveraging the energy efficiency and scalability of the SNNs, as discussed in Chapter I. However, many of these hardware do not have the capability to conduct on-chip learning. The goal of this chapter is to address the issue of lacking an efficient and effective learning algorithm that is suitable for SNN hardware.

One popular way to utilize the SNN hardware is training an ANN counterpart off-line using conventional learning algorithms, converting the well-trained ANN into an SNN, and then downloading the learned weights into the specialized hardware [55, 56, 57, 58]. However, such a training method fails to provide the on-line learning capability, which is an expected feature for many specialized neuromorphic hardware that target future smart IoT devices.

Over the past few decades, there have been many efforts from both the artificial-intelligence community and the neuroscience community to develop learning algorithms for SNNs. Spike-timing-dependent plasticity (STDP), which was first observed

in real biological experiments, was proposed as an empirically successful learning rule that could be used for unsupervised learning [43, 44, 45, 46]. In a typical STDP protocol, the synaptic weight is updated according to the relative order of spikes and the difference between the presynaptic and postsynaptic spike timings. Unsupervised learning is useful in discovering the underlying structure of the data, yet it is not as powerful as supervised learning in many real-life applications, at least at the current stage. There also exist various algorithms for supervised learning in SNNs, such as SpikeProp [47], ReSuMe [48], tempotron learning rule [49, 16, 14], and PSD rule [15, 50]. SpikeProp is one of the earliest proposed methods for supervised learning in SNNs and it is analogous to the backpropagation employed in conventional ANNs. The tempotron rule relies on a more biologically plausible mechanism. Both of these methods are based on the gradient descent method and they both limit each neuron in the SNN to fire only once, making them well-suited for classification applications where the output is a one-hot code. However, it is not convenient to employ such a learning algorithm for a softmax classifier or a universal function approximator. ReSuMe and the PSD rule both originate from the Widrow-Hoff rule. They have the advantage that they can learn precise spatiotemporal patterns of desired spikes, which makes them attractive for systems in which the information is mainly modulated on the timings of the spikes. However, how to apply these learning rules to a multilayer neural network is not obvious. This limitation impedes these algorithms from being employed in a deep neural network, which has achieved many astonishing results recently [9, 10, 8]. In addition, all of these learning rules aim at learning precise firing timings of neurons. It is, however, still debatable what the best way is to encode information using spikes. Therefore, learning exact spike timing might not be the optimum and the most efficient method for many real-life applications.

In this chapter, we take a different approach [59]. We do not attempt to learn the precise spike timings of neurons. Rather, learning rules that aim to achieve de-

sired firing densities are developed. Nevertheless, the spike timings of neurons are employed to estimate the gradient components in the learning process. Furthermore, how to propagate the errors from the output neurons to each synapse is studied. The proposed method leads to a learning process that is similar to a conventional backpropagation. The gradients estimated from spike timings are exploited to conduct gradient descent learning. By developing such a learning rule, we can take advantage of the power efficiency and scalability of a specialized SNN hardware. In addition, the developed learning algorithm stays relatively compatible with the learning in a conventional ANN. Therefore, many theories and techniques developed for the well-established ANN-based learning, such as momentum and mini-batch, can be applied to the proposed learning algorithm. Despite the similarities, many unique features associated with the proposed learning algorithm provide new opportunities. For example, in contrast to the layer-by-layer backpropagation in a conventional ANN, a direct backpropagation in SNN is possible by properly utilizing the spike timings of neurons, thus reducing the computational effort and improving the backpropagation speed. Another example is that the trained SNN can infer with a progressive precision, thus accelerating the inference process and reducing the energy consumption of the SNN hardware.

2.2 Estimating Gradients from Spike Timings

The following notations are used throughout this chapter.

1) A discrete-time stochastic process is denoted with an uppercase symbol and is indexed by n, for example, $X[n]$.

2) A deterministic discrete-time signal is denoted by a lowercase symbol and is indexed by n, for example, $x[n]$.

3) A column vector is denoted by a bold symbol such as $\mathbf{x}$, and its elements are denoted by x_i, where $i = 1, 2, \cdots$.

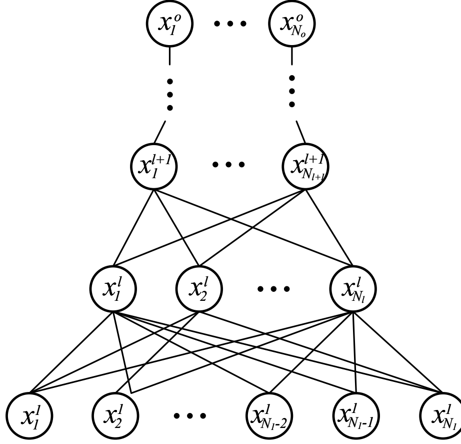


Figure 2.1: Illustration of a multilayer neural network. A neuron located at the lth layer is denoted as x_i^l, where i represents the index of that neuron.

4) The differentiation of a function f with respect to a vector is an element-wise operation. For example, $\partial f/\partial \mathbf{x} = [\partial f/\partial x_1, \partial f/\partial x_2, \cdots]^T$.

5) The expectation of a random variable is denoted by μ, and the arithmetic average of signal $x[n]$ over a finite duration is denoted by $\overline{x}$.

6) $Pr(\cdot)$ and $E[\cdot]$ are used to represent the probability and expectation, respectively.

Fig. 2.1 shows a multilayer SNN with the naming convention used in this chapter. In this figure, a neuron that is located at the lth layer is denoted as x_i^l, where i represents the index of that neuron. There are, in total, N_l such neurons in the lth layer. For a pair of neurons, a presynaptic neuron x_i^l and a postsynaptic neuron x_j^{l+1}, their output spike trains are denoted as

$$x_i^l[n] = \sum_m \delta\left[n - n_{i,m}^l\right] \tag{2.1}$$

$$x_j^{l+1}[n] = \sum_m \delta\left[n - n_{j,m}^{l+1}\right] \tag{2.2}$$

where $\delta[n]$ is the Kronecker delta function and $n_{i,m}^l$ and $n_{j,m}^{l+1}$ are spike timings for the

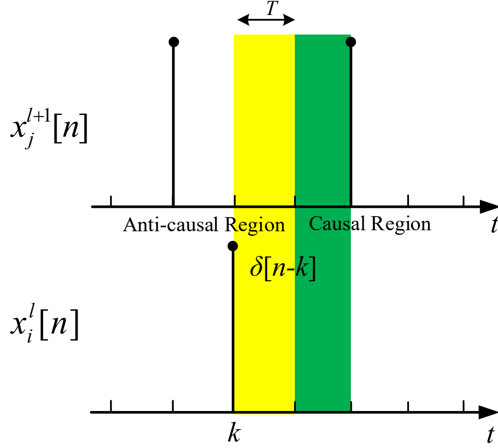


Figure 2.2: Illustration of the two regions divided by the spike timing of a presynaptic neuron. The causal and anti-causal regions are defined according to the causal relationship between presynaptic and postsynaptic spikes.

m^{th} spikes from neuron x_i^l and neuron x_j^{l+1}, respectively. As our primary interests are in training hardware-based SNNs, we restrict ourselves to discrete-time systems and the usage of a constant excitatory postsynaptic potential. This way of representing spikes is very popular in the hardware realization of SNNs, considering its ease of implementation and routing.

It was shown in [60] that a presynaptic spike partitions the time axis into two regions: a causal region and an anti-causal region, as shown in Fig. 2.2. Consequently, we can define a time sequence and its sample mean as

$$stdp_{ij}^l[n] = x_i^l[n-T]\left(1 - x_i^l[n-T-1]\right)\left(x_j^{l+1}[n] - x_j^{l+1}[n-1]\right) \tag{2.3}$$

$$\overline{stdp_{ij}^l} = \sum_{n=T+1}^{D_L} stdp_{ij}^l[n]/(D_L - T) \tag{2.4}$$

where T is the time delay associated with the neuron model used and D_L is the learning duration, which serves as a design parameter. The quantity $stdp_{ij}^l[n]$ measures the causality between the presynaptic and postsynaptic spikes, which is similar to the

quantity measured in an STDP protocol.

We consider a class of stochastic neuron model with the dynamics of

$$X_j^{l+1}[n] = H\left(\sum_{i=1}^{N_l} w_{ij}^l X_i^l[n-T] + S_j^{L+1}[n-T]\right) \tag{2.5}$$

For analysis purposes, we treat spikes as stochastic processes. The spike trains $x_i^l[n]$ and $x_j^{l+1}[n]$ shown in (2.1) and (2.2) are particular realizations of $X_i^l[n]$ and $X_j^{l+1}[n]$ in (2.5). $H(\cdot)$ is the Heaviside function. $S_j^{l+1}[n]$ is a random process that models the internal state of neuron x_j^{l+1}. We are interested in finding out how the mean firing rate of neuron x_j^{l+1} is related to the mean firing rate of its input neuron x_i^l. Before embarking on deriving this relationship, let us assume that the following two conditions hold.

C1) $X_i^l[n]$ and $X_k^l[n]$ are independent for $k = 1, 2, \cdots, N_l$, and $k \neq i$.

C2) $X_i^l[n]$ and $X_j^{l+1}[n]$ are strictly stationary processes, and $C_{X_i^l, X_j^{l+1}}(n, m) = 0$ for $n \neq m - T$, where $C_{X,Y}(\cdot)$ stands for the cross-covariance function.

We first show that under conditions C1) and C2), the mean firing rate of neuron x_j^{l+1}, i.e., μ_j^{l+1}, is a function of the mean firing rates of its input spikes, μ^l, or mathematically, $\mu_j^{l+1} = g(\boldsymbol{\mu}^{\mathbf{l}})$. In addition, we show that $g(\cdot)$ is differentiable with respect to $\boldsymbol{\mu}^{\mathbf{l}}$, and its mth derivative $\partial g^{(m)}/\partial \boldsymbol{\mu}^{\mathbf{l}} = \mathbf{0}$ for $m > 1$. Here, $\mathbf{0}$ denotes a zero vector in which all elements are zero. Note that the time index n is dropped for the sake of a cleaner notation because we consider strictly stationary processes.

The mean firing rate of neuron x_j^{l+1} can be expressed as

$$\begin{aligned}
\mu_j^{l+1} &= Pr(X_j^{l+1} = 1) \\
&= \sum_{b_1^l=0}^{1} \cdots \sum_{b_{N_l}^l=0}^{1} \left\{ Pr(X_j^{l+1} = 1 | X_1^l = b_1^l, \cdots, X_{N_l}^l = b_{N_l}^l) \prod_{i=1}^{N_l} \left[\mu_i^l (2b_i^l - 1) - b_i^l + 1\right] \right\} \\
&= g(\boldsymbol{\mu}^{\mathbf{l}})
\end{aligned} \tag{2.6}$$

where b_i^l is a binary auxiliary variable for the convenience of derivation. It can be shown that the first derivative of $g(\boldsymbol{\mu}^{\mathbf{l}})$ with respect to μ_i^l is

$$\frac{\partial g}{\partial \mu_i^l} = \sum_{b_1^l=0}^{1} \cdots \sum_{b_{N_l}^l=0}^{1} \Bigg[Pr(X_j^{l+1} = 1 | X_1^l = b_1^l, \cdots, X_{N_l}^l = b_{N_l}^l) \cdot \left(\prod_{k=1, k\neq i}^{N_l} \left[\mu_i^l (2b_k^l - 1) - b_k^l + 1 \right] \right) (2b_i^l - 1) \Bigg] \tag{2.7}$$

Clearly, $\partial g / \partial \mu_i^l$ is not a function of μ_i^l, which implies that $\partial g^{(m)} / \partial \boldsymbol{\mu}^{\mathbf{l}} = \mathbf{0}$ for $m > 1$.

According to the law of large numbers, we have

$$\begin{aligned} E\left[\overline{stdp_{ij}^l}\right] &= Pr(X_j^{l+1} = 1, X_i^l = 1, X_i^{l'} = 0) - Pr(X_j^{l+1} = 1, X_i^l = 0, X_i^{l'} = 1) \\ &= \left[Pr(X_j^{l+1} = 1 | X_i^l = 1) - Pr(X_j^{l+1} = 1 | X_i^l = 0) \right] \mu_i^l (1 - \mu_i^l) \end{aligned} \tag{2.8}$$

In (2.8), X_i^l denotes the random variable on which $X_j^{(l+1)}$ depends, whereas $X_i^{l'}$ denotes the random variable of which $X_j^{(l+1)}$ is independent. From (2.8), through expanding $g(\boldsymbol{\mu}^{\mathbf{l}})$ in a Taylor series and using the fact that $\partial g^{(m)} / \partial \boldsymbol{\mu}^{\mathbf{l}} = \mathbf{0}$ for $m > 1$, we have

$$\begin{aligned} \frac{E\left[\overline{stdp_{ij}^l}\right]}{\mu_i^l (1 - \mu_i^l)} &= Pr(X_j^{l+1} = 1 | X_i^l = 1) - Pr(X_j^{l+1} = 1 | X_i^l = 0) \\ &= g(\boldsymbol{\mu}^{\mathbf{l}}) + \frac{\partial g}{\partial \mu_i^l}(1 - \mu_i^l) - g(\boldsymbol{\mu}^{\mathbf{l}}) - \frac{\partial g}{\partial \mu_i^l}(-\mu_i^l) \\ &= \frac{\partial \mu_j^{l+1}}{\partial \mu_i^l} \end{aligned} \tag{2.9}$$

The derivation of (2.9) is based on assumption C1) and C2). We then examine the validity of (2.9) qualitatively when C1) and C2) do not hold rigorously. C1) assumes that the spike timings for different input neurons are independent. Even though the density of each neuron might be highly correlated, the spike timing of an individual neuron can be largely independent. The mild assumption that the

spike timing of each neuron is somewhat uncorrelated holds for most SNNs. C2) implies that $S_j^{l+1}[n]$ is also strictly stationary and that it should be independent of X_i^l and X_j^{l+1}. Rigorously, $S_j^{l+1}[n]$ depends on all $X_i^l[m]$ in which $m < n - T$ for any neuron model with a memory, such as the popular leaky integrate-and-fire model. In practice, however, the dependency of $S_j^{l+1}[n]$ on the firing history of a presynaptic neuron is significantly diluted by the firing histories of other independent presynaptic neurons as well as the modulus or noisy reset operations that are associated with the postsynaptic neuron. In addition, the dependency can be weakened to an acceptable level through proper noise injection. This arrangement is illustrated in Section 2.3. A natural extension of (2.9) is to define the time sequence $stdp_{ij}^l[n]$ in such a way that more samples can be included for each estimation. This approach is illustrated in (2.10). In the equation, WIN_{STDP} is a design parameter that is used to specify the window size of the summation. This method is inspired by the biological STDP, in which an exponential integration window is employed. The purpose of the parameter WIN_{STDP} is to include the effects of delayed perturbated outputs, which might be caused by the memory of $S_j^{l+1}[n]$.

$$\begin{aligned} stdp_{ij}^l[n] = {} & x_i^l[n-T]\left(1 - x_i^l[n-T-1]\right) \\ & \cdot \left(\sum_{m=1}^{WIN_{STDP}} x_j^{l+1}[n+m-1] - \sum_{m=1}^{WIN_{STDP}} x_j^{l+1}[n-m] \right) \end{aligned} \tag{2.10}$$

Intuitively, (2.9) indicates that $\frac{\partial \mu_j^{l+1}}{\partial \mu_i^l}$ can be estimated by observing how the postsynaptic neuron alters its statistical behavior in response to an input spike that serves as a small perturbation to the network. Even though perturbations from various presynaptic neurons might cause the same postsynaptic neuron to spike, contributions from each presynaptic neuron can be evaluated simultaneously as long as the spike timings of each of the presynaptic neurons are reasonably uncorrelated. For example, at any given time k, as shown in Fig. 2.2, neuron x_j^{l+1} has equal probability

to fire at both regions when the input spike from neuron x_i^l is absent. When the input spike is present, the spike from neuron x_j^{l+1} is more likely to occur at one side of k depending on whether the synapse is excitatory or inhibitory. The contributions of other input neurons appear to be noise, and they can be easily filtered out if they are not correlated. To further decorrelate the spike timings of each neuron in an SNN, a stochastic neuron can be employed. More conveniently, a technique called quantization noise injection, which was introduced in [60], can be utilized. Therefore, (2.9) can be readily employed in a large network, and individual gradients can be estimated simultaneously. This approach has the same spirit as simultaneous perturbation stochastic approximation (SPSA) [61].

Next, we assume that (2.11) can approximately describe the input-output relationship in the chosen neuron model, where $f_j^{l+1}(\cdot)$ is a differentiable function that depends on the dynamics of the spiking neuron model that is used. The actual form of $f_j^{l+1}(\cdot)$ is not important in our derivation because it serves as only an intermediate quantity that is substituted eventually. Conceptually, $f_j^{l+1}(\cdot)$ can be obtained, for example, through function fitting.

$$\mu_j^{l+1} \approx f_j^{l+1}\left(\sum_i w_{ij}^l \mu_i^l\right) \tag{2.11}$$

Then, with (2.9) and (2.11), we arrive at

$$\frac{\partial \mu_j^{l+1}}{\partial w_{ij}^l} = \mu_i^l f_j^{l+1\prime}\left(\sum_i w_{ij}^l \mu_i^l\right) = \frac{E\left[\overline{stdp_{ij}^l}\right]}{w_{ij}^l(1-\mu_i^l)} \tag{2.12}$$

Equation (2.12) resembles the STDP learning rule in the literature. However, in contrast to a conventional STDP rule, a denominator term is included. Mathematically speaking, including the weight gives at least the sign information. If negative weights are allowed, then it is necessary for a term to change the sign in (2.12), which

would otherwise induce a wrong direction for the gradient descent. In addition, the introduction of the weight denominator ensures an upper bound on w_{ij}^l, which serves a similar purpose as the weight-decay technique that is widely used in ANNs [62].

Equations (2.9) and (2.12) provide theoretical guidelines to estimate the gradients in an SNN in order to conduct gradient descent learning. In practice, we use $\overline{stdp_{ij}^l} / \left[\overline{x_i^l}(1-\overline{x_i^l})\right]$ and $\overline{stdp_{ij}^l} / \left[w_{ij}(1-\overline{x_i^l})\right]$ to approximate $\partial \mu_j^{l+1} / \partial \mu_i^l$ and $\partial \mu_j^{l+1} / \partial w_{ij}^l$, where $\overline{x_i^l} = \sum_{n=T+1}^{D_L} x_i^l[n] / (D_L - T)$

To feature gradient descent learning, we need to propagate errors at the output neurons back to each synapse in the neural network. This process can be achieved through a chain rule that is similar to that used in a conventional ANN, as shown in (2.13).

$$\frac{\partial \mu_k^o}{\partial w_{ij}^l} = \frac{\partial \mu_k^o}{\partial \mu_j^{l+1}} \cdot \frac{\partial \mu_j^{l+1}}{\partial w_{ij}^l} \tag{2.13}$$

In (2.13), the term $\partial \mu_j^{l+1} / \partial w_{ij}^l$ can be computed according to (2.12), whereas the term $\partial \mu_k^o / \partial \mu_j^{l+1}$ can be obtained by propagating the gradient layer by layer through

$$\frac{\partial \mu_k^o}{\partial \mu_j^{l+1}} = \sum_{i_o=k}^{k} \sum_{i_{o-1}=1}^{N_{o-2}} \sum_{i_{l+2}=1}^{N_{l+2}} \sum_{i_{l+1}=j}^{j} \prod_{p=l+1}^{o-1} \frac{\partial \mu_{i_{p+1}}^{p+1}}{\partial \mu_{i_p}^{p}}. \tag{2.14}$$

This process is similar to the backpropagation used in a conventional ANN.

Alternatively, a direct propagation method shown in (2.15) is proposed to estimate the gradient.

$$\frac{\partial \mu_k^o}{\partial \mu_j^{l+1}} = \frac{E\left[\overline{cstdp_{jk}^{l+1}}\right]}{\mu_j^{l+1}(1-\mu_j^{l+1})} \tag{2.15}$$

In (2.15), $cstdp_{jk}^{l+1}[n]$ is a quantity that is similar to the one defined in (2.3). The only difference is that $\overline{cstdp_{jk}^{l+1}}$ measures the relationship between the $(l+1)^{th}$ layer and the output layer. The delay across multiple layers must be considered in this case. Here, (2.15) is an extension of (2.9) in the sense that instead of using perturbation to estimate the gradient of the output of a neuron with respect to its input, we estimate

the gradient across a network of neurons by observing how the input spike affects the output firing probability.

To verify (2.12)-(2.15), we conduct simulations on a two-hidden-layer neural network. The operations of the neural network largely follow the conventions used in TrueNorth [38] because it is the most recently developed, powerful general-purpose neuromorphic hardware. In [38], spikes from neurons can occur only synchronously with a time unit called a tick. This setting guarantees a one-to-one mapping between software and hardware at a tick level, albeit the internal evaluations of the neuron states are asynchronous to save energy. In the remainder of this chapter, a tick is used as the minimum temporal resolution as well as the unit for time-related quantities, e.g., WIN_{STDP}.

The configuration of the neural network is 80-30-100-1, where each number represents the number of neurons at each layer, from the input layer to the output layer. A modified integrate-and-fire neuron model, shown in (2.16) and (2.17), is used.

$$x_j^{l+1}[n] = \begin{cases} 0, & V_j^{l+1}[n] < th_j^{l+1} \\ 1, & V_j^{l+1}[n] \geq th_j^{l+1} \end{cases} \tag{2.16}$$

$$V_j^{l+1}[n] = \max\left(0, V_j^{l+1}[n-1] + \sum_i w_{ij}^l x_i^l[n-1] - L_j^{l+1} - x_j^{l+1}[n-1] \cdot th_j^{l+1}\right) \tag{2.17}$$

In the model, $x_j^{l+1}[n]$ and $V_j^{l+1}[n]$ are the output and membrane potential of a neuron x_j^{l+1} at tick n, th_j^{l+1} is the threshold to fire, and L_j^{l+1} represents the leakage. It has been shown in [60] that such a neuron model behaves similarly to a first-order $\Sigma - \Delta$ modulator, and the quantization noise associated with this model is helpful in achieving less correlated spike timings. In other words, we can randomize the spike timing of each neuron without explicitly using random number generators. In addition, this modified model is one of the models employed in the TrueNorth

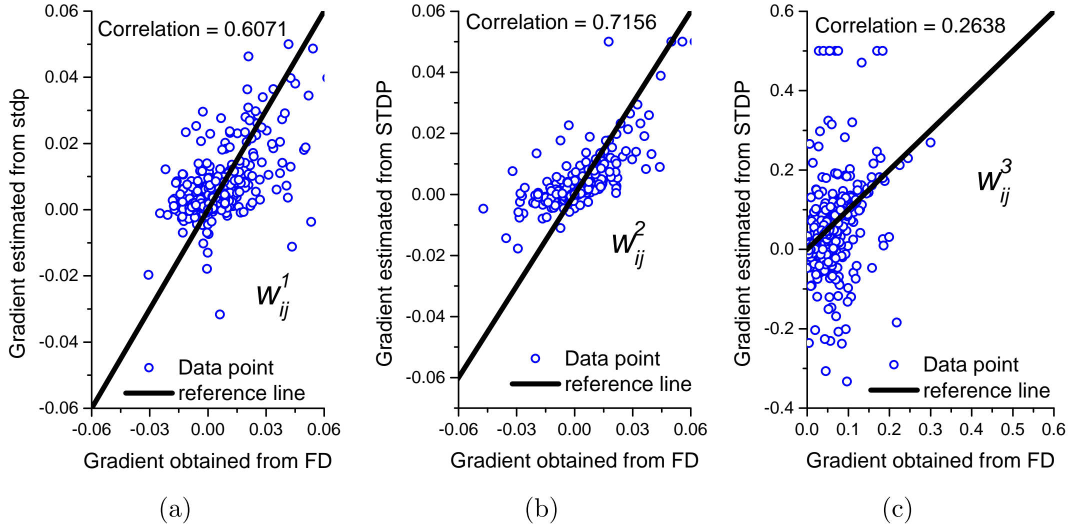


Figure 2.3: Comparison of the gradients obtained from numerical simulations with the gradients obtained from STDP for (a) the first-layer synapses w_{ij}^1, (b) the second-layer synapses w_{ij}^2, and (c) the third-layer synapses w_{ij}^3

chips [63]. Therefore, we utilize this model in this chapter unless otherwise stated. Nevertheless, our proposed algorithm is not restricted to this modified model. For example, it can also be applied to a conventional leaky integrate-and-fire (LIF) model if noise is properly injected. This approach is demonstrated in Section 2.3. In our simulations, input neurons in the network are injected with excitatory currents at every tick. The injected currents are randomly chosen at the beginning of learning and are fixed throughout the learning. More information on the input encoding is detailed in Section 2.3.

Fig. 2.3 shows the scatter charts that compare the gradients estimated from the spike timing and gradients calculated numerically. Ten sets of experiments are conducted and 100 weights from each layer are randomly chosen for each set of experiments. A thousand data points, in total, are collected in the figure for each layer. The numerical results shown in Fig. 2.3 are obtained with the finite-difference (FD) method. In other words, a small perturbation is applied to the weight and the gradient is obtained by dividing the change at the output by the amount of perturbation that

is applied. Owing to the complicated dynamics of the SNN and the limited computational resources, the gradient obtained from the FD method is not the true gradient but is instead a noisy version of the true gradient. These gradients asymptotically approach the true gradient as the number of evaluation ticks increases. Nevertheless, a comparison with such noisy gradients can provide some useful insights into how well the spike timing can be used for estimating gradients. As shown in (2.12), when the weight is small, the quantization noise in the density of the spike might induce a large estimated gradient variation. Therefore, a limiting operation is needed to limit the maximum and minimum gradients obtained from the spike timing information. Detailed information for this clamping is shown in Table 2.1.

The estimated gradients and the gradients obtained numerically match well in Fig. 2.3, which demonstrates the effectiveness of the proposed algorithm. It is observed in Fig. 2.3(c) that the correlation for w_{ij}^3 is comparatively low. It is found in the simulations that few negative outliers in Fig. 2.3(c) are responsible for this low correlation. The reason is that the clamping values for all layers in Table 2.1 are chosen symmetrically for convenience, yet the gradients associated with the last-layer weight are actually non-negative. In practice, this non-negative characteristic can be exploited during learning.

Table 2.1: Information of limiting operations used to obtain data in Fig. 2.3

Layer index	Maximum/minimum gradient allowed	# of outliers
1	± 0.05	1
2	± 0.05	4
3	± 0.5	16

To evaluate the importance of the parameter WIN_{STDP}, which is used in (2.10), simulations are conducted to examine the results obtained with different window sizes. Fig. 2.4(a) shows the correlations obtained between the estimated gradients and the numerical gradients. As shown in the figure, estimating with different window

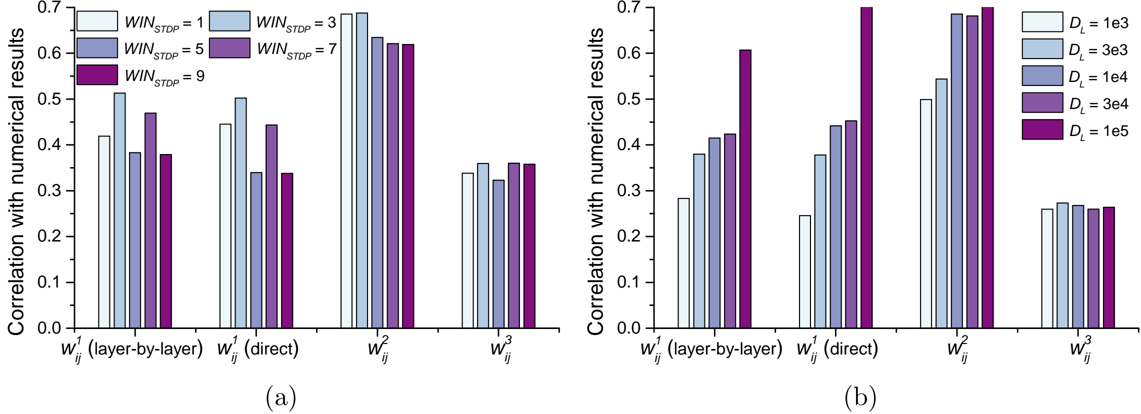


Figure 2.4: Correlations between the estimated gradients and the gradients obtained from the FD numerical method for (a) different window size WIN_{STDP} and (b) different evaluation duration D_L. The results obtained for all three layers of synaptic weights (w^1_{ij}, w^2_{ij}, and w^3_{ij}) are compared. Two different backpropagation methods (layer-by-layer and direct) are also compared.

sizes results in similar accuracies. Preliminary numerical studies on the effect of the window size on learning also show that changing the window size does not yield a noticeable difference. Therefore, in this work, we focus on the case with a window size of 1.

Another set of simulations is conducted to study how the evaluation duration affects the accuracy of the estimated gradients. As shown in Fig. 2.4(b), a general trend is that the longer the evaluation duration, the more accurate the estimated gradients. This relationship is consistent with other stochastic approximation methods because any possible unbiased noise can be filtered out through averaging.

With gradients estimated through spike timings, a stochastic gradient descent method can be readily employed for learning. Following the convention in a standard backpropagation algorithm, we define an error function as

$$E = \frac{1}{2}\sum_{k=1}^{N_o}(e^o_k)^2 \tag{2.18}$$

where $e^o_k = \overline{x^o_k} - t^o_k$ is the error at each output neuron. Here, t^o_k is the target mean

firing rate of neuron x_k^o.

The weight update Δw_{ij}^l can be calculated as

$$\Delta w_{ij}^l = -\alpha \cdot \sum_{k=1}^{N_o} e_k^o \cdot \frac{\partial \mu_k^o}{\partial w_{ij}^l} \tag{2.19}$$

where α is the learning rate and the term $\partial \mu_k^o / \partial w_{ij}^l$ can be obtained from (2.13). Updating weights according to (2.19) leads to a reduction in the error function toward the gradient-decent direction.

It is worth noting that the proposed learning algorithm can be readily extended to other popular learning schemes, such as unsupervised learning and reinforcement learning, when the target is to minimize some forms of cost functions, even though this chapter focuses mainly on supervised learning, which has achieved great success in real-life applications.

2.3 Simulation Results

In Section 2.2, we demonstrate that spike timing information can be readily employed for estimating the gradient components needed in a gradient descent algorithm. In this section, we apply the proposed learning algorithm to two neural networks. The sizes of the neural networks are chosen according to two examples demonstrated in [2] in such a way that a direct comparison can be made. The MNIST benchmark task is employed to examine the proposed algorithm. The MNIST dataset contains, in total, 70000 28×28 images of handwritten digits. The number of images in the training and testing sets are 60000 and 10000, respectively. The dataset is categorized into 10 classes, which correspond to ten integers (0 - 9), and each image has an associated label. Unless otherwise stated, for all of the training examples in this section, we use a training set that contains the first 500 images from the standard MNIST training set to accelerate the simulation. For testing, we use all of the 10000 images

from the standard MNIST testing set. It should be noted that the results obtained with such an experiment setting are only for verifying the proposed techniques and exploring the design spaces. Benchmark performance obtained with the full training set is reported in Section 2.3.3.

To feed the double-precision real values, which are used to encode the grayscale images, into the SNN, proper encoding mechanisms are needed. We use a pulse-density modulation scheme, which is a rate-based encoding method. Real values from the images in the MNIST dataset are injected into the input-layer neurons as incremental membrane potentials at every tick. Combined with the modified LIF model, this encoding scheme behaves similarly to a $\Sigma - \Delta$ modulator, which is capable of converting high-resolution data to low-resolution bit streams through pulse density modulation. The firing rates of the input-layer neurons are then proportional to the intensities of the corresponding pixels. Such an encoding method leads to a simple implementation in hardware while achieving the desired rate encoding.

For both neural networks, 10 output neurons, which correspond to 10 digits, are used. The target of learning is that when a digit is applied to the neural network, the output neuron that corresponds to the correct digit should fire with a high density of μ_H, whereas all of the other neurons fire with a lower density of μ_L. The firing density is measured through $\overline{x_k^o}$. To test the trained neural network, a digit is presented to the network. After an inference duration of D_I, the output neuron with the highest firing density $\overline{x_k^o}$ is chosen as the winning neuron, and its corresponding digit is chosen as the classification result. All the results presented in this section are obtained from 10 independent runs. Error bars that correspond to the 95% confidence interval are plotted together with the simulation results.

2.3.1 One-Hidden-Layer Neural Network

As most useful feedforward neural networks have at least one hidden layer, the first example that we consider is a one-hidden-layer neural network with 784 input neurons, 300 hidden-layer neurons, and 10 output neurons. Nevertheless, neural networks with only two layers are also useful in some cases. For example, a two-layer neural network with a special input encoding similar to the radial basis function has been demonstrated in [60]. Because learning in a two-layer neural network is essentially a sub-problem of learning in a multilayer (more than two) network, we do not study them in this chapter separately. Nevertheless, most conclusions and techniques developed in this section can be readily applied to two-layer neural networks as well.

As demonstrated in [60], the gradients estimated from STDP started saturating and diverging from the actual gradients as the density of spike trains reaches a certain limit. This saturation occurs because it is difficult to tell whether a postsynaptic spike is a causal spike or an anti-causal spike when the presynaptic spike train is too dense. To tackle this issue, it was suggested in [60] that a clock that is fast enough to avoid dense spike trains should be used. This approach is similar to avoiding the hidden unit in a conventional ANN to be driven close to 1 or 0, which would otherwise lead to a significantly slowed learning process. Despite its effectiveness, this method of manually adjusting weights or the clock frequency is inconvenient. In this work, we propose to leverage a biologically inspired refractoriness to achieve the desired sparsity. More specifically, each neuron has a refractory period after firing. During the refractory period, it is not allowed to fire again. By utilizing this technique, dense spike trains can be avoided. One potential drawback with a fixed refractory period is that all neurons that are saturated are highly correlated in their spike timings. To tackle this problem, a random refractory period technique is proposed. In a discrete-

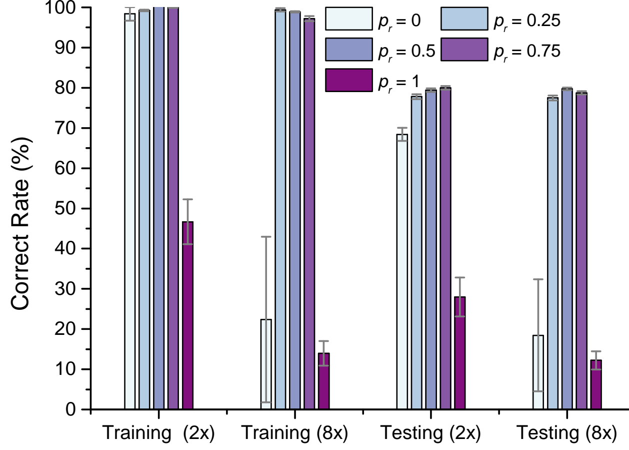


Figure 2.5: Comparison of the training and testing correct rates achieved with different levels of refractoriness and different initial weights. The refractory mechanism is helpful in avoiding dense spike trains, which helps improve the learning results.

time implementation, it is convenient to implement according to (2.20).

$$x_j^{l+1}[n] = \begin{cases} 0, & V_j^{l+1}[n] < th_j^{l+1} \\ 1, & V_j^{l+1}[n] \geq th_j^{l+1} \ \& \ x_j^{l+1}[n-1] = 0 \\ 1 - R & V_j^{l+1}[n] \geq th_j^{l+1} \ \& \ x_j^{l+1}[n-1] = 1 \end{cases} \tag{2.20}$$

where R is a random variable with a Bernoulli distribution $B[1, p_r]$. Here, p_r is a design parameter that is used for controlling the sparsity. A larger p_r can lead to sparser spike trains.

Fig. 2.5 compares the learning results achieved with different initial weights and p_r. Two sets of initial weights are employed. One set of weights is initialized uniformly from the interval $[0, 2]$, i.e., $w_{ij}^l \sim U[0, 2]$, where $U[0, 2]$ stands for a uniform distribution between 0 and 2. The results obtained with these initial weights are labeled with "2×" in the figure. Another set of weights is initialized such that $w_{ij}^l \sim U[0, 8]$. The results obtained with these initial weights are labeled with "8×" in the figure. For small initial weights ($w_{ij}^l \sim U[0, 2]$), a reasonable learning result can be achieved even for the case in which $p_r = 0$ because saturation has been avoided through the

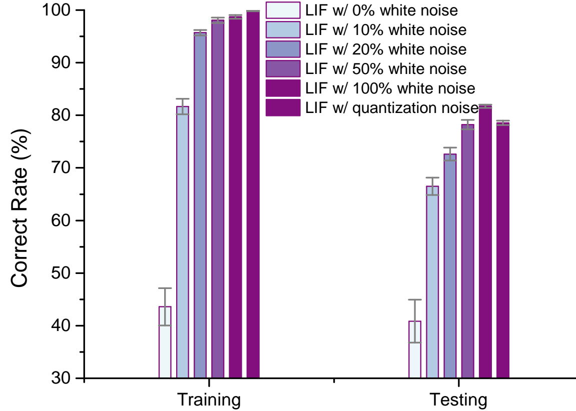


Figure 2.6: Comparison of the training and testing correct rates achieved with the LIF neuron model and the modified LIF model. The results obtained with the conventional LIF model with white noise residue injection are labeled as "LIF w/ white noise", whereas the results obtained with the modified LIF model is labeled as "LIF w/ quantization noise."

proper choice of small initial weights. This circumstance corresponds to the case in which proper initial weights are chosen to avoid the hidden layer unit being driven close to 0 or 1 when training a conventional ANN. When the initial weights are large ($w_{ij}^l \sim U[0, 8]$), however, the learning performance is significantly deteriorated for the $p_r = 0$ case because of the aforementioned detrimental effect of saturated spike trains. It is noted that learning is not successful for the case $p_r = 1$ regardless of the selection of initial weights because neurons that are saturated always have high correlations in spike timings. Owing to the proposed stochastic refractory period technique, good learning results are achieved when a proper p_r is employed.

To study the effectiveness of the proposed learning algorithm applied to a conventional integrate-and-fire neuron model, simulations are conducted for different levels of noise injection, as shown in Fig. 2.6. Noise is injected into the neuron model as noisy residue. In other words, a random residue is added to the membrane voltage after each spike. The injected noise is uniformly distributed with the range from zero to a percentage of the threshold value of that neuron. For example, the 50% white noise in Fig. 2.6 means that the noise injected into the neuron obeys a distribution,

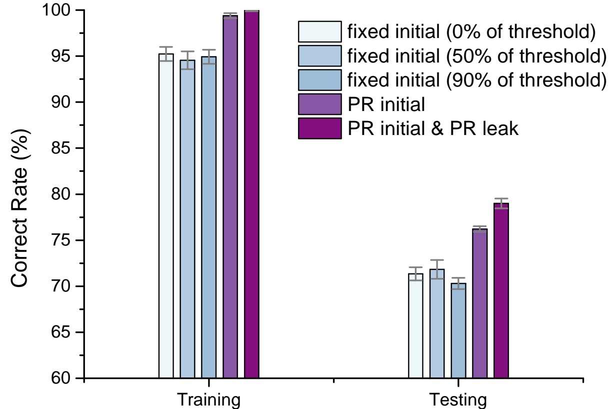


Figure 2.7: Comparison of the training and testing correct rates achieved with different initial conditions. The case with pseudo-random initial membrane voltages outperforms the cases with fixed initial membrane voltages. A pseudo-random leakage technique is also employed to further improve the learning performance.

$U[0, 0.5 \times th_i^l]$. The results obtained with the modified integrate-and-fire model that are described in (2.16) and (2.17) are also shown for comparison. The corresponding results are labeled as "LIF w/ quantization noiss". As the amount of injected noise increases, the learning is more effective. This result is expected because the proposed algorithm relies on the assumption that the spike timings of unconnected neurons should stay relatively uncorrelated. The conventional LIF model with noise injection can achieve a reasonably low correlation, yet random number generators are required for this purpose. On the other hand, the modified LIF model can decorrelate spike timings without explicitly injecting noise.

Another design consideration in our proposed learning algorithm is the initial condition of the neuron. In many applications, we need to reset neurons to certain states for each new input. Therefore, a proper initial condition needs to be set up. We propose to use a pseudo-random initial condition such that the initial membrane voltage of a neuron obeys a uniform distribution, e.g., $x_i^l[0] \sim U[0, th_i^l]$. The reason for choosing such an initial condition is that the membrane voltages of an SNN in a steady state approximately follow a uniform distribution. Therefore, a warm start can be achieved by setting the initial condition as a uniformly distributed random variable.

The results obtained with such a pseudo-random initial condition are compared with the fixed initial conditions in Fig. 2.7. As shown in the figure, even though any initial condition can feature effective learning, the proposed pseudo-random initial condition achieves the best performance. The main reason that the random initial condition outperforms others is that such an initial condition helps to achieve lower correlations among the input spikes. For the MNIST dataset, many pixels that correspond to strokes have values equal to one. This circumstance leads to highly correlated input spikes even when the modified LIF neuron model is used. By setting the initial condition differently, the correlations can be somewhat lowered.

With the same spirit, a pseudo-random leakage is also added at the input layer to further decorrelate the spike timings caused by the saturated intensities. The leakage for each neuron is assigned randomly beforehand and is fixed for the whole learning process. At each tick, the leakage is subtracted from the membrane voltage according to the neuron dynamics shown in (2.17). From another perspective, the pseudo-random leakage is helpful in breaking the possible symmetry that exists in the input data. Many input pixels from the MNIST dataset have the value of one. Through introducing the random leakage, we can break this symmetry in the data. The symmetry-breaking technique has been widely used by many machine learning researchers for weight initialization [62] and asymmetric connections in convolutional neural networks (CNNs) [2]. The results obtained with this technique are also compared in Fig. 2.7. The advantage of pseudo-random initial conditions and leakage for neurons is that no pseudo-random/true-random number generators are actually needed in the hardware implementation. The values can be conveniently stored in an on-chip static random access memory (SRAM) array or can be hardcoded in the logic.

In Section 2.2, it is shown that a longer learning duration yields more accurate estimated gradients. Therefore, it is expected that the learning performance can be

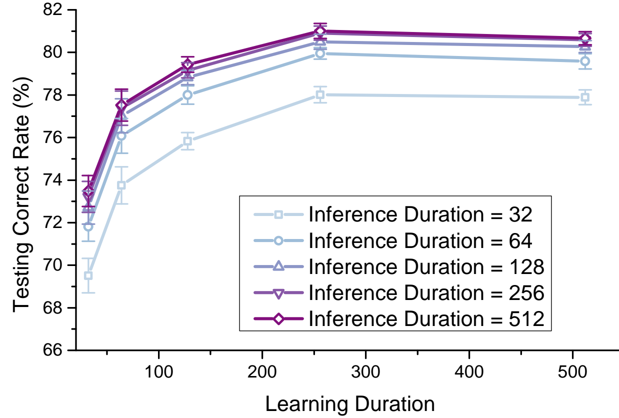


Figure 2.8: Comparison of the testing correct rates achieved with different learning and inference durations. The longer the learning or inference duration is, the higher the correct rate.

improved through lengthening the learning duration. To investigate the effect of the learning duration on the learning performance, simulations are conducted, and the obtained results are compared in Fig. 2.8. In the figure, five different learning durations are used: 32, 64, 128, 256 and 512. Five different inference durations are also used to evaluate the learned weights. A general trend shown in the figure is that increasing either the learning or inference duration helps in improving the recognition accuracy. For both the learning and inference duration, saturations occur at approximately 256, beyond which the improvement is marginal. Despite the fact that the best learning results are achieved when the learning duration is long, learning with a short duration also yields impressive results. This finding arises because stochastic gradient descent learning is quite robust against noise as long as it is not biased. Furthermore, it has been demonstrated recently that a noisy gradient is actually beneficial in learning, especially for a very deep neural network [64]. Therefore, a recommendation is to utilize a small learning duration at the beginning of the learning to speed up the learning process as well as to reduce the power consumption. The learning duration should be gradually lengthened to obtain more and more accurate gradients.

2.3.2 Two-Hidden-Layer Neural Network

The second example is a two-hidden-layer neural network with 784 neurons in the input layer, 300 neurons in the first hidden layer, 100 neurons in the second hidden layer, and 10 neurons in the output layer.

Because many conclusions that we draw for the one-hidden layer neural network also apply to the two-hidden-layer neural network, we focus mainly on investigating how different methods of propagating the errors affect the learning performances. Simulations are conducted for the two different backpropagation methods discussed in Section 2.2: the standard layer-by-layer backpropagation and the direct backpropagation. As shown in Fig. 2.9, similar performances are achieved by two backpropagation methods, which agrees with the results shown in Fig. 2.4(a) and Fig. 2.4(b). For comparison purposes, we also plot the results obtained with the one-hidden-layer network in Fig. 2.9. The recognition rates achieved with the two-hidden-layer network are higher when a learning duration of a moderate length is used (specifically, $D_L \geq 256$ in the figure). In addition, the two-hidden-layer neural network requires a longer learning duration to achieve a satisfactory result compared to its one-hidden-layer counterpart. This finding is consistent with the observation in ANNs that a deeper network tends to yield better results, yet it is harder and slower to train.

Even though the conventional layer-by-layer backpropagation can always be used along with our proposed algorithm, the unique direct backpropagation method can be helpful when the number of output-layer neurons is much smaller than the number of hidden-layer neurons, thereby providing more design freedom. For the lth layer in the network, $N_l N_{l+1}$ multiply-accumulate (MAC) operations are needed for a layer-by-layer backpropagation, whereas only $N_l N_o$ MAC operations are needed for a direct propagation. Significant savings in the number of MAC operations can be achieved when $N_o \ll N_l$. We do pay the price of spending more memory to store the STDP information across multiple layers. Therefore, we trade more memory spaces for

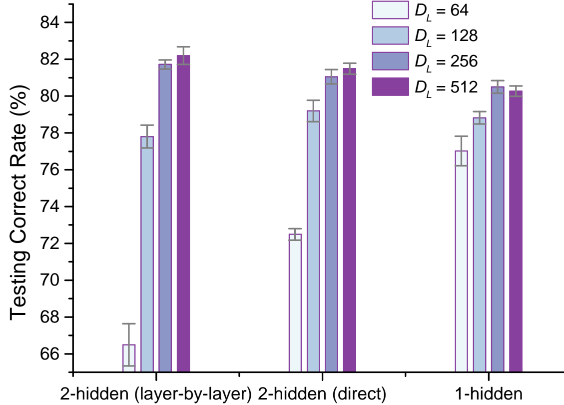


Figure 2.9: Comparison of the testing correct rates achieved with the two different backpropagation schemes. The two methods achieve similar performances. The two-hidden-layer neural network can yield better performance, but it requires a longer learning duration.

fewer computations. The memory requirement for storing $\overline{cstdp^l_{jk}}$ is $N_o \cdot \sum_{i=1}^{o-1} N_i$. Fortunately, this memory requirement does not scale as badly as the synaptic weight memory, which is on the order of $O(N^2)$, where N is the average number of neurons for one layer. For neural networks that are employed in most applications, the output layer has far fewer neurons compared to the preceding layers. Indeed, a function of a deep neural network is to extract useful information from a high-dimensional input, layer by layer. Therefore, the number of output neurons in a typical neural network is on the order of $O(1)$. Consequently, the memory requirement for this type of error backpropagation is approximately on the order of $O(N)$.

2.3.3 MNIST Benchmark

To demonstrate the effectiveness of the proposed learning algorithm, the standard MNIST benchmark is employed. Here, 60000 training data are used for training a one-hidden-layer neural network and a two-hidden-layer neural network. The trained networks are examined with the testing set, which includes 10000 digits. No pre-processing technique is used for a fair comparison. The obtained testing results for these two networks are compared with the results in the literature in Table 2.2. Clas-

Table 2.2: Comparison of the classification accuracies for the MNIST benchmark task.

Ref.	Type	Configuration	Learning algorithm	Recognition rate
[44]	SNN	784 input neurons + 6400 excitatory neurons + 6400 inhibitory neurons + readout circuit	Unsupervised, STDP	95%
[43]	SNN	784 input neurons + 300 output neurons + readout circuit	Unsupervised, STDP	93.5%
[58]	SDBN	784-500-500-10	ANN to SNN mapping	94.1%
[14]	SCNN	N/A	Tempotron	91.3%
[56]	SNN	784-1200-1200-10	ANN to SNN mapping	98.6%
	SCNN	28x28-12c5-2s-64c5-2s-10o	ANN to SNN mapping	99.1%
[57]	SDBN	484-256-10	ANN to SNN mapping	89%
[65]	SDBN	784-500-40	Contrastive divergence	91.9%
[66]	SNN	784 input + 10 output neurons each with 200 dendrites and 5000 synapses	Morphological learning	90.26%
[2]	ANN	784-300-10	Stochastic gradient descent	95.3%
		784-300-100-10	Stochastic gradient descent	96.95%
This work	SNN	784-300-10	weight-dependent STDP	97.2%
		784-300-100-10	weight-dependent STDP	97.8%

sification accuracies of 97.2% and 97.8% are achieved by the two neural networks, respectively. The proposed learning algorithm can achieve better classification correct rates compared to ANNs with the same configurations that are trained with sophisticated algorithms. Compared to the state-of-the-art result 98.6% in [56], our result is only slightly worse, especially considering that the size of our neural network is nine times smaller than the network used in [56] in terms of the number of synapses. Moreover, different from the ANN-to-SNN conversion method employed in [56], our proposed learning algorithm can conduct on-line learning directly on hardware SNNs, which is an expected feature for many energy-stringent applications.

In the table, unsupervised learning in [44] and [43] and the contrast divergence learning in [57] are similar to clustering. Other decision logics in addition to the neural network are needed to perform the classification. Moreover, it is not obvious how these learning algorithms can be used to train a universal function approximator that is needed in many applications, e.g., reinforcement learning.

2.3.4 Inference with a Progressive Precision

The results in the previous sections are obtained with the inference methods that are mapped directly from those used in a conventional ANN, for simplicity. In other words, we wait until the output of the neural network converges to the steady-state result and then we read out the results. An SNN, however, provides new opportunities for more rapid estimation of the results. For example, if we train the neural network such that the output neuron that corresponds to the correct digit fires with a firing density of μ_H and other output neurons fire with a density of μ_L. Then, we have a noise margin of $\mu_H - \mu_L$ such that a correct inference can still be achieved as long as the noise or any disturbance is less than this margin. Similar to the signal outputted by a $\Sigma - \Delta$ modulator, output signals from neurons are buried in high-frequency quantization noise. Counting the number of spikes is essentially filtering the

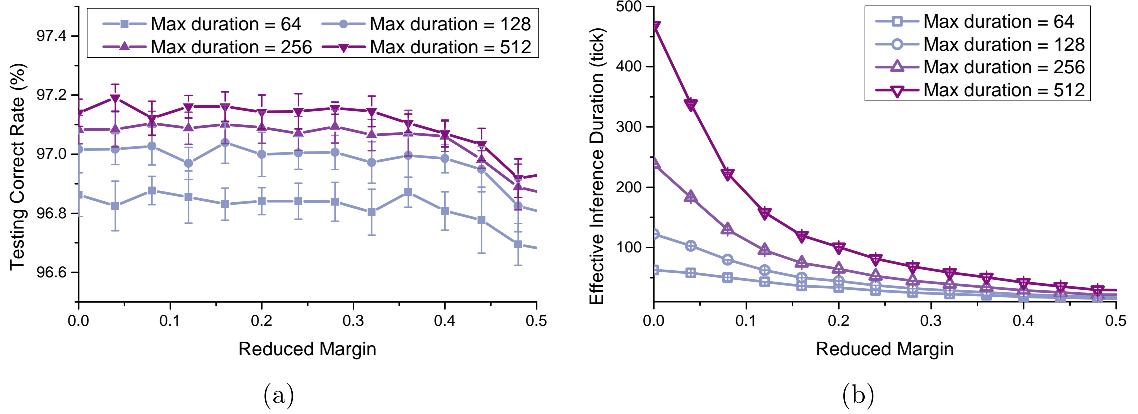


Figure 2.10: Comparison of a) the recognition rate and b) the effective inference durations needed for different levels of reduced margin. The results are obtained with the one-hidden-layer neural network.

high-frequency noise. A longer inference duration can lead to less quantization noise and, consequently, a more reliable result. This finding is similar to the well-known progressive precision in the stochastic computation [67]. When the image presented is easy to recognize, the neural network is able to produce the answer confidently. Therefore, we do not have to wait until the noise is fully removed. This results in a rapid inference. On the contrary, when the image is hard to classify, then we have to wait longer to obtain the answer.

Fig. 2.10(a) and Fig. 2.10(b) show the testing correct rate and effective inference durations that are needed to complete one classification. A trained one-hidden-layer neural network is used for illustration. At each tick, the density outputted from each of the output neurons is computed. If the density of one neuron is larger than $\mu_H - M/2$, and the densities from all of the other neurons are less than $\mu_L + M/2$, then the inference is considered to be completed and the output neuron with the largest spike density is chosen as the answer, where M is the reduced margin. Otherwise, the inference continues until a maximum allowed inference duration is reached. The effective inference duration in the figure is obtained by averaging the inference durations in 10000 testing cases. As shown in the figure, as the margin reduces, the length of

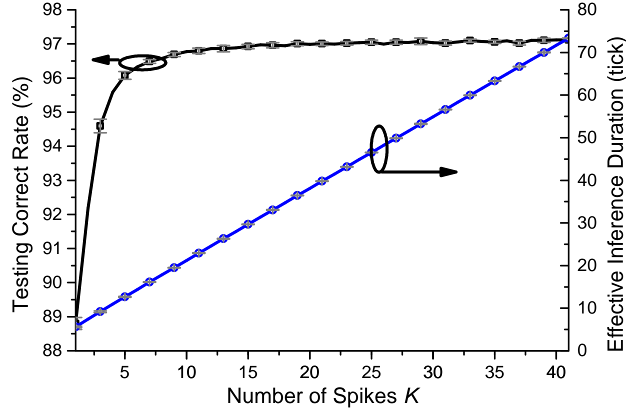


Figure 2.11: The recognition accuracy and the corresponding effective inference duration needed when the first-to-spike-K-spikes readout is employed. The results are obtained with the one-hidden-layer neural network.

the effective inference duration is significantly shortened. The classification accuracy, however, does not start dropping until the reduced margin reaches 0.3, where the quantization noise starts having a noticeable effect on the testing results. It should be noted that there are some testing cases where the neural network is not able to give a confident answer regardless of how long the inference duration is. For these testing cases, the results are always produced when the maximum allowed inference duration is reached. Therefore, a trend is that the longer the maximum inference duration is, the longer the effective inference duration.

Another way to demonstrate the inference with a progressive precision is shown in Fig. 2.11. The output neuron that first generates K spikes is determined to be the winning neuron, and the corresponding digit is read out as the inferred result. The recognition rates on the testing-set images are shown in the figure for different values of K. The number of ticks needed before an inference can be obtained is also recorded. As shown in the figure, an accuracy as high as 89% can be achieved with an effective inference duration of only 5.6 ticks. The accuracy enhances rapidly when K increases. The growth in accuracy starts saturating when K reaches 10 in the figure.

2.4 Chapter Summary

In this chapter, we formulate an on-line learning algorithm for multilayer spiking neural networks. The proposed learning method can estimate the gradient components with the help of the spike timings in an SNN. The readily available gradient information is then exploited in stochastic gradient descent learning. How the error can be propagated back to each layer is studied. A direct backpropagation is proposed in addition to the conventional layer-by-layer approach. The newly proposed algorithm is employed in two neural networks for the purposes of demonstration. To feature more effective learning, techniques such as random refractory period and pseudo-random initial conditions are proposed. Furthermore, the progressive precision provided by a trained SNN is leveraged to accelerate the inference process. Extensive parametric studies are conducted to verify the proposed techniques as well as to examine many aspects of the proposed learning rules. To further demonstrate the effectiveness of the algorithm, the MNIST benchmark test is conducted. Recognition accuracies of 97.2% and 97.8% are achieved with the neural networks trained by the proposed algorithm. With the hardware-friendly learning algorithm presented in this chapter, efficient SNN hardware can be built. This is discussed in Chapter III.

CHAPTER III

A Low-Power Hardware Architecture for On-Line Supervised Learning in Spiking Neural Networks

3.1 Introduction

To tackle the problem of lacking effective on-chip supervised learning algorithms for hardware SNN, a hardware-friendly learning rule is proposed in Chapter II for training multilayer SNN on-line. With the gradient information estimated from the spike timings (STs) in the neural network, a stochastic gradient-descent learning is presented.

In this chapter, based on the learning algorithm outlined in Chapter II, we propose an efficient hardware architecture [68]. The proposed event-triggered architecture provides good scalability and low power consumption. Several simplifications and adaptations on the original algorithms are proposed to increase the performance and reduce the power consumption of the system. A cache structure is proposed to leverage the sparsity that exists in neural networks in order to reduce the memory requirement of the system. In addition, a background ST update technique is proposed to boost the system throughput as well as the energy efficiency of the system. The proposed hardware architecture is implemented in a 65-nm CMOS technology and various performance metrics are simulated and reported.

3.2 Hardware Architecture

3.2.1 Algorithm Adaptations

Even though the learning algorithm developed in Chapter II is already suitable for VLSI implementation, a few simplifications can be made to improve the power efficiency of the system further. The MNIST hand-written digit recognition benchmark is employed to study various aspects of the proposed hardware architecture. The original 28×28 MNIST images are down-sampled to 16×16 to accommodate a design that can be simulated, synthesized, placed, and routed in a reasonable amount of time. The network configuration we use in this chapter is 256-50-10, where each number represents the number of neurons in each layer, from the input layer to the output layer.

Memory is the most critical element in hardware implementation of a neural network. It easily dominates both the power consumption and silicon area. Therefore, the bitwidth of the synaptic weights needs to be carefully chosen to minimize the memory size while not degrading the performance. To investigate the trade-off existing in picking the word length, parametric simulations have been conducted for different bitwidth, and the obtained results are compared in Fig. 3.1. The error bars in this figure indicate a 95% confidence interval. In the simulations, the maximum weight is fixed as 1023 while the weight resolution is swept. To speed up the simulations, only 100 images in the training set are used for training. The trained network is evaluated with all the images in the test set. From the figure, we can draw the conclusion that a bitwidth of 20 is necessary to maintain an acceptable system performance. We pick 24 as the bitwidth of the synaptic weight in our design to allow some margins.

Another simplification originates from the division operation in the original algorithm. The term $1 - \mu_i^l$ in (2.9) and (2.12) can be approximated by 1, assuming sparse

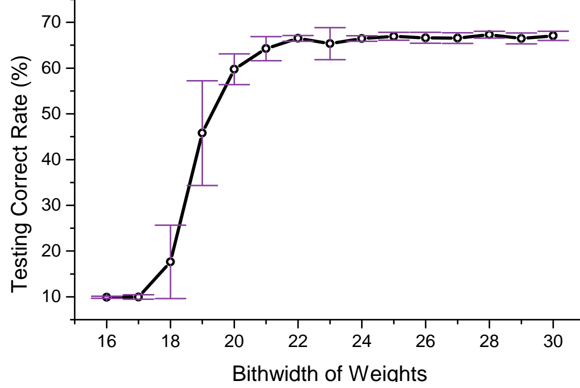


Figure 3.1: Effect of shortening bitwidth of the synaptic weights on the classification results.

spike trains. In addition, to circumvent the slow and power-hungry division, we adopt the approximate division technique proposed in [60]. In an approximate division, the divisor is rounded toward zero to the first power of two. Mathematically, the new divisor can be computed as

$$w_{ij}^{l}{}' = \mathrm{sgn}\left(w_{ij}^{l}\right) \cdot 2^{\lfloor \log_2 \|w_{ij}^{l}\| \rfloor} \tag{3.1}$$

where $\mathrm{sgn}(\cdot)$ represents the sign function. In (3.1), we use w_{ij}^{l} as an example. A similar approximation can be applied to $\overline{x_i^l}$ as well. With the approximate division technique, division can be conveniently implemented as a round-and-shift operation, which remarkably simplifies the hardware complexity.

In the neuron model shown in (2.20), it is expected that a random number generator is available to achieve the effect of the random refractoriness. To avoid using linear feedback shift register to generate random numbers, we propose to use the membrane voltage as the source of randomness. It is observed in (2.20) that random numbers are only needed for firing neurons whose membrane potentials exceed the firing threshold. Thus, we can utilize the last few digits in the membrane potentials of firing neurons as the random numbers to decide whether a refractory event should occur or not. With such a random refractoriness, two consecutive spikes only occur

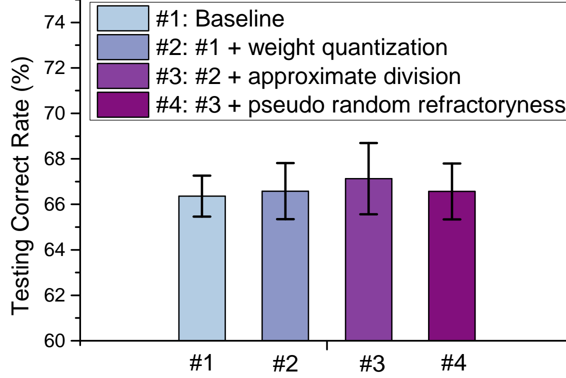


Figure 3.2: Comparison of the recognition rates achieved with the proposed simplifications on the original algorithm.

occasionally. Therefore, the term $\left(1 - x_i^l[n - T - 1]\right)$ in (2.10) is dropped for the ease of implementation.

Fig. 3.2 compares the testing recognition rates of the neural networks employing above-mentioned techniques. Again, only 100 images are used for training to accelerate the evaluation process. It is obvious that none of the proposed simplifications in hardware implementation induces any noticeable performance degradation.

3.2.2 Layer Unit

The diagram of the circuit used to implement one layer of the neural network is illustrated in Fig. 3.3. Input of this layer is the address representing spikes from the preceding layer. Corresponding synaptic weights are read out from the weight memory upon a new spike event, and the membrane potential for each neuron in the layer is updated. After all spikes from the preceding layer are committed for the current tick, neurons in this layer are evaluated. Spikes are generated according to the dynamics shown in (2.20). A priority encoder is then employed to encode output spikes into corresponding addresses.

The proposed neuron circuit is also shown in Fig. 3.3. It is designed to minimize the number of adders, which is normally the most bulky and power-hungry component in a spiking neuron. In the proposed neuron circuit, only one adder and three

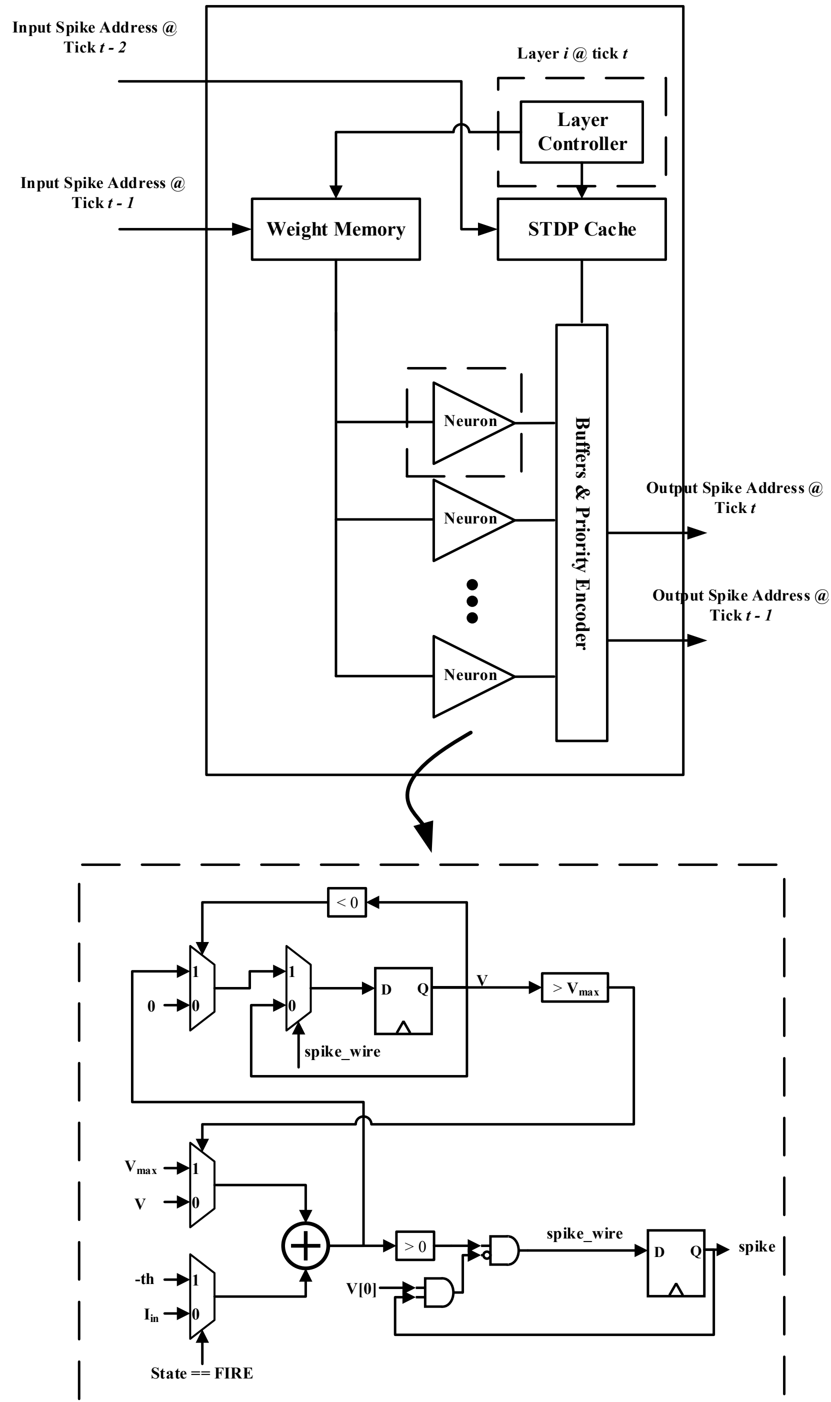


Figure 3.3: Schematic of the circuit implementing one layer of the neural network.

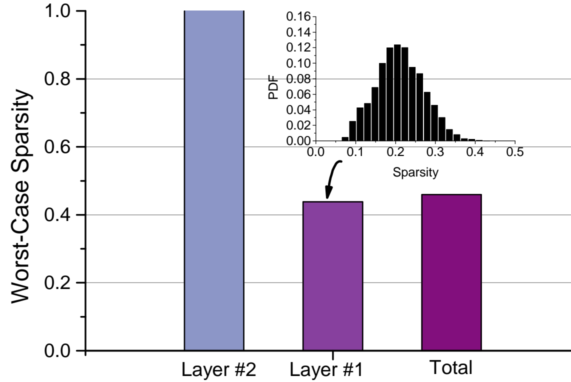


Figure 3.4: Illustration of the sparsity in the employed neural network over 10000 testing images.

comparators are needed. Among those comparators, two of them are trivial sign detectors.

3.2.3 Leveraging Sparsity

To feature effective learning, memory units are needed to store spike timing information in our proposed algorithm. In the most straightforward implementation, the numbers of words used to store ST information and synaptic weights are the same. In other words, each synapse has one dedicated ST field in the memory. Even though the ST information generally demands much shorter word length compared to the synaptic weight, it is still a burden to the system in order to store all the ST information. Fortunately, as pointed out in [60, 59], sparsity in neural networks can be leveraged to reduce the memory requirement. Sparsity is an important feature of many neural networks. It is not only discovered in biological neural networks but is also controlled in many artificial neural networks [69]. Indeed, many real-world signals are sparse in nature, such as image, audio, and video, resulting in sparsely activated input-layer and hidden-layer neurons when such signals are presented to neural networks. Fig. 3.4 demonstrates the sparsity in the investigated neural network. The worst-case sparsities in the figure are obtained from a testing set of 10000 MNIST images. It can be observed that the input layer (layer #1) has a low aver-

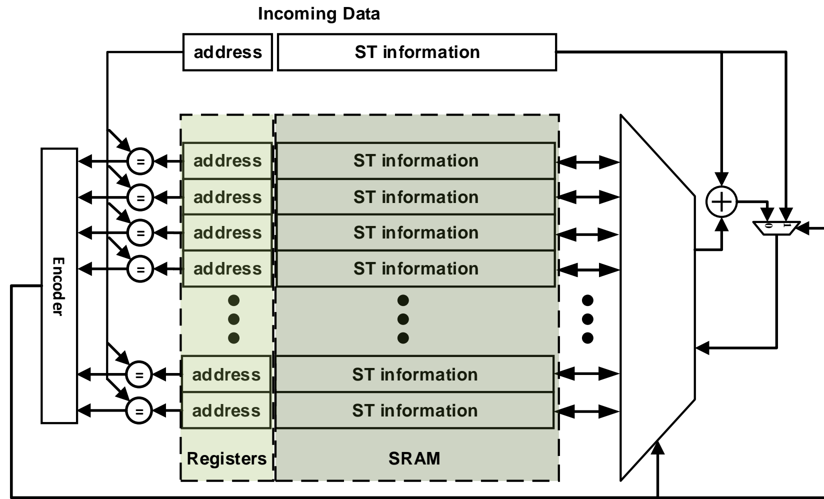


Figure 3.5: Cache structure employed to store the active ST information.

age sparsity of approximately 0.2 and a worst-case sparsity of 0.4. That is, at most 40% of input neurons are activated for any image in the testing set. As the input layer typically has most neurons in a neural network, the total worst-case sparsity is mainly determined by the input layer, as shown in Fig. 3.4. Furthermore, sparsity regulation is often employed to regulate hidden-layer neurons in order to meet certain sparsity requirement. It is demonstrated that learning with sparsity is more biologically plausible as well as more effective [69]. To leverage the spasity that widely exists in neural networks, we observe that only recently active synapses have non-zero ST information. Therefore, we propose to use a cache structure to store ST information, as illustrated in Fig. 3.5. It is similar to a fully associative cache structure. Whenever a new entry comes in, the content addressable memory first searches if the entry with the same address is already in the cache. On a cache miss, a new entry in the cache structure is created, and the newly-coming ST information is directly stored. Such a structure can help reduce the memory requirement significantly, considering that only the ST information associated with active synapses are stored.

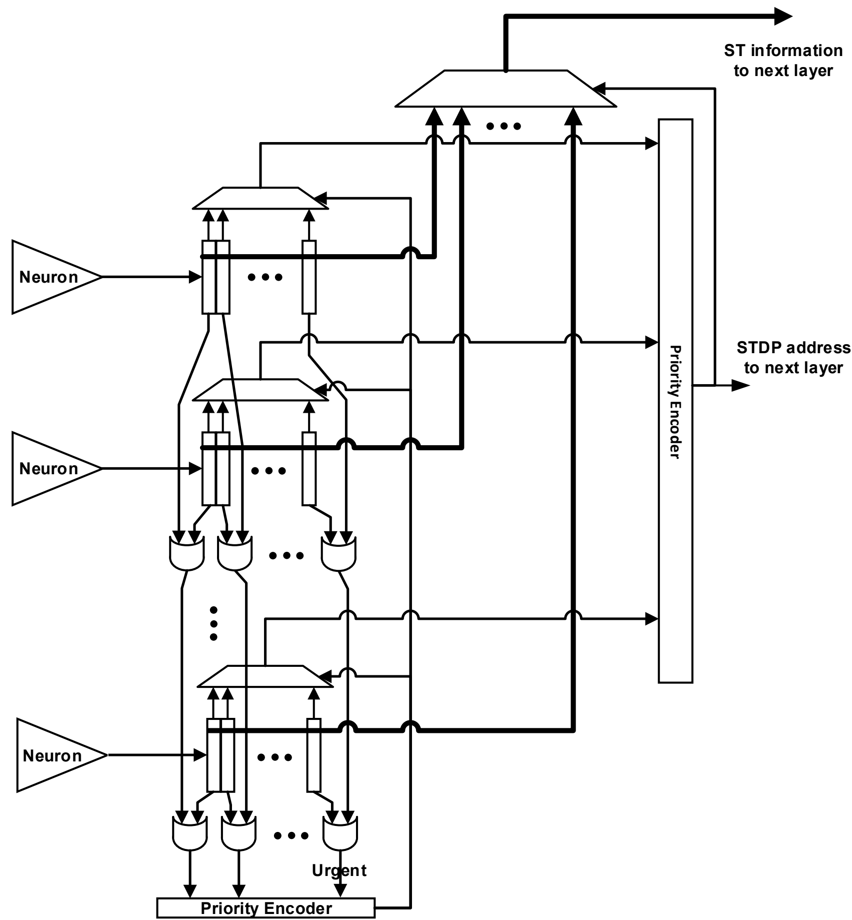


Figure 3.6: Schematic of the circuit that implements the background ST information updating technique.

3.2.4 Background ST Update

In the learning algorithm, the ST information needs to be updated while the neural network is running. The frequency of updating the STDP cache serves as a design parameter. One extreme is to hold all spike timings for each neuron and to update the STDP cache only once in the end. This way of updating minimizes the frequency of visiting the STDP cache, yet requires a large memory structure to store all the spike timings. Another extreme is to update the STDP cache immediately after each spike. This method avoids the need for a large memory array to hold spike timings but slows down the system significantly. Intuitively, weight memory

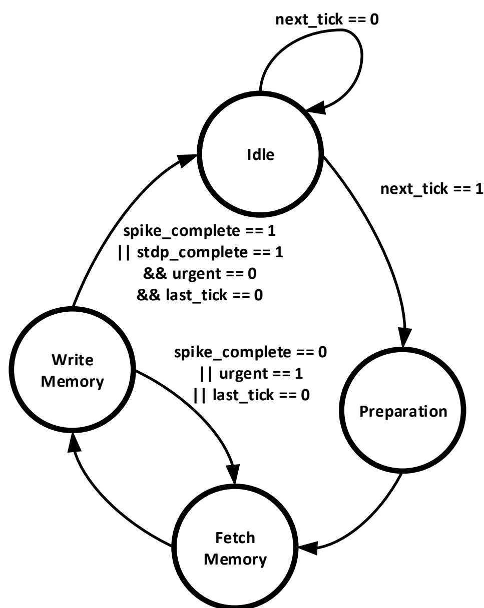


Figure 3.7: State diagram of the background ST updating scheduler.

and STDP cache are visited in similar frequencies in every tick. However, both read and write operations are needed for the STDP cache updating, whereas only the read operations are conducted during a membrane voltage update. Therefore, conducting immediate STDP field update inevitably prolongs the duration of each tick.

In contrast to the membrane voltage updating where an immediate response is required, the ST field updating can be delayed without affecting the computational results. Therefore, we propose hiding the STDP updating in the background using the circuit shown in Fig. 3.6. Local buffers are employed to hold the spike timing information temporarily. A scheduling algorithm with a state diagram shown in Fig. 3.7 is utilized to control whether a ST update should be conducted. The basic idea is that updating of the ST is only carried out when there are membrane voltage updates currently going on. The ST update is enforced whenever an urgent flag is set, which indicates that the storing capacity of the buffers is reached and the update has to be conducted otherwise information will be lost. Another situation to enforce the ST updates is when the current tick is the last tick in a learning iteration. This way

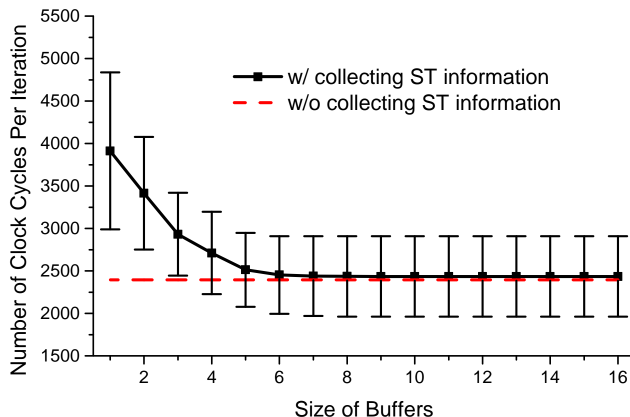


Figure 3.8: Comparison of the number of clock cycles per forward iteration for different buffer depth.

of scheduling helps conduct ST updating in the background. It attempts to update the ST information immediately while ensuring that the duration of each tick is only slightly affected. Fig. 3.8 compares the number of clock cycles for each forward iteration as the length of the buffer varies. When a shallow buffer is used, more clock cycles are needed for each forward iteration. Updating of the ST information significantly slows down the system in this case. As the size of the buffer reaches five, the improvement in the number of clock cycle per forward iteration starts saturating.

3.3 CMOS Implementation Results

The proposed hardware architecture is implemented in TSMC 65 nm technology along with all the techniques proposed in this chapter. The chip layout is illustrated in Fig. 3.9. The chip occupies an area of 1.8mm^2 including pads. Synaptic weight memories, which are implemented in SRAM, take most of the area. It is also noted that the STDP cache is much smaller compared to the weight memory. Table 3.1 shows the estimated power consumption breakdown for the learning mode and the inference mode, respectively. The numbers in the table are estimated with Cadence Innovus. The results are obtained with the post-layout parasitics and the circuit-switching activities obtained from gate-level simulations. It is observed that most

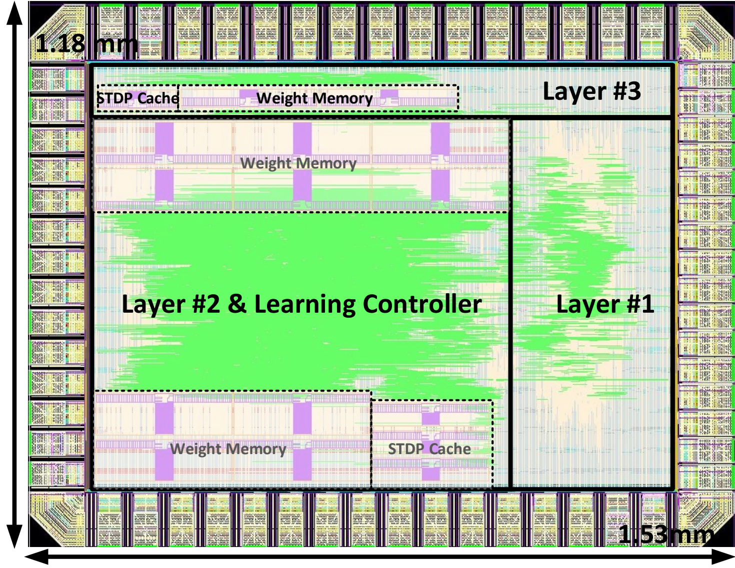


Figure 3.9: Chip layout of the CMOS implementation.

power is dissipated in the second layer, as it involves most synaptic operations. In addition, the inference mode consumes less power than the learning mode, as the ST information and the synaptic weights are not updated in the inference mode.

Table 3.1: Power consumption breakdown

	Learning	Inference
Layer #1	20.97 mW (20.14%)	21.15 mW (23.17%)
Layer #2	49.11 mW (47.16%)	43.64 mW (47.80%)
Layer #3	7.34 mW (7.05%)	7.47 mW (8.19%)
Learning controller	12.13 mW (11.66%)	4.74 mW (5.19%)
Others	13.28 mW (12.76%)	14.30 mW (15.66%)
Total	104.12 mW	91.30 mW

One very useful feature of an SNN is the capability to infer with a progressive precision, as demonstrated in Chapter II. It provides an additional knob for the designer to optimize the system performance dynamically. Fig. 3.10 plots the time and energy needed for a certain inference accuracy when the first-to-spike-K-spikes scheme is employed. That is, the output neuron that first spike K spikes is chosen as the winning neuron and the recognized digit is read out accordingly. As shown in the

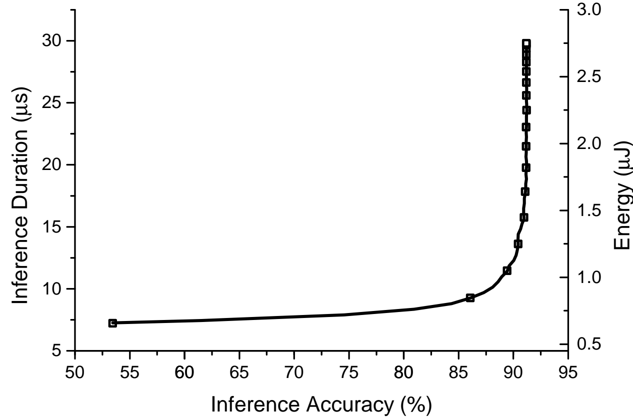


Figure 3.10: Time and energy needed per inference as a function of the inference accuracy.

figure, a growing recognition accuracy can be achieved by spending more time and energy in inference, providing additional freedom that can be configured at run-time.

Table 3.2: Summary and comparison with prior works

	This work	[[70]]	[[57]]	[[71]]
Network size	256-50-10	256-256	484-256	256-256
Bitwidth of synapses	24 bit	8 bit and 13 bit	1 bit	4 bit
Memory size	358.3 Kb	1.31 Mb	256 Kb	256 Kb
Technology	65-nm	65-nm	45-nm	45-nm
Core area	1.1 mm^2	3.1 mm^2	4.2 mm^2	4.2 mm^2
Power consumption	104.12 mW @ learning, 91.3 mW @ inference	228.1 mW @ learning, 218 mW @ inference	45pJ/spike	-
Clock frequency	166.7 MHz	235 MHz @ learning, 310 MHz @ inference	1 KHz	1 MHz

Table 3.2 summarizes the performances of the CMOS implementation. The design is also compared with the state-of-the-art SNN implementations in the literature. Compared to other SNN chips, one remarkable feature of the design presented in this chapter is that it is a multilayer neural network that is capable of conducting on-chip supervised learning. Such a piece of hardware can be beneficial to many

energy-starved applications where built-in intelligence is needed.

3.4 Chapter Summary

In this chapter, we propose a hardware architecture for supervised learning in multilayer SNNs based on the algorithm presented in Chapter II. Learning can be conducted directly on-chip with the algorithm. Several adaptations are made to the original learning algorithm to reduce the complexity of the circuit while not degrading the performance noticeably. An event-triggered architecture is proposed to improve the throughput as well as the energy efficiency of the system. In addition, the sparse nature of an SNN is leveraged in the design to reduce the memory requirement. A background ST update is also utilized to speed up the inference and learning process of the algorithm. The design is implemented in TSMC 65-nm technology and how inferring with a progressive precision can be beneficial in saving energy is demonstrated. It is expected that the learning algorithm and hardware architecture proposed in Chapter II and this chapter will help accelerate the development of energy-efficient deep spiking neural network hardware.

CHAPTER IV

A Low-Power Accelerator for Action-Dependent Heuristic Dynamic Programming

4.1 Introduction

In addition to the efforts of building customized SNN hardware, there is also a lot of interest in accelerating ANN applications with specialized hardware in recent years. Compared to SNN accelerators, ANN accelerators can take advantage of many state-of-the-art network architectures, algorithms, and techniques that have been developed over the past decade. Recently, many specialized accelerators have been developed for ANNs [72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84]. Various design techniques have been proposed to improve the efficiency and throughput of the neural network accelerators. Most of these accelerators are for deep neural networks, especially the deep convolutional neural network, which is one of the most popular and powerful neural networks employed widely in image and audio recognition.

Adaptive dynamic programming (ADP) is a powerful algorithm in solving various decision-making and control problems [85, 86, 87, 88, 89]. Through approximating the solution to the Bellman equation, the ADP algorithm can generate optimal or near-optimal solutions for many real-life problems. The ADP algorithm is considered one type of reinforcement-learning algorithm. It is also known as adaptive critic design,

approximate dynamic programming, neurodynamic programming, etc. Many ADP algorithms have been successfully implemented in the form of software running on a general-purpose processor [90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103]. Among various types of ADP algorithms, the action-dependent heuristic dynamic programming (ADHDP) algorithm is one of the most popular and most powerful ADP algorithms [90, 91, 97, 98, 100, 102], as this algorithm does not require any pre-knowledge about the model of the system to be controlled.

Despite being effective as an algorithm itself, the highly iterative ADP algorithms running on a general-purpose processor in the form of software fail to provide energy-efficient solutions to various applications where power consumption is of importance. For example, potential applications for the ADP algorithm are mobile autonomous robots with a small form factor [35, 34, 33] and future internet-of-things (IoT) devices. For these microrobots and IoT devices that chiefly rely on energy scavenging from the environment or energy stored on a tiny battery, energy consumption is of utmost importance. Therefore, it is necessary to resort to specialized accelerators in order to meet the stringent requirements of both the speed and energy consumption.

Even though an ADP-based reinforcement learning also utilizes neural networks, there are some different design challenges and tradeoffs in building accelerators for ADP algorithms [104]. For example, most existing deep-learning accelerators only implement the inference phase, as the learning is assumed to be accomplished somewhere else. Such an operating model indeed works well for supervised learning. The slow and energy-consuming learning process can be conducted on the graphic processing units in the data centers. Users of the neural network accelerator can then download the trained weights onto the chip and the accelerator is ready to conduct some classification or inference tasks. On the other hand, most ADP algorithms target controlling plants or making optimal decisions in a dynamic environment. Under this circumstance, each ADP accelerator needs to learn how to choose the optimal policy

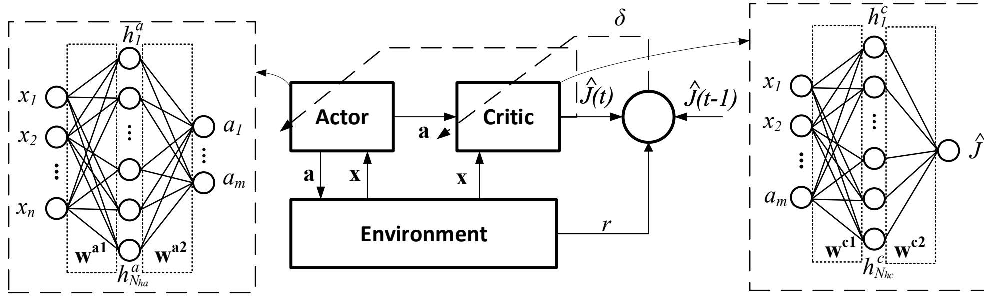


Figure 4.1: Illustration of the actor-critic configurations used in the ADHDP algorithm. Two neural networks, critic network and actor network, are employed in the algorithm to approximate functions need to be learned.

for the plant or environment it is interacting with in an on-line fashion. Therefore, learning for an ADP accelerator is most likely to be a real-time task.

In this chapter, we introduce a hardware architecture as well as design methodologies for ADHDP accelerators [105]. A tile-based computing is employed to provide good scalability for the accelerators. The designed accelerators are also flexible, as they can be programmed with instructions in order to run ADHDP algorithms with different configurations. Low-power operations are achieved through reducing the data movements by utilizing and partitioning data buffers. Furthermore, as we focus on building accelerators that can conduct learning efficiently, a virtual update technique is introduced to leverage some unique computational patterns in the ADHDP algorithm in order to shorten the computational time and increase the energy efficiency.

4.2 Action-Dependent Heuristic Dynamic Programming

In this section, a few concepts for the ADP and ADHDP are reviewed. For brevity, only important terminologies that are closely related to this chapter are covered.

4.2.1 Actor-Critic Networks

Suppose the discrete-time system under control can be modeled by

$$\mathbf{x}(t+1) = f\left[\mathbf{x}(t), \mathbf{a}(t)\right] \tag{4.1}$$

where $\mathbf{x}(t)$ is the n-dimensional state vector at time t, $\mathbf{a}(t)$ is the m-dimensional action vector, and $f(\cdot)$ is the model of the system. The target of the algorithm is to maximize the reward-to-go J, expressed as follows

$$J\left[\mathbf{x}(t)\right] = \sum_{k=1}^{\infty} \gamma^{k-1} r\left[\mathbf{x}(t+k)\right] \tag{4.2}$$

where γ is the discount factor used to promote the reward received in the near future over long-term reward and $r\left[\mathbf{x}(t)\right]$ is the reward received at state $\mathbf{x}(t)$.

The reward-to-go can be maximized through solving the Bellman equation

$$J^*\left[\mathbf{x}(t)\right] = \max_{\mathbf{a}(t)} \{ (r\left[\mathbf{x}(t+1)\right] + \gamma J^*\left[\mathbf{x}(t+1)\right]) \} \tag{4.3}$$

where $J^*\left[\mathbf{x}(t)\right]$ denotes the optimal value function under the optimal policy. The optimal policy $\mathbf{a}^*(t)$ is obtained by maximizing the right-hand side of (4.3).

Solving the Bellman equation directly is intractable for many problems with practical sizes. The complexity grows exponentially with the size of the problem, which is well known as the curse of the dimensionality. To circumvent this difficulty, the ADP algorithm solves the Bellman equation approximately with the help of function approximators. The model-free ADP algorithm we consider in this chapter is the ADHDP algorithm, which is one of the most popular model-free ADP algorithms [90, 91, 97, 98, 100, 102]. It is closely related to the well-known Q-learning algorithm that is widely used in the artificial intelligence community. The model-free ADP algorithm does not need a model for the plant or the environment. The algorithm

learns the model in the process of interacting with the plant or the environment. The configuration of the ADHDP algorithm is illustrated in Fig. 4.1. In the figure, $\mathbf{x}$ and $\mathbf{a}$ represent state and action vectors, respectively. $\mathbf{h^a}$ and $\mathbf{h^c}$ are N_{ha}-dimensional and N_{hc}-dimensional output vectors from the hidden units in the actor and critic network, respectively. $\mathbf{w^{a1}}$, $\mathbf{w^{a2}}$, $\mathbf{w^{c1}}$, and $\mathbf{w^{c2}}$ are synaptic weights in the networks. Two neural networks are used as universal function approximators in this algorithm. One neural network, called the critic network, is employed to generate $\hat{J}[\mathbf{x}(t)]$, which is an estimation of $J[\mathbf{x}(t)]$. The critic network attempts to learn $J[\mathbf{x}(t)]$ through adjusting the synaptic weights in the neural network in order to minimize the temporal difference error as shown in (4.4).

$$\delta(t) = \hat{J}[\mathbf{x}(t-1)] - \gamma\hat{J}[\mathbf{x}(t)] - r[\mathbf{x}(t)] \tag{4.4}$$

The second neural network is called an actor network. Its function is to generate an action vector $\mathbf{a}(t)$ that maximizes the estimated reward-to-go $\hat{J}[\mathbf{x}(t)]$. Action vector outputted by the actor network is fed to the critic network. The actor then adjusts its synaptic weights to maximize $\hat{J}[\mathbf{x}(t)]$.

4.2.2 Learning Algorithm

In the learning process, we need to train the two neural networks such that the defined cost function can be minimized. The most popular and efficient way to train a neural network is the stochastic gradient descent learning based on backpropagation [106]. Errors at output layers are propagated back to each synapse in the network, layer by layer. There are two phases in the ADHDP algorithm: critic update phase and actor update phase. Multiple iterations are involved in both phases. Each iteration contains a forward operation and a backward operation. The algorithm is illustrated in Fig. 4.2.

Inputs : $\mathbf{w^{a1}}$, $\mathbf{w^{a2}}$, $\mathbf{w^{c1}}$, $\mathbf{w^{c2}}$: weights for the actor and critic neural network
I_a, I_c: The maximum number of iterations allowed for updating actor and critic networks in one time step
E_a, E_c: Thresholds to control whether an update can be terminated

1 $t = 0$
2 Actor network forward operation: compute $\mathbf{a}(t)$
3 Critic network forward operation: compute $\hat{J}\,[\mathbf{x}(t)]$
4 Output action $\mathbf{a}(t)$ and obtain the updated states $\mathbf{x}(t+1)$, reward $r[\mathbf{x}(t+1)]$, and termination request REQ_{term} from the environment or the plant
5 **while** $REQ_{term} \neq 1$ **do**
6     $t = t + 1$, $i_c = 0$, $i_a = 0$
7   Actor network forward operation: compute $\mathbf{a}(t)$
8   Critic network forward operation: compute $\hat{J}\,[\mathbf{x}(t)]$
9   Compute the temporal difference $\delta(t)$
10   **while** $\left(i_c < I_c \;\&\&\; \frac{\delta(t)^2}{2} \geq E_c\right)$ **do**
11   Critic network backward operation: update $\mathbf{w^{c2}}$ and $\mathbf{w^{c1}}$
12   Critic network forward operation: compute $\hat{J}\,[\mathbf{x}(t-1)]$
13   Compute the temporal difference $\delta(t)$
14         $i_c = i_c + 1$
15   Compute the cost function $\frac{e_a^2}{2}$
16   **while** $\left(i_a < I_a \;\&\&\; \frac{e_a^2}{2} \geq E_a\right)$ **do**
17   Actor network backward operation: update $\mathbf{w^{a2}}$ and $\mathbf{w^{a1}}$
18   Actor network forward operation: compute $\hat{J}\,[\mathbf{x}(t)]$
19   Compute the cost function $\frac{e_a^2}{2}$
20         $i_a = i_a + 1$
21   Output action $\mathbf{a}(t)$ and obtain the updated states $\mathbf{x}(t+1)$, reward $r[\mathbf{x}(t+1)]$, and termination request REQ_{term} from the environment or plant

Output: $\mathbf{w^{a1}}$, $\mathbf{w^{a2}}$, $\mathbf{w^{c1}}$, $\mathbf{w^{c2}}$: updated weights for the actor and critic neural network

Figure 4.2: Pseudocode for the ADHDP algorithm

4.2.2.1 Forward Opeartion

For each iteration in the actor update phase, the forward operation is carried out according to (4.5)-(4.9).

$$h_i^a = \sigma\left(\sum_{j=1}^{n} w_{ij}^{a1} x_j\right) \tag{4.5}$$

$$a_i = \sigma\left(\sum_{j=1}^{N_{ha}} w_{ij}^{a2} h_j^a\right) \tag{4.6}$$

$$\mathbf{p^c} = \begin{bmatrix} \mathbf{a} \\ \mathbf{x} \end{bmatrix} \tag{4.7}$$

$$h_i^c = \sigma\left(\sum_{j=1}^{m+n} w_{ij}^{c1} p_j^c\right) \tag{4.8}$$

$$\hat{J} = \sum_{i=1}^{N_{hc}} w_i^{c2} h_i^c \tag{4.9}$$

where $\sigma(\cdot)$ is the activation function. Popular choices are hyperbolic tangent function, sigmoid function, and the rectified linear unit. For the critic forward phase, only (4.8)-(4.9) are carried out. (4.5)-(4.7) are not necessary as the weights in the actor network remain the same, which leads to the same action vector.

4.2.2.2 Backward Operation

During the backward operation in the critic update phase, $\mathbf{w^{c1}}$ and $\mathbf{w^{c2}}$ are updated according to (4.10) and (4.11).

$$\Delta w_i^{c2} = \alpha \delta h_i^c \tag{4.10}$$

$$\Delta w_{ij}^{c1} = \alpha e_i^{c1} \sigma'\left(\sum_{k=1}^{m+n} w_{ik}^{c1} p_k^c\right) p_j^c \tag{4.11}$$

where $e_j^{c1} = \delta w_j^{c2}$ is the error at the hidden unit h_i^c, and α is the learning rate.

During the backward operation in the actor update phase, $\mathbf{w^{a1}}$ and $\mathbf{w^{a2}}$ are updated according to (4.12) and (4.13).

$$\Delta w_{ij}^{a2} = \alpha e_i^{a2} \sigma' \left(\sum_{k=1}^{N_{ha}} w_{ik}^{a2} h_k^a \right) h_j^a \tag{4.12}$$

$$\Delta w_{ij}^{a1} = \alpha e_i^{a1} \sigma' \left(\sum_{k=1}^{n} w_{ik}^{a1} x_k \right) x_j \tag{4.13}$$

where $e_j^{a1} = \sum_{i=1}^{m} \left[e_i^{a2} \cdot \sigma' \left(\sum_{k=1}^{N_{ha}} w_{ik}^{a2} h_k^a \right) \cdot w_{ij}^{a2} \right]$, $e_j^{a2} = \sum_{i=1}^{N_{hc}} \left[e_i^{c1} \cdot \sigma' \left(\sum_{k=1}^{m+n} w_{ik}^{c1} p_k^c \right) \cdot w_{ij}^{c1} \right]$, and $e_j^{c1} = e_a w_j^{c2}$ are backpropagated errors at h_j^a, a_j and h_j^c, respectively. $e_a^2/2$ is the cost function that needs to be minimized for the actor network. In many applications, the desired reward-to-go is 0, i.e. no punishment (negative reward). In this case, a convenient choice is $e_a = \hat{J}\left[\mathbf{x}(t)\right]$ [90, 91, 97, 98, 100, 102].

4.3 Hardware Architecture

The proposed hardware architecture for the ADP accelerator is shown in Fig. 4.3. The accelerator consists of three major blocks: datapath, memory, and controller. The datapath is the core of the ADP accelerator. It handles all the arithmetic operations needed in the ADHDP algorithm. The memory unit contains all the on-chip storage units, including a static random-access memory (SRAM) array used to store synaptic weights, registers for holding neuron states, and input buffers for reducing data movements. The controller oversees operations of the whole accelerator and executes programmed instructions in order.

4.3.1 On-Chip Memory

Memory in our system can be divided into three categories based on the purposes they serve: synapse memory, neuron memory, and data buffers. The most critical and also the largest memory block is the synapse memory, as the number of synapses grows

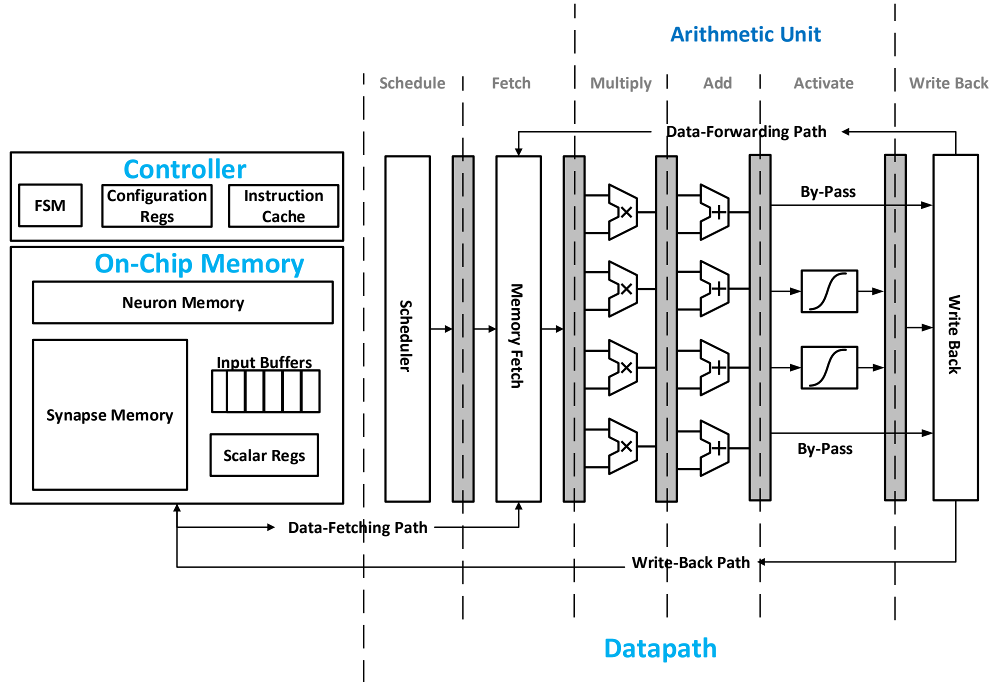


Figure 4.3: Hardware architecture for the proposed accelerators. Data-level parallelism is exploited through utilizing multiple datapath lanes. A reconfigurable five/six-stage pipeline is used for the datapaths.

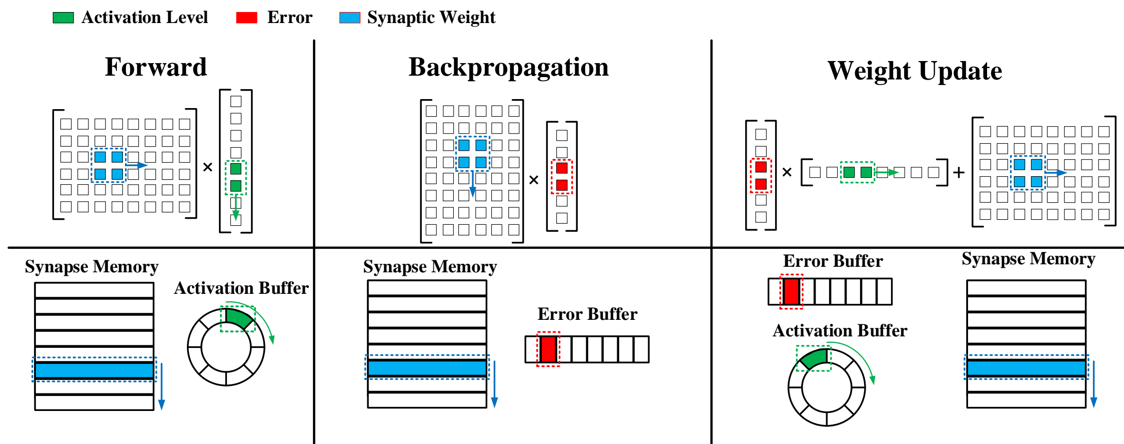


Figure 4.4: Illustration of data flow and memory access patterns in the proposed accelerators. Data buffers are employed to exploit the locality of the data. Synaptic weights needed in on tile operation is stored in one row.

quadratically with the size of the neural network. In this design, we use SRAM array for storing synaptic weights. Neuron memory is where the activation levels of neurons in the network are stored. It is implemented with an array of registers in this design. Data buffers are storage units used to hold the input, intermediate, and output data temporarily in order to accelerate the computation and save computational energy.

Data flow and memory access patterns employed in the proposed accelerator are shown in Fig. 4.4. The computations in the forward operations shown in (4.5) - (4.9) are mostly matrix multiplication operations. Similar to most machine-learning accelerators [72, 73, 74, 77], we adopt a tile-based matrix multiplication strategy, where the matrix is partitioned into several smaller blocks. The size of the tile is determined by the number of data lanes available in the system. In this design, the number of lanes is set to four, as this is enough for applications targeted by this work. Nevertheless, the proposed architecture and design methodology are scalable, so more lanes can readily be added into the design to accommodate larger problems.

For the forward operation, we adopt a row-wise multiplication. The neuron activation vector is first loaded from the neuron memory to the activation buffer. The activation buffer is a circular buffer and it rotates a complete circle when multiplying each row in the matrix. Loading the data from the neuron memory to the input buffer has the advantage that the data in the buffer can be reused without accessing the relatively-large neuron memory repeatedly, thereby saving power and time. Synaptic weights in the SRAM are arranged in a way such that weights corresponding to one tile are stored in the same row for easy access.

For the backward operation, there are two major steps: error backpropagation and weight update. The error backpropagation operation is also a matrix-vector multiplication. Similar to the forward operation, tile-based multiplication is used. However, the multiplication in this case is done column-wise instead of row-wise. Such an arrangement has the advantage that the access for the memory is always

sequential, providing a more regular memory access pattern when off-chip memory is used. In the weight update operation, two vectors are multiplied to form a matrix that is added to the old synaptic weight matrix. In this case, elements in the row vector are stored in the circular buffer, whereas elements associated with the column vector are stored in the linear buffer. The error backpropagation and weight update operations are scheduled in alternate clock cycles in order to reuse the same row of synaptic weights. Therefore, for one backward operation, each entry in the synaptic weight SRAM only needs to be read and written once.

4.3.2 Datapath

The datapath is partitioned into five/six-stage reconfigurable pipelines: schedule, fetch, multiply, add, activate, and write back.

4.3.2.1 Datapath Operations

In the schedule stage, instructions fetched from the instruction memory are decoded to obtain necessary information for scheduling operations with data. In the proposed single-instruction-multiple-data (SIMD) architecture, one instruction may contain workloads that need multiple clock cycles to complete. Therefore, the instruction fetching and decoding occur selectively with the help of the controller. The scheduler needs to generate and latch addresses for data to be fetched in the fetch stage as well as to inspect any potential data hazard. Upon detecting that the data needed in the speculatively-scheduled operation are not ready in the input buffer, the scheduler looks for possibilities of data forwarding directly from the memory or write-back buffer. If even the data forwarding is not able to resolve the data hazard, a STALL operation is inserted into the pipeline as a null operation, waiting for the data needed to be computed. In the fetch stage, data is read from input buffers or memory and is latched in corresponding pipeline registers.

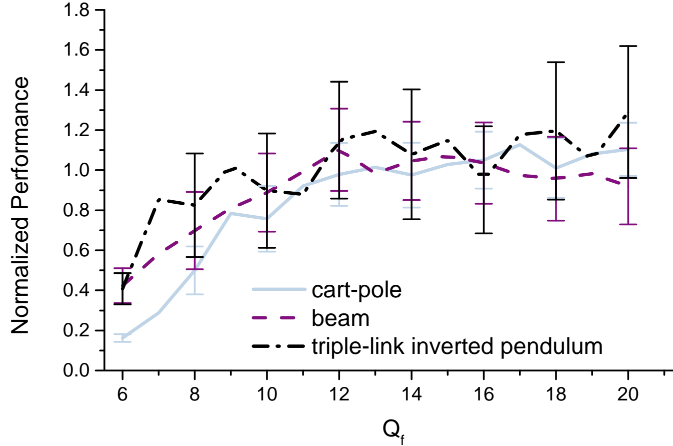


Figure 4.5: Comparison of the learning performances achieved with different levels of data quantization for three classic ADP benchmarks. The obtained performances are normalized with respect to those obtained from double-precision floating-point computations.

The multiply and add stage conduct multiplication and addition operations, respectively. Adders in the add stage can be configured as parallel adders, adder trees, or a mixture of both depending on the operations conducted. The activate stage implements the activation function employed in neural networks. The hyperbolic tangent function is employed in this design, as it is the most popular choice for ADP algorithm in the literature [90, 91, 97, 98, 100, 102]. In the proposed design, the activation function is implemented with piecewise linear interpolation, similar to those employed in [72, 107]. Depending on the operation conducted, the activation stage may be bypassed, as the activation operation is only needed in the forward phase. In this case, the six-stage pipeline is reduced to a five-stage pipeline. After arithmetic computations, the write-back stage in the end of the pipe writes computed results back to storage units depending on the instruction executed.

4.3.2.2 Datapath Quantization

One important consideration in designing customized accelerators is the choice of bitwidth used to represent data in the system. It is the norm to use a fixed-point

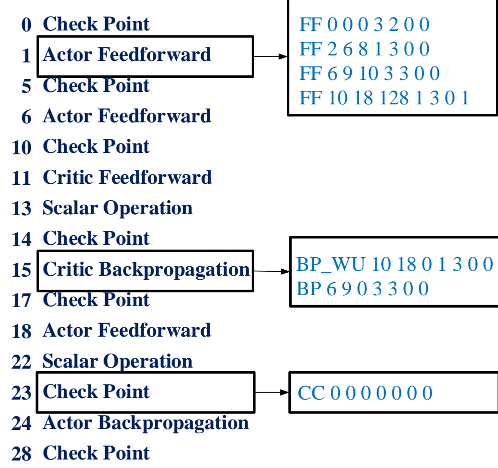
number representation in machine-learning accelerators [72, 73, 74, 75, 76, 77, 78, 79, 80, 81] because of its ease of implementation and good computational efficiency. To provide some guidelines in determining the proper bitwidth in our system, we conduct parametric simulations on the learning performance of the ADP algorithm under different bitwidths. Three most popular benchmark tasks for the ADP algorithm are employed in the parametric study: The cart-pole balancing problem [90, 97, 91, 100], the beam-balancing problem [99, 101], and the triple-link inverted pendulum problem [90, 97]. In these three tasks, the target is to control the system such that the states of the system stay within some pre-defined ranges. In our experiment setting, once the states of the system under control exceed the desired ranges, punishment (or a negative reward) is provided. For each task, data used in the algorithm, including synaptic weights, neuron states, and other intermediate variables, are quantized to numbers with a fractional bitwidth of Q_f. Learning processes are then carried according to the ADP algorithm. The learning performance obtained for each task with different levels of quantizations are compared in Fig. 4.5. For each task, 50 runs are conducted, where a run contains several trials. Each trial is a complete process from beginning to end. A trial is ended either when the maximum time is reached (1000 time steps in these experiments) or the states under control exceed certain limits. The learning performance is measured by the total time that the plants are successfully maintained in the desired states in all trials. In Fig. 4.5, performances obtained with different bitwidth are normalized to the performances obtained from the computations with a double-precision floating-point number representation. Error bars in the figure correspond to a 95% confidence interval. As shown in the figure, performances achieved with quantized data start matching those obtained with double-precision data when the bitwidth for the fractional part reaches 12 bit. We use a 6-bit integer part (including a 1-bit sign information) and an 18-bit fractional part to represent data in our accelerators. The extra six bits in the fractional part compared to the

Op Code	Source Addr	Synapase Addr	Destination Addr	# of Row	# of Column	Offset	Config

(a)

Op Code	Operation
FF	Forward Operation
SCA	Scalar Operation
BP_WU	Error Backpropagation & Weight Update
BP	Error Backpropagation
WU	Weight Update
CC	Controller Operation

(b)



(c)

Figure 4.6: Illustration of the instructions used in the accelerators. (a) Format of the instruction. (b) List of all operation codes and their corresponding operations. (c) A sample program for implementing the ADHDP algorithm shown in Fig. 4.2.

12-bit lower limit is to provide some tolerance in the design.

4.3.3 Controller

The main role of the controller is to determine the instruction flow. The format of instructions developed for our accelerator is shown in Fig. 4.6(a). The operation code field specifies the type of the instruction. There are six types of instructions in our accelerator, as shown in Fig. 4.6(b).

The code "FF" corresponds to the forward operation, which is the most common instruction. The code "SCA" is for scalar operation such as calculating the temporal difference as shown in (4.4). The operation code "BP_WU" is used for hidden-layer units where both error backpropagation and weight update are needed. Code "BP" and "WU" are for error backpropagation and weight update, respectively. They are

used when only one type of operation is needed. For example, "WU" code can be used for the input layer when error backpropagations are not needed. The code "CC" calls for controller operation. It can be used, for example, to implement the conditional jumps in Fig. 4.2. The fields "Source Addr", "Synapse Addr", and "Destination Addr" specify the addresses for the source data, the address for the synaptic weight and the addresses to write back, respectively. In our design, the synapse memory has its own address space, whereas all other registers share a unified address space. The "# of Row" and "# of Column" fields indicate the size of the matrix operated on. The field of "Offset" specifies any offset in computing the matrix multiplication. For example, as shown in Fig. 4.2, backpropagation for the actor network only needs to be done for $\mathbf{a}(t)$. Therefore, elements associated with $\mathbf{x}(t)$ should be skipped through specifying the offset. The "Config" field is used for configuration purposes, for example, to specify whether to bypass the activation stage in the datapath.

With all the fields specified in the instructions, the scheduler can schedule operations based on this information. In the proposed architecture, one instruction specifies all operations conducted on one matrix. An example of the instructions corresponding to the pseudocode shown in Fig. 4.2 is illustrated in Fig. 4.6(c). Only a portion of the instructions are shown for the purpose of brevity. All instructions in the figure correspond to a series of operations conducted by the datapath except for the "Check Point" operation where the controller conducts a conditional jump with the help of an FSM.

4.4 Virtual Update Algorithm

In this section, we examine a few unique features of the ADP algorithm. These features are then exploited to improve speed and energy efficiency of the accelerator.

4.4.1 Algorithm

In the ADP algorithm, it is the norm to conduct many internal cycles in order to minimize the cost function for each input vector [90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102]. This corresponds to the second and the third *while* loop in Fig. 4.2. The maximum number of internal loops for each input vector is typically in the range of 10 to 100. In other words, many iterations are carried out for the same input vector, attempting to minimize the cost function at the current time step. Therefore, it may be worth conducting some pre-processing if the same input vector is used repeatedly. Such a simplification is indeed possible by inspecting the unrolled *while* loop.

Without loss of generality, let us focus on the update in the critic network. Weights in the input layer of the critic network are updated according to (4.11), followed by the immediate forward operation in (4.8). For the ease of explanation, (4.11) and (4.8) are rewritten in (4.14) and (4.15) with the dependence on the loop index i_c explicitly indicated. Note that $\mathbf{p^c}$ in the equations is not a function of i_c, as the input vector and selected action remain the same when the critic network is updating.

$$\Delta w_{ij}^{c1}(i_c) = \alpha e_i^{c1}(i_c)\sigma' \left[\sum_{k=1}^{m+n} w_{ik}^{c1}(i_c) p_k^c \right] p_j^c \tag{4.14}$$

$$h_i^c(i_c+1) = \sigma \left[\sum_{j=1}^{m+n} w_{ij}^{c1}(i_c+1) p_j^c \right] \tag{4.15}$$

By substituting (4.15) into (4.14) with the help of the relationship $w_{ij}^{c1}(i_c+1) = w_{ij}^{c1}(i_c) + \Delta w_{ij}^{c1}(i_c)$, one can obtain

$$o_i^c(i_c) = o_i^c(0) + E_i(i_c)\Lambda_c \tag{4.16}$$

where

$$\Lambda_c = \sum_{j=1}^{m+n} (p_j^c)^2 \tag{4.17}$$

$$o_i^c(0) = \sum_{j=1}^{m+n} w_{ij}^{c1}(0) p_j^c \tag{4.18}$$

$$E_i(i_c) = \sum_{k=0}^{i_c} \alpha e_i^{c1}(k) \sigma'[o_i^c(k)] \tag{4.19}$$

$o_i^c(i_c)$ is the input to neuron h_i^c in the i_c^{th} iteration. Λ_c is the sum of squares of activation levels of all input neurons, which is independent of i_c. In this case, we only need to update $E_i(i_c)$ in each iteration. $o_i^c(0)$ and Λ_c need only to be computed and stored once. Therefore, activation levels of the hidden layer units in the i_c^{th} iteration can be conveniently calculated based on results obtained from the previous iterations. Even though no actual weight update or forward operations are conducted, it appears to neurons in other layers as if the weights in the input layer were updated. We call this technique the virtual update technique.

When the update loop is terminated either because the maximum number of iterations I_c is reached or because the cost function is below a certain threshold, synaptic weights associated with the input layer are updated according to (4.20).

$$\Delta w_{ij}^{c1} = E_i(i_c) p_j^c \tag{4.20}$$

It is worth mentioning that the virtual update algorithm accelerates the learning process through reordering effective operations more efficiently instead of using approximations. Therefore, the proposed technique does not reduce the precision of the ADP algorithm.

4.4.2 Hardware Implementation Considerations

In order to exploit the proposed virtual update technique, one extra instruction "VU" is added to our instruction set introduced in Section 4.3. This instruction implements (4.16) - (4.19) in two groups of operations. The first group of operations is to compute and store Λ_c when the current input vector is presented for the first time. The second group of operations are the multiply-and-add operations shown in (4.16). The operation of accumulating $E_i(i_c)$ is merged to the normal "BP" or "BP_WU" operations without introducing any overhead in computational time. It is worth noting that all newly-added operations, as shown in (4.16)-(4.19), scale linearly with the size of the network, whereas the original backward and forward operations scale quadratically. Therefore, the virtual update technique can help save significant computational efforts.

To implement the virtual update algorithm, $o_i^c(0)$ and $E_i(i_c)$ need to be stored. They can be stored conveniently in the synapse memory, recognizing that the weight memory is not utilized during the virtual update operation. Indeed, the virtual update technique avoids both writing synaptic weights in the weight update phase and reading weights in the forward phase, leaving the weight memory free during that period. Compared to the synaptic weights stored in the synapse SRAM, the additional memory overhead caused by the virtual update technique is negligible, especially when the size of the network is large.

The pseudocode for the *while* loop of updating the critic network with the proposed virtual update technique is shown in Fig. 4.7. If the current iteration is not the last one allowed by the maximum number of iterations, the virtual update algorithm is used to compute neuron activation levels of hidden-layer neurons in the next iteration, otherwise the conventional update is used. It should be ensured that when exiting the *while* loop, normal weight update has to be conducted once according to (4.20), as weights are actually not updated during previous iterations.

```
1  while (i_c < I_c && δ(t)^2/2 ≥ E_c) do
2      Backward operation: update w^{c2}
3      if (i_c == I_c − 1) then
4          Backward operation: update w^{c1}
5          Forward operation: compute h^c
6      else
7          Virtual update: compute h^c
8      Forward operation: compute Ĵ[x(t − 1)]
9      Compute the temporal difference δ(t)
10     if (δ(t)^2/2 < E_c) then
11         Backward operation: update w^{c1}
12     i_c = i_c + 1
```

Figure 4.7: Pseudocode for the *while* loop corresponding to the critic update when the virtual update algorithm is employed

The computational complexity of the virtual update algorithm is compared with that of the baseline in Table 4.1. The number of arithmetic operations per time step is used for comparison. In the table, N_i and N_h represent the number of input-layer neurons and hidden-layer neurons, respectively. L specifies the number of iterations in one time step. Its value should be in the range of $[1, I_c]$ or $[1, I_a]$ depending on whether the network is the critic or the actor. "MAC", "MUL", and "ADD" denote multiply-accumulate, multiply, and add operations, respectively. It is worth noting that the complexities listed in the table are valid for the case where $L > 1$. When $L = 1$, two algorithms have the same computational complexity, as no virtual update takes place. As shown in the table, the virtual update technique significantly reduces the number of operations needed. Even though the technique is only applicable for synapses between the input layer and the hidden layer, the savings in computational efforts is remarkable, as most weights in neural networks concentrate in between these two layers. The actual percentage of savings is reported in Section 4.5.

Table 4.1: Comparison of the computational complexity of conventional update and virtual update

	Regular update	Virtual update
Forward	N_iN_hL **MAC**	$(L-1)N_h$ **MAC** $+N_i$ **MUL** $+N_iN_h$ **MAC**
Backward	N_iN_hL **MAC**	$(L-1)N_h$ **ADD** $+N_iN_h$ **MAC**
Total operations (**MAC/MUL/ADD**)	$2N_iN_hL$	$2(L+N_i-1)N_h+N_i$
Complexity	$O(N_iN_hL)$	$O((L+N_i)N_h)$

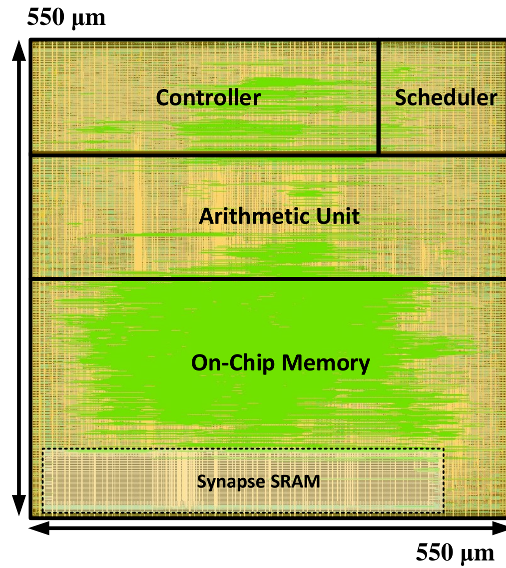


Figure 4.8: Chip layout and floorplan of the accelerator chip with the virtual update algorithm

4.5 Design Examples

The hardware architectures and techniques discussed in previous sections are implemented in TSMC 65-nm CMOS technology and the obtained simulation results are presented in this section. In order to examine all aspects of the proposed design methodology, simulators are developed in a high-level programming language. These simulators model behaviors of the final chip and they are employed to measure the input-output relationships and clock cycle needed to accomplish certain tasks. Area, speed, and power consumption are evaluated based on post-layout circuit-level

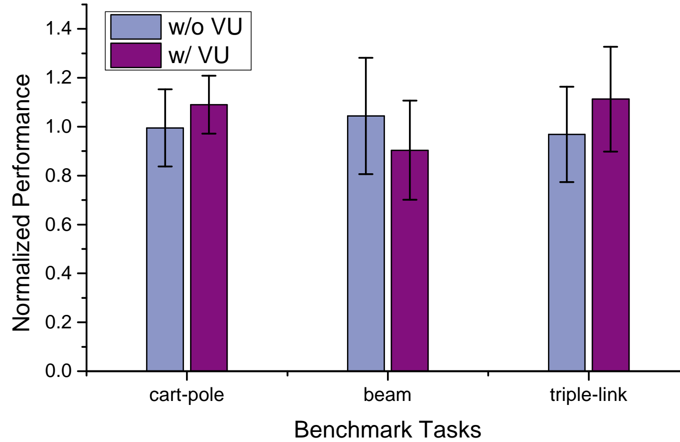


Figure 4.9: Comparison of the learning performances achieved by the accelerators and the software approach for three commonly used benchmarks. The results obtained from the accelerators are normalized to those obtained from software. The error bars correspond to a confidence interval of 95%.

simulation results.

Two accelerators are implemented. One implementation is equiped with the proposed virtual update algorithm, whereas another one is the baseline design with the conventional update. Both chips have similar chip layouts. Therefore, only the one with virtual update is shown in Fig. 4.8 for brevity. On-chip memories, including synapse memory, neuron memory, and input buffers, take most of the space. The arithmetic unit, which contains multipliers, adders, and the activation block, is the second largest block. The controller and scheduler occupy the rest of the area. Both the baseline and the upgraded accelerators are designed to operate at a clock frequency of 175 MHz. Such a clock frequency is more than enough for the accelerators to perform all the benchmark tasks that are discussed in this work in real time. In order to evaluate the performance of the accelerators in conducting reinforcement-learning tasks, the three most common control benchmarks used in Section 4.3 are employed. The same metric, accumulated time steps, is used for comparison. The performances achieved by the accelerators are normalized with respect to the performance achieved by the software approach implemented on a general-purpose processor. The obtained

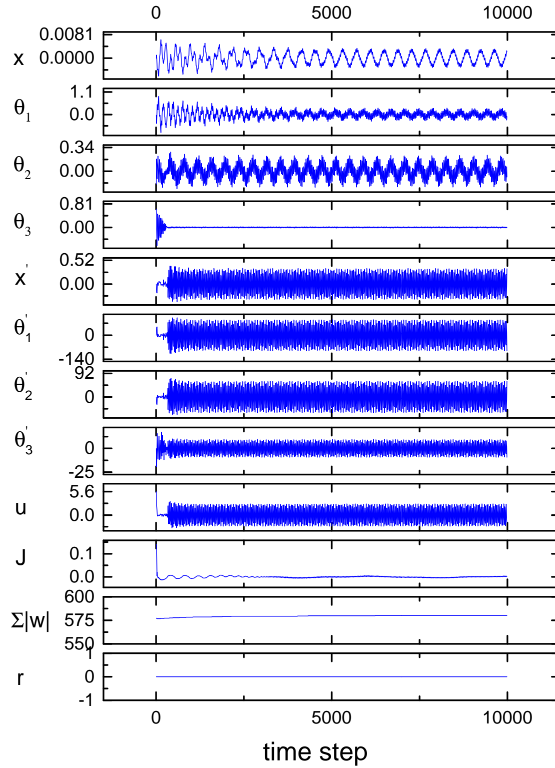


Figure 4.10: Typical waveforms obtained in the triple-link inverted pendulum task with the baseline accelerator. In the figure, the unit for distances and angles are meter and degree, respectively.

results are shown in Fig. 4.9. As the virtual update algorithm does not use any approximation or assumption in the computation, the obtained results should be the same with the baseline design when quantization is absent. Nevertheless, there are slight differences in the computed results, which are caused by the different orderings of quantization. The results in the figure are obtained from the behavior-level model of the chip. Mathematical models used for simulating these benchmark tasks can be found in [90, 91, 97, 99, 100, 101], and they are omitted here for brevity. As shown in the figure, the accelerators are able to achieve a similar performance compared to the processor that computes with double-precision floating-point numbers.

To provide more insight into how well the accelerator performs on a complicated task, one set of typical waveforms obtained from the triple-link task for successful learning is demonstrated in Fig. 4.10. In the figure, x, θ_1 - θ_3 and their corresponding

derivatives x' and θ_1' - θ_3' are eight state variables that the ADP accelerator observes, u is the applied control voltage, J is the estimated reward-to-go, $\Sigma|w|$ is the sum of absolute values of all weights, and r is the reward signal, which is -1 if the states of the plant exceed the target range. Initial conditions for the plants are set as the following. x and x' are initialized as zero. $\theta_1 - \theta_3$ and $\theta_1' - \theta_3'$ are initialized randomly. They obey uniform distributions $U[-1°, 1°]$ and $U[-0.5°/s, 0.5°/s]$, respectively. The target is to control x to be within the range of $[-1\text{m}, 1\text{m}]$ and to maintain θ_1 - θ_3 in the range $[-20°, 20°]$ while the applied voltage is bounded by $\pm 30V$. More detailed information on the triple-link inverted pendulum balancing task can be found in [90, 97]. It is demonstrated in Fig. 4.10 that the accelerator successfully learns the control policy and maintains the states of the system well within the target range.

To demonstrate the efficacy of the proposed virtual update technique, Fig. 4.11(a)-Fig. 4.11(c) compare the number of clock cycles, power consumption, and energy efficiency of the accelerators with and without the proposed technique. Different sizes of neural networks and control tasks are examined. In the figures, results with the labels "4-6-1, 5-6-1" and "4-10-1, 5-12-1" are obtained from the cart-pole task, where the two sets of numbers refer to the sizes of the critic and actor networks, respectively. The results with the label "8-20-1, 9-20-1" are obtained from the triple-link inverted pendulum task. In addition, specifications of the proposed ADHDP accelerator are summarized in Table 4.2.

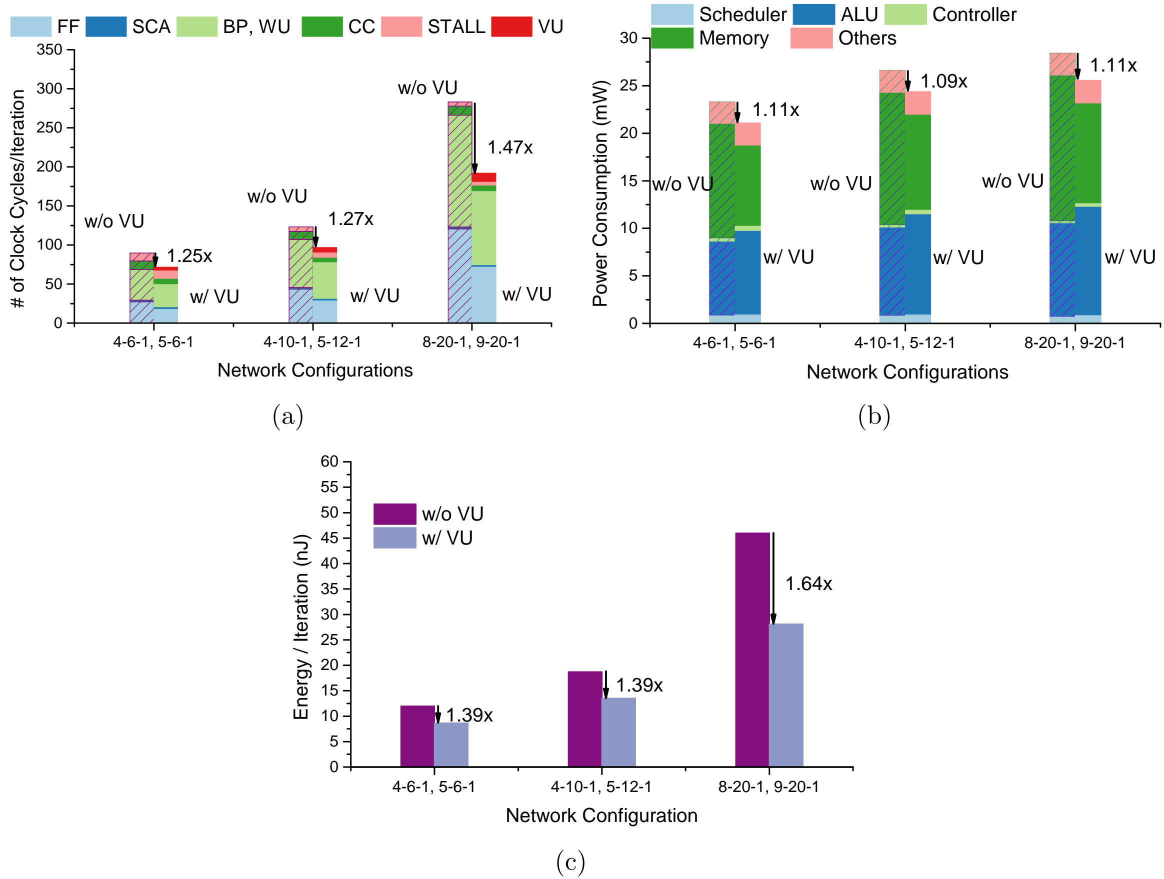
Table 4.2: Specifications of the ADHDP accelerator

Technology	TSMC 65nm
Area	550 μm $\times$ 550 μm
Number of lanes	4
Arithmetic precision	24-bit fixed-point
Supply voltage	1.2 V
Clock frequency	175MHz
Power consumption	25 mW

The normalized number of clock cycle breakdowns are compared in Fig. 4.11(a).

One trend that can be observed for both the baseline accelerator and the accelerator with the virtual update is that the forward and backward operations occupy most of the clock cycles and the percentages that these two operations occupy increase as the size of the neural networks becomes large. Indeed, as the size of the neural network increases, the number of operations that can flow through the pipeline without being interrupted by the control or branch operation increases. The proposed virtual update algorithm effectively shortens the number of clock cycles needed for each task. The improvement grows as the size of the problem increases. A 1.47 times improvement is achieved for the triple-link inverted pendulum benchmark task. The main reason for the growing improvement is that the virtual update algorithm effectively replaces quadratically-scaled operations with linearly-scaled operations. Therefore, the savings in the number of clock cycles increases with the size of the problem. To give a comparison between the accelerators presented in this chapter and a software running on a general-purpose processor, the ADP algorithm is programmed and is run on an Intel Xeon processor. On average, the accelerator is two orders of magnitude faster than the software approach.

Fig. 4.11(b) compares the power consumption of the two accelerators. The accelerator with virtual update has a slightly lower power consumption compared to the baseline design, thanks to many fewer memory operations, as illustrated in the figure. Another observation made from Fig. 4.11(b) is that the virtual update tends to increase the power consumption in the arithmetic unit. This can be attributed to two reasons. The first reason is that the virtual update increases the utilization rate of the arithmetic unit. Another reason is that multiplexers are added in the arithmetic unit to allow for more operations needed in the virtual update algorithm, which contributes to the additional power. The increased power consumption in the arithmetic unit is offset by the savings in memory operations, resulting in a net savings in the power consumption. The energy efficiencies are compared in Fig. 4.11(c). Through



(a)

(b)

(c)

Figure 4.11: Comparison of (a) the numbers of clock cycles needed for every critic/actor update iteration, (b) the power consumption, and (c) the energy consumption for every critic/actor update iteration for the ADP accelerators with and without the virtual update technique. The first two groups of data are obtained from the cart-pole balancing task, whereas the third group of data is obtained from the triple-link inverted pendulum task.

accumulating the improvements in both the number of clock cycles per iteration and the power consumption, energy efficiency of the accelerator with the virtual update technique has been improved as many as 1.64 times for the triple-link inverted pendulum task. Again, as the size of the network increases, the improvement in the energy efficiency grows.

4.6 Chapter Summary

In this chapter, we present a hardware architecture for ADHDP accelerators. Through leveraging the data-level parallelism and data locality, scalable and programmable accelerators with high throughput and high energy efficiency are demonstrated. In addition, to exploit the iterative nature of the ADP algorithm, a virtual update technique is proposed to skip unnecessary computations, improving the throughput and power consumption. We demonstrate two design examples that are with and without the proposed technique. Extensive simulations are conducted to demonstrate the efficacy of the design strategies and techniques. It is shown that the proposed virtual update algorithm can effectively improve the energy efficiency of the accelerator by a factor of 1.64 for the most complicated benchmark task we employ. Such a good energy efficiency and high throughput open the door for complicated ADP algorithms to be deployed in various energy-constraint applications where optimal decision-making or control are needed.

CHAPTER V

Memory Reliability Enhancement Through On-Chip Compensation and Error Correction

5.1 Introduction

In addition to the algorithm- and architecture-level optimizations and techniques that are employed in previous chapters in order to reduce the power consumption needed in neuromorphic computing, circuit-level low-power design techniques can also be very effective. The most popular low-power circuit-level strategy is the low-voltage design, considering the dynamic power consumption, which is the dominant source of power consumption in most CMOS digital systems, scales quadratically with the supply voltage. This is illustrated in Fig. 5.1. Even though the leakage power sets a lower bound on how low the supply voltage can drop, the voltage can be reduced to a subthreshold or a near-threshold voltage to boost the energy efficiency remarkably.

Subthreshold circuit design has attracted many researchers' attentions in recent years thanks to many emerging applications such as wireless sensor networks and the IoT technology. Many ultra-low-power subthreshold CMOS systems have been demonstrated recently with significantly improved energy efficiencies compared to their superthreshold counterparts. Subthreshold logic design is relatively straightforward. The design process is essentially an optimization task that involves the

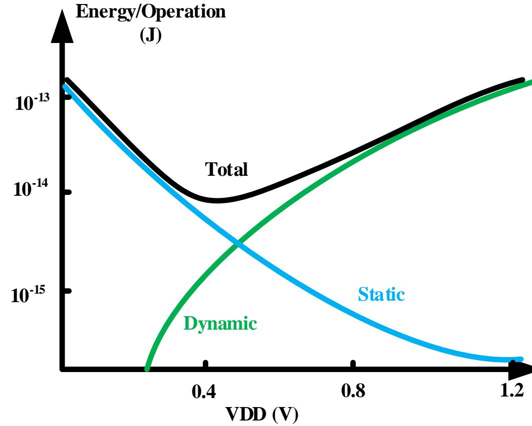


Figure 5.1: Illustration of the energy per operation versus the supply voltage.

power-delay trade-off. On-chip subthreshold memory design, however, are much more challenging. Despite the success in reducing the power consumption by operating circuits in the subthreshold region, reliability becomes the most serious issue. The reduced supply voltage shrinks both the write and read noise margin significantly. A more problematic issue is the serious degradation in the worst-case read stability and writability because of global and local process variations.

Another requirement for the memory in a neural network hardware is a high integration density. The size of the memory required by neuromohrphic computing is enormous even for a moderate-size task. For different tasks, synaptic weights obtained from learning need to be stored in the non-volatile memory when not being used. In the future, it is expected that both the size of the problem and the number of problems to which neuromorphic computing are applicable will grow rapidly. As a reference, it is estimated that there are around 10^{14} synapses in a human brain. Even if we treat each synapse as a 1-bit memory cell, the capacity of a human brain is in the order of 100 Tb! Although the storage capacity of artificial neuromorphic hardware might not need to be as high as a human brain, large storage spaces can lead to much more powerful neuromorphic hardware. Instead of performing a single simple

task, the hardware with more storage capacity is able to process multiple complex inference and control tasks with human-level intelligence. One solution to avoid the large storage requirement is cloud storage or cloud computing. However, storing the synaptic weights or conducting the inference in the cloud might not be feasible for many applications where the overhead of communication is large or low-latency inference is needed. Therefore, high-density and low-power non-volatile memories are critical components for future low-power neuromorphic computing.

The growing storage demands increasingly push the advance of flash memories. Higher and higher density is achieved through technology scaling, multi-level storage, as well as migration from a 2-D flash array to 3-D flash. All this technology advancement, however, is achieved by sacrificing the reliability of the memory. Despite the high density that modern memory systems can achieve, reliability becomes a major problem [54]. In addition to the mature flash memories, many emerging memory technologies also provide new opportunities for more aggressive memory scaling. With the advent of the memristor technology, the crossbar-based memristor memory becomes more and more attractive as an alternative high-density non-volatile memory [108][109]. Despite its high density and high durability, the memristor crossbar suffers from problems such as stochastic variations [110][111] and sneak path [112][113][114], which jeopardize the reliability of the memory.

In order to tackle the reliability issue existing in the memory system of neuromorphic hardware, two techniques are introduced in this chapter. The first technique utilizes an on-chip closed-loop compensation to mitigate the global variation, increasing the reliability of the on-chip SRAM memory [115]. The second technique relies on a more efficient error-correction method that can be used to improve the error resilience of the non-volatile memory [116].

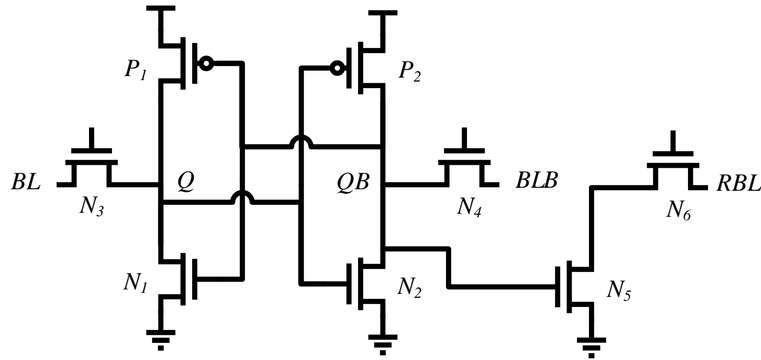


Figure 5.2: Schematic of an 8T SRAM cell.

5.2 Counteract Variations in SRAM

In this section, how to improve the reliability of on-chip SRAM is discussed. First, the static noise margin (SNM), the most important metric used to measure the stability of an SRAM cell, is modeled semi-analytically, providing a fast and direct way to analyze and estimate the reliability of the SRAM cells. An on-chip compensation method is then introduced to mitigate the degraded stability of SRAM cells.

5.2.1 Modeling of Variations in SNM

Characterizing variation in cell stability is necessary, considering the large number of cells involved in an SRAM system. This is even more critical for SRAM systems operating in the subthreshold region because of the larger variations in static noise margin (SNM). Monte Carlo (MC) simulations are often carried out to capture the worst-case read stability. However, MC simulations for a large array are often computationally prohibitive. Importance sampling [117][118] was proposed as a powerful technique to characterize the variation of SNM. By transforming the sampling density function, MC simulations based on importance sampling are much more efficient than

traditional full MC simulations. Nevertheless, many samples are still needed. Therefore, an accurate statistical model in an analytical or a semi-analytical form is able to help designers estimate the performance of the SRAM under process variation in the early design phase without resorting to time-consuming MC simulations repeatedly.

Fig. 5.2 shows the typical configuration of an 8T SRAM cell. In the figure, the two cell inverters consisting of N_1, P_1 and N_2, P_2 are of interest. N_3-N_6 can be omitted for the purpose of analyzing the read (hold) SNM except that one has to be cautious about the systematic mismatch introduced by N_5 and N_6. The voltage transfer characteristic (VTC) of an inverter is an important tool to analyze the stability of an SRAM cell. Most voltage-based SNMs can be calculated if the VTCs of cell inverters are known. The drain current of an NMOS transistor operating in the subthreshold region can be modeled as shown in (5.1) [119].

$$I = I_0 \frac{W}{L} e^{\frac{V_{GS}-V_{th}}{nU_T}} \left(1 - e^{-\frac{V_{DS}}{U_T}}\right) \tag{5.1}$$

where V_{th} is the threshold voltage, U_T is the thermal voltage, n is the subthreshold factor, W and L are width and length of the transistor, and I_0 is a technology-dependent fitting parameter.

For a CMOS inverter, let us define a quantity, called transition voltage V_T, as the input voltage that drives the output of the inverter to $V_{DD}/2$. Mathematically, this can be written as $VTC(V_T) = V_{DD}/2$ where $VTC(\cdot)$ is the voltage transfer characteristic of the inverter. It is well known that the variation of threshold voltage, which is mainly caused by the random dopant fluctuation (RDF), is the dominant source of the variation in cell stability in a subthreshold memory system [120][121]. This is because the exponential dependence of current on threshold voltage dilutes the impacts brought by other sources. Therefore, only the variation in the threshold voltage is modeled in this chapter. Threshold voltages are typically modeled as

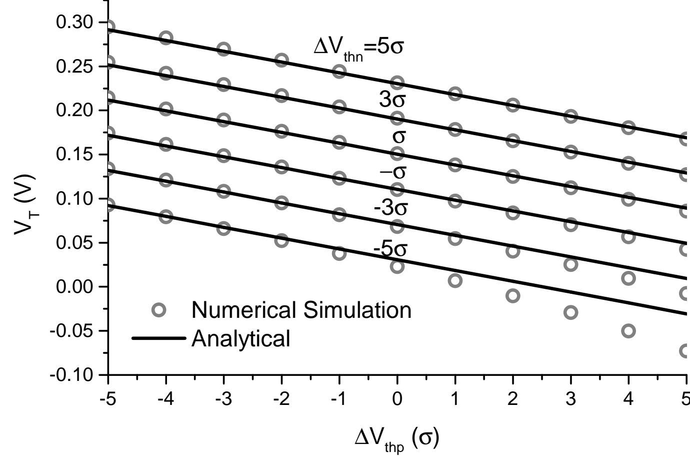


Figure 5.3: Transition voltages of an SRAM cell inverter as the threshold voltages of the NMOS and PMOS transistor vary. The solid line is obtained with (5.2), whereas the circles are obtained from the circuit simulation tool.

independent, normally distributed random variables with standard deviations of the form $A_{V_{th}}/\sqrt{WL}$ [122], where $A_{V_{th}}$ is a technology-dependent constant. It can be shown that the transition voltage of a cell inverter obeys a normal distribution with a standard deviation of

$$\sigma\left(V_T\right)=\sqrt{\left(\frac{n_p}{n_n+n_p}\right)^2\frac{A_{V_{thn}}^2}{W_nL_n}+\left(\frac{n_n}{n_n+n_p}\right)^2\frac{A_{V_{thp}}^2}{W_pL_p}} \tag{5.2}$$

The transition voltage obtained from (5.2) and the one from the simulation tool are compared in Fig. 5.3. In this chapter, a 65-nm technology is employed in the simulation for demonstration purposes and all simulation results are obtained with the Cadence Spectre. The threshold voltages of both NMOS and PMOS transistors vary from -5σ to 5σ, corresponding to the worst variation of $5\sqrt{2}\sigma$. The analytical results match well with the numerical results except for the region where V_T is negative or close to zero. This discrepancy, however, is forgivable as a near- or sub-zero V_T implies an unreliable cell that cannot hold the data. In practice, most SRAM designs have the objective of controlling the worst-case SNM to be larger than zero with some margin.

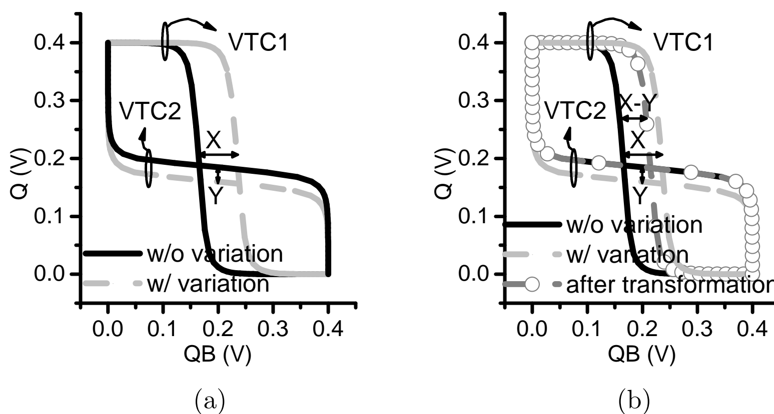


Figure 5.4: Illustration of (a) two VTCs in the nominal case and two VTCs under process variation, and (b) VTCs under variation that are shifted by the same amount such that VTC2 is moved back to its original location.

In the noise margin calculation, two single-sided SNMs can be found, which correspond to two lobes of the butterfly curve. The double-sided SNM of an SRAM cell, which characterizes the worst-case noise margin, is defined as the minimum of these two single-sided SNMs. For analysis purposes, results for a single-sided SNM called SNM_S are developed first. The results will then be used to find the double-sided SNM called SNM_D. In order to find the single-sided SNM variation, a mapping function $g(x, y)$ is needed such that

$$M = g(X, Y) \tag{5.3}$$

where X and Y are random variables representing variations in the transition voltages of the two VTCs used to calculate SNM and M is the single-sided SNM we are interested in.

It is difficult to get an analytical expression for $g(\cdot)$ accurately. One alternative way is to obtain $g(\cdot)$ numerically by running through different x and y. Careful examination of the definition of SNM, however, reveals that a function $g(\cdot)$ with one argument is sufficient for the mapping. Suppose the variation in transistors pushes

VTCs of the inverters away from their original places by the amount of X and Y, as shown in Fig. 5.4(a). Then, the change in V_T is also X and Y. By shifting both VTC1 and VTC2 by $-Y$, VTC2 returns to its nominal place as shown in Fig. 5.4(b). Therefore, the single-sided SNM can be obtained as

$$M = g(S) + Y \tag{5.4}$$

where

$$S = X - Y \tag{5.5}$$

Therefore, only a one-argument function $g(s)$ is needed. Function $g(s)$ can be obtained by keeping the VTC of one inverter fixed and shifting the VTC of another inverter for different s. The obtained single-sided SNM is the value of $g(s)$.

To obtain an analytical expression for the distribution of a single-sided SNM, another random variable is defined as

$$T = g(X - Y) \tag{5.6}$$

Without loss of generality, let us focus on the single-sided noise margin corresponding to the upper-left lobe of the butterfly curve. $g(s)$ is a strictly increasing function in this case. Through the change of variable [123], the probability density function (PDF) of M can be obtained as

$$f_M(m) = \int_{-\infty}^{+\infty} \frac{f_X\left(g^{-1}(t) + m - t\right) f_Y(m - t)}{g'\left(g^{-1}(t)\right)} dt \tag{5.7}$$

A typical $g(s)$ is plotted in Fig. 5.5. There are three regions in $g(s)$. When s is small, SNM_S is proportional to s with a slope approximately equal to 1, because this is the region where VTC1 constrains SNM_S. When s is large, SNM_S is proportional to s with a slope approximately equal to 0 because this is the region in which VTC2

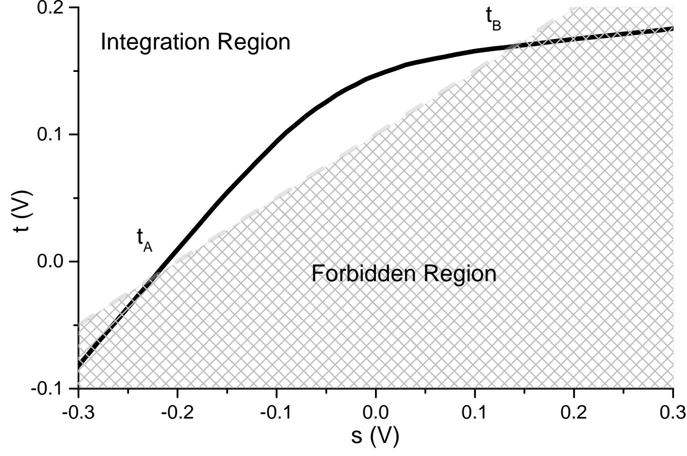


Figure 5.5: A typical mapping function.

constrains SNM_S. Based on the nature of this curve, it is convenient to do a piecewise linear fitting on $g(s)$ such that

$$g(s) \approx k_i s + a_i \tag{5.8}$$

for

$$s \in (s_i, s_{i+1}), i = 1, 2, \cdots, N \tag{5.9}$$

where (s_i, t_i) are knots for the piecewise linear fitting.

Then, (5.7) can be approximately written as

$$f_M(m) \approx \sum_{i=1}^{N} \int_{t_i}^{t_{i+1}} \frac{1}{k_i} f_X \left(\frac{t - a_i}{k_i} + m - t \right) f_Y(m - t) dt \tag{5.10}$$

As X and Y are normally distributed random variables with a mean of zero and standard deviations shown in (5.2), the integral in (5.10) can be analytically derived and the final expression is

$$f_M(m) \approx \sum_{i=1}^{N} [C_i(m, t_{i+1}) - C_i(m, t_i)] \, \phi \left(\frac{m - a_i}{\overline{\sigma_i}} \right) \tag{5.11}$$

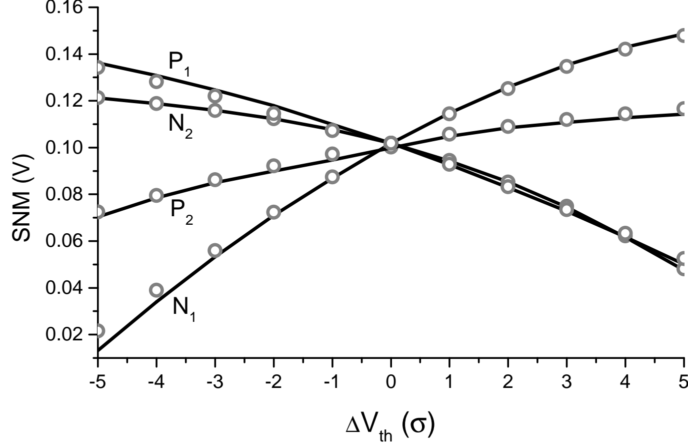


Figure 5.6: Comparison of the sensitivities obtained from $g(s)$ and the circuit simulation tool. Two sets of results match well, demonstrating that $g(s)$ can be employed for estimating the SNM.

where

$$C_i(m, t_i) = \Phi\left(\frac{(t-m)\overline{\sigma_i}^2 + (a_i - m)(k_i - 1)\sigma_Y^2}{k_i \sigma_X \sigma_Y \overline{\sigma_i}}\right) \tag{5.12}$$

$$\overline{\sigma_i} = \sqrt{k_i^2 \sigma_X^2 + (k_i^2 - 1)^2 \sigma_Y^2} \tag{5.13}$$

In the equations, $\phi(\cdot)$ and $\Phi(\cdot)$ are the PDF and the cumulative density function (CDF) of the normal distribution.

It is noted that all the information needed for calculating the SNM is contained in $g(s)$, given (5.2) is held. To illustrate this, SNM calculated based on (5.2) and (5.4) are compared with the results obtained directly from the simulation tool in Fig. 5.6. Two sets of results match well. Compared to the conventional linear model, the proposed method requires similar or even less amounts of circuit simulations to model the SNM variations. For example, 41 circuit simulations are necessary to generate 41 VTCs for points in Fig. 5.6 in order to utilize the linear model, whereas only one circuit simulation is needed for the proposed algorithm to generate the VTC in the nominal case. For SRAM cells operating in the superthreshold region, it is generally assumed that the total SNM variation is a linear combination of SNM variations caused by each transistor. Therefore, the slope of each curve at $\Delta V_{th} = 0$ in Fig. 5.6

is employed as the sensitivity of the SNM with respect to the threshold voltage of that transistor. Correctness of this approach, however, is challenged in [120], as it is found that sensitivity curves shown in Fig. 5.6 are dependent on each other. One observation from (5.11) is that the PDF of a single-sided SNM is the sum of several modulated Gaussian PDFs, meaning that the distribution of SNM_S is not necessarily normal. In the case where the variation of the transition voltage, which is ultimately determined by the variation in threshold voltages, is not large, the variation of s is kept local. $g(s)$, in this case, can be represented by one straight line. Consequently, (5.11) can be simplified to

$$f_M(m) \approx \phi\left(\frac{m - m_0}{\overline{\sigma}}\right) \tag{5.14}$$

where

$$\overline{\sigma} = \sqrt{k^2\sigma_X^2 + \left(k^2 - 1\right)^2 \sigma_Y^2} \tag{5.15}$$

and m_0 is the SNM in the nominal case and k is the localized slope of $g(\cdot)$.

Equation (5.14) shows that the distribution of a single-sided SNM is normal when the variation is not too large, which is the assumption that is widely used in the literature. Consequently, the conventional way of representing the total SNM variation as a linear combination of SNM variations caused by transistors is an acceptable approximation. However, when the variation is large, the distribution starts deviating from a Gaussian one. This observation is particularly useful because the variation of the threshold voltage is growing as the technology advances, especially considering that the transistors used for SRAM cells are usually with minimum sizes.

The simulation results of single-sided SNMs obtained from (5.11) are compared with 20000-run MC simulations results obtained from the circuit simulation tool in Fig. 5.7(a). To increase the efficiency of the MC simulations, an importance sampling with 1.5 times larger standard deviation is employed. One nominal-case circuit sim-

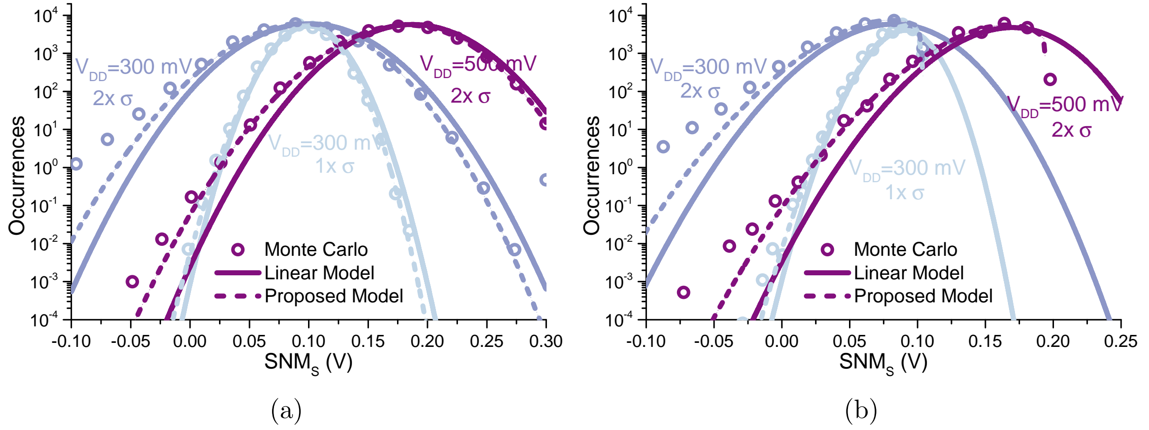


Figure 5.7: Comparison of the distributions of (a) the single-sided SNMs and (b) the double-sided SNMs obtained from the conventional linear superposition model, proposed model, and MC simulations. The results obtained with exaggerated (2x) variations are also plotted to show the trend as variations grow in advanced technologies.

ulation is needed to help build the mapping function. The shifting of the VTC and the calculation of SNM_S are conducted in a programming tool. (5.2) is employed to obtain the values of σ_X and σ_Y. For comparison purposes, the SNM distribution obtained with the conventional superposition method [124][125][126] are also plotted. To demonstrate the trend as the variation in threshold voltage increases, the results obtained with doubled variations are also plotted. In Fig. 5.7(a), the distributions obtained from the MC simulations match well with our proposed model for the region where SNM is positive. When the SNM is negative, the distribution obtained from the MC simulations start deviating from the proposed model as a consequence of the discrepancy between the estimated V_T and the measured V_T shown in Fig. 5.3. However, this estimation error when SNM is negative (or close to zero), again, is forgivable as one rarely cares about SNM that is close to zero or even negative. To illustrate this point, the same SRAM array with a supply voltage of 0.5 V is also simulated. This is a more realistic supply voltage for an SRAM array with such a large threshold voltage variations. In this case, the proposed estimation method accurately predicts the distribution of the SNM. There exists a discrepancy between the proposed model

and the conventional linear superposition model, and this discrepancy grows as the variation increases.

After obtaining the distribution of SNM_S, the tail of the PDF of the double-sided SNM can be estimated by [120]

$$f(SNM_D) = 2f_{SNM_S}(1 - F_{SNM_S}) \tag{5.16}$$

The exact distribution of SNM_D can actually be derived with the help of an inequality

$$g^{-1}(T) - 2T + 2M < 0 \tag{5.17}$$

This inequality corresponds to the unshaded region shown in Fig. 5.5. Suppose the two intersects of the mapping function and the boundary of the region corresponding to (5.17) are t_A and t_B. Then the PDF of the double-sided SNM is given by

$$f_{SNM_D}(m) = \int_{t_A}^{t_B} \frac{f_X\left(g^{-1}(t) + m - t\right) f_Y(m - t)}{g'\left(g^{-1}(t)\right)} dt \tag{5.18}$$

Once (5.18) is obtained, the piecewise linear fitting technique described before can be used to calculate the PDF of the double-sided SNM. Equations similar to (5.11)-(5.13) can be obtained except, in this case, the sum only runs through knots between t_A and t_B. One thing should be noted is that t_A and t_B are functions of m. Therefore, they have to be calculated for each m that is of interest.

The distributions of the double-sided SNMs obtained from the model in (5.18) are compared with the results estimated from the conventional linear model and MC simulations, as shown in Fig. 5.7(b). The proposed model predicts the whole distribution well, whereas the conventional model can only predict the tail with a limited accuracy. To show the trend of increasingly worsened variations in advanced technologies, the results obtained with twice as many variations as the employed

technology (65 nm) are also shown. A similar trend is observed in the double-sided SNMs as well: the proposed method can accurately predict positive SNMs. We also compare the results obtained with a supply voltage of 0.5 V. This is, again, a more realistic supply voltage for such serious threshold voltage variations. As the variation gets worse, the discrepancies between the results obtained from the MC simulations and the conventional linear superposition method increase, whereas the proposed method matches well with the MC simulation for regions where SNMs are positive.

5.2.2 Adaptive Body Biasing

To counteract threshold voltage variations, one straightforward way is to increase the transistor sizing in a cell [121]. This way of remedy, however, introduces a large area penalty. Mismatches in transistors induced by the RDF are uncorrelated [122]. Therefore, compensation for local variation is not feasible. The global variation, however, can be sensed and compensated accordingly. To counteract performance and yield degradation caused by the variation, body biasing is introduced as an effective and efficient technique. In [127], an operational amplifier is employed to adjust the body bias. This method, however, has at least three disadvantages. First, an op-amp with a wide input common mode range and a rail-to-rail output is hard to build under subthreshold supply voltages. Second, a current source, a block that is often absent in SRAM, is needed. Third, the offset of the employed op-amp might introduce a significant error. In [128], two inverters are employed as error amplifiers and feedback is used to correct the body bias. However, the stability of the feedback system is not discussed and the circuit only supports a balanced P-N ratio. In an SRAM cell, however, a stronger NMOS is often desired for a more robust write operation. In [129], an open-loop system is employed to generate a proper bias. However, considering process, voltage, and temperature variations, a feedback system is more suitable. In this section, an adaptive body biasing circuit is proposed to compensate for the global

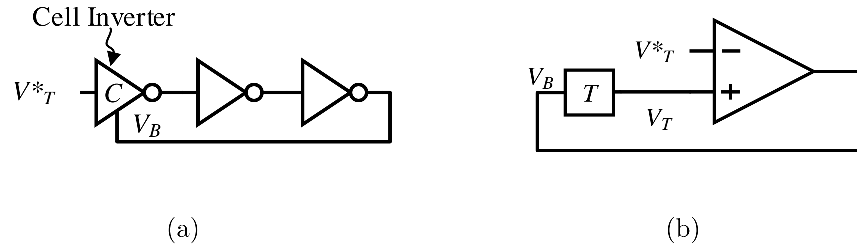


Figure 5.8: (a) Conceptual diagram of the proposed adaptive body biasing circuit. (b) Simplified equivalent circuit of (a).

variation. Simple circuit structures are used to achieve a negligible area and power consumption overhead. A stabilization technique inspired by the ring amplifier is introduced to stabilize the feedback loop.

To mitigate the threshold voltage variation, an adaptive body biasing technique is introduced in this section. A simple yet powerful body bias generation circuit is proposed, which senses global threshold voltage variations and then adjusts the body bias of SRAM cells for compensation. The circuit conceptually consists of only inverters, as shown in Fig. 5.8(a). To better understand how the proposed circuit functions, its simplified equivalent circuit is shown in Fig. 5.8(b). The first-stage sensing inverter senses and amplifies the difference between the desired transition voltage V_T^* and the actual transition voltage V_T. Two other inverters are employed as an error amplifier to further amplify the error. The error signal is then fed back to the cell inverter for a feedback control. The block T shown in the figure is the transfer function that models the relationship between the bias voltage of the cell inverter and the transition voltage of that inverter.

In this section, we use the body bias of PMOS transistors as an example, as this type of body biasing does not require any special option such as the triple-well process. Nevertheless, the proposed technique can be readily applied to the body biasing of NMOS transistors. Two examples of the proposed biasing circuit are shown in Fig. 5.9(a) and Fig. 5.9(b). The circuit shown in Fig. 5.9(b) requires an additional

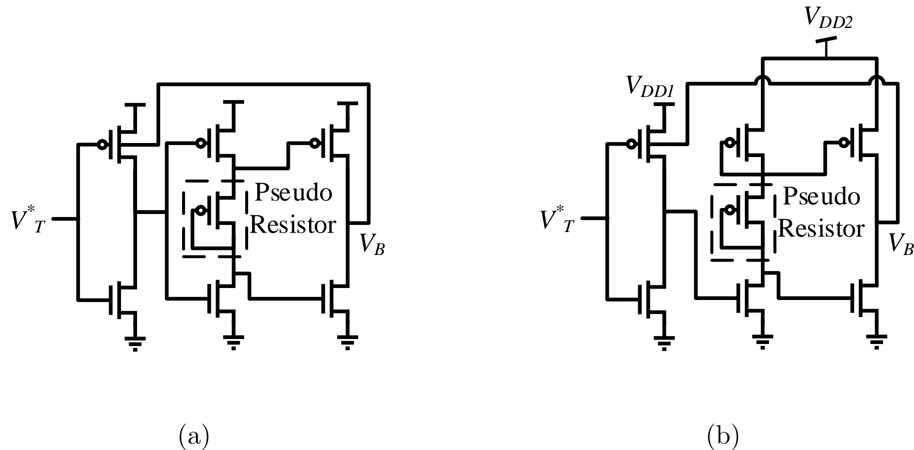


Figure 5.9: Configuration of the proposed adaptive body biasing circuit. (a) One example of generating body bias to mitigate the read SNM degradation when the global variation is present. V_T^* is the desired transition voltage. (b) Another example of generating body bias to mitigate the read SNM degradation. In the figure, $V_{DD2} > V_{DD1}$.

supply voltage so that the tuning range of the body bias can be enlarged. Despite its simplicity, one thing that needs to be ensured for the proposed adaptive biasing circuit is the stability. Inspired by the stabilization technique employed in a ring amplifier [130], an offset is introduced by the pseudo-resistor in Fig. 5.9(a) and Fig. 5.9(b) in order for the two transistors at the output stage to conduct a small amount of current in the steady state. Therefore, a dominant pole is formed at the output stage, which helps stabilize the feedback circuit. In addition, the output of the biasing circuit is connected to bodies of many transistors in an SRAM array. The parasitic capacitance and body leakage current also help stabilize the loop.

5.2.3 Overheads

Fig. 5.10 shows a diagram of how the proposed body biasing circuit can be included in an SRAM array. The layout of the sensing inverters should resemble inverters in an SRAM cell to reduce systematic mismatch. One possible way of achieving

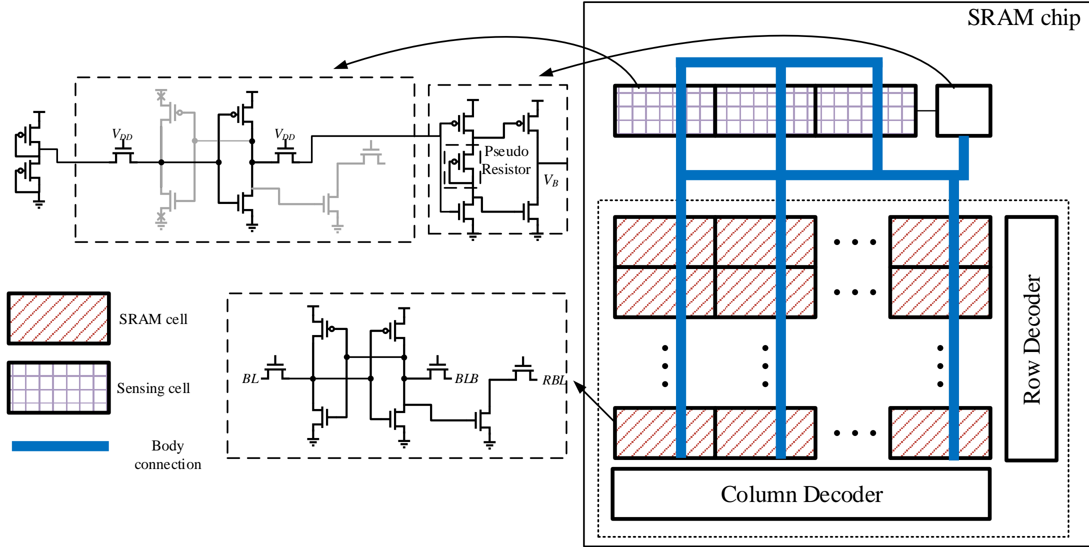


Figure 5.10: Configuration of the proposed adaptive body biasing circuit in an SRAM array.

this similarity is to have the same device placement and interconnect routing in the sensing cells as those in an SRAM cell. Contacts, however, are used differently to form the connection needed in a sensing inverter, as shown in Fig. 5.10. Furthermore, in order for the sensing inverters to accurately capture the global threshold voltage, the number of sensing cells, K, has to be large enough to overcome the local variation. If we assume the biasing circuit has a large enough gain to bias the transition voltage of the sensing inverter to the desired voltage, then the variation (both global and local) of the transition voltage of an SRAM cell inverter with the adaptive biasing is

$$\sigma'(V_T) = \sqrt{(\sigma^2_{sense}(V_T)) + \sigma^2(V_T)} \tag{5.19}$$

where $\sigma_{sense}(V_T) = \sigma(V_T)/\sqrt{K}$

The above result is obtained by assuming there is no spatial correlation in the threshold voltage variation. This assumption is supported by [122], where no noticeable spatial correlation in threshold voltages was observed. To suppress the variation

introduced by the sensing cell inverters, we require that $\sigma_{sense}(V_T) \ll \sigma(V_T)$. According to (5.19), a K of 16 is enough to effectively control the variation introduced by the inaccurate sensing to be approximately 3% of the local mismatch in the target technology.

The second stage of the proposed circuit splits input from the sensing stage into two outputs with an offset. The pseudo-resistor employed also helps in tracking variations in supply voltage. The output stage is used to bias bodies of PMOS transistors. Both the huge amount of parasitic capacitance associated with body nodes and the leakage current from the body help in stabilizing the circuit. It is found in the implementation that a phase margin of around 90 degrees can be met once the output is applied to an array of SRAM.

Current flowing in the output stage should be designed larger than the junction leakage current such that a moderate gain can be kept to maintain the correct voltage value. The overhead, in terms of the power consumption, can be estimated as

$$\frac{I_{ABB}}{I_{DLeak}} = \frac{K}{2N} e^{\frac{V_{DD}}{2nU_T}} + \frac{BI_B}{I_D} \tag{5.20}$$

where K is the number of sensing inverters, and N is the number of SRAM cells, B is a design parameter, I_B is the junction leakage current, and I_D is the subthreshold leakage current. When the size of the SRAM array is small, the power consumption of the proposed adaptive biasing circuit is dominated by the sensing stage, corresponding to the term $\frac{K}{2N} e^{\frac{V_{DD}}{2nU_T}}$ in (5.20). This is because a moderate number of inverters are needed to overcome the local variation. For example, a rough estimation according to (5.20) is that 16 sensing inverters incur a power consumption that is less than 1% of the leakage power of an SRAM array with 40 K cells. The power consumption of sensing inverters is quickly diluted as the size of the SRAM array increases. When the size of the SRAM array grows large, power is mostly consumed by the output stage

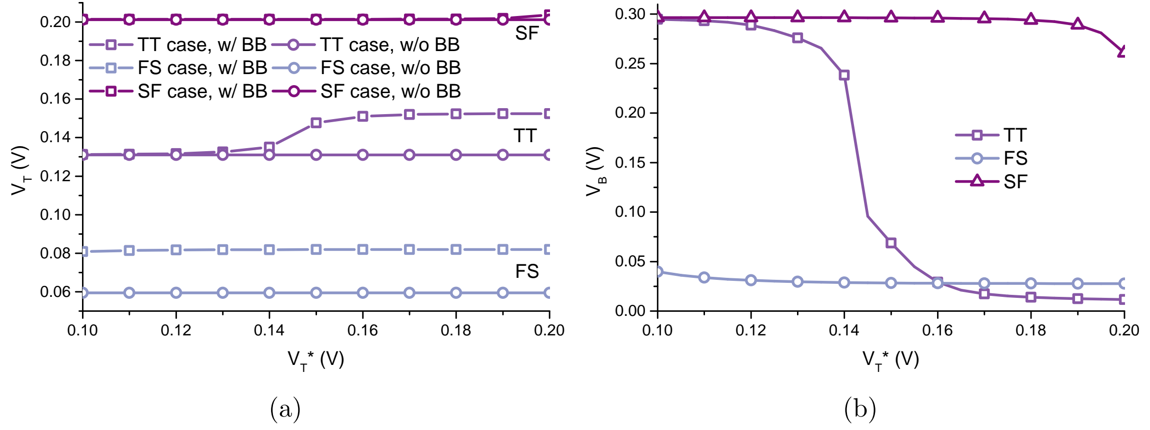


Figure 5.11: (a) Transition voltage and (b) body bias voltage of an SRAM cell inverter at different process corners with the help of the circuit shown in Fig. 5.9(a).

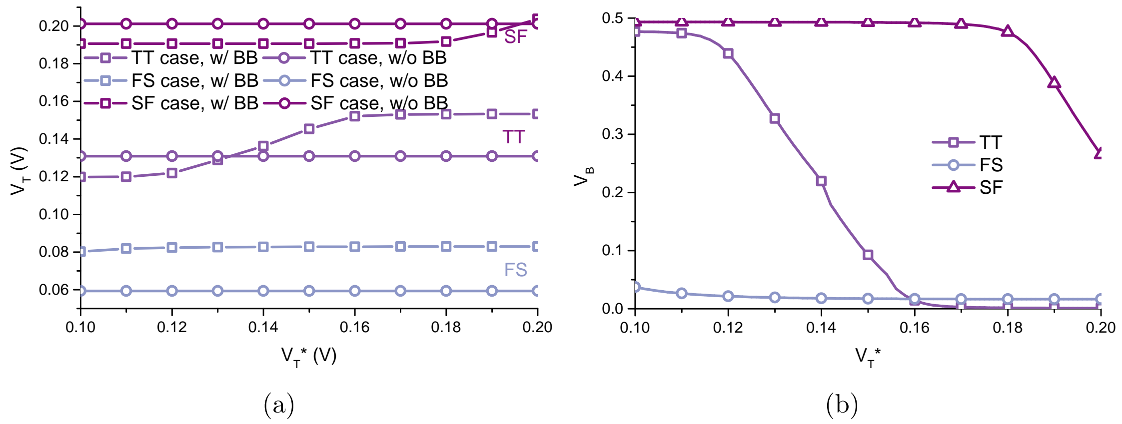


Figure 5.12: (a) Transition voltage and (b) body bias voltage of an SRAM cell inverter at different process corners with the help of the circuit shown in Fig. 5.9(b).

in order to counteract the body leakage. This power consumption scales linearly with the size of the SRAM array. The junction leakage in a transistor is normally much smaller than the subthreshold leakage [131], which keeps the term I_B/I_D small. It is found that the power consumption of the proposed biasing circuit is less than 0.4% and 1% of the leakage power of a 100 Kb SRAM under a supply voltage of 0.3 V for circuits shown in Fig. 5.9(a) and Fig. 5.9(b), respectively.

5.2.4 Implementation Examples

Simulations are carried out to demonstrate the effectiveness of the proposed body biasing circuit. Fig. 5.11(a) compares the transition voltage V_T of an SRAM cell inverter with and without the proposed body biasing. The comparison is conducted for various desired transition voltage V_T^* under different process corners. Fig. 5.11(b) shows how the corresponding body bias varies with V_T^* under different process corners. In the TT (NMOS typical/PMOS typical) case, V_T obtained with the proposed adaptive body bias follows the desired transition voltage V_T^* applied at the input of the biasing circuit. As V_T^* increases, the body bias signal V_B generated by the proposed circuit decreases to strengthen the PMOS transistor in order to shift the VTC towards the right. In this case, Fig. 5.11(b) shows that the body bias V_B can swing rail to rail. This is one advantage of the proposed body biasing scheme. In the FS case (NMOS fast/PMOS slow), the generated body bias stays at a low voltage, attempting to compensate for the weak PMOS transistor because of the global threshold voltage variation. Limited by the finite tuning range of the body bias, the actual V_T is not able to follow the desired V_T^* accurately. Nevertheless, the resultant V_T is closer to the desired value, leading to a less spreading V_T. In the SF corner (NMOS slow/ PMOS fast), the body bias signal is close to V_{DD}, leading to a V_T close to the uncompensated value. This is because the body bias is bounded by the supply voltage. In order to provide a more effective compensation for the SF corner, the circuit shown in Fig. 5.9(b), which has a wider tuning range, is simulated. The obtained results are compared in Fig. 5.12(a) and Fig. 5.12(b). The generated body bias and the obtained V_T in the FS corner are similar to those achieved in Fig. 5.11(a) and Fig. 5.11(b), as the lowest body bias available in both circuits are zero. In the TT case, however, the range where V_T follows V_T^* obviously enlarges, thanks to the larger tuning range of V_B, as illustrated in Fig. 5.12(b). In both the TT and SF corner, the body of the PMOS transistor can be reversely biased to weaken the PMOS transistor

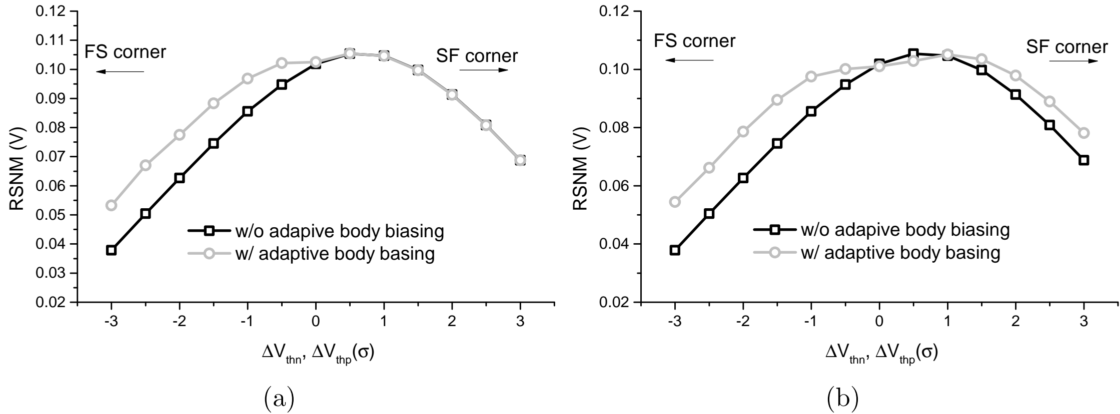


Figure 5.13: Comparison of the read SNM obtained with and without the proposed adaptive body biasing circuit as the threshold voltages of the PMOS transistor and the NMOS transistor vary. (a) The results are obtained with the circuit shown in Fig. 5.9(a). (b) The results are obtained with the circuit shown in Fig. 5.9(b)

that is stronger than desired because of process variations.

To study how the proposed adaptive body biasing circuit can be helpful in mitigating the degradation of SNM, which is caused by global process variations, Fig. 5.13(a) and Fig. 5.13(b) compare the double-sided SNMs of an SRAM cell with and without the proposed body bias scheme under different levels of variations. In the figures, variations in the threshold voltages of both NMOS and PMOS transistors are swept together to study the worst-case scenario. For example, in Fig. 5.13(a) and Fig. 5.13(b), the numbers on the x-axis represent the amount of variations ΔV_{thn} and ΔV_{thp}. As shown Fig. 5.13(a), the read SNM is improved wherever a body biasing is effective. A 15% improvement is achieved for the worst-case SNM with the proposed biasing technique. The percentage of improvement is normalized with respect to the nominal SNM. In Fig. 5.13(b), the SNMs at both the SF and FS corners are improved as the body bias voltage can swing both sides.

It is worth pointing out that even though the proposed adaptive biasing circuit is aimed at reducing SNM variation caused by global process variations, it can also be used for variations introduced by other sources, e.g. the negative bias temperature instability (NBTI) effect [132].

5.3 Improved One-Pass Chase Decoding of BCH Codes

To counteract the increasing unreliability introduced by the aggressive memory scaling, error corrections are often needed. Bose-Chaudhuri-Hocquenghem(BCH) codes are widely employed in flash memories to support a high memory throughput with low energy consumption [133]. During the normal operation, a hard-decision decoding of BCH code is used. When storage cells are damaged, the soft-decision BCH decoding may be activated to provide strong error-correcting performance. Chase decoding is a good candidate for this purpose because of its compatibility with a regular hard-decision BCH decoder and the ease of implementation. The conventional Chase decoding algorithm, however, is only practical when the number of extra error-correcting bits is few, as the computational complexity of the algorithm scales exponentially with the number of extra correcting bits. To extend the error-correcting capability of the Chase decoding algorithm, a one-pass Chase decoding method was proposed in [134]. The decoding complexity of this algorithm is significantly lowered, enabling the soft-decision Chase decoding with better error-correcting capabilities to be deployed in real applications.

To implement the one-pass Chase decoding algorithm, hardware architecture was discussed in [135]. Zeros in the error locator polynomial corresponding to the flipped bits were taken out to maintain the order of the polynomial, reducing the computational efforts. In that work, it was stated that most area and power consumption were spent on the highly parallel Chien search that runs as fast as the polynomial update block. An interpolation-based Chase decoding algorithm was developed in [136] to avoid the expensive parallel Chien search. A 2.3 times higher efficiency was reported in that work.

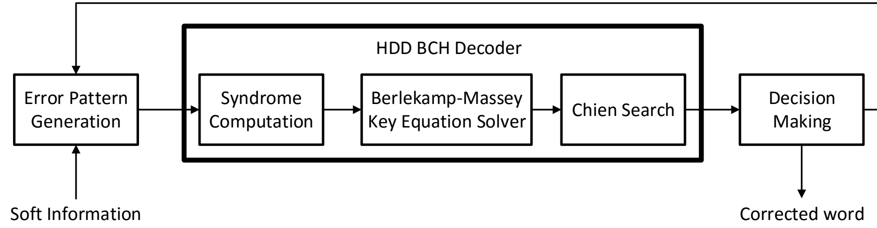
To better accommodate the one-pass Chase decoding algorithm, an eligibility verification algorithm is proposed in this chapter. The algorithm checks if the obtained error locator polynomial from the polynomial update algorithm or the Berlekamp's

algorithm is able to generate a correct error pattern such that the corrected code is a valid code word. The main motivation of the proposed algorithm is that an invalid polynomial can be easily detected without actually finding out all roots of that polynomial. The problem of checking the eligibility is first converted into a problem of calculating polynomial modulus. The calculation of the modulus can then be conveniently solved by repeated squaring. In addition, to further reduce the computational complexity, as well as the critical path delay in a hardware implementation, a polynomial inversion algorithm is proposed. An efficient hardware architecture is then proposed to build the algorithm. To verify the efficacy of the algorithm and hardware architecture, a design example is presented. Compared to the conventional exhaustive searching, the proposed algorithm achieves a reduction of 88% in gate counts.

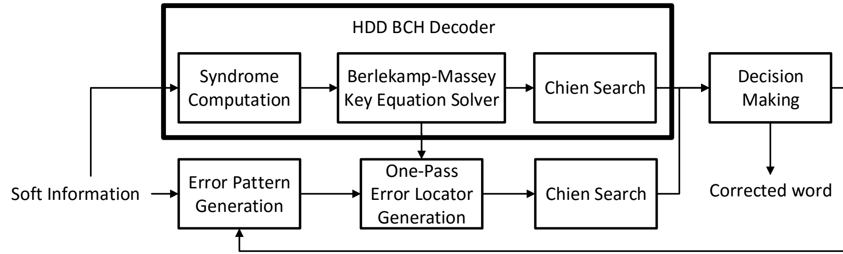
5.3.1 One-Pass Chase Decoding

The standard way of decoding a BCH code is through the Berlekamp's algorithm. It is a hard-decision decoding (HDD) algorithm. That is, all bits in a received word are treated equally. Such an HDD decoding algorithm works well whenever the number of errors is less than t, the error-correction capability of BCH codes. The Chase decoding is a type of soft-decision decoding (SDD) method. An SDD method leverages the soft information, i.e. the reliability of each bit in a word, to decode. The main motivation of the Chase decoding algorithm is that a less reliable bit has a higher chance of being wrong. Therefore, the Chase decoding algorithm flips η least reliable bits in a received word one by one and then attempts to decode the modified word to see if the corrupted word can be recovered correctly.

The most straightforward and the widely adopted way of implementing the Chase decoding in hardware is to use the existing BCH HDD circuit iteratively with control circuit generating different testing patterns [137][138], as illustrated in Fig. 5.14(a). For every error pattern, a new testing word is generated and the HDD decoder is



(a)



(b)

Figure 5.14: Illustration of (a) the conventional and (b) the one-pass Chase decoding of BCH code.

applied with the attempt to find the correct word. This way of decoding requires applying the power-hungry Berlekamp key equation solver and the Chien search module 2^{η} times for decoding. Such an exponential growth in computational complexity restricts η to be a small number (typically one or two), yielding only a marginal performance improvement compared to an HDD decoder.

To circumvent the aforementioned difficulty, a one-pass Chase decoding algorithm was proposed in [134]. The original work was for Reed-Solomon (RS) codes, but it can be easily adapted for BCH codes. This algorithm is based on the observation that the error-locator polynomial for a new testing word can be easily obtained if the polynomial for another testing word, which is close to the new word in Hamming distance, is known, as shown in Fig. 5.14(b). Therefore, the Berlekamp's algorithm needs only to be applied once to obtain the starting error-locator polynomial. Other error-locator polynomials corresponding to different error patterns can then be derived easily from the starting polynomial by using the polynomial update algorithm in

[134][135].

5.3.2 Eligibility Verification Algorithm

With the one-pass Chase decoding algorithm, the decoding of a BCH code with Chase decoding can be very efficient. Nevertheless, the highly iterative Chien search is still needed for each newly-generated codeword to examine whether the obtained codeword is a valid one. This process has been shown to be the most power and area consuming part in the algorithm [135]. In this section, how this costly step can be by-passed is discussed.

It has been shown in [139] that the sufficient and necessary conditions for an error locator polynomial to locate errors such that a legal (not necessarily correct) word can be recovered are as follows:

1) The error locator polynomial $\Lambda(x)$ has exactly e distinct non-zero roots in GF(2^m).

2) $L_\Lambda = e$, where L_Λ is the length of the linear feedback shift register (LFSR) described by $\Lambda(x)$.

It is also known that $d_\Lambda \leq L_\Lambda$ [140], where d_Λ is the degree of the polynomial $\Lambda(x)$. Consequently, the process of verifying an error locator polynomial can be divided into two cases:

1) If $d_\Lambda \neq L_\Lambda$, then the error locator polynomial is not a valid one.

2) If $d_\Lambda = L_\Lambda$, then check whether the number of distinct non-zero roots of $\Lambda(x)$ is equal to d_Λ. If they are not equal, then the error locator polynomial is not a valid one.

The condition $d_\Lambda = L_\Lambda$ can be easily checked by identifying the location of the first non-zero coefficient in $\Lambda(x)$. Furthermore, it is found in simulations that this condition is satisfied in most of the time even when the error locator polynomial is not a correct one. Therefore, counting the number of roots that $\Lambda(x)$ has is the key

to determining whether the obtained error locator polynomial is valid. Zero roots in $\Lambda(x)$ can be identified by checking whether Λ_0 is zero. $\Lambda(x)$ with a zero root is discarded right away without further processing. Therefore, in the remainder of this section, it is assumed that $\Lambda(x)$ does not have a root that is equal to zero.

An auxiliary polynomial $d(x)$ is defined according to (5.21), where the operator $\gcd(a, b)$ stands for finding the greatest common divisor of a and b. As $x^{2^m} - x$ has all the elements in $\mathrm{GF}(2^m)$ as its roots [138], it can be shown that the degree of $d(x)$ is equal to the number of roots that $\Lambda(x)$ has.

$$d(x) := \gcd\left(x^{2^m} - x, \Lambda(x)\right) \tag{5.21}$$

Euclideans algorithm can be employed here to obtain a further simplified expression

$$d(x) = \gcd\left(\left(x^{2^m} - x\right) \bmod \Lambda(x), \Lambda(x)\right) \tag{5.22}$$

Following (5.22), it can be proven that the sufficient and necessary condition for $\Lambda(x)$ to have d_Λ distinct non-zero roots in $\mathrm{GF}(2^m)$ is

$$x^{2^m} \bmod \Lambda(x) = x \tag{5.23}$$

(5.23) is too expensive to be computed directly when m is large. Fortunately, squaring and multiply [141] can be utilized here to save a significant amount of computational labor. In addition, because 2^m is a power of 2, what we really need is just repeated squaring (that is, not even multiply). More specifically, we can calculate x^{2^m} iteratively as follows, starting from the trivial case:

$$x^{2^{\lfloor \log_2(d_\Lambda - 1)\rfloor}} \bmod \Lambda(x) = x^{2^{\lfloor \log_2(d_\Lambda - 1)\rfloor}} \tag{5.24}$$

We then can compute $x^{2^{i+1}} \bmod \Lambda(x)$from $x^{2^i} \bmod \Lambda(x)$ as

$$x^{2^{i+1}} \bmod \Lambda(x) = f_i(x) \bmod \Lambda(x) \tag{5.25}$$

where

$$f_i(x) = \left(x^{2^i} \bmod \Lambda(x)\right)^2 \tag{5.26}$$

By doing this, only $m - \lfloor \log_2(d_\Lambda - 1) \rfloor$ polynomial modulo operations are needed to compute $x^{2^m} \bmod \Lambda(x)$. Each polynomial modulo operation is at most of order t. To carry out the modulo operation, an old-school long division can be employed. Each modulo operation takes about d_Λ^2 multiplications. One problem with this straightforward implementation is that the critical path is long, as shown in Section 5.3.3. Considering that the divisors in all modulo operations are $\Lambda(x)$ (see (5.22)-(5.26)), it is worth spending some effort on converting $\Lambda(x)$ into a form with which the division in the following stages can be performed more efficiently. Inspired by the algorithm in [142], we propose the following polynomial inversion algorithm to help improve the efficiency and critical path delay of the polynomial division.

Let $\Lambda^r(x)$ represent the polynomial with all the coefficients arranged in a reverse order of $\Lambda(x)$. That is, $\Lambda^r(x) = \Lambda_0 x^{d_\Lambda} + \Lambda_1 x^{d_\Lambda - 1} + \cdots + \Lambda_{d_\Lambda}$. Similar notations apply for other polynomials. Then it can be shown that the reverse quotient polynomial can be computed as

$$q_i^r(x) = \hat{\Lambda}^r(x) f_i^r(x) \bmod x^{d_\Lambda - 1} \tag{5.27}$$

where $\hat{\Lambda}^r(x)$ is defined as the inverse polynomial of $\Lambda^r(x)$ such that

$$\Lambda^r(x)\hat{\Lambda}^r(x) = 1 \bmod x^{d_\Lambda - 1} \tag{5.28}$$

$\hat{\Lambda}^r(x)$ can be conveniently computed through an iterative algorithm. Let $\hat{\Lambda}_i^r(x)$ be the polynomial such that $\Lambda^r(x)\hat{\Lambda}_i^r(x) = 1 \bmod x^i$, then it can be shown that by

setting the initial condition and iterative updating equation as (5.29) and (5.30), one can obtain $\hat{\Lambda}^r(x)$.

$$\hat{\Lambda}^r_1(x) = \Lambda^{-1}_{d_\Lambda} \tag{5.29}$$

$$\hat{\Lambda}^r_i(x) = \Lambda^r(x) \left(\hat{\Lambda}^r_j(x)\right)^2 \bmod x^i \tag{5.30}$$

where $j \geq \lceil i/2 \rceil$.

It is noted that in (5.30), only unknown coefficients need to be calculated. The process of computing $\hat{\Lambda}^r(x)$ takes one inversion, $(d_\Lambda - 1)/2$ squaring, $(d_\Lambda - 1)^2/4 + (d_\Lambda - 1)/2$ multiplication and $(d_\Lambda - 1)^2/4 - (d_\Lambda - 1)/2$ addition when $d_\Lambda - 1$ is a power of two. As will be shown in Section 5.3.3, the computational complexity of deriving $\Lambda^r(x)$ is much less than the complexity of computing (5.25). Furthermore, pre-computing $\Lambda^r(x)$ can actually save efforts for computing (5.25).

One thing should be noted is that the derivation of the proposed algorithm implicitly assumes that the BCH code is not shortened. Therefore, the proposed algorithm is not rigorously applicable to shortened BCH codes. It is theoretically possible for the error locator polynomial of a shortened BCH code to have the correct number of roots over GF(2^m), yet one or more of these roots are not in the valid range. In this case, the proposed algorithm fails to detect the invalid error locator polynomial, whereas the exhaustive Chien search is still able to. Simulation, however, shows that the probability of this malfunction is unnoticeably low. Therefore, we argue that the proposed algorithm can be applied to a shortened BCH code for practical purposes.

5.3.3 VLSI Architecture

Even though the proposed algorithm can be used for polynomials with arbitrary degrees, it is, in practice, enough to check polynomials with a degree of t. By doing this, the hardware complexity can be significantly reduced without a noticeable performance degradation. Fig. 5.15 compares the performance achieved by an SDD

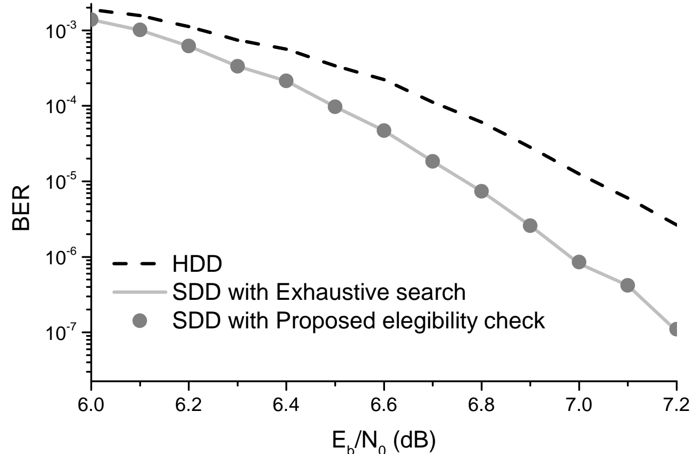


Figure 5.15: Comparison of the error-correction performances of an HDD, an SDD with exhaustive polynomial search, and an SDD with the proposed eligibility checking algorithm. The BCH code used here is a $(4200, 4096)$ code over GF(2^{13}).

employing the proposed eligibility verification algorithm with the performance of an SDD using direct polynomial search and the performance of an HDD. In the proposed eligibility checking algorithm, d_Λ is set to t. That is, the verification process only runs on the polynomials with a degree of t. For the polynomials with degrees less than t, they are adopted as valid solutions. It is shown in the figure that, the proposed eligibility verification algorithm can effectively identify valid error locator polynomials. The performance of the proposed algorithm is degraded neither by applying the algorithm to a shortened BCH code nor by only running verification on polynomials with a degree of t.

A VLSI architecture of the proposed eligibility checking algorithm is shown in Fig. 5.16. There are three main blocks: block I and II are for polynomial multiplication and block III is used for polynomial inversion. The pipelining can be used in order to improve the throughput of the system. There are mainly three finite field operations shown in Fig. 5.16 besides the standard multiplex, delay and compare operations. The first operation is subtracting polynomials in a finite field. This operation can be done by simply doing bitwise XOR operations on each coefficient of the two polynomials. The second operation is squaring a polynomial in a finite field, which can be achieved

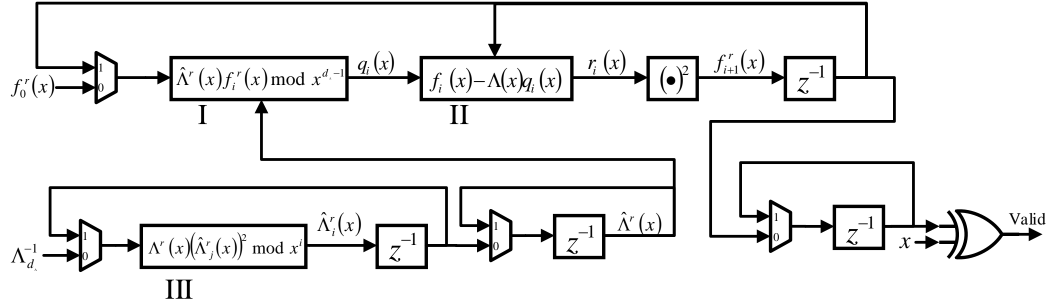


Figure 5.16: A VLSI architecture of the proposed eligibility checking algorithm. Block I and II conduct multiply and modulo operations, and Block III conducts polynomial inversion operation.

by squaring each coefficient of the polynomial. The third operation is to multiply and modulo polynomials. This is the main operation in the proposed algorithm. It can be formulated as a matrix-vector multiplication where the matrix is a Toeplitz matrix. Thanks to the unique property of Toeplitz matrices, the multiplication-and-modulo operation can be efficiently conducted by employing the circuit shown in Fig. 5.17. In this figure, the diagonal multipliers share the same multiplicands b_i. The multipliers at the same columns share the same multipliers a_i. Products at the same row are then added by a XOR tree to get the final results c_i. Since a finite field multiplication can be expressed as a matrix-vector multiplication, matrices associated with the shared multiplicands only need to be calculated once and distributed along the diagonals, reducing gate counts and the critical path delay.

The hardware complexity of the main blocks in the proposed eligibility checking circuit is summarized in Table 5.1. The proposed circuit has an area-latency product in the order of $(m - \log_2 t)\, t^2$. This is much less than nt, the area-latency product of a conventional exhaustive search. The area-latency product is defined as the product of the number of multipliers and the number of clock cycles needed to complete the task. It serves as a quick estimation of how the complexity of the circuit grows with

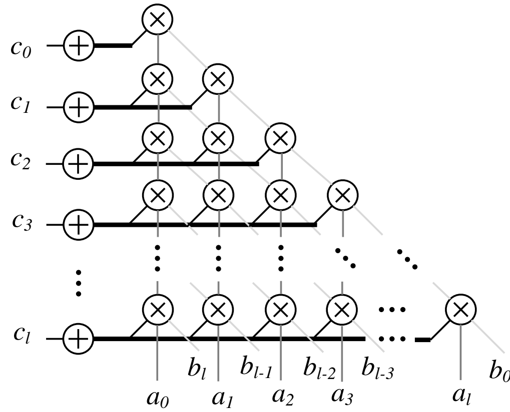


Figure 5.17: Diagram of the proposed polynomial multiplication array. Same multiplicands b_i are shared along diagonals, and multipliers a_i are shared along columns. Multiplication results c_i are obtained by adding each row with XOR trees.

the size of the problem. The reason that only finite field multipliers are counted is that they dominate the area of the circuit. To give an immediate comparison, the gate count ratio between the proposed algorithm and the conventional method is less than $(n-k)/n$, which is the redundancy ratio of an error-correction code. The redundancy ratios in most memory systems are much less than one. Even though the employed non-constant multiplier takes larger area than the constant multiplier employed in the Chien search, the savings in area-latency product is still significant, as will be shown in Section 5.3.4. In addition, the diagonal-sharing technique mentioned above also helps reduce the gate counts effectively. Furthermore, compared to the straightforward implementation with a long division that requires $(m - \log_2(t))\, t^2$ multipliers, the proposed algorithm only needs approximately $3\,(m - \log_2(t))\, t^2/4$ multipliers, reducing the gate count by roughly 25%. This saving is achieved by pre-computing the inverse polynomial $\hat{\Lambda}^r(x)$.

5.3.4 Design Example

In this section, the proposed eligibility checking circuit is implemented for a $(4200, 4096)$ code over GF(2^{13}). Inversion of $\Lambda_{d_\Lambda}^{-1}$ and block III are pipelined with

Table 5.1: Summary of hardware complexity of the proposed eligibility verification circuit

	# of multiplier	# of adder	Critical path delay	# of cycles
I	$\frac{t(t+2)}{4}$	$\frac{t(t-2)}{4}$	$D_{mul} + D_{add} \cdot \log_2\left(\frac{t-1}{2}\right)$	$m - \lfloor \log_2(t-1) \rfloor$
II	$\frac{(t-1)(t+2)}{2}$	$\frac{t(t-1)}{2}$	$D_{mul} + D_{add} \cdot \log_2(t-1)$	$m - \lfloor \log_2(t-1) \rfloor$
III	$\frac{t}{2}$	$\frac{t}{2} - 1$	$D_{mul} + D_{add} \cdot \log_2\left(\frac{t}{2}\right)$	t

block II and III. The process of eligibility verification takes 11 clock cycles.

The comparisons of the gate counts and the critical path delay are summarized in Table 5.2. In this design example, the finite field multiplier and squaring circuit that were proposed in [143] are used. The multiplier in [143] is not the optimal choice in terms of gate counts. It is adopted in our design example because of its simplicity. More sophisticated multipliers such as those in [144] can be employed to further reduce the gate counts. In the table, the numbers of flip-flops are directly read out from the synthesized netlists and the combinational gate counts reported by the employed synthesis tool are converted to equivalent NAND2 gate counts for comparison.

Table 5.2: Comparison of hardware complexity and critical path delay

Algorithm	Sub-block	NAND2	Registers	Critical path delay
Proposed algorithm	Polynomial inversion	634	94	$D_{AND} + 6D_{XOR}$
	Polynomial multiplication	39K	104	$2D_{AND} + 13D_{XOR}$
	Total	46K	453	$2D_{AND} + 13D_{XOR}$
Exhaustive polynomial search		416K	485	$6D_{AND} + 6D_{XOR}$

As shown in the table, the gate counts of the proposed eligibility checking circuit are only around 12% of the exhaustive Chien search. This number can be further projected to estimate the overall savings in the area of the entire decoder. In [135], it is shown that the polynomial searching block occupies an area that is 85% of the total area of the decoder. Therefore, it is estimated that the proposed eligibility

verification circuit can reduce the area of the one-pass Chase decoder by 75% while having a similar performance. Furthermore, thanks to the polynomial inversion step, the critical path delay of the proposed circuit is reduced to a value similar to that of the Chien search.

The proposed design and the conventional Chien search block are synthesized in a 65-nm technology using Synopsys Design Compiler. The synthesized designs are then automatically placed and routed with Cadence Encounter. The areas of the obtained layouts are reported in Table 5.3. The netlists obtained after place and route are simulated using Synopsys Finesim with extracted interconnect parasitics. The power consumption and the critical path delay are simulated. The maximum clock frequency reported in Table 5.3 is calculated according to the simulated critical path delay with a 10% margin. As noted from Table 5.3. The proposed eligibility checking circuit is 20 times more power-efficient than the conventional exhaustive Chien search. The savings in power consumption is larger than the area saving ratio. This can be attributed to the fact that the conventional Chien search has a larger activity factor.

Table 5.3: Summary of the proposed design

	This work	Conventional
Equivalent NAND2 gate count	49K	418K
Area after place and route	$67,600\mu m^2$	$640,000\mu m^2$
Power consumption @ 1.2 V & 400MHz	21.9 mW	408mW
Maximum clock frequency	568 MHz	455 MHz

5.4 Chapter Summary

In this chapter, architecture- and circuit-level techniques are employed to reduce the power consumption of the memories in neuromorphic systems while improving the reliability. Two studies are presented.

The first study focuses on compensating for the variations in on-chip SRAM memories. The variation in SNMs is analyzed. A statistical model for SNMs is proposed to provide a quick way to estimate the reliability of the SRAM cells, thus avoiding the repeated uses of time-consuming MC simulations at the design time. To counteract global process variations, an adaptive body bias generation circuit is proposed. Feedback is employed to control the transition voltage of a cell inverter. A stabilization technique inspired by the ring amplifier is applied to the proposed bias generation circuit. Two examples are provided along with the simulations results. It is shown that a 15% improvement can be achieved for the worst-case read SNM.

In the second study, we present a novel eligibility verification algorithm aimed at avoiding the area and power consumption penalty incurred by the parallel Chien search in a conventional one-pass Chase soft-decision BCH decoder. The proposed algorithm can effectively check the correctness of a derived error locator polynomial by counting the number of roots it has. In addition, an iterative polynomial inversion algorithm is presented to reduce the area and the critical path delay. A hardware architecture for the proposed algorithms is also presented in this chapter. Hardware complexity is carefully examined. A design example is implemented for a (4200, 4096) code over $GF(2^{13})$. The obtained gate counts and critical path delay are compared with a conventional design. Our newly proposed design achieves more than 88% area reduction while having a similar critical path delay. This translates into a 75% reduction in the overall decoder area.

CHAPTER VI

Conclusion

6.1 Summary and Contributions

The looming end of Moore's law drives researchers to look for promising alternatives to the conventional ways of computing. This dissertation aims at developing energy-efficient neural network hardware for energy-constrained applications. In order to achieve this ultimate goal, innovations in algorithms, architectures, and circuits are made.

Chapter II introduces a bio-inspired learning algorithm that can train spiking neural networks in hardware effectively and efficiently. It investigates the possibility of using an STDP-like supervised learning algorithm in a deep neural network. Through estimating the gradients based on spike timing information, a learning process similar to that of a conventional ANN can be achieved. Two neural networks performing MNIST hand-written digits recognition tasks are employed as examples to demonstrate the efficacy of the proposed learning algorithm.

With the learning algorithm developed in Chapter II, an efficient hardware architecture is proposed in Chapter III. The new architecture is based on an event-driven computational model. In addition, a cache structure is employed to reduce the memory requirement, considering the sparsity in the neural networks. Furthermore, local storage buffers are leveraged to hide the latency of updating spike timing information,

which helps boost the throughput of the system. The architecture is implemented in a 65-nm technology and how the inference duration can be utilized as a run-time knob to trade off accuracy for energy and throughput is demonstrated.

Chapter IV presents a neural network accelerator for the ADP algorithm. Data-level parallelism in the matrix-vector multiplication is exploited through a SIMD architecture. The proposed accelerator can be programmed through the customized instruction set in order to accommodate various tasks. In addition, a virtual update algorithm is proposed to exploit the inherent computational patterns existing in the ADHDP algorithm. The proposed technique effectively increases the throughput and reduces the power consumption of the accelerator, resulting in a significant improvement in energy efficiency.

Chapter V addresses the concern about the reliability of the memory circuitry used in neural network hardware. Two techniques are proposed in this chapter to deal with both volatile SRAM memory and non-volatile memory such as flash. A feedback compensation technique is presented to counteract the global process variation, improving the worst-case read static noise margin. In order to increase the throughput, as well as the energy efficiency of the error-correction circuit used in flash memory, an algorithm is proposed to break the bottleneck existing in a conventional one-pass Chase decoding algorithm. With the proposed algorithm and hardware architecture, the energy efficiency of the error-correction circuitry is significantly improved.

6.2 Future Work

There is still a long way to go before achieving brain-like computing in hardware. The problem becomes even more complicated when power consumption is also a major design consideration.

One future direction is to employ the proposed learning algorithm in more complicated neural networks, such as convolutional neural networks and recurrent neural

networks. Even though the proposed learning algorithm is still valid in these networks, suitable hardware architectures need to be developed to accommodate the more complicated network structures with a weight sharing.

Another research direction is to build neuromorphic hardware based on emerging nanotechnologies, e.g. memristors. One problem of using memristors in neural network hardware is that a neural network that is trained off-line with the device model might not work well after being mapped to a physical network on the chip because of both the spatial and temporal variations in memristor devices. With the proposed on-line learning algorithm, however, the training can occur directly on-chip. Such on-chip learning can effectively compensate for the variations.

6.3 Related Publications

Some chapters of this dissertation are based on following publications:

Chapter II:

[59] N. Zheng and P. Mazumder, "Online supervised learning for hardware-based multilayer spiking neural networks through the modulation of weight-dependent spike-timing-dependent plasticity," *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, no. 99, pp. 1-16. © 2017 IEEE.

[60] N. Zheng and P. Mazumder, "Hardware-friendly actor-critic reinforcement learning through modulation of spike-timing-dependent plasticity," *IEEE Transactions on Computers*, vol. 66, no. 2, pp. 299-311, 2017. © 2017 IEEE.

Chapter III:

[68] N. Zheng and P. Mazumder, "A low-power hardware architecture for on-line supervised learning in multi-layer spiking neural networks," *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on.* Under review.

Chapter IV:

[105] N. Zheng and P. Mazumder, "A scalable low-power reconfigurable acceler-

ator for action-dependent heuristic dynamic programming," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. PP, no. 99, pp. 1-12. © 2017 IEEE.

Chapter V:

[115] N. Zheng and P. Mazumder, "Modeling and mitigation of static noise margin variation in subthreshold SRAM cells," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 10, pp. 2726-2736, 2017. © 2017 IEEE.

[116] N. Zheng and P. Mazumder, "An efficient eligible error locator polynomial searching algorithm and hardware architecture for one-pass Chase decoding of BCH codes," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 64, no. 5, pp. 580-584, 2017. © 2017 IEEE.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[3] H. A. Rowley, S. Baluja, and T. Kanade, "Neural network-based face detection," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 20, no. 1, pp. 23–38, 1998.

[4] D. Psaltis, A. Sideris, and A. A. Yamamura, "A multilayered neural network controller," *IEEE control systems magazine*, vol. 8, no. 2, pp. 17–21, 1988.

[5] M. Kawato, K. Furukawa, and R. Suzuki, "A hierarchical neural-network model for control and learning of voluntary movement," *Biological cybernetics*, vol. 57, no. 3, pp. 169–185, 1987.

[6] M. D. Odom and R. Sharda, "A neural network model for bankruptcy prediction," in *Neural Networks, 1990., 1990 IJCNN International Joint Conference on.* IEEE, 1990, pp. 163–168.

[7] T. Kimoto, K. Asakawa, M. Yoda, and M. Takeoka, "Stock market prediction system with modular neural networks," in *Neural Networks, 1990., 1990 IJCNN International Joint Conference on.* IEEE, 1990, pp. 1–6.

[8] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?" *Journal of Machine Learning Research*, vol. 11, no. Feb, pp. 625–660, 2010.

[9] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[11] D. Goodman and R. Brette, "Brian: a simulator for spiking neural networks in python," *Frontiers in neuroinformatics*, vol. 2, 2008.

[12] N. K. Kasabov, "Neucube: A spiking neural network architecture for mapping, learning and understanding of spatio-temporal brain data," *Neural Networks*, vol. 52, pp. 62–76, 2014.

[13] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on neural networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

[14] B. Zhao, R. Ding, S. Chen, B. Linares-Barranco, and H. Tang, "Feedforward categorization on aer motion events using cortex-like features in a spiking neural network," *IEEE transactions on neural networks and learning systems*, vol. 26, no. 9, pp. 1963–1978, 2015.

[15] Q. Yu, H. Tang, K. C. Tan, and H. Li, "Precise-spike-driven synaptic plasticity: Learning hetero-association of spatiotemporal spike patterns," *Plos one*, vol. 8, no. 11, p. e78318, 2013.

[16] Q. Yu, H. Tang, K. C. Tan, and H. Li, "Rapid feedforward computation by temporal encoding and learning with spiking neurons," *IEEE transactions on neural networks and learning systems*, vol. 24, no. 10, pp. 1539–1552, 2013.

[17] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of physiology*, vol. 117, no. 4, pp. 500–544, 1952.

[18] A. N. Burkitt, "A review of the integrate-and-fire neuron model: I. homogeneous synaptic input," *Biological cybernetics*, vol. 95, no. 1, pp. 1–19, 2006.

[19] S. Eberhardt, T. Duong, and A. Thakoor, "Design of parallel hardware neural network systems from custom analog vlsi'building block'chips," *NEURON*, vol. 3, p. 2, 1989.

[20] Y. Maeda, H. Hirano, and Y. Kanata, "A learning rule of neural networks via simultaneous perturbation and its hardware implementation," *Neural Networks*, vol. 8, no. 2, pp. 251–259, 1995.

[21] N. Mauduit, M. Duranton, J. Gobert, and J.-A. Sirat, "Lneuro 1.0: A piece of hardware lego for building neural network systems," *IEEE transactions on Neural Networks*, vol. 3, no. 3, pp. 414–422, 1992.

[22] P. Mazumder and Y.-S. Jih, "A new built-in self-repair approach to vlsi memory yield enhancement by using neural-type circuits," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 12, no. 1, pp. 124–136, 1993.

[23] G. Bell, "Bell's law for the birth and death of computer classes: A theory of the computer's evolution," *IEEE Solid-State Circuits Society Newsletter*, vol. 13, no. 4, pp. 8–19, 2008.

[24] Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle *et al.*, "Greedy layer-wise training of deep networks," *Advances in neural information processing systems*, vol. 19, p. 153, 2007.

[25] Q. V. Le, "Building high-level features using large scale unsupervised learning," in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 8595–8598.

[26] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[28] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[29] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[30] Y. Lee, S. Bang, I. Lee, Y. Kim, G. Kim, M. H. Ghaed, P. Pannuto, P. Dutta, D. Sylvester, and D. Blaauw, "A modular 1 mm die-stacked sensing platform with low power i c inter-die communication and multi-modal energy harvesting," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 1, pp. 229–243, 2013.

[31] I. Lee, G. Kim, S. Bang, A. Wolfe, R. Bell, S. Jeong, Y. Kim, J. Kagan, M. Arias-Thode, B. Chadwick *et al.*, "System-on-mud: Ultra-low power oceanic sensing platform powered by small-scale benthic microbial fuel cells," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 4, pp. 1126–1135, 2015.

[32] Y.-P. Chen, D. Jeon, Y. Lee, Y. Kim, Z. Foo, I. Lee, N. B. Langhals, G. Kruger, H. Oral, O. Berenfeld *et al.*, "An injectable 64 nw ecg mixed-signal soc in 65 nm for arrhythmia monitoring," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 1, pp. 375–390, 2015.

[33] P. Mazumder, D. Hu, I. Ebong, X. Zhang, Z. Xu, and S. Ferrari, "Digital implementation of a virtual insect trained by spike-timing dependent plasticity," *Integration, the VLSI Journal*, vol. 54, pp. 109–117, 2016.

[34] N. O. Pérez-Arancibia, K. Y. Ma, K. C. Galloway, J. D. Greenberg, and R. J. Wood, "First controlled vertical flight of a biologically inspired microrobot," *Bioinspiration & Biomimetics*, vol. 6, no. 3, p. 036009, 2011.

[35] R. J. Wood, "The first takeoff of a biologically inspired at-scale robotic insect," *IEEE transactions on robotics*, vol. 24, no. 2, pp. 341–347, 2008.

[36] D. Hu, X. Zhang, Z. Xu, S. Ferrari, and P. Mazumder, "Digital implementation of a spiking neural network (snn) capable of spike-timing-dependent plasticity (stdp) learning," in *14th IEEE International Conference on Nanotechnology*. IEEE, 2014, pp. 873–876.

[37] O. Vermesan and P. Friess, *Internet of things-global technological and societal trends from smart environments and spaces to green ICT*. River Publishers, 2011.

[38] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[39] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J.-M. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, 2014.

[40] K. A. Boahen, "Point-to-point connectivity between neuromorphic chips using address events," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 47, no. 5, pp. 416–434, 2000.

[41] R. Serrano-Gotarredona, M. Oster, P. Lichtsteiner, A. Linares-Barranco, R. Paz-Vicente, F. Gómez-Rodríguez, L. Camuñas-Mesa, R. Berner, M. Rivas-Pérez, T. Delbruck *et al.*, "Caviar: A 45k neuron, 5m synapse, 12g connects/s aer hardware sensory–processing–learning–actuating system for high-speed visual object recognition and tracking," *IEEE Transactions on Neural Networks*, vol. 20, no. 9, pp. 1417–1438, 2009.

[42] N. Srinivasa and J. M. Cruz-Albrecht, "Neuromorphic adaptive plastic scalable electronics: analog learning systems," *IEEE pulse*, vol. 3, no. 1, pp. 51–56, 2012.

[43] D. Querlioz, O. Bichler, P. Dollfus, and C. Gamrat, "Immunity to device variations in a spiking neural network with memristive nanodevices," *IEEE Transactions on Nanotechnology*, vol. 12, no. 3, pp. 288–295, 2013.

[44] P. U. Diehl and M. Cook, "Unsupervised learning of digit recognition using spike-timing-dependent plasticity," *Frontiers in computational neuroscience*, vol. 9, 2015.

[45] T. Masquelier, "Relative spike time coding and stdp-based orientation selectivity in the early visual system in natural continuous and saccadic vision: a computational model," *Journal of computational neuroscience*, vol. 32, no. 3, pp. 425–441, 2012.

[46] T. Masquelier and S. J. Thorpe, "Unsupervised learning of visual features through spike timing dependent plasticity," *PLoS Comput Biol*, vol. 3, no. 2, p. e31, 2007.

[47] S. M. Bohte, J. N. Kok, and H. La Poutre, "Error-backpropagation in temporally encoded networks of spiking neurons," *Neurocomputing*, vol. 48, no. 1, pp. 17–37, 2002.

[48] F. Ponulak and A. Kasinski, "Supervised learning in spiking neural networks with resume: sequence learning, classification, and spike shifting," *Neural Computation*, vol. 22, no. 2, pp. 467–510, 2010.

[49] R. Gütig and H. Sompolinsky, "The tempotron: a neuron that learns spike timing–based decisions," *Nature neuroscience*, vol. 9, no. 3, pp. 420–428, 2006.

[50] Q. Yu, R. Yan, H. Tang, K. C. Tan, and H. Li, "A spiking neural network system for robust sequence recognition," *IEEE transactions on neural networks and learning systems*, vol. 27, no. 3, pp. 621–635, 2016.

[51] L. Chang, R. K. Montoye, Y. Nakamura, K. A. Batson, R. J. Eickemeyer, R. H. Dennard, W. Haensch, and D. Jamsek, "An 8t-sram for variability tolerance and low-voltage operation in high-performance caches," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 4, pp. 956–963, 2008.

[52] B. H. Calhoun and A. P. Chandrakasan, "A 256-kb 65-nm sub-threshold sram design for ultra-low-voltage operation," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 3, pp. 680–688, 2007.

[53] S. Pal and A. Islam, "9-t sram cell for reliable ultralow-power applications and solving multibit soft-error issue," *IEEE Transactions on Device and Materials Reliability*, vol. 16, no. 2, pp. 172–182, 2016.

[54] K. Takeuchi, "Scaling challenges of nand flash memory and hybrid memory system with storage class memory & nand flash memory," in *Proceedings of the IEEE 2013 Custom Integrated Circuits Conference*. IEEE, 2013, pp. 1–6.

[55] Y. Cao, Y. Chen, and D. Khosla, "Spiking deep convolutional neural networks for energy-efficient object recognition," *International Journal of Computer Vision*, vol. 113, no. 1, pp. 54–66, 2015.

[56] P. U. Diehl, D. Neil, J. Binas, M. Cook, S.-C. Liu, and M. Pfeiffer, "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing," in *2015 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2015, pp. 1–8.

[57] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha, "A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm," in *2011 IEEE custom integrated circuits conference (CICC)*. IEEE, 2011, pp. 1–4.

[58] P. OConnor, D. Neil, S.-C. Liu, T. Delbruck, and M. Pfeiffer, "Real-time classification and sensor fusion with a spiking deep belief network," *Neuromorphic Engineering Systems and Applications*, p. 61, 2015.

[59] N. Zheng and P. Mazumder, "Online supervised learning for hardware-based multilayer spiking neural networks through the modulation of weight-dependent spike-timing-dependent plasticity," *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, no. 99, pp. 1–16, 2017.

[60] N. Zheng and P. Mazumder, "Hardware-friendly actor-critic reinforcement learning through modulation of spike-timing-dependent plasticity," *IEEE Transactions on Computers*, vol. 66, no. 2, pp. 299–311, 2017.

[61] J. C. Spall, "An overview of the simultaneous perturbation method for efficient optimization," *Johns Hopkins apl technical digest*, vol. 19, no. 4, pp. 482–492, 1998.

[62] G. E. Hinton, "A practical guide to training restricted boltzmann machines," in *Neural Networks: Tricks of the Trade*. Springer, 2012, pp. 599–619.

[63] A. S. Cassidy, P. Merolla, J. V. Arthur, S. K. Esser, B. Jackson, R. Alvarez-Icaza, P. Datta, J. Sawada, T. M. Wong, V. Feldman *et al.*, "Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores," in *Neural Networks (IJCNN), The 2013 International Joint Conference on*. IEEE, 2013, pp. 1–10.

[64] A. Neelakantan, L. Vilnis, Q. V. Le, I. Sutskever, L. Kaiser, K. Kurach, and J. Martens, "Adding gradient noise improves learning for very deep networks," *arXiv preprint arXiv:1511.06807*, 2015.

[65] E. Neftci, S. Das, B. Pedroni, K. Kreutz-Delgado, and G. Cauwenberghs, "Event-driven contrastive divergence for spiking neuromorphic systems," 2013.

[66] S. Hussain, S.-C. Liu, and A. Basu, "Improved margin multi-class classification using dendritic neurons with morphological learning," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2014, pp. 2640–2643.

[67] A. Alaghi and J. P. Hayes, "Survey of stochastic computing," *ACM Transactions on Embedded computing systems (TECS)*, vol. 12, no. 2s, p. 92, 2013.

[68] N. Zheng and P. Mazumder, "A low-power hardware architecture for on-line supervised learning in multi-layer spiking neural networks," *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*, 2018. Under review.

[69] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 2011, pp. 315–323.

[70] P. Knag, J. K. Kim, T. Chen, and Z. Zhang, "A sparse coding neural network asic with on-chip learning for feature extraction and encoding," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 4, pp. 1070–1079, 2015.

[71] J.-s. Seo, B. Brezzo, Y. Liu, B. D. Parker, S. K. Esser, R. K. Montoye, B. Rajendran, J. A. Tierno, L. Chang, D. S. Modha *et al.*, "A 45nm cmos neuromorphic chip with a scalable architecture for learning in networks of spiking neurons," in *Custom Integrated Circuits Conference (CICC), 2011 IEEE.* IEEE, 2011, pp. 1–4.

[72] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, no. 4. ACM, 2014, pp. 269–284.

[73] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 2014, pp. 609–622.

[74] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[75] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture.* IEEE Press, 2016, pp. 267–278.

[76] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: ineffectual-neuron-free deep neural network computing," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on.* IEEE, 2016, pp. 1–13.

[77] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "Dlau: A scalable deep learning accelerator unit on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 513–517, 2017.

[78] Y. Sun and A. C. Cheng, "Machine learning on-a-chip: A high-performance low-power reusable neuron architecture for artificial neural networks in ecg classifications," *Computers in biology and medicine*, vol. 42, no. 7, pp. 751–757, 2012.

[79] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on.* IEEE, 2012, pp. 356–367.

[80] J. Kung, D. Kim, and S. Mukhopadhyay, "A power-aware digital feedforward neural network platform with backpropagation driven approximate synapses,"

in *Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on*. IEEE, 2015, pp. 85–90.

[81] A. Savich, M. Moussa, and S. Areibi, "A scalable pipelined architecture for real-time computation of mlp-bp neural networks," *Microprocessors and Microsystems*, vol. 36, no. 2, pp. 138–150, 2012.

[82] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 27–40.

[83] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 1–12.

[84] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 13–26.

[85] D. V. Prokhorov and D. C. Wunsch, "Adaptive critic designs," *IEEE transactions on Neural Networks*, vol. 8, no. 5, pp. 997–1007, 1997.

[86] F. L. Lewis and D. Vrabie, "Reinforcement learning and adaptive dynamic programming for feedback control," *IEEE circuits and systems magazine*, vol. 9, no. 3, 2009.

[87] F.-Y. Wang, H. Zhang, and D. Liu, "Adaptive dynamic programming: An introduction," *IEEE computational intelligence magazine*, vol. 4, no. 2, 2009.

[88] F. L. Lewis, D. Vrabie, and K. G. Vamvoudakis, "Reinforcement learning and feedback control: Using natural decision methods to design optimal adaptive controllers," *IEEE Control Systems*, vol. 32, no. 6, pp. 76–105, 2012.

[89] D. Wang, H. He, and D. Liu, "Adaptive critic nonlinear robust control: A survey," *IEEE Transactions on Cybernetics*, vol. 47, no. 10, pp. 3429–3451, 2017.

[90] J. Si and Y.-T. Wang, "Online learning control by association and reinforcement," *IEEE Transactions on Neural Networks*, vol. 12, no. 2, pp. 264–276, 2001.

[91] D. Liu, X. Xiong, and Y. Zhang, "Action-dependent adaptive critic designs," in *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, vol. 2. IEEE, 2001, pp. 990–995.

[92] M. S. Iyer and D. C. Wunsch, "Dynamic re-optimization of a fed-batch fermentor using adaptive critic designs," *IEEE Transactions on Neural Networks*, vol. 12, no. 6, pp. 1433–1444, 2001.

[93] D. Han and S. Balakrishnan, "State-constrained agile missile control with adaptive-critic-based neural networks," *IEEE Transactions on Control Systems Technology*, vol. 10, no. 4, pp. 481–489, 2002.

[94] S. Ferrari and R. F. Stengel, "Online adaptive critic flight control," *Journal of Guidance, Control, and Dynamics*, vol. 27, no. 5, pp. 777–786, 2004.

[95] C.-K. Lin, "Adaptive critic autopilot design of bank-to-turn missiles using fuzzy basis function networks," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 35, no. 2, pp. 197–207, 2005.

[96] S. Ferrari, J. E. Steck, and R. Chandramohan, "Adaptive feedback control by constrained approximate dynamic programming," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 38, no. 4, pp. 982–987, 2008.

[97] H. He, Z. Ni, and J. Fu, "A three-network architecture for on-line learning and optimization based on adaptive dynamic programming," *Neurocomputing*, vol. 78, no. 1, pp. 3–13, 2012.

[98] F. Liu, J. Sun, J. Si, W. Guo, and S. Mei, "A boundedness result for the direct heuristic dynamic programming," *Neural Networks*, vol. 32, pp. 229–235, 2012.

[99] Z. Ni, H. He, and J. Wen, "Adaptive learning in tracking control based on the dual critic network design," *IEEE transactions on neural networks and learning systems*, vol. 24, no. 6, pp. 913–928, 2013.

[100] Y. Sokolov, R. Kozma, L. D. Werbos, and P. J. Werbos, "Complete stability analysis of a heuristic approximate dynamic programming control design," *Automatica*, vol. 59, pp. 9–18, 2015.

[101] Z. Ni, H. He, X. Zhong, and D. V. Prokhorov, "Model-free dual heuristic dynamic programming," *IEEE transactions on neural networks and learning systems*, vol. 26, no. 8, pp. 1834–1839, 2015.

[102] C. Mu, Z. Ni, C. Sun, and H. He, "Air-breathing hypersonic vehicle tracking control based on adaptive dynamic programming," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 3, pp. 584–598, 2017.

[103] Q. Wei, D. Liu, F. L. Lewis, Y. Liu, and J. Zhang, "Mixed iterative adaptive dynamic programming for optimal battery energy control in smart residential microgrids," *IEEE Transactions on Industrial Electronics*, vol. 64, no. 5, pp. 4110–4120, 2017.

[104] N. Zheng and P. Mazumder, "A low-power circuit for adaptive dynamic programming," in *VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID), 2018 31th International Conference on.* IEEE, 2018.

[105] N. Zheng and P. Mazumder, "A scalable low-power reconfigurable accelerator for action-dependent heuristic dynamic programming," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. PP, no. 99, pp. 1–12, 2017.

[106] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics).* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[107] D. Larkin, A. Kinane, V. Muresan, and N. OConnor, "An efficient hardware architecture for a neural network activation function generator," in *International Symposium on Neural Networks.* Springer, 2006, pp. 1319–1327.

[108] Y. Yilmaz and P. Mazumder, "Threshold read method for multi-bit memristive crossbar memory," in *Electronic System Design (ISED), 2011 International Symposium on.* IEEE, 2011, pp. 217–222.

[109] I. E. Ebong and P. Mazumder, "Self-controlled writing and erasing in a memristor crossbar memory," *IEEE Transactions on Nanotechnology*, vol. 10, no. 6, pp. 1454–1463, 2011.

[110] R. Naous, M. Al-Shedivat, and K. N. Salama, "Stochasticity modeling in memristors," *IEEE Transactions on Nanotechnology*, vol. 15, no. 1, pp. 15–28, 2016.

[111] I. Salaoru, A. Khiat, Q. Li, R. Berdan, C. Papavassiliou, and T. Prodromakis, "Origin of the off state variability in reram cells," *Journal of Physics D: Applied Physics*, vol. 47, no. 14, p. 145102, 2014.

[112] S. Kim, J. Zhou, and W. D. Lu, "Crossbar rram arrays: Selector device requirements during write operation," *IEEE Transactions on Electron Devices*, vol. 61, no. 8, pp. 2820–2826, 2014.

[113] J. Zhou, K.-H. Kim, and W. Lu, "Crossbar rram arrays: Selector device requirements during read operation," *IEEE Transactions on Electron Devices*, vol. 61, no. 5, pp. 1369–1376, 2014.

[114] M. A. Zidan, H. A. H. Fahmy, M. M. Hussain, and K. N. Salama, "Memristor-based memory: The sneak paths problem and solutions," *Microelectronics Journal*, vol. 44, no. 2, pp. 176–183, 2013.

[115] N. Zheng and P. Mazumder, "Modeling and mitigation of static noise margin variation in subthreshold SRAM cells," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 10, pp. 2726–2736, Oct 2017.

[116] N. Zheng and P. Mazumder, "An efficient eligible error locator polynomial searching algorithm and hardware architecture for one-pass Chase decoding of BCH codes," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 64, no. 5, pp. 580–584, 2017.

[117] T. Doorn, E. Ter Maten, J. Croon, A. Di Bucchianico, and O. Wittich, "Importance sampling monte carlo simulations for accurate estimation of sram yield," in *Solid-State Circuits Conference, 2008. ESSCIRC 2008. 34th European.* IEEE, 2008, pp. 230–233.

[118] T. Kida, Y. Tsukamoto, and Y. K. Renesas, "Optimization of importance sampling monte carlo using consecutive mean-shift method and its application to sram dynamic stability analysis," in *Quality Electronic Design (ISQED), 2012 13th International Symposium on.* IEEE, 2012, pp. 572–579.

[119] M. Alioto, "Ultra-low power vlsi circuit design demystified and explained: A tutorial," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 59, no. 1, pp. 3–29, 2012.

[120] B. H. Calhoun and A. P. Chandrakasan, "Static noise margin variation for subthreshold sram in 65-nm cmos," *IEEE Journal of solid-state circuits*, vol. 41, no. 7, pp. 1673–1679, 2006.

[121] B. Zhai, S. Hanson, D. Blaauw, and D. Sylvester, "A variation-tolerant sub-200 mv 6-t subthreshold sram," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 10, pp. 2338–2348, 2008.

[122] N. Drego, A. Chandrakasan, and D. Boning, "Lack of spatial correlation in mosfet threshold voltage variation and implications for voltage scaling," *IEEE Transactions on Semiconductor Manufacturing*, vol. 22, no. 2, pp. 245–255, 2009.

[123] J. H. Drew, D. L. Evans, A. G. Glen, and L. M. Leemis, "Transformations of random variables," in *Computational Probability.* Springer, 2017, pp. 47–56.

[124] K. Agarwal and S. Nassif, "The impact of random device variation on sram cell stability in sub-90-nm cmos technologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 86–97, 2008.

[125] A. J. Bhavnagarwala, X. Tang, and J. D. Meindl, "The impact of intrinsic device fluctuations on cmos sram cell stability," *IEEE journal of Solid-state circuits*, vol. 36, no. 4, pp. 658–665, 2001.

[126] K. Takeuchi, R. Koh, and T. Mogami, "A study of the threshold voltage variation for ultra-small bulk and soi cmos," *IEEE Transactions on Electron Devices*, vol. 48, no. 9, pp. 1995–2001, 2001.

[127] M. Sumita, S. Sakiyama, M. Kinoshita, Y. Araki, Y. Ikeda, and K. Fukuoka, "Mixed body bias techniques with fixed v/sub t/and i/sub ds/generation circuits," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 1, pp. 60–66, 2005.

[128] Y. Pu, J. P. De Gyvez, H. Corporaal, and Y. Ha, "An ultra-low-energy multi-standard jpeg co-processor in 65 nm cmos with sub/near threshold supply voltage," *IEEE Journal of Solid-State Circuits*, vol. 45, no. 3, pp. 668–680, 2010.

[129] H. Mostafa, M. Anis, and M. Elmasry, "A novel low area overhead direct adaptive body bias (d-abb) circuit for die-to-die and within-die variations compensation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 10, pp. 1848–1860, 2011.

[130] Y. Lim and M. P. Flynn, "A 100 ms/s, 10.5 bit, 2.46 mw comparator-less pipeline adc using self-biased ring amplifiers," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 10, pp. 2331–2341, 2015.

[131] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic, *Digital integrated circuits*. Prentice hall Englewood Cliffs, 2002, vol. 2.

[132] H. Mostafa, M. Anis, and M. Elmasry, "Adaptive body bias for reducing the impacts of nbti and process variations on 6t sram cells," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 12, pp. 2859–2871, 2011.

[133] F. Sun, K. Rose, and T. Zhang, "On the use of strong bch codes for improving multilevel nand flash memory storage capacity," in *IEEE Workshop on Signal Processing Systems (SiPS): Design and Implementation*, 2006.

[134] Y. Wu, "Fast chase decoding algorithms and architectures for reed–solomon codes," *IEEE Transactions on Information Theory*, vol. 58, no. 1, pp. 109–129, 2012.

[135] X. Zhang, J. Zhu, and Y. Wu, "Efficient one-pass chase soft-decision bch decoder for multi-level cell nand flash memory," in *2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, 2011, pp. 1–4.

[136] X. Zhang, "An efficient interpolation-based chase bch decoder," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 60, no. 4, pp. 212–216, 2013.

[137] C.-H. Yang, T.-Y. Huang, M.-R. Li, and Y.-L. Ueng, "A 5.4 soft-decision bch decoder for wireless body area networks," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 9, pp. 2721–2729, 2014.

[138] Y.-M. Lin, C.-L. Chen, H.-C. Chang, and C.-Y. Lee, "A 26.9 k 314.5 mb/s soft (32400, 32208) bch decoder chip for dvb-s2 system," *IEEE journal of solid-state circuits*, vol. 45, no. 11, pp. 2330–2340, 2010.

[139] I. Giacomelli, "Improved decoding algorithms for reed-solomon codes," *arXiv preprint arXiv:1310.2473*, 2013.

[140] Y. Wu, "New list decoding algorithms for reed-solomon and bch codes," in *2007 IEEE International Symposium on Information Theory*. IEEE, 2007, pp. 2806–2810.

[141] J.-P. Deschamps, *Hardware implementation of finite-field arithmetic.* McGraw-Hill, Inc., 2009.

[142] M. Sudan, "Algebra and computation," February 2012, lecture notes in computer science available : http://people.csail.mit.edu/madhu/ST12/scribe/lect06.pdf.

[143] X. Zhang and K. K. Parhi, "Fast factorization architecture in soft-decision reed-solomon decoding," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 4, pp. 413–426, 2005.

[144] A. Halbutogullari and Ç. K. Koç, "Mastrovito multiplier for general irreducible polynomials," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 503–518, 2000.