creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

Ph.D. DISSERTATION

Designing a Flexible, Efficient, and Scalable Brain Simulation Accelerator

유연성, 효율성, 확장성 높은 두뇌 시뮬레이션 가속기 설계

BY

Hunjun Lee

AUGUST 2023

DEPARTMENT OF ELECTRICAL AND
COMPUTER ENGINEERING
SEOUL NATIONAL UNIVERSITY

Designing a Flexible, Efficient, and Scalable Brain Simulation Accelerator

유연성, 효율성, 확장성 높은
두뇌 시뮬레이션 가속기 설계

지도교수 김 장 우
이 논문을 공학박사 학위논문으로 제출함

2023년 6월

서울대학교 대학원

전기 · 정보 공학부

이 헌 준

이헌준의 공학박사 학위 논문을 인준함

2023년 6월

위 원 장:	김 재 준	(인)
부위원장:	김 장 우	(인)
위 원:	심 재 웅	(인)
위 원:	허 재 혁	(인)
위 원:	김 한 준	(인)

Abstract

To fully understand and utilize the compute capabilities of the brain, neuroscientists rely on brain simulations that incorporate the concept of time into their operating model. The simulation involves computing the neuronal state changes over time and communicating the spikes via synapses. It also simulates learning by evaluating how the synapses change their weights in response to the spiking activity of the neurons. To explore various behaviors of the brain and thus make great advances, researchers demand a simulation framework to support large-scale simulations in both a flexible and efficient manner. However, existing simulation frameworks are too slow and inefficient due to their software-based simulation methodology. The hardware-based accelerators have mitigated the performance overhead to some extent, but they are still sub-optimal due to their inflexible hardware designs and excessive computation and synchronization overhead at each time interval.

In this dissertation, we introduce the FlexBrain project to design a flexible, efficient, and scalable brain simulation framework. In this project, we design three different hardware designs (i.e., *FlexLearn*, *NeuroEngine*, and *NeuroSync*) to meet the three goals in a step-by-step manner. First, FlexLearn achieves high flexibility by identifying and supporting 17 representative sub-rules, which can be combined to simulate target learning rules. Second, NeuroEngine enables event-driven computations of the neuronal state changes to greatly reduce the excessive computation overhead using its specialized datapath and memory units. Third, NeuroSync enables a scalable and accurate simulation by adopting speculative synchronization while supporting rollback and recovery mechanisms to ensure simulation accuracy. The three architectures can be seamlessly integrated to design FlexBrain simulator, which serves as an optimal brain simulation framework.

keywords: Brain Simulation, Neuromorphic Computer, Spiking Neural Network, Acceleration System

student number: 2018-24440

Contents

Abstract **i**

Contents **ii**

List of Tables **v**

List of Figures **vi**

1 Introduction **1**

2 Background **6**

2.1 Organization of the Brain 6

2.2 Modeling the Brain 7

2.3 Simulating the Brain 9

2.3.1 Single Core Simulation 9

2.3.2 Multicore Simulation 11

3 FlexLearn: Fast and Highly Efficient Brain Simulations Using Flexible On-Chip Learning **13**

3.1 Motivation & Design Goals 13

3.1.1 High Computational Overheads 13

3.1.2 Limitations of the Existing Accelerators 15

3.1.3 Design Goals 16

3.2 FlexLearn Architecture 16
3.2.1 Target Learning (Sub-)Rules 17
3.2.2 Efficient Integration of the Datapaths 30
3.2.3 End-to-End Brain Simulation Processor 32
3.3 Evaluation 34
3.3.1 Experimental Setup 34
3.3.2 Flexible & Efficient Brain Simulations 36
3.3.3 Low Hardware Costs 38

4 NeuroEngine: A Hardware-Based Event-Driven Simulation System for Advanced Brain-Inspired Computing 40
4.1 Motivation 40
4.1.1 Categories of Neuron Models 40
4.1.2 Time-Driven & Event-Driven Simulation 42
4.2 Challenges 45
4.2.1 Complex Computations 45
4.2.2 Complex Scheduling 47
4.2.3 Resource Contention 47
4.3 NeuroEngine Architecture 48
4.3.1 Optimized Datapath 48
4.3.2 Composite Queue 50
4.3.3 Lazy Update 53
4.4 Implementation 58
4.4.1 NeuroEngine Hardware 58
4.4.2 Software Stack 59
4.5 Evaluation 61
4.5.1 Experimental Setup 61
4.5.2 Efficiency of the Hardware-based Solutions 62
4.5.3 Low Event-Driven Computation Costs 63

4.5.4 Eliminated Redundant Event Scheduling 65
4.5.5 Speedup Using Lazy Update 66
4.5.6 Fast & Efficient Event-Driven Simulations 66
4.5.7 Iso-area Comparisons 67

5 NeuroSync: A Scalable and Accurate Brain Simulator Using Safe and Efficient Speculation 69

5.1 Motivation 69
5.2 Key Ideas 70
5.2.1 Speculative Brain Simulation 72
5.2.2 Optimal Speculative Simulation 76
5.3 NeuroSync Architecture 80
5.3.1 Baseline Simulator 81
5.3.2 RR manager 82
5.3.3 On-Demand trace provider 83
5.3.4 Deferred Learning handler 84
5.4 Simulation Front-Ends 85
5.5 Evaluation 85
5.5.1 Experimental Setup 85
5.5.2 Results 87

6 A Flexible, Efficient, and Scalable Simulation System 93

7 Related Work 95

8 Conclusion 99

Abstract (In Korean) 120

List of Tables

3.1 Summary of representative brain simulation chips 16
3.2 Summary of the representative sub-rules for learning 17
3.3 Microcodes to configure the datapaths 31
3.4 RTL synthesis results of Loihi accelerator and FlexLearn 34
3.5 Evaluated biological neural networks 35
3.6 Evaluated learning rule configurations 36

4.1 Chip area of Time and NeuroEngine. 61
4.2 Evaluated brain simulation benchmarks. 61
4.3 Area and power consumption of the datapath. 63

5.1 Evaluated simulation benchmarks. 86
5.2 Chip area comparison (mm^2). 89

List of Figures

2.1 Biological neurons and synapses 7
2.2 Basic mechanism of a biological network. 8
2.3 Illustration of the multicore brain simulation. 10

3.1 Breakdown of the simulation latency of Brunel network on the server-class CPU and GPU with representative learning rule configurations . 14
3.2 Changes in traces with respect to spikes 19
3.3 Datapaths for calculating traces 20
3.4 Datapaths for long-term plasticity 24
3.5 Datapath for short-term plasticity: FD 27
3.6 Datapaths for homeostasis 27
3.7 Datapath Integration in FlexLearn Architecture 30
3.8 Example flexible brain simulation processor with FlexLearn 32
3.9 Working model of our example processor 33
3.10 Speedup of FlexLearn over the CPU 37
3.11 Improvements in the simulation efficiency of FlexLearn over the CPU, GPU, and Loihi. The dotted lines indicate the simulations not supported by Loihi. 38
3.12 Area and power consumption of the per-rule datapaths and FlexLearn 39

4.1 Behavior difference of the one-stage and two-stage neurons for different input patterns. 40
4.2 Neuron computations in time-driven simulation and event-driven simulation. 42
4.3 Potential of the event-driven simulation. 44
4.4 Occurrence of a dead event. 46
4.5 Overview of the NeuroEngine architecture. 48
4.6 Optimized datapath for neuron computation. 49
4.7 Redundant event handling in the composite queue. 52
4.8 The working example of the event-driven simulator with and without the lazy update. 54
4.9 Example of the lazy update with a lazy neuron turning into an active neuron. 55
4.10 Datapath for safe weight initialization. 56
4.11 Programming interface and compilation toolchain. 59
4.12 The simulation latency profiling of CPU, GPU, and `Time`. 63
4.13 Total energy consumption of the datapath. 64
4.14 Reduced number of neuron computations. 65
4.15 Reduced latency with increased parallelism. 65
4.16 Improvements in the simulation efficiency of NeuroEngine over time-driven and event-driven baselines. 67
4.17 The speedup and energy efficiency of `NeuroEngine` over the iso-area `Time` baseline. 68

5.1 Profile result on the (a) underutilized cycles and (b) energy breakdown of the time-precise simulator using `Izhikevich` benchmark. 69
5.2 Profile result on the ratio of the computations that can be speculatively executed without waiting for the spikes from other cores. 71

5.3 Coincidence measure of the time-precise simulator and speculative simulators with relaxed synchronization. 71
5.4 Mechanism of the rollback and recovery procedure in Safe Speculation. The figure illustrates the voltage changes of the destination neuron (n_0) and the weight changes of the fan-in synapses. 74
5.5 Trace update stage of the baseline and Collective Trace Update. 76
5.6 Area and latency overhead of the rollback and recovery due to post-learning (`Post Learn`) and other types of computations (`Others`). The results are normalized to the total overhead. 77
5.7 Learning and the second-order rollback and recovery procedure of the baseline and the optimized mechanism using Deferred Learning. . . . 78
5.8 Overview of the NeuroSync architecture. 80
5.9 Working mechanism of RR manager. 82
5.10 Working mechanism of On-Demand trace provider. 83
5.11 Working mechanism of Deferred Learning handler. 84
5.12 Breakdown of the simulation cycles on three different baseline simulators (Normalized to the total cycles of `Time-Precise`). 87
5.13 Relative remote trace memory access of `NeuroSync` normalized to that of `Time-Precise` 88
5.14 Total amount of dirty copies of `NeuroSync` normalized to that of `Speculative` 88
5.15 Improvements in the brain simulation efficiency of `NeuroSync` over `Time-Precise` and `Speculative`. 90
5.16 Latency of the CPU-based solution and `NeuroSync` normalized to the busy cycles of `NeuroSync`. 91
5.17 Speedup and energy efficiency of `NeuroSync` normalized to those of `Time-Precise` at various chip scales. 91

Chapter 1

Introduction

One of the grand challenges in neuroscience is to understand the mechanism of the brain, modeled as a biological network of neurons and synapses. To achieve the goal, researchers heavily rely on simulations that incorporate the concept of time to closely mimic how the brain behaves over time. In the simulations, each neuron keeps internal states that change over time. If the internal states meet a certain condition, the neuron fires a spike and transfers the spike to the connected neurons via synapses. The spike triggers learning to change the weight of the synapses it passes through and affects the internal states of the connected neurons by the weight. By evaluating how the neuronal and synaptic states evolve, researchers can more closely understand the underlying mechanism of the brain (e.g., information processing and memory formation [24, 62, 64]) and utilize them for various purposes (e.g., object detection and cures for brain disorders [31, 133]).

To evaluate a wide spectrum of biological networks and thus to fully understand the brain, the simulator should meet the three key requirements. First, the simulations demand a highly flexible simulator supporting diverse learning rules to describe different biological learning patterns [85]. For example, the simulator should support learning rules related to temporary weight changes (lasting up to a few seconds) as well as learning rules related to permanent weight changes. Second, the simulator should be

efficient to simulate long-term behaviors of the brain, such as memory formation [12] and disease recovery [133], which span over a long time scale (e.g., months or years). Lastly, the simulator should support large-scale networks such as full-scale anatomical data (e.g., Drosophila [109], visual cortex [99]) to identify various biological features not observed in small-scale networks. **As a result, demystifying the brain demands a flexible, efficient, and scalable simulator capable of simulating diverse biological phenomena.**

To satisfy the flexibility demands, neuroscientists typically rely on general-purpose processors offering high programmability (e.g., CPUs [14, 45, 48, 56], GPUs [34, 58, 87, 139]). Unfortunately, their limited performance makes it difficult for neuroscientists to achieve fast and highly efficient brain simulations. The CPU-based accelerators (e.g., SpiNNaker [97]) provide high flexibility to simulate any type of learning rules. GPU-based simulators (e.g., CARLsim [87], GeNN [139]) are also programmable with faster simulation speeds by exploiting the inherent parallelism in neuronal dynamics. However, they are orders of magnitude slower and energy-inefficient compared to the customized hardware-based accelerators [72, 142], making them sub-optimal when simulating large-scale networks over long time intervals.

To achieve higher performance, the accelerators like Intel's Loihi [26] or IBM's TrueNorth [5] achieve high performance by optimizing their designs for brain simulations. The accelerators have mitigated the performance overhead of the software-based simulators to some extent. However, they are still sub-optimal due to the following challenges.

- **Diverse Model Support:** First, the accelerators are inflexible, so they can simulate only a limited set of neurons and learning rules. A recent study [72] proposes a flexible digital neuron capable of simulating the diverse neuron models; however, it only supports a limited part of the brain simulation (i.e., neuron computation). Therefore, neuroscientists desperately demand a highly flexible accelerator supporting diverse neurons and learning rules.

- **Intensive Computation Overhead:** Second, the existing works still suffer from the intense computational overhead to update the internal state of the neurons. The reason is that they adopt a time-driven mechanism where the simulator updates all neurons at every time quantum of the simulation called timestep (e.g., 0.1 ms). As the number of timesteps linearly increases with the simulation interval, evaluating the internal states becomes a critical performance bottleneck, even for the specialized accelerators.
- **Unscalable Architecture:** Lastly, the existing works suffer from underutilization and excessive memory access overhead as the simulator scales. In typical accelerator designs, a core waits until all the spikes are properly transferred to the destination cores. As the spike transfer latency increases with scale, the accelerator suffers from higher underutilization. Also, the per-core data increases as the simulator scales and introduces a critical memory access overhead.

To that end, we initiate the FlexBrain project, which effectively solves the three challenges in the existing hardware designs. In the project, we propose three architectures to tackle the three challenges in a step-by-step manner.

- **FlexLearn:** FlexLearn is a highly flexible on-chip learning architecture to support biologically plausible learning rules. It can be seamlessly augmented to existing simulators to enable fast and highly efficient brain simulations. To design FlexLearn, we first identify 17 representative sub-rules from an extensive analysis of prior neuroscience publications on learning. The identified sub-rules serve as the primitives of the diverse learning rules; a learning rule is essentially a combination of the sub-rules. After classifying the sub-rules into six categories based on their biological characteristics, we design per-category datapaths which serve as the basic building blocks of FlexLearn. Next, to reduce the hardware costs, we compact the datapaths by exploiting the common sub-expressions shared among the sub-rules. To improve the performance of the datapaths, we identify dependencies between the categories and maximize parallelism between them. The resulting design, FlexLearn, is capable

of simulating the diverse learning rules through microcodes by selectively activating the necessary datapaths.

- **NeuroEngine:** NeuroEngine is a hardware-based event-driven simulator for fast and efficient brain simulations. It faithfully implements the event-driven simulation for biological neural networks with complex neuron models and synaptic delays. We propose novel schemes ranging from the specialized datapath design to memory units to maximize the benefits of the event-driven simulations. First, we propose a datapath to efficiently support the closed-form solution by splitting a complex computation into multiple sub-computations and pipelining them. Then, we approximate the existing prediction model to further reduce the hardware cost without affecting functionality. Next, we propose a composite queue structure by combining two different queues with auxiliary data structures. The composite queue detects the computations that become redundant when scheduling the events and automatically removes them from the queue. Lastly, NeuroEngine parallelizes the computation and the spike transfer by using the lazy update method. By using the method, it identifies the computations that can bypass the event scheduling and parallelize them with other computations.
- **NeuroSync:** NeuroSync is a novel speculative brain simulation accelerator to enable scalable simulations. The key idea of NeuroSync is to speculatively change the simulation order to maximize the computational efficiency while guaranteeing simulation accuracy. We propose two key mechanisms as follows for both accurate and efficient speculative simulations. First, we propose a speculative simulation where the cores speculatively advance the timestep within a predefined period without waiting for the other cores to finish computations and communications. We also propose a rollback and recovery mechanism to identify and handle the mis-speculated states. Second, we identify and exploit a new data reuse opportunity. We observe that the memory accesses to update the traces, which represent the spiking activity of the neurons, incur a critical energy overhead. To mitigate the overhead, NeuroSync col-

lectively updates a single trace over multiple timesteps at once and temporally reuses the traces during the collective update procedure.

For evaluation, we implement a custom cycle-accurate simulator to evaluate the three architectures and fully release the simulator as an open source to contribute to the brain simulation communities. Then, we write Register-Transfer Level (RTL) code in Verilog and synthesize the code using open-source (i.e., 45-nm NanGate FreePDK45) and commercially available (i.e., 28-nm and 65-nm Samsung) cell library to extract the timing, energy, and area parameters. Using the simulator, we compare the performance of the three architectures against that of the CPU- and GPU-based simulators and the baseline architecture modeled after Loihi [26].

The rest of the dissertation is organized as follows. Chapter 2 illustrates an overview of the brain and how neuroscientists model and simulate the brain. Chater 3 describes how FlexLearn enables flexible simulations with different learning rules. Chater 4 describes how NeuroEngine improves the simulation efficiency using an event-driven simulation methodology. Chapter 5 describes how NeuroSync reduces the synchronization overhead using a speculative synchronization with rollback and recovery mechanism. Chapter 7 discusses the related publications, and Chapter 8 concludes the dissertation.

Chapter 2

Background

2.1 Organization of the Brain

In the brain, neurons transmit electrochemical signals, called spikes, over synapses to communicate with each other. A neuron consists of three major components: dendrites, a soma, and axons (Figure 2.1). Dendrites act as receivers of spikes sent from neighboring neurons through synapses. The spikes are then relayed to the soma, the body of the neuron. The soma maintains the neuron's internal states (e.g., voltage), which tracks the temporal history of the received spikes. When the neuron's voltage reaches the threshold voltage, a pre-defined threshold, the soma fires a spike and transmits the spike to other neurons through the axons. During the process, the voltage slowly decays over time and maintains a steady state if no input spikes are received for a long period of time.

When a neuron receives a spike through a synapse, the neuron's voltage changes with respect to the type and weight of the synapse. Synapses are typically classified into two types: excitatory and inhibitory synapses. Excitatory synapses increase the voltage, whereas inhibitory synapses decrease the voltage. In addition to the type of a synapse, the amount of change in the voltage depends on the strength of the synapse, known as synaptic weight. As a result, a neuron's voltage tends to depend more heavily

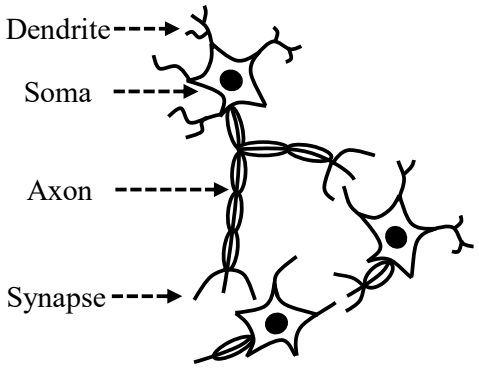


Figure 2.1: Biological neurons and synapses

on spikes from the neurons connected via synapses with higher weights rather than those from synapses with lower weights. Such spike-triggered changes in the voltage incur the complex behaviors of the human brain [50].

Learning, also known as synaptic plasticity, is the ability of synapses to strengthen or weaken over time and is widely believed as an underlying mechanism of memory [59,78,80,102]. The changes in synaptic weights depend on various biological phenomena: spiking activity and voltage of the connected neurons, synaptic weight value, and even the external stimulus. Depending on how each learning rule correlates which phenomena, it mimics different behavior of the brain. Also, various sub-rules explaining different biological phenomena interact with each other to form a single learning rule and explain high level functions of the brain (e.g., memory formation [57, 141]).

2.2 Modeling the Brain

The brain is modeled as a directed graph whose nodes are neurons, and the edges are synapses. Figure 2.2 illustrates an example of the internal dynamics of a biological network. A neuron n_0 is connected to the other neuron n_1 via a synapse $(n_0 - n_1)$. In this case, n_1 is the destination neuron for n_0 and n_0 is the source neuron for n_1. Also, the synapse is a fan-out synapse of the source neuron and a fan-in synapse of

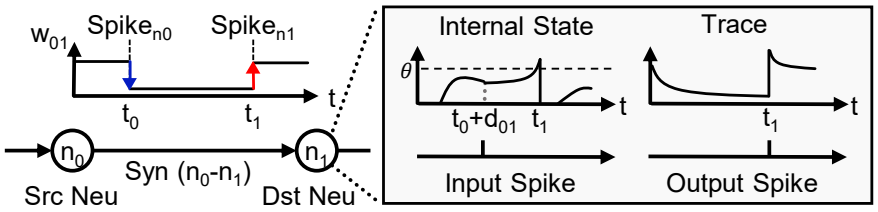


Figure 2.2: Basic mechanism of a biological network.

the destination neuron. In other words, n_0 is a source neuron for the synapse, and n_1 is the destination neuron for the synapse. Also, a neuron can be connected to multiple destination neurons.

Each neuron keeps time-varying internal states including the voltage that change with the incoming spikes and time. It also keeps traces that change with the output spikes of the neuron and time. If one of the states called voltage exceeds a certain threshold, the neuron fires a spike. Then, the spike triggers learning which changes the weight of the synapses with respect to the traces of the connected neurons. After that, the spike is transferred to the destination neurons. In turn, it affects the states regarding the weight and delay of the synapses it passes through.

In the example, the source neuron (n_0) fires a spike at t_0. Thus, the weight of the fan-out synapse ($n_0 - n_1$) changes due to the spike at t_0 (marked with a blue line). At the same time, the spike is transferred to the destination (n_1) and changes the states by the weight (w_{01}) after delay (d_{01}). A spike also triggers learning of the fan-in synapses; the spike at t_1 from the destination neuron (n_1) changes the weight (w_{01}) with respect to the traces (marked with a red line). Note that learning triggered by the source neuron (n_0) is called pre-learning, and learning triggered by the destination neuron (n_1) is called post-learning. Such temporal dynamics of the neurons and the synapses form the basis of the cognitive functions in the brain.

Algorithm 1: A typical brain simulation procedure.

```
1  # t_cur: current simulation timestep
2  # neurons: a list of all neurons
3  # n.fan-out: a list of outgoing edges (dst,wgt,delay)
4  # n.fan-in: a list of incoming edges (src,wgt,delay)
5  # n.states: internal states of neuron n
6  # n.acc[t]: accumulated weight (wgt) for neuron n at t
7  # spike_buf: a buffer for the generated spikes
8  While(t_cur < t_max):
9      For(n in neurons): # neuron comp. & trace update
10         fired = neu_comp(n.states, n.acc[t_cur])
11         If (fired): spike_buf.append(n)
12         trace_update(n.traces, fired)
13     For (n in spike_buf): # learning & wgt accum.
14         For ((src,w,d) in n.fan-in): # post-learning
15             post_learn(src.traces, n.traces,w)
16         For((dst,w,d) in n.fan-out): # pre-learning
17             pre_learn(n.traces, dst.traces,w)
18         For((dst,w_new,d) in n.fan-out): # wgt accum.
19             dst.acc[t_cur + d] += w_new
20     t_cur += Δt, spike_buf.reset()
```

2.3 Simulating the Brain

2.3.1 Single Core Simulation

Simulating the brain involves evaluating the neuronal and the synaptic state changes during a fixed amount of time to simulate the temporal dynamics. Algorithm 1 shows an example simulation procedure. The simulation divides time into discrete timesteps (Δt), representing the time quantum of the simulation (e.g., 1 ms in biological time). There are four stages for each timestep to accurately evaluate the state changes. The stages include:

Neuron Computation: First, the simulator iterates over all the neurons and updates the internal states (e.g., voltage) (Line 9–11) using the accumulated weight of the incoming spikes (`n.acc[t_cur]`). When the voltage exceeds a certain threshold,

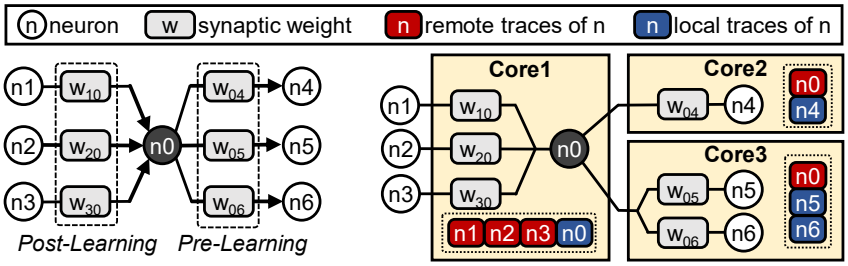


Figure 2.3: Illustration of the multicore brain simulation.

the neuron fires a spike. The simulator keeps the list of neurons that fire a spike (`spike_buf`). Note that the internal states change with both incoming spikes and time; thus, the simulator performs neuron computation even in the absence of incoming spikes.

Trace Update: Then, the simulator updates the traces which represent the spiking activities of neurons (Line 12). The traces are used at the learning stage. Similar to the internal states, the traces evolve over time; therefore, the simulator updates them even in the absence of an output spike.

Learning: Next, the simulator evaluates the weight changes of the synapses. It performs post-learning for the fan-in synapses of the neuron that fires a spike. For each synapse, it updates the weight using the traces of the source and destination neuron (Line 14–15). After that, the simulator performs pre-learning for the fan-out synapses (Line 16–17).

Weight Accumulation: Finally, the simulator iterates over the fan-out synapses for the neuron that fires a spike. For each destination neuron, it accumulates the updated weight (`wgt_new`) after the delay (`delay`) (Line 18–19).

After the four stages, the simulator proceeds the timestep (Line 20) and terminates if the simulation time reaches a pre-defined deadline (`t_max`). By sequentially iterating over the timesteps, the simulator precisely models the dynamics of the brain.

2.3.2 Multicore Simulation

To simulate large-scale networks, existing works adopt a multicore simulator where each core simulates a dedicated set of neurons and fan-in synapses of the neurons [8, 23, 26, 94]. Figure 2.3 illustrates how the multicore simulator performs brain simulations. In the example, a neuron (n_0) is connected to three source neurons ($n_1 - n_3$) and three destination neurons ($n_4 - n_6$). Core 1 simulates n_0, Core 2 simulates n_4, and Core 3 simulates n_5 and n_6, respectively. For simplicity, we do not include the cores simulating $n_1 - n_3$ in the figure. We use the term local neurons to indicate the dedicated neurons and remote neurons to indicate the source neurons of the local neurons (e.g., $n_1 - n_3$ for Core 1).

For learning, the core requires traces of the source and destination neuron for a target synapse; therefore, the core keeps and updates the traces of both local neurons (i.e., local traces) and remote neurons (i.e., remote traces). When a neuron (n_0) fires a spike, Core 1 performs post-learning which updates the weight for all the fan-in synapses (w_{10}, w_{20}, w_{30}). At the same time, it transfers the spike to the destination cores (Core 2 and Core 3). After receiving the spike, the cores perform pre-learning for the fan-out synapses (w_{04}, w_{05}, w_{06}) of the source neuron (n_0) and weight accumulation using the updated weight. The spike also affects the remote traces of the source neuron (n_0) at the destination cores.

However, multicore simulation may suffer from inaccuracy. As the cores operate in parallel, some cores proceed to the next timestep while other cores are still updating the states of the previous timestep. In such a case, a core may suffer from a *belated spike arrival* where the core receives a spike that should have arrived at a previous timestep. The belated spike arrival causes simulation inaccuracy as neuron computation for the destination neurons, trace update for the source neuron, weight accumulation, and learning are all affected by the timing of the spike.

Therefore, existing simulators [8, 26, 38, 39] operate in a *time-precise* manner. In the time-precise simulation, all the cores wait until all the other cores finish the com-

putations and communications. Even if a core finishes all the required computations, it does not proceed to the next timestep as it may receive a belated spike after it advances the timestep. Instead, the core waits until it receives a synchronization message which indicates that all the cores finish the required computations and communications. Using the method, the simulator guarantees that all the cores simulate the same timestep. Therefore, it prevents a belated spike arrival and does not suffer from simulation inaccuracy.

Chapter 3

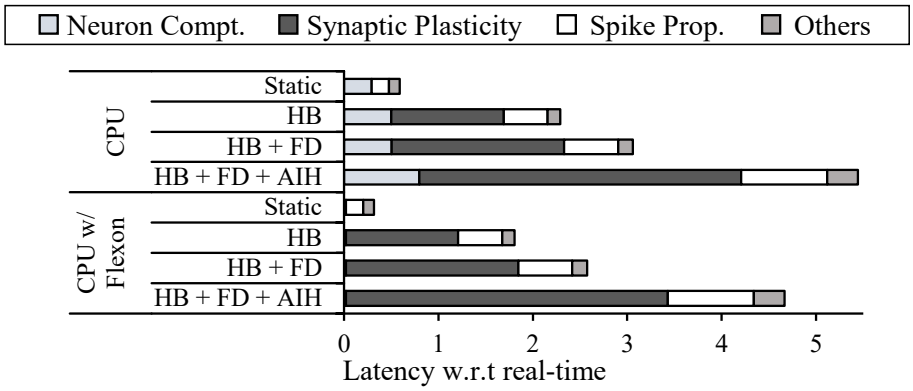
FlexLearn: Fast and Highly Efficient Brain Simulations Using Flexible On-Chip Learning

3.1 Motivation & Design Goals

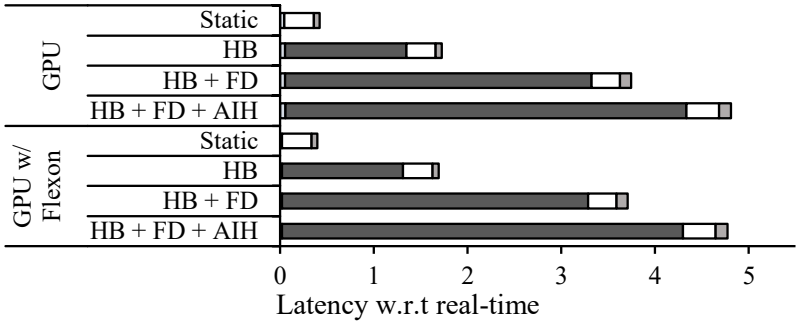
3.1.1 High Computational Overheads

To examine which of the brain simulation stages contributes the most to the latency, we profile a representative biological neural network used by Brunel [19], with diverse learning rules. The Brunel network consists of 5,750 neurons. The neurons are connected randomly with a 10% probability, resulting in the network having 3,306,250 synapses. Learning rules affect only 2,116,000 synapses while the remaining synapses remain constant. The simulation uses a timestep of 0.1 ms to run for 1 second of biological time (i.e., 10,000 timesteps). The underlying server-class CPU and GPU are 12-core Intel Xeon E5-2650 v4 CPU [1] and NVIDIA Titan X (Pascal) GPU [2], respectively. We use NEST [45] simulator for the CPU and GeNN [139] for the GPU.

Figure 3.1 shows the latency breakdown with different learning configurations; `Others` indicates the unclassified functions (e.g., network construction and random input generation). The profiling results show that the synaptic plasticity stage incurs the highest latency. The details of the learning configurations are discussed in Sec-



(a) Intel Xeon E5-2650 v4 CPU



(b) NVIDIA Titan X (Pascal) GPU

Figure 3.1: Breakdown of the simulation latency of Brunel network on the server-class CPU and GPU with representative learning rule configurations

tion 3.2.1. When the learning configuration enables only one sub-rule (`HB`), the synaptic plasticity stage contributes to 51.9% and 75.0% of the total latency on the CPU and the GPU, respectively. Enabling more sub-rules increases the contribution of the synaptic plasticity stage to the total latency; the synaptic plasticity stage incurs 88.9% of the total latency on the GPU when three sub-rules are enabled (`HB+FD+AIH`).

We further profile the simulation latency in case the CPU and the GPU cores offload the neuron computation to Flexon [72], the state-of-the-art digital neuron supporting diverse neuron models. With the digital neuron, the synaptic plasticity stage incurs up to 73.0% and 89.6% of the total latency on the CPU and the GPU, respec-

tively. In addition, the simulations lead to 5.4× and 4.7× higher latency with respect to the real-time on the CPU and the GPU, respectively. (i.e., it takes 5.4 and 4.7 seconds to simulate 1 second of biological time). Such slow simulation speeds are detrimental as various learning rules operate on large timescales (e.g., hours to days [62,131,141]). In summary, the synaptic plasticity stage is the major performance bottleneck and should be accelerated for fast and highly efficient simulations.

3.1.2 Limitations of the Existing Accelerators

As different learning rules work together to explain various phenomena of the human brain, brain simulation accelerators should be highly flexible by supporting diverse learning rules. Unfortunately, we find that the existing accelerators are either slow or inflexible. Table 3.1 summarizes the limitations of the representative accelerators. First, to achieve high flexibility, SpiNNaker [97] is equipped with low-power CPU cores; however, its performance is limited by the CPU cores and incurs higher latency than the other accelerators. Second, IBM's TrueNorth [5] is a highly efficient neurosynaptic processor. TrueNorth achieves highly efficient neuron computation; however, it lacks on-chip learning support. There are efforts to implement on-chip learning on TrueNorth [93] by using its neuron cores to emulate the learning; however, it incurs high overheads as additional neuron cores and connections are required to emulate the synaptic behavior. Third, Loihi [26], the state-of-the-art neuromorphic accelerator with on-chip learning support, has limited support toward diverse neuronal behaviors and various learning rules. It supports only a limited set of biological phenomena which can be expressed with a simple sum-of-products formula. For instance, it cannot simulate STP which greatly enriches a biological neural network's dynamical behaviors [129], complex spike pairing methods such as suppression model, and the effect of neurons' membrane potential on stabilization. Fourth, Flexon [72] is a flexible digital neuron capable of simulating the diverse neurons of the human brain. It achieves high flexibility in terms of neurons; however, it only supports the neuron computation

Table 3.1: Summary of representative brain simulation chips

Name	Performance	Flexibility		Brain Simulations
		Learning	Neuron	
SpiNNaker [97]	Low	High	High	✓
TrueNorth [5]	High	Low	Low	✗
Loihi [26]	High	Moderate	Low	✗
Flexon [72]	High	None	High	✗
FlexLearn (Ours)	**High**	**High**	**High**	✓

stage of the brain simulation. Therefore, a flexible hardware accelerator supporting the diverse learning rules is a key requirement of brain simulations.

3.1.3 Design Goals

Motivated by the limited performance and flexibility of the existing accelerators, we need a new highly flexible and efficient accelerator achieving the following design goals. First, it should be flexible by supporting the diverse learning rules of the human brain. Second, it should achieve high efficiency in simulation speed and energy efficiency. Third, it should be easily applicable to the existing accelerators to realize the end-to-end brain simulation (e.g., digital neurons [72, 115]).

3.2 FlexLearn Architecture

In this section we present *FlexLearn*, a flexible and highly efficient on-chip learning accelerator for brain simulations. We first perform an extensive analysis of prior neuroscience publications to figure out various learning rules. With the analysis, we identify 17 representative sub-rules and classify them into six different categories. Next, we design per-category datapaths and use the datapaths as the building blocks of FlexLearn (Section 3.2.1). After designing the datapaths, we integrate and pipeline the datapaths according to their dependencies (Section 3.2.2). Then, we design and present an

Table 3.2: Summary of the representative sub-rules for learning

Biological Role	Category	Sub-rule
Long-Term Plasticity (LT)	1. Hebbian (HB)	1.1. Nearest Pair [64]
		1.2. All-to-All Pair [44]
		1.3. Triplet Pair [96]
		1.4. Suppression Pair [41]
		1.5. Momentary Voltage [16]
		1.6. Average Voltage [24]
	2. Weight Dependence (WD)	2.1. Additive [120]
		2.2. Multiplicative [131]
		2.3. Sublinear [51]
		2.4. Mixed [134]
	3. Reward-Based (RB)	3.1. Three-Factor STDP [83]
Short-Term Plasticity (STP)	4. Fast Dynamics (FD)	4.1. Separate [4]
		4.2. Merged [128]
Homeostasis	5. Activity-Dependent Homeostasis (ADH)	5.1. Spike Dependent [134]
		5.2. Voltage Dependent [24]
	6. Activity-Independent Homeostasis (AIH)	6.1. Linear Decay [16]
		6.2. Exponential Decay [44]

example brain simulation processor by integrating FlexLearn with the state-of-the art flexible digital neuron to to perform end-to-end brain simulations (Section 3.2.3).

3.2.1 Target Learning (Sub-)Rules

To identify the target learning rules of FlexLearn, we perform an extensive analysis of prior neuroscience publications on the human brain [4, 16, 24, 41, 44, 51, 64, 83, 96, 120, 128, 131, 134]. The literature analysis indicates the critical need for supporting three major biological roles of learning (i.e., Long-Term Plasticity, Short-Term Plasticity, Homeostasis) whose detailed biological characteristics can be flexibly configured as a combination of sub-rules from the six categories. (i.e., Hebbian, Weight De-

pendence, Reward-Based, Fast Dynamics, Activity-Dependent Homeostasis, Activity-Independent Homeostasis)

Then, we observe that different learning rules share a set of common sub-rules from the categories and the flexible integrations of the sub-rules can implement various high-level functions of the brain [65]. Thus, implementing the sub-rules would be the most cost-effective way to implement the wide-spectrum of learning rules. More importantly, if a new learning rule is to be simulated in future, it can be effectively done by adding only unique sub-rules required by the new rule. In this way, the sub-rule implementation continues to enable flexible and efficient learning simulations. Table 3.2 summarizes our target learning rules and their sub-rules.

Next, to minimize the sub-rule implementation costs, we design a set of per-category datapaths. For the purpose, we first indentify five types of traces which track the temporal history of the biological phenomena affecting the amount of weight changes in different sub-rules. After designing the datapaths for the traces, we design the datapaths for how the sub-rules in different categories correlate the traces. By separating the datapath for the traces and the method for correlating them, we can reuse the trace datapath for different sub-rules. The datapaths are then integrated to achieve high flexibility. When integrating the datapaths, we minimize the hardware costs by eliminating the redundant sub-datapaths shared among the datapaths. In this section, we explain the mathematical models of the sub-rules in detail to show how we design and optimize the datapaths.

To facilitate mathematical descriptions of the learning rules, let us consider a synapse which transmits spikes from a pre-synaptic neuron i and a post-synaptic neuron j and the synapse's synaptic weight is w_{ij}. Then, assume that we are running a brain simulation for an interval of Δt whose current biological time is t. In such a condition, a learning rule of the synapse can be seen as a function which produces the synaptic state of the synapse (e.g., the value of w_{ij}) at time $t + \Delta t$ by taking as input 1) the synaptic state at time t, 2) whether the neurons fired a spike during $(t, t + \Delta t]$,

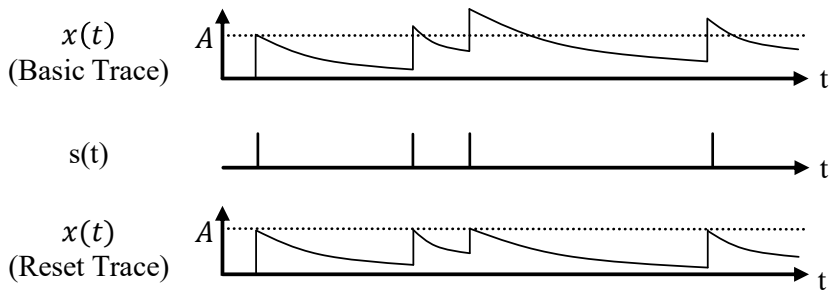


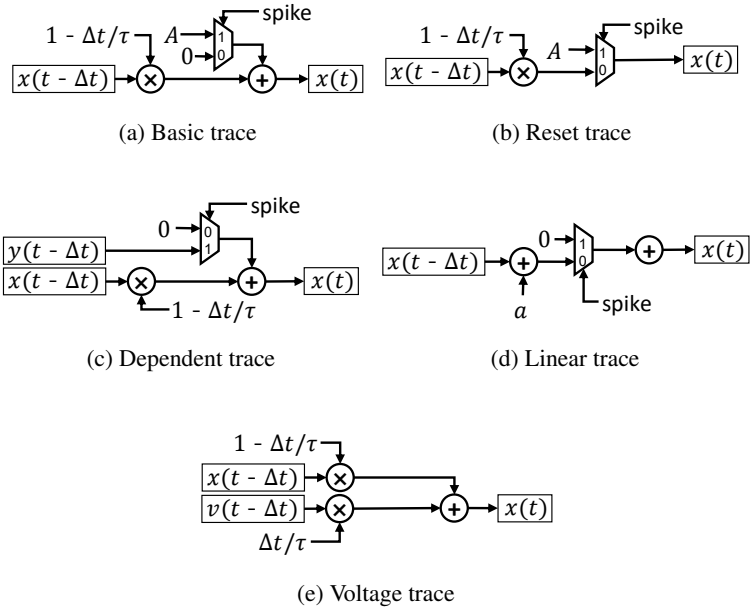
Figure 3.2: Changes in traces with respect to spikes

and 3) the set of learning rules applied to the synapse. Using these mathematical definitions, we first describe and design datapaths for the traces that capture the various biological phenomena required in the 17 sub-rules (Section 3.2.1). Then, we describe how each sub-rule correlates the trace values to calculate the amount of weight change in the synapses and design the datapaths implementing them (Section 3.2.1–3.2.1).

Traces

In the human brain, both the prior and the current activities of a neuron can incur changes in synaptic weight. To track the prior activities and the previous values within a single variable, various learning rules utilize the concept of traces [85]. The most basic form of a trace, $x(t)$ ($x(t) \geq 0$), for a neuron is defined as $x(t) = (1 - \frac{\Delta t}{\tau}) \cdot x(t - \Delta t) + A \cdot s(t)$ where $s(t)$ denotes occurence of a spike at time t and τ denotes the decay constant of the trace value. If the neuron fires the spike at time t, $s(t) = 1$; otherwise, $s(t) = 0$. Whenever the neuron fires a spike, it updates its trace value $x(t)$ by a constant A and decays its value over time in the absence of a spike. Such dynamics of a trace enable accumulation of spiking activity of the neuron within a single variable.

We find that the learning rules utilize five types of per-neuron traces: basic, reset,



(a) Basic trace

(b) Reset trace

(c) Dependent trace

(d) Linear trace

(e) Voltage trace

Figure 3.3: Datapaths for calculating traces

dependent, linear, and voltage traces. The mathematical definitions for the traces are:

$$x(t) = \begin{cases} (1 - \frac{\Delta t}{\tau}) \cdot x(t - \Delta t) + A \cdot s(t) & \text{basic trace} \\ (1 - \frac{\Delta t}{\tau}) \cdot x(t - \Delta t) \cdot \overline{s}(t) + A \cdot s(t) & \text{reset trace} \\ (1 - \frac{\Delta t}{\tau}) \cdot x(t - \Delta t) + e(t - \Delta t) \cdot s(t) & \text{dependent trace} \\ (a + x(t - \Delta t)) \cdot \overline{s}(t) & \text{linear trace} \\ (1 - \frac{\Delta t}{\tau}) \cdot x(t - \Delta t) + \frac{\Delta t}{\tau} \cdot v(t - \Delta t) & \text{voltage trace} \end{cases}$$

where the $\overline{s}(t) = 1$ in the absence of the spike and $\overline{s}(t) = 0$ if the neuron fires the spike. Also, e in the dependent trace represents an arbitrary trace (e.g., reset trace) that the trace depends on. Then, a in the linear trace is a pre-defined constant, and $v(t - \Delta t)$ in the voltage trace is the neuron's membrane potential at $t - \Delta t$. Figure 3.2 demonstrates how traces change with respect to the neuron's spiking activities. In the example, when a neuron fires a spike, the basic trace increases by A whereas the reset trace is set to A. In this way, the basic trace captures all previous spiking history; however, the reset trace tracks only the most recent spiking activity. Figures 3.3a–3.3e

show the datapaths implementing the five traces; `spike` in the figures denotes whether the neuron fires a spike at the given timestep.

Long-Term Plasticity

Long-Term Plasticity (LT) is the first type of the biological roles we consider. Two forms of LT exist: Long-Term Depression (LTD) and Long-Term Potentiation (LTP) which weaken and strengthen inter-neuron connections, respectively [130]. Due to their long-lasting effects, LTD and LTP are thought to be highly relevant to memory [100]. The high importance of LT has attracted significant interests from neuroscientists ever since Hebb's postulate [53] was proposed.

Three biological phenomena take part in the LT: the interaction between the pre- and post- synaptic neurons, the synaptic weight, and the external stimulus. The sub-rules from the three categories (i.e., Hebbian (HB), Weight Dependence (WD), Reward-Based (RB)) describe each of the phenomena and are combined to take part in the LT. LT is modeled as:

$$w(t) = w(t - \Delta t) + \overbrace{RB(WD_{+}(t) \cdot HB_{+}(t)}^{LTP} + \overbrace{WD_{-}(t) \cdot HB_{-}(t))}^{LTD} \tag{3.1}$$

where Δt denotes the timestep, and $w(t)$ is the synaptic weight at time t. $WD_{+}(t)$ and $WD_{-}(t)$ represent $WD(t)$ for LTP and LTD, respectively. The same applies to $HB_{+}(+)$ and $HB_{-}(t)$.

The STDP learning rule, for example, one of the most well-known methods for explaining the long term plasticity, consists of two sub-rule categories: HB and WD. Mathematically, the standard form of STDP is defined as:

$$w(t) = w(t - \Delta t) + \overbrace{(w_{max} - w(t - \Delta t)) \cdot x_i(t) \cdot s_j(t)}^{LTP} \underbrace{- w(t - \Delta t) \cdot y_j(t) \cdot s_i(t)}_{LTD}$$

where w_{max} is a pre-defined largest weight value for the synapse and the term $w_{max} - w(t-\Delta t)$ and $w(t-\Delta t)$ represent $WD_{+}(t)$ and $WD_{-}(t)$. For the HB term, each neu-

ron keeps two seperate traces $x(t)$ and $y(t)$ for pre-synapse and post-synapse, respectively. In addition, the synaptic weight decreases (LTD) when the presynpatic neuron fires (i.e., $s_i(t) = 1$), and the synaptic weight increases (LTP) when the postsynaptic fires (i.e., $s_j(t) = 1$). The $x(t)$ from the pre-synaptic neuron (i.e., $x_i(t)$) and $y(t)$ from the post-synaptic neuron (i.e., $y_j(t)$) represent each case when multiplied with spikes of the post- and pre- synaptic neuron $s_j(t)$ and $s_i(t)$. According to the definition, various long-term plasticity models have different 1) model for explaining the HB term, 2) function for WD, and 3) existence of the RB term.

Hebbian Term: This category captures the interaction between the pre- and post-synaptic neurons in terms of their spiking activities and the membrane voltage of the post-synaptic neuron. Six different sub-rules fall into this category: nearest pair, all-to-all pair, triplet pair, suppression pair, momentary voltage, and average voltage. When updating the synaptic weight, the first four models consider various spike pairs between the pre- and post- neurons by utilizing different traces. On the other hand, the last two sub-rules account for the pre-synaptic neuron's activities and the membrane voltage of the post-synaptic neuron when changing the synaptic weight.

First, the nearest-neighbor pair model considers only the two latest spikes, one from the pre-synaptic neuron and the other from the post-synaptic neuron, using the reset trace. Second, the all-to-all pair model utilizes the basic trace to make a pre-synaptic spike and a post-synaptic spike interact with all prior post-synaptic spikes and all prior pre-synaptic spikes, respectively. The two sub-rules use the HB term defined as:

$$HB_+(t) = x_i(t) \cdot s_j(t), \quad HB_-(t) = y_j(t) \cdot s_i(t) \tag{3.2}$$

The two aforementioned sub-rules only pair a single post-synaptic spike with a single pre-synaptic spike. However, such single-spike pairs cannot describe the frequency effects on synaptic weight updates [96], requiring additional traces. To do so,

the triplet and suppression sub-rules use:

$$HB_{+}(t) = x_{i1}(t) \cdot y_{j2}(t) \cdot s_j(t)$$
$$HB_{-}(t) = y_{j1}(t) \cdot x_{i2}(t) \cdot s_i(t)$$

where $x_{i1}(t)$ and $y_{j1}(t)$ correspond to $x_i(t)$ and $y_j(t)$ in Equation 3.2, and $x_{i2}(t)$ and $y_{j2}(t)$ are newly adopted traces to explain the effect of multiple spike pairs (e.g., pre-post-pre). Specifically, triplet sub-rule employs the basic traces as the two new traces:

$$x_{i1}(t) = (1 - \Delta t/\tau_{x1}) \cdot x_{i1}(t - \Delta t) + s_i(t)$$
$$x_{i2}(t) = (1 - \Delta t/\tau_{x2}) \cdot x_{i2}(t - \Delta t) + s_i(t)$$

where τ_{x1} and τ_{x2} denote the decay constants for the traces and $y_{j1}(t)$ and $y_{j2}(t)$ follow the same equation. On the other hand, suppression sub-rule defines $x_{i1}(t)$ and $y_{j1}(t)$ as the dependency traces and $x_{i2}(t)$ and $y_{j2}(t)$ as the reset traces, resulting in:

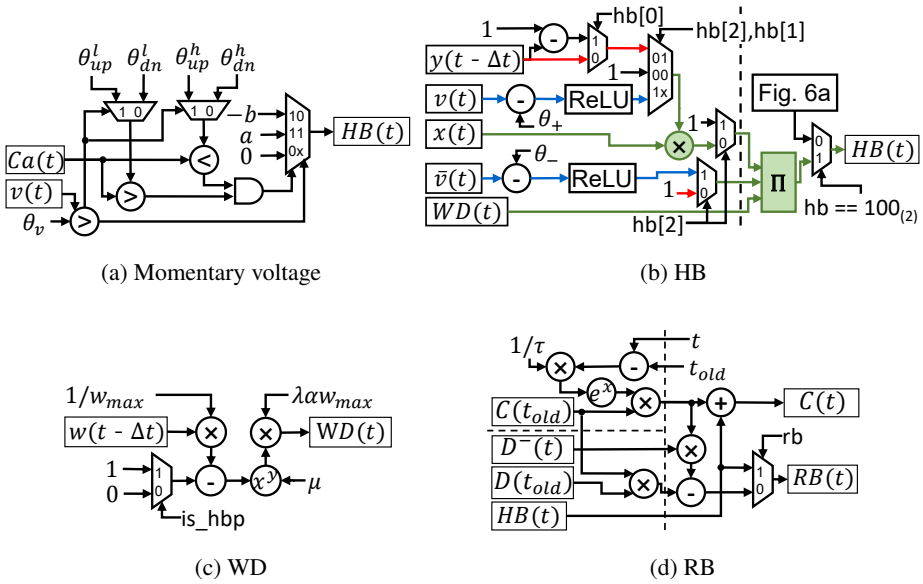
$$x_{i1}(t) = (1 - \Delta t/\tau_{x1}) \cdot x_{i1}(t - \Delta t) + s_i(t) \cdot x_{i2}(t - \Delta t)$$
$$x_{i2}(t) = (1 - \Delta t/\tau_{x2}) \cdot x_{i2}(t - \Delta t) \cdot \overline{s}_i(t) + s_i(t)$$

as the dynamics of the traces.

Until now, we discussed the HB term whose synaptic weight updates depend on spike pairings and the current synaptic weights. However, prior neuroscience publications report that the updates can also depend on the membrane voltage of post-synaptic neurons [16, 24]. Two sub-rules model this phenomenon in different manner: momentary voltage [16] and average voltage [24].

The momentary voltage model determines the direction of the synaptic weight according to the instant voltage ($v(t)$) of the post-synaptic neuron. It introduces an additional basic trace (called calcium trace) $Ca_j(t)$ to determine if a synaptic weight update should be performed or not. The model is defined as:

$$Ca_j(t) = (1 - \Delta t/\tau) \cdot Ca_j(t - \Delta t) + J_c \cdot s_j(t)$$
$$HB_{+}(t) = a \cdot s_i(t) \quad \text{if } v(t) > \theta_v \text{ and } \theta^l_{up} < Ca_j(t) < \theta^h_{up}$$
$$HB_{-}(t) = b \cdot s_i(t) \quad \text{if } v(t) \leq \theta_v \text{ and } \theta^l_{dn} < Ca_j(t) < \theta^h_{dn}$$



(a) Momentary voltage (b) HB

(c) WD (d) RB

Figure 3.4: Datapaths for long-term plasticity

where J_c is a pre-defined constant, $\theta^{\{l,h\}}_{\{up,dn\}}$ define the upper and the lower bounds of facilitation and depression, the a, and b is a pre-defined constant determining the amount of weight change when the condition is met. Figure 3.4a shows the datapath implementing the momentary voltage model; `hb` in the figure denotes the microcode in Table 3.3.

Second, the average voltage model adopts different form of voltage; the low pass filtered voltage determines the amount of synaptic weight updates. The mathematical definition of the average voltage model introduces a voltage trace $\overline{v}_{\pm}(t)$ and is defined as:

$$x_i(t) = (1 - \Delta t/\tau_x) \cdot x_i(t - \Delta t) + s_i(t)$$

$$\overline{v}_-(t) = (1 - \Delta t/\tau_-) \cdot \overline{v}_-(t - \Delta t) + \Delta t/\tau_- \cdot v(t - \Delta t)$$

$$\overline{v}_+(t) = (1 - \Delta t/\tau_+) \cdot \overline{v}_+(t - \Delta t) + \Delta t/\tau_+ \cdot v(t - \Delta t)$$

$$HB_+(t) = x_i(t) \cdot ReLU(\overline{v}_+(t) - \theta_-) \cdot ReLU(v(t) - \theta_+) \cdot s_j(t)$$

$$HB_-(t) = ReLU(\overline{v}_-(t) - \theta_-) \cdot s_i(t)$$

where $x_i(t)$ defines a basic trace and $ReLU(x) = max(0, x)$. Also, θ_+ and θ_- define

the lower bounds for voltage and voltage trace, respectively, and $\tau_{\pm}$ are pre-defined decay constants.

We reduce the hardware cost by making the datapaths share circuit for the common sub-datapaths. Figure 3.4b shows the integrated datapath for the HB term. The red line and the blue line in the figure represent the datapath for the triplet pair model and the average voltage model respectively. Also, the green line in the figure represents the common sub-expression shared among the two sub-rules. As shown in the figure, we reduce the integrated datapath by making the different sub-rules share the same circuit for the common sub-expression.

Weight Dependence: As discussed in Equation 3.1, the amount of synaptic weight changes due to LT depends on the definition of $WD_{\pm}(t)$. Weight dependence, the synaptic weight updates depending on the current synaptic weight, can lead to stability and biologically plausible distribution of synaptic weights [46]. From prior neuroscience publications, we find that four biologically plausible definitions for $WD_{\pm}(t)$ exist: additive [120], multiplicative [131], sublinear [51], and mixed [134] models. The mathematical definition is:

$$WD_{\pm}(t) = \begin{cases} \lambda, -\lambda\alpha & \text{additive} \\ \lambda w_{max}(1 - \frac{w}{w_{max}}), -\lambda\alpha w & \text{multiplicative} \\ \lambda w_{max}(1 - \frac{w}{w_{max}})^{\mu}, -\lambda\alpha w_{max}(\frac{w}{w_{max}})^{\mu} & \text{sublinear} \\ c_p, -c_d w & \text{mixed} \end{cases}$$

where the left and right formula indicates WD_{+} and WD_{-}, respectively. λ denotes a learning rate, α is the ratio of LTP and LTD, μ is a constant, and c_p and c_d represent the average amounts of potentiation and relative depression, respectively, after one spike pairing. Among the models, additive model has no dependency on the current synaptic weight and imposes hard boundaries on synaptic weights. Figure 3.4c shows the datapath implementing the four $WD_{\pm}(t)$ models; `is_hbp` in the figure denotes whether the datapath is used for ltp or for ltd. Instead of using multiplexers with microcodes to support various types of sub-rules, we select specific parameters to emulate different

sub-rules based on the micrcocode (e.g., $\mu = 1$ for multiplcative sub-rule).

Reward-Based STDP: The aforementioned sub-rules implement unsupervised learning; however, reinforcement learning can be described in a biologically plausible description [63]. Additional traces for the external stimulus track the rewards for eligible actions.

To incorporate the concept of reward to the learning rules, three-factor STDP model [63], an implementation of reward-based STDP, requires two additional traces: dopamine (D_i) and eligibility (C_{ij}) traces represent low pass filtered version of the dopaminergic spikes and HB term outcomes ($HB_{\pm}(t)$). Mathematically, three-factor STDP is defined as:

$$C_{ij}^{-}(t) = e^{(t-t_{old})/\tau_c} \cdot C_{ij}(t_{old})$$

$$C_{ij}(t) = C_{ij}^{-}(t) + WD_{\pm}(t) \cdot HB_{\pm}(t) \cdot s_{i,j}(t)$$

$$D_j^{-}(t) = (1 - \Delta t/\tau_d) \cdot D_j(t - \Delta t), \;\; D_j(t) = D_j^{-}(t) + s_d(t)$$

$$RB(t) = (C_{ij}^{-}(t) \cdot D_j^{-}(t) - C_{ij}(t_{old}) \cdot D_j(t_{old})) \cdot s_{i,j,d}(t)$$

$$t_{old} = t \cdot s_{i,j,d}(t) + t_{old} \cdot \overline{s}_{i,j,d}(t)$$

where the $s_d(t)$ is the incoming dopaminergic spikes and t_{old} is the latest spiking time of the pre-synaptic-, post-synaptic- or dopaminergic spikes. Figure 3.4d illustrates the datapath for three-factor STDP; `rb` in the figure denotes the microcode in Table 3.3.

Short-Term Plasticity

Along with the LT, an inherent dynamics lasting 100 ms to about a second exists in biological synapses [49, 79]. The short-lasting dynamics is called Short-Term Plasticity (STP). STP has been shown to encode temporal information in biological neural networks by changing the effective synaptic weights according to inter-spike intervals [42]. From a computational point of view, it is plausible that STP may be an underlying mechanism of a number of processes which occur in daily life (e.g., speech recognition) [129].

Fast Dynamics: The Fast Dynamics (FD) category is resposible for exaplaining the

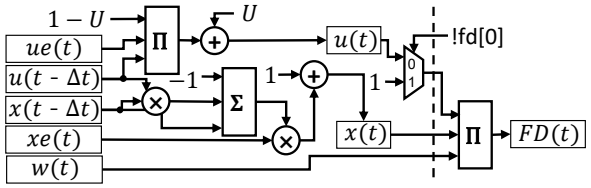
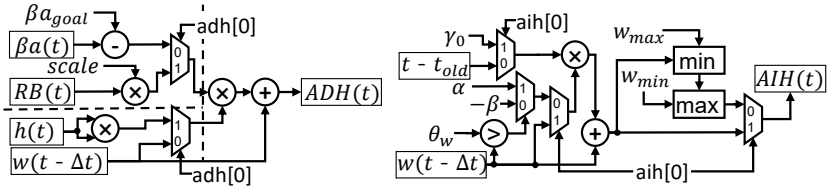


Figure 3.5: Datapath for short-term plasticity: FD



(a) ADH (b) AIH

Figure 3.6: Datapaths for homeostasis

STP: separate [4] and merged [128] model. First, the separate model is defined as:

$$P(t) = \frac{(P_0 + (\tau_P - 1) \cdot P(t - \Delta t))}{\tau_P} + f \cdot (1 - P(t - \Delta t)) \cdot s_i(t)$$

$$FD(t) = P(t) \cdot w(t)$$

where P_0, f, and τ_p is a constant and $P(t)$ is a state variable tracking the spiking activities of the pre-synaptic neuron. Also, the result of $FD(t)$ denotes the effective synaptic weight, which lasts only temporally, at time t. Although the model is simple and easy to implement, it can model only either the facilitating or the depressing effect caused by a sequence of spikes; the two effects cannot be taken into account concurrently.

Merged model avoids the limitation by employing two state variables (i.e., $x(t)$ and $u(t)$) rather than one:

$$xe(t) = (1 - \Delta t/\tau_{rec}) \cdot xe(t - \Delta t) \cdot \overline{s}_i(t) + s_i(t)$$

$$ue(t) = (1 - \Delta t/\tau_{fac}) \cdot ue(t - \Delta t) \cdot \overline{s}_i(t) + s_i(t)$$

$$x(t) = 1 + (x(t - \Delta t)\dot{(}1 - u(t - \Delta t)) - 1) \cdot xe(t - \Delta t)$$

$$u(t) = U + u(t - \Delta t) \cdot (1 - U) \cdot ue(t - \Delta t)$$

$$FD(t) = x(t) \cdot u(t) \cdot w(t) \cdot s_i(t)$$

where $xe(t)$ and $ue(t)$ denotes the temporal distance between the two latest pre-synaptic spikes with reset traces with pre-defined decay constant $\tau_{\{fac,rec\}}$. The state variables, $x(t)$ and $u(t)$ account for the depressing and the facilitating effects, respectively, of a series of spikes, and the two state variables are updated only when $s_i(t) = 1$. The variables are affected only when the pre-synaptic spike occurs. For instance, if the frequency of the pre-synaptic spikes increases, the value of $x(t)$ decreases and reduces the effective weight. The datapath implementing the two STP models is shown in Figure 3.5; `fd` in the figure denotes the microcode in Table 3.3.

Homeostasis

The last type of biological role synaptic plasticity affects is the Homeostasis; the ability of biological neural networks to modulate the activity of synapses [124, 132]. Homeostasis counter-balances the synaptic weight changes by LT in a way the synaptic weight distribution of an SNN does not deviate from a stable form. Depending on whether the activity of the neuron is relevant to the synaptic change, sub-rules are classified into two different categories: Activity-Dependent Homeostasis (ADH) and Activity-Independent Homeostasis (AIH) term. The general model for explaining the effect of homeostasis is as follows:

$$w(t) = ADH(t) + AIH(t)$$

where the $ADH(t)$ and $AIH(t)$ represent AIH term and ADH term respectively.

Activity-Dependent Homeostasis: Two sub-rules fall into this category: spike dependent [134] and voltage dependent [24] models. Spike dependent model changes the direction and the amount of synaptic weight update depending on the activity of the post-synaptic neuron. The model is defined as:

$$a(t) = (1 - \Delta t/\tau) \cdot a(t - \Delta t) + s_j(t) \cdot \Delta t/\tau$$
$$ADH(t) = \beta \cdot w(t - \Delta t) \cdot (a_{goal} - a(t))$$

where $a(t)$ is a basic trace to reflect the spiking activity. The β is a scaling constant

reflecting the amount of weight changes induced by the sub-rule, and a_{goal} is a constant reflecting the desired value of the $a(t)$ value.

The other model, voltage dependent model, depresses the amount of synaptic weight updates by LT as follows:

$$h(t) = (1 - \Delta t/\tau) \cdot h(t - \Delta t) + \Delta t/\tau \cdot v(t - \Delta t) + s_i(t)$$
$$ADH(t) = \frac{h(t)^2}{scale} \cdot RB(t)$$

where $h(t)$ is a slight variation of the voltage trace, the average of the membrane potential and spiking activity of the post-synaptic neuron over a long timescale. Also, $scale$ is a scaling constant to reflect the amount of weight changes.

Activity-Independent Homeostasis: The two sub-rules in this category, linear decay [16] and exponential decay [44] models, do not take neuronal activity into account. First, linear decay model is defined as:

$$AIH(t) = \begin{cases} w(t - \Delta t) + \alpha(t - t_{old}) \cdot s_{i,j}(t) & \text{if } w(t - \Delta t) > \theta_w \\ w(t - \Delta t) - \beta(t - t_{old}) \cdot s_{i,j}(t) & \text{if } w(t - \Delta t) < \theta_w \end{cases}$$
$$t_{old} = t \cdot s_{i,j} + t_{old} \cdot \overline{s}_{i,j}$$

where α and β is a constant determining the amount of weight change by the sub-rule, and t_{old} refers to the latest spiking time of the pre-synaptic- and post-synaptic neuron. Depending on the current synaptic weight, the synaptic weight can saturate to extreme values. This model can stabilize synaptic weights against spontaneous activity. Second, exponential decay model emulates exponential decays by employing a decay constant γ_0 as follows:

$$AIH(t) = \gamma_0 \cdot w(t - \Delta t)$$

By doing so, the model can be used as a depression mechanism in the absence of other depression mechanisms [44]. Figure 3.6 shows the datapaths for the homeostatic plasticity mechanisms; `adh` and `aih` in the figure denotes the microcode in Table 3.3.

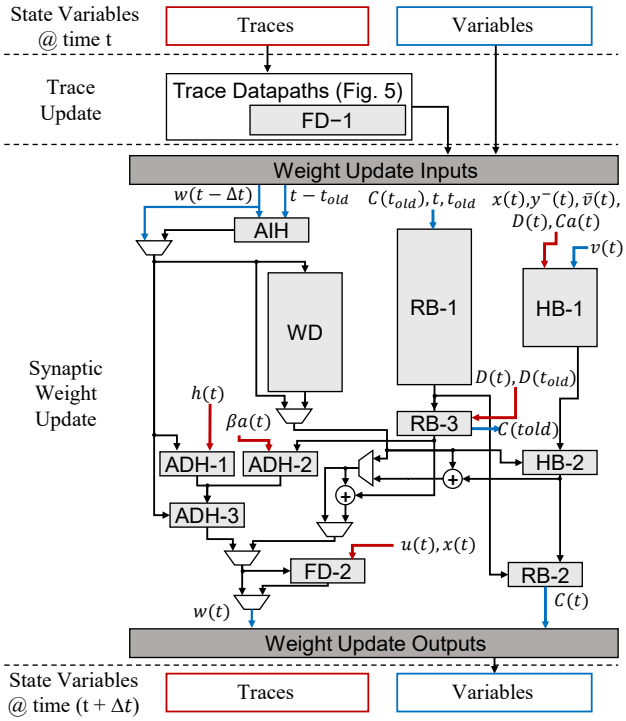


Figure 3.7: Datapath Integration in FlexLearn Architecture

3.2.2 Efficient Integration of the Datapaths

We now present FlexLearn architecture, a flexible and highly efficient on-chip learning accelerator supporting the diverse learning rules. FlexLearn integrates the per-category datapaths to achieve high flexibility. Figure 3.7 shows how FlexLearn integrates the datapaths. Given the current neuronal state (e.g., membrane voltage, neuron trace) and synaptic state (e.g., synaptic weight) variables at time t, the integrated datapath produces the updated state variables (e.g., neuron trace, synatpic weight) for the next timestep $t + \Delta t$ according to the specified learning rules.

For higher performance, FlexLearn splits the datapaths into smaller sub-datapaths. For example, the `ADT` datapath is split into three sub-datapaths `ADT-{1,2,3}`. By doing so, FlexLearn can implement an 8-stage pipeline to improve throughput. Furthermore, employing the sub-datapaths allows data-parallel sub-datapaths to run in

Table 3.3: Microcodes to configure the datapaths

Signal	**Description**	**Values**	
hbp[2:0] / hbd[2:0]	HB category sub-rules for LTP and LTD	0: Nearest Pair 2: Triplet Pair 4: Momentary Voltage 6: Disable	1: All-to-All Pair 3: Suppression Pair 5: Average Voltage
wd[2:0]	WD category sub-rules	0: Additive 2: Sublinear 4. Disable	1: Multiplicative 3: Mixed
rb	RB sub-rule	0: Enable	1: Disable
fd[1:0]	FD category sub-rules	0: Separate 2: Disable	1: Merged
adh[1:0]	ADH category sub-rules	0: Spike Dependent 2: Disable	1: Voltage Dependent
aih[1:0]	AIH category sub-rules	0: Linear Decay 2: Disable	1: Exponential decay

parallel. As an example, when sub-rules `RB` and `HB` are enabled, sub-datapaths `RB-1` and `HB-1` can run in parallel. Lastly, we identify some sub-rules that update weights for the synapses only when any of the connected neurons to the synapses fire spikes (e.g., HB). To improve efficiency, we perform weight update only for the syanpses connected to neurons that fire spikes for such sub-rules.

As the datapaths should be enabled only when the target learning rule demands them, FlexLearn provides 16-bit microcodes to selectively activate the datapaths (Table 3.3). Using the datapaths, brain simulators can program FlexLearn to simulate their target learning rules. The microcodes instruct which of the datapaths should be activated; the input to the multiplexers within FlexLearn is determined by parsing the given microcodes. One 16-bit microcode corresponds to a learning rule for a synapse. By assigning different microcodes to synapses, FlexLearn facilitates highly flexible

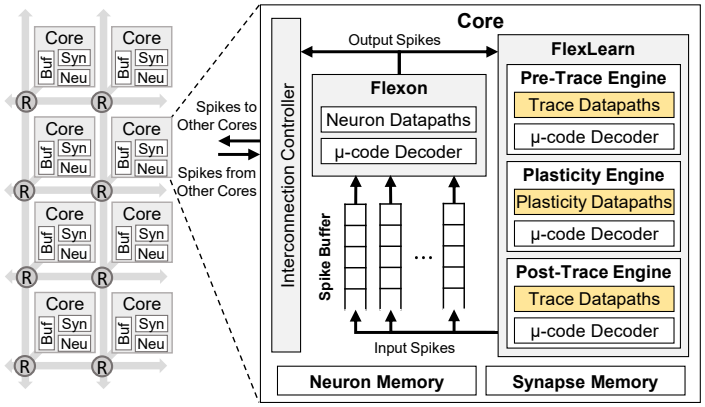


Figure 3.8: Example flexible brain simulation processor with FlexLearn

brain simulations involving diverse learning rules.

3.2.3 End-to-End Brain Simulation Processor

The datapaths of FlexLearn provide highly flexible learning; however, end-to-end brain simulations demand high flexibility in both learning and neurons. To enable the end-to-end simulations, we design and present an example brain simulation processor by integrating FlexLearn archiecture on top of the scalable architecture similar to Loihi [26]. Loihi is the state-of-the-art brain simulation hardware which supports up to 128 neuromorphic cores per chip and a scalable asynchronous Network-on-Chip (NoC) for up to 16,348 chips. Despite the advantages in scalability, Loihi has limited support for both the learning rules and the neurons of the human brain; Loihi supports only 11 out of our target 17 sub-rules, and CUBA-variant neurons. To achieve high flexibility in neurons, our accelerator employs Flexon [72]. Flexon is the state-of-the-art digital neuron supporting the diverse neuron behaviors within the human brain; however, it only supports the neuron computation stage of the brain simulation. By integrating the datapaths with Flexon, our processor achieves highly flexible learning (using FlexLearn) and neurons (using Flexon) at the same time.

Figure 3.8 illustrates the architecture of our processor. The cores of the accelera-

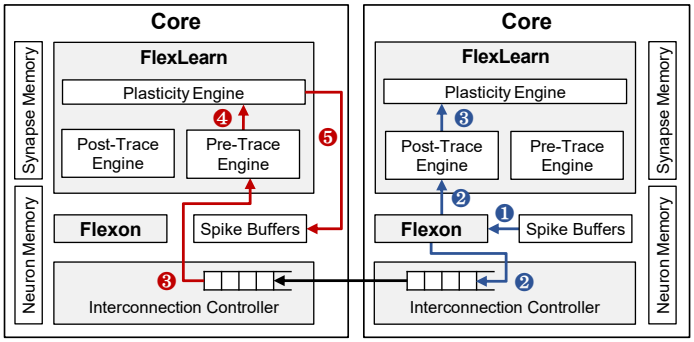


Figure 3.9: Working model of our example processor

tor simulate on-chip learning and neuron computation, and are interconnected using the 2D mesh interconnection network of Loihi. Each core consists of the datapaths of FlexLearn and Flexon, spike buffers storing incoming spikes toward the core's neurons, an interconnection controller, and neuron and synapse memory. The datapaths of FlexLearn and Flexon simulate on-chip learning and neurons, respectively. Each core can process up to 35 neurons and 28,000 synapses using a 0.23-MB per-core SRAM which serves as the neuron and the synapse memory. When a neuron of a source core fires a spike to a neuron of another destination core, the spike goes through the interconnection network to the destination core.

Figure 3.9 illustrates the working model of our processor. The processor simulates a timestep of a brain simulation as follows. First, Flexon retrieves input spikes from the spike buffers (❶), and then updates the target neuron's internal state using the datapaths of Flexon. The information on whether the target neuron fires a spike gets propagated to the post-trace engine, and also to the destination neurons (possibly on other cores) if the neuron fires a spike (❷). After updating the neuron's post-trace, a variable tracking the activities of the post-synaptic neuron, the plasticity engine receives the trace values and performs synaptic weight updates according to the synapse's learning rules (❸). When the target neuron is on another core, a notification of the spike is sent to the destination core through the interconnection network. Then, the destination core's

Table 3.4: RTL synthesis results of Loihi accelerator and FlexLearn

Accelerator	Component	Area [mm^2]	Power [mW]
128-Core Loihi	Neuron	0.4	35.3
	Learning	5.3	542.2
	SRAM	380.7	10360.2
	Total	386.4	10937.7
128-Core FlexLearn	Neuron	2.1	396.0
	Learning	27.7	3224.9
	SRAM	380.7	10360.2
	Total	410.5	13981.1

pre-trace engine receives the notification as the spike is generated by the source/pre-synaptic neuron (❸ on the destination core). The pre-trace engine then updates the pre-trace value of the destination neuron and, applies the learning rules of the synapse, and the spike information is stored to the spike buffers of the destination core to trigger (❹–❺ on the destination core). By doing so, the destination core initiates neuron computation of the destination neuron. In this way, our processor achieves not only highly flexible and efficient brain simulations, but also high scalability using the inter-core communication.

3.3 Evaluation

3.3.1 Experimental Setup

To evaluate FlexLearn, we compare a 128-core simulation processor using FlexLearn against server-class CPU, GPU, and 128-core Loihi-style baseline processor. In this section, for brevity, we use the terms FlexLearn and Loihi to indicate the brain simulation processor using FlexLearn with Flexon and Loihi-style learning and neuron engine, respectively. 12-core Intel Xeon E5-2650 v4 CPU [1] and NVIDIA Titan X (Pascal) GPU [2] serve as the CPU and the GPU. The number of cores for the accelerators is chosen to be equal to that of Loihi [26]. The performance numbers for

Table 3.5: Evaluated biological neural networks

Name	Neurons	Synapses	Compatibility				
			CPU	GPU	TrueNorth	Loihi	FlexLearn
Brunel [19]	5,750 CUB	Static: 1,190,250 Plastic: 2,116,000	✓	✓	✗	✓	✓
Vogels [135]	10,000 COBE	Static: 1,680,000 Plastic: 320,000	✓	✓	✗	✗	✓
Izhikevich [61]	1,000 Izhi.	Static: 360,000 Plastic: 640,000	✓	✓	✗	✗	✓

CPU and GPU were obtained from NEST [45] and GeNN [139] simulators, respectively. To measure the energy efficiency of the CPU and the GPU, we use Intel Performance Counter Monitor [137] and nvprof [3]. For Loihi accelerator and FlexLearn, we write Register-Transfer Level (RTL) code in Verilog, and synthesize the code using 45-nm NanGate FreePDK45 library [123]. The Static Random Access Memory (SRAM) costs are measured using CACTI 6.5 [86]. The synthesis result indicates that both accelerators operates in 250 MHz. Table 3.4 shows the synthesis results. Our FlexLearn accelerator greatly improves the flexibility over Loihi with only 6.3% chip area and 27.8% power overheads.

We use the biological neural networks used by Brunel [19], Vogels et al. [135], and Izhikevich [61] as the benchmarks. Table 3.5 summarizes their characteristics. Along with the biological neural networks, we utilize four representative learning rule combinations (Table 3.6). The combinations are applied to the plastic synapses. Note that the underlying hardware may not support some configurations due to its inflexibility. For example, TrueNorth lacks support for all of the evaluated learning rule combinations, and Vogels and Izhikevich are not supported by Loihi due to its inflexibility in neuron model. On the other hand, FlexLearn is highly flexible and supports all of the configurations.

Table 3.6: Evaluated learning rule configurations

Type	Learning Rules	Compatibility				
		CPU	GPU	TrueNorth	Loihi	FlexLearn
A	1.5. Mementary 2.1. Additive 6.1. Linear Decay	✓	✓	✗	✓	✓
B	1.3. Triplet 2.2. Multiplicative 6.2. Exponential	✓	✓	✗	✓	✓
C	1.4. Suppression 2.2. Multiplicative	✓	✓	✗	✗	✓
D	1.2. All-to-All 2.2. Multiplicative 4.1. Separate	✓	✓	✗	✗	✓

3.3.2 Flexible & Efficient Brain Simulations

In this experiment, we evaluate the brain simulation efficiency of our FlexLearn processor against the server-class CPU and GPU, and Loihi. We employ two performance metrics for the evaluation: simulation speed and energy consumption. For the comparison, we measure the latency and the energy consumption of the hardware when simulating the benchmarks for 10,000 timesteps. 10,000-timestep simulations emulate 1 s of biological time for Brunel whose timestep is 0.1 ms, and 10 s for Vogels and Izhikevich network as their timesteps are 1 ms.

Our results clearly indicate that FlexLearn significantly improves the brain simulation speed and energy efficiency. First, Figure 3.10 shows the speedups of our FlexLearn processor over the CPU and the Flexon-augmented CPU. FlexLearn achieves the harmonic mean speedups of 40.32× over the CPU, and 27.44× over the Flexon-augmented CPU. As Flexon targets only the neuron computation, the Flexon-augmented CPU fails to mitigate the high performance overheads of the learning rules. On the

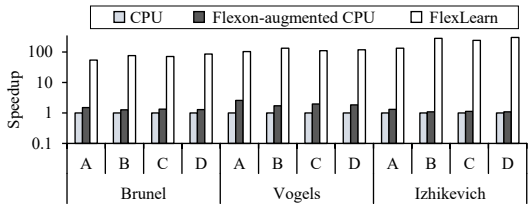


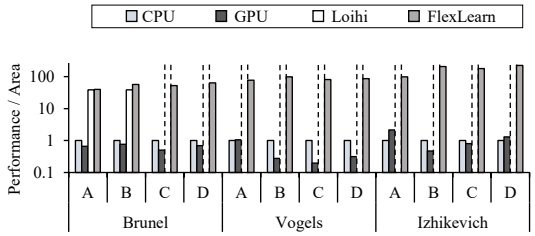
Figure 3.10: Speedup of FlexLearn over the CPU

other hand, our FlexLearn processor greatly improves the performance by providing highly flexible learning rule support.

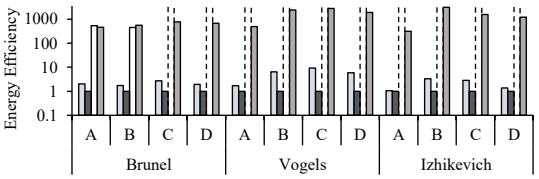
Second, Figure 3.11 shows the per-area performance improvements and the energy efficiency improvements of our FlexLearn processor over the other platforms. FlexLearn achieves harmonic mean performance improvement per area of 30.07× and 37.12× over the CPU and the GPU, respectively. In terms of the energy efficiency, FlexLearn achieves the harmonic mean improvements of 126.87× and 262.16× over the CPU and the GPU, respectively.

Moreover, FlexLearn outperforms Loihi despite improved flexibility by efficiently integrating the datapaths to minimize the hardware costs. Compared to Loihi, FlexLearn improves the performance per area and the energy efficiency by 1.41× and 1.17×, respectively, in harmonic means. Note that the CPU outperforms the GPU as the brain simulations do not have enough parallelism to fully utilize the GPU. Our analysis shows that the small synapse counts per neuron are the primary reason for the underutilization.

Our experimental results also indicate the high flexibility of FlexLearn over Loihi. In Figure 3.11, we observe that Loihi supports only two simulations. On the other hand, FlexLearn supports all of the simulations by implementing the 17 sub-rules; Loihi supports only 11 of the 17 sub-rules. Furthermore, by integrating Flexon with FlexLearn, our FlexLearn processor supports various neuron models as well.



(a) Per-area performance improvements over the CPU



(b) Energy efficiency improvements over the GPU

Figure 3.11: Improvements in the simulation efficiency of FlexLearn over the CPU, GPU, and Loihi. The dotted lines indicate the simulations not supported by Loihi.

3.3.3 Low Hardware Costs

Aimed at minimizing the hardware costs, our FlexLearn datapath exploits the common sub-expressions between the sub-rules of the same category. We further reduce the datapath by integrating the datapath for LTP and LTD. To evaluate the efficiency of the integration, we compare the hardware costs of FlexLearn's integrated datapath to a naïve integration of the datapaths. The naïve integration simply places all of the datapaths and activates one of the datapaths at a time with respect to a given input. Figure 3.12 compares the chip area and the power consumption of each datapaths, the naïve integration, and FlexLearn's integrated datapath. The results show that FlexLearn demands 27.8% smaller chip area and consumes 40.8% less power than the naïve integration. By exploiting the characteristics of the learning rules, FlexLearn effectively reduces the hardware costs.

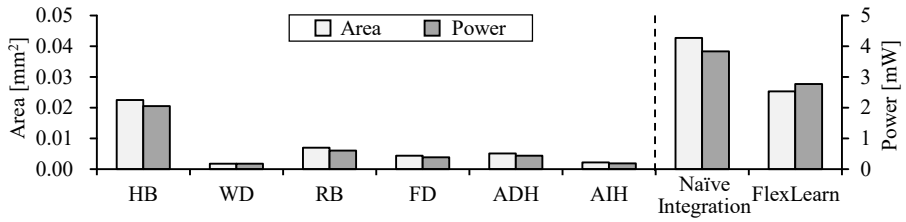


Figure 3.12: Area and power consumption of the per-rule datapaths and FlexLearn

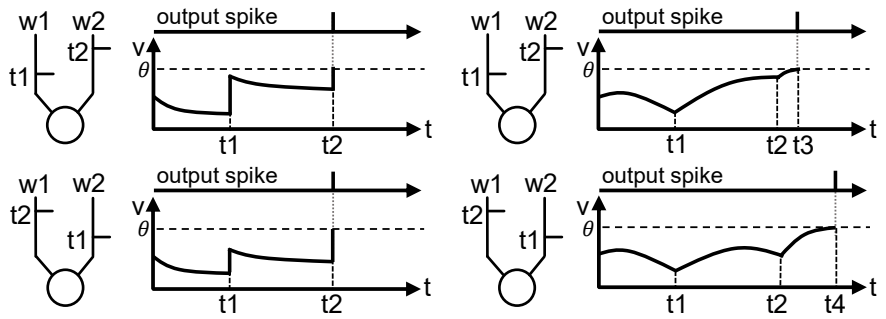
Chapter 4

NeuroEngine: A Hardware-Based Event-Driven Simulation System for Advanced Brain-Inspired Computing

4.1 Motivation

4.1.1 Categories of Neuron Models

The neuron models which mimic the internal dynamics of the neurons can be classified into two categories depending on how they respond to the input spikes: the one-stage and the two-stage neuron [115]. The one-stage neuron changes its membrane voltage



(a) one-stage neurons (b) two-stage neurons

Figure 4.1: Behavior difference of the one-stage and two-stage neurons for different input patterns.

with an infinite slope, and thus, fires a spike only when it receives a spike. On the other hand, the two-stage neuron gradually changes its voltage and takes time to fire a spike after receiving a spike. By reacting to the input spikes with a non-instant slope, the two-stage neurons can process more complex relationships between input spikes. Quantitatively, a previous study [115] identifies that a two-stage neuron processes information with 50% higher resolution than the one-stage neurons. Figure 4.1 illustrates an example of the one-stage neuron and the two-stage neuron receiving two different input spike patterns and shows how the two-stage neuron enables richer temporal dynamics. In the first case, a neuron receives a former spike at t_1 with larger weight w_1 and the latter spike at t_2 with smaller weight w_2. The order of the two spikes is reversed in the second case. The one-stage neuron fires a spike only when it receives a spike, and the timing of the output spike is the same in both cases (t_2 in Figure 4.1a); whereas the two-stage neuron takes time to fire a spike, and the timing of the output spikes differ (t_3, t_4 in Figure 4.1b).

To that end, emerging studies focus on using two-stage neuron models for various tasks. For example, studies by Smith [117–119] emphasize the computational power of the two-stage neuron models and use them on spike-based image classification workloads. Other works apply a spike-based backpropagation on two-stage neuron models to achieve the state-of-the-art accuracy on classification workloads [91, 114]. Recent studies are exploring the potential of simulating the networks with two-stage neuron models on diverse use cases. Ponulak et al. reproduce the brain's navigating function using the two-stage neuron [98]. As another example, two-stage neurons are adopted for feature extraction (i.e., LASSO) [125] or solving satisfaction problem [36].

Among the two-stage neurons, we mainly focus on biexponential model for its

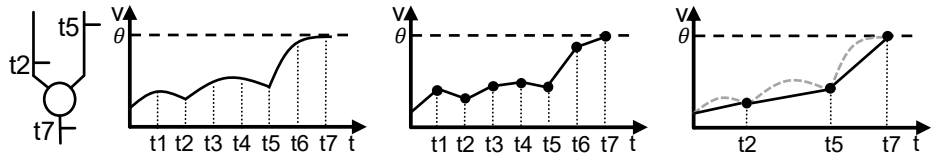
high applicability in recent brain-inspired computing tasks. It is defined as:

$$\frac{dg_{type}}{dt} = -\frac{g_{type}}{\tau_{type}} + \sum_i \delta(t - (t_i + d_{ij})) \cdot w_{ij} \;\; (type = exc, inh)$$
$$\frac{dv}{dt} = -\frac{v}{\tau_v} + g_{exc} - g_{inh} \tag{4.1}$$
$$\text{if } v > \theta \text{, then fire a spike and } v = v_0$$

where v is a membrane voltage which decays with the time constant τ_v and θ indicates the threshold voltage. The $\sum_i \delta(t - (t_i + d_{ij})) \cdot w_{ij}$ indicates the accumulated weight using the firing time of the source neuron (t_i), the delay (d_{ij}) and weight (w_{ij}) between the source and the destination neuron. Also, g_{exc} and g_{inh} refer to the current, which enables the gradual increase in the voltage by low-pass filtering the accumulated weights. The current decays with the time constants τ_{exc} and τ_{inh}. Biexponential model is both computationally powerful and efficient, making it widely used in various studies [44, 115]. Also, a recent neuromorphic benchmark study [25] emphasizes its high applicability to various workloads.

4.1.2 Time-Driven & Event-Driven Simulation

There are two representative approaches to simulate the brain: time-driven and event-driven. The time-driven mechanism performs neuron computation for all neurons every timestep (Figure 4.2b). Algorithm 1 is a representative example of the time-driven simulation. In the time-driven simulation, the simulator iterates over all the neurons for each timestep and updates the states based on the previous states and the accumulated



(a) neuronal dynamics (b) time-driven (c) event-driven

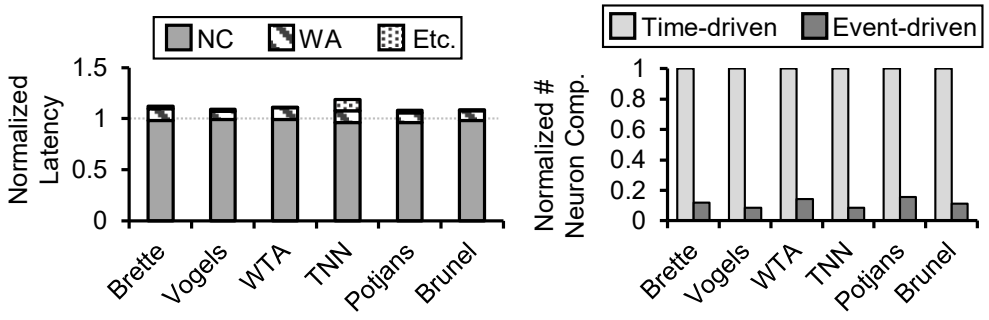
Figure 4.2: Neuron computations in time-driven simulation and event-driven simulation.

Algorithm 2: An event-driven brain simulation

```
1  # event_queue: the event queue (target_neuron, event_time)
2  # n.last: timestep of the last update for neuron n
3
4  While (t_cur < t_max):
5   While ((n,t) = event_queue.top() and t == t_cur):
6    event_queue.pop()  # neuron computation
7    fired = state_update(n.states, n.acc[t], n.last) # Eq.2
8    n.last = t
9    If (fired): spike_buf.append(n)
10   t_pred = predict(n.states)  # spike prediction (Eq.3)
11   event_queue.schedule(n, t_pred)
12  For (src in spike_buf):  # weight accumulation
13   For ((dst,w,d) in src.fanout):
14    dst.acc[t_cur + d] += w
15    event_queue.schedule(dst, t_cur + d)
16  t_cur += Δt, spike_buf.reset()
```

weight (Line 9–12).

On the other hand, the event-driven mechanism calculates the change in the internal states over several timesteps at once (Figure 4.2c). Instead of updating all neurons every timestep, the simulator performs neuron computations only when a neuron receives or fires a spike. The simulator uses the term *events* to indicate such occasions and adopts an event queue to handle the events. Algorithm 2 illustrates an example of the event-driven simulation. For each timestep, the simulator checks the event queue where the events are stored in chronological order. Then, it performs neuron computation for each event using an equation (i.e., Equation 4.2) and updates the last time the neuron was updated (Line 5–8). After that, the event-driven simulator calculates the time when a neuron fires a spike and schedule an event (i.e., *out-event*) at the calculated timestep (Line 10–11). Ideally, the simulator calculates the exact time when the neuron will fire a spike; however, existing works use approximation methods to



(a) Latency breakdown of the time-driven simulation

(b) Number of neuron computations

Figure 4.3: Potential of the event-driven simulation.

conservatively predict the firing time due to the high complexity of the calculation. Note that the one-stage neurons do not schedule an out-event as they fire a spike only when it receives a spike. After processing all the events for the timestep, the simulator performs weight accumulation (Line 12–14). The simulator schedules an event for the destination neuron at the timestep where it receives the spike (`t_cur + d`) to ensure that it performs neuron computation for the spike (Line 15). We refer to such an event as an *in-event*.

The event-driven simulation exhibits higher efficiency than the time-driven simulation by updating the internal states only when an event exists. To examine the potential of the event-driven simulation, we breakdown the simulation latency of the baseline time-driven simulator on six benchmarks (Refer to Section 4.5.1 for the evaluation setup). The result shows that the time-driven simulator suffers from the excessive neuron computation overhead (Figure 4.3a). The `NC` and `WA` refer to the time taken in neuron computation and weight accumulation, respectively, and `Etc` indicates subordinate functions, including routing and synchronization overhead. The latency is normalized to the total latency of each benchmark. Note that the total latency is smaller than the combined value as the computations can be parallelized. The result indicates that `NC` is a critical performance bottleneck taking up to 99.3% of the total latency. Then, we profile the number of neuron computations required in the event-driven sim-

ulator (Figure 4.3b). Our results show that the event-driven simulator can greatly reduce the neuron computation overhead with up to 91.4% fewer computations than the time-driven counterpart.

4.2 Challenges

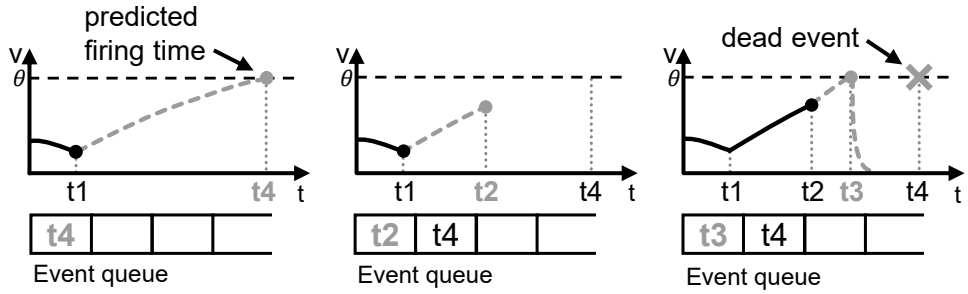
The event-driven simulation is potentially highly efficient over the time-driven simulation by reducing the number of neuron computations; however, we identify three challenges in realizing an efficient event-driven simulation for brain simulations with two-stage neurons and synaptic delays.

4.2.1 Complex Computations

Unlike the time-driven method which updates the states over a fixed interval, the event-driven simulation updates the states over arbitrary time intervals. Thus, it requires a closed-form solution for the target model which is highly complex for two-stage neurons. The closed-form solution for biexponential model is defined as:

$$
\begin{aligned}
g_{type,t} &= g_{type,t'} \cdot e^{-\Delta t/\tau_{type}} + \overbrace{\sum_i w_{ij} \cdot s_i(t)}^{I_{type,t}} \\
\tau_{vg_{type}} &= (\tau_v \cdot \tau_{type})/(\tau_v - \tau_{type}) \\
v_t &= v_{t'} \cdot e^{-\Delta t/\tau_v} + \\
&\sum_{type} \tau_{vg_{type}} \cdot g_{type,t'} \cdot (e^{-\Delta t/\tau_v} - e^{-\Delta t/\tau_{type}})
\end{aligned}
\tag{4.2}
$$

where the $g_{type,t}$ (current) and v_t (voltage) illustrate the state variables of the neuron j at time t (Equation 4.1). $s_i(t)$ denotes the occurence of a spike of a neuron i at time t. Δt is the time interval between the current time t and the last update time t'. As the equation illustrates, the datapath needs to support complex computations with varying intervals, Δt. As the simulations are sparse in events, the Δt leads to high bit precision requirements for the computation.



(a) neuron computation (@t1) (b) receive a spike before the predicted time (@t1) (c) update the predicted firing time (@t2)

Figure 4.4: Occurrence of a dead event.

Moreover, the event-driven simulations require complex operations to predict when a neuron fires a spike. Unlike the one-stage neuron which fires right after receiving a spike, the two-stage neuron takes time to fire a spike after receiving a spike (Figure 4.1). To calculate the firing time, some approaches rely on the LUTs where the precalculated results are stored [43, 90, 92, 106]; however, the LUT demands too large memory capacities to be implemented in hardware. To mitigate the complexity of solving the equation, some other works propose a linear approximation model to conservatively predict the firing time [30, 77]. The prediction for the biexponential model is:

$$
\begin{aligned}
A_v &= v_t + \sum_{type} g_{type,t} \cdot \tau_{vg_{type}}, \quad A_{type} = -g_{type,t} \cdot \tau_{vg_{type}} \\
slope &= \begin{cases} -A_v/\tau_v - A_{exc}/\tau_{exc} - A_{inh}/\tau_{inh}, & \text{if } A_v > 0 \\ -A_v/\tau_v - A_{exc}/\tau_{exc} - A_{inh}/\tau_v, & \text{if } A_v < 0 \end{cases} \\
t_{pred} &= (\theta - v_t)/slope
\end{aligned}
\tag{4.3}
$$

where $slope$ and t_{pred} refer to an approximate slope and the resulting predicted time, respectively. The prediction model requires power- and area-consuming divide operations and incur high computational overhead.

4.2.2 Complex Scheduling

We identify two challenges in designing an efficient event scheduler in the brain simulations. First, both the in-events and the out-events are scheduled at an arbitrary timestep as shown in Algorithm 2 (Line 12, 16). Therefore, the scheduler should sort the events at a low cost to process the events in sequential order. Second, the in-events and the out-events interact with each other and may become redundant. In this case, the scheduler should detect the redundant events and remove them from the event queue. For example, an out-event becomes redundant if a neuron receives a spike before the predicted firing time. The predicted firing time changes after receiving the spike, and the scheduled out-event becomes useless. Figure 4.4 illustrates how the out-event becomes a useless event which we refer to as a *dead event*. After the neuron computation at t_1, the neuron predicts the next firing time as t_4 and schedules an out-event (Figure 4.4a). Right after scheduling the out-event, a new in-event which indicates that a neuron receives a spike at t_2 is scheduled (Figure 4.4b). Due to the event, the simulator performs neuron computation at t_2, and predicts a new firing event, indicating that the neuron is predicted to fire a spike at t_3 (Figure 4.4c). As the firing time is updated to t_3, the previously scheduled out-event at t_4 becomes useless, turning into a dead event.

4.2.3 Resource Contention

In the event-driven simulation, the simulator suffers from the resource contention in event scheduling when processing neuron computation and weight accumulation simultaneously. Using the event-driven mechanism to update the states, the number of neuron computations is highly reduced, and the weight accumulation also becomes a major source of performance bottleneck; therefore, it is highly demanding to parallelize neuron computation and weight accumulation. However, as described in Algorithm 2, the event-driven simulator schedules events for both neuron computation (Line 12) and weight accumulation (Line 16). To that end, the concurrent processing of the two operations is a challenging task with the resource contention in the event

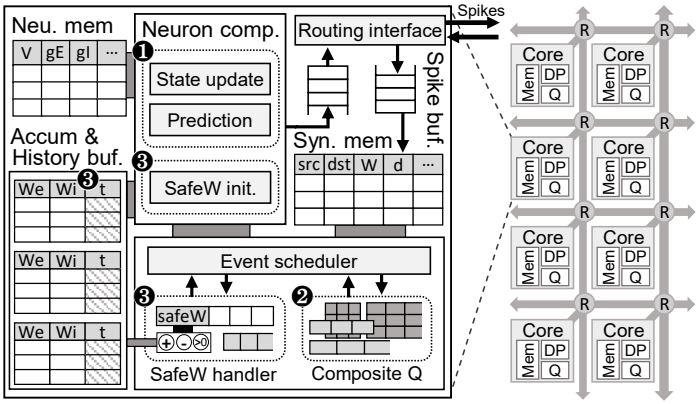


Figure 4.5: Overview of the NeuroEngine architecture.

scheduler.

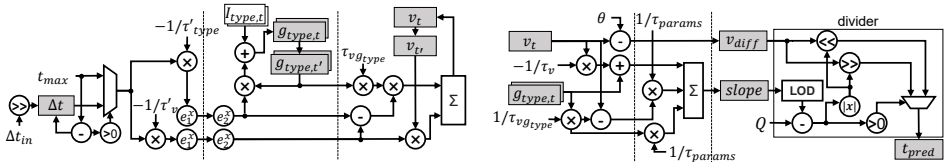
4.3 NeuroEngine Architecture

In this section, we present NeuroEngine, a fast and efficient hardware-based event-driven simulation system. We apply simple datapaths, composite queue scheduler, and lazy update to minimize the computation and event scheduling overhead. Figure 4.5 illustrates the overall architecture of NeuroEngine. First, our simple datapaths with reduced bit precision and approximate divider offer fast and efficient neuron computations (❶). Second, our composite queue structure for scheduling the events prevents redundant event scheduling (❷). Lastly, we propose a novel method called lazy update to bypass event scheduling and parallelize operations (❸).

4.3.1 Optimized Datapath

Pipelined Datapath w/ Reduced Bit Precision

We design an optimal datapath for neuron computation by reducing the integer-part-bit precision for representing the Δt. The key idea is to shift and split the time interval into multiple sub-intervals and pipeline the computations. First, we identify that the



(a) State update (b) Prediction

Figure 4.6: Optimized datapath for neuron computation.

Δt can be shifted as it is multiplied by the inverse of the time constants, $1/\tau$ (i.e., $1/\tau_v$, $1/\tau_{type}$) (Equation 4.2). By performing the right shift on the Δt and the left shift on the $1/\tau$, we can reduce the precision for representing the interval ($\Delta t \cdot 1/\tau = (2^{-n} \cdot \Delta t) \cdot (2^n \cdot 1/\tau)$). We further reduce the bits by splitting the Δt into smaller sub-intervals so that the Δt does not exceed the supporting bit precision. Although splitting increases the number of computations, the overhead is minimal (Section 4.5.3). After that, we fully pipeline the datapaths into three stages with forwarding paths to prevent stalls when consecutively updating the sub-intervals. Figure 4.6a illustrates our state update datapaths. The $g_{type,t}$ and v_t refers to the internal states in Equation 4.2. Also, the datapaths for the exponential function is divided into two parts (i.e., e_1^x and e_2^x) for efficient pipeline. The $1/\tau^{'}$s are the inverse of the time constants after performing the left shift. The datapaths shift and split the time interval into sub-intervals, t_{max} for each event. Then, the computations for each sub-interval pass through the pipeline stages. If the two consecutive computations target the same neuron, the updated states at each stage are forwarded.

Efficient Prediction w/ an Approximate Divider

We design an approximate divider which shifts the dividend by $\lceil \log_2 divisor \rceil$ instead of performing a complex division. Such an abstraction achieves high efficiency by eliminating the iterations required in the conventional fixed-point divider. The approximate divider avoids the loss of functionality as the shift-based division produces pessimistic results. The result may lead to additional false-positive predictions; how-

ever, the resulting overhead is minimal (Section 4.5.3). Then, we pipeline the datapaths to reduce the critical path latency. Figure 4.6b illustrates our prediction datapaths including the approximate divider. The Q denotes the fractional-part-bit precision, and τ_{params} is a constant determined by the sign of A_v. The LOD module returns the index of the first '1' in the input value. Our approximate divider allows the fixed-point division to operate in a single clock cycle, improving latency as well as the power and area consumption of the prediction.

4.3.2 Composite Queue

We design a composite queue structure composed of heterogeneous data structures to detect and remove redundant events. First, we design an optimal queue structure for the in-event and the out-event, respectively, after analyzing their unique features. Then, we identify three conditions where different events interact and become redundant. Next, we propose two auxiliary data structures to detect the interaction and remove an event if it becomes redundant. By combining the optimal queue with auxiliary data structures, we propose our composite queue-based event scheduler.

From our analysis, we identify the key features of the in-event and the out-event. We identify two different features of the in-event which indicates the arrival of the input spikes. First, multiple in-events for a single neuron coexist if a neuron receives spikes with different synaptic delays. Second, the range of the synaptic delay is fixed, and the in-events reside only within a fixed time window. In contrast, only one out-event is scheduled for a neuron, and an out-event can occur in the distant future without any time constraint. Also, it turns into a dead event and should be removed in some cases (Figure 4.4). Using the identified features, we design an optimal queue structure to schedule each event:

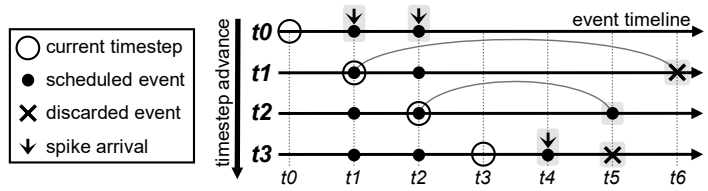
- **In-event queue:** In-event queue consists of multiple FIFO queues to efficiently handle multiple coexisting events in a time window. Each FIFO queue handles in-events for each timestep within the maximum synaptic delay, d_M; therefore, it keeps only

d_M queues. It uses the queues in a circular manner and each queue holds in-events for $t_0, t_0 + d_M, t_0 + 2d_M ...$ as timestep goes on ($0 \leq t_0 < d_M$). Then, NeuroEngine sequentially iterates over the FIFO queues, naturally processing the events in the time order.

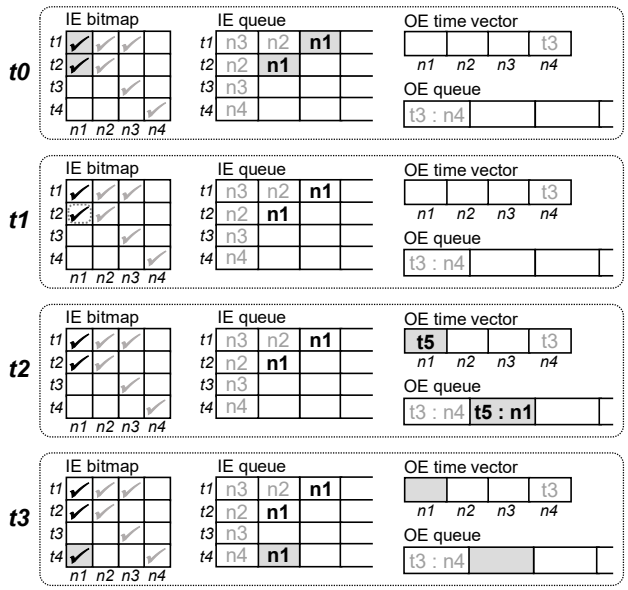
- **Out-event queue:** We adopt a Structured Heap Queue (SHQ) [21] as the out-event queue for efficient deletion of events and scheduling of events over a long time range. The SHQ is a modified version of the pipelined heap queue (PHQ) [60] which stores only a single event for each neuron and provides an arbitrary deletion of a neuron. Such characteristics highly fit well with the identified features of the out-events. Note that popping the event and deletion of an arbitrary event induces a stall cycle.

There are three conditions where an event becomes redundant as a result of the interaction between the in-events and the out-events. First, even if a neuron receives two spikes at the same timestep, a single event-driven computation can process the aggregated information of the two spikes. Thus, the in-event for the second spike becomes a duplicate event and do not need to be scheduled. Second, an out-event becomes useless if an in-event is scheduled before the out-event (Figure 4.4). In this case, the scheduler should remove the out-event. Lastly, an out-event does not need to be scheduled if an in-event is already scheduled before the predicted time of the out-event. We propose two auxiliary data structures and mechanisms to handle the interactions efficiently:

- **In-event bitmap:** The in-event queue is equipped with a bitmap that tracks the existence of the scheduled in-events. The entry indicates whether an in-event exists in each FIFO queue. The in-event bitmap provides an overview of the existence in the in-event and enables NeuroEngine to identify redundant events and prevent scheduling them. Before scheduling a new in-event, the scheduler uses the bitmap to check if an in-event is already scheduled at the timestep. Also, before scheduling a new out-event, the scheduler checks if an in-event is already scheduled at an earlier timestep than the predicted time.



(a) Progress of event scheduling for neuron $n1$.



(b) Changes in contents of the composite queue over time.

Figure 4.7: Redundant event handling in the composite queue.

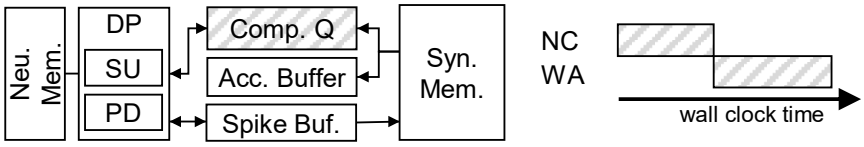
- **Out-event time vector:** The out-event queue holds an additional data structure called an out-event time vector that stores the scheduled time of the out-events. Using the out-event time vector, the scheduler can detect the dead event without accessing the out-event queue. Before NeuroEngine schedules a new in-event, the scheduler uses the out-event time vector to examine if an out-event is scheduled at an earlier timestep. Then, the scheduler removes the out-event from the out-event queue.

Figure 4.7 illustrates an example of how NeuroEngine handles redundant events using the composite queue. In this example, there are four neurons ($n1 \sim n4$) and we

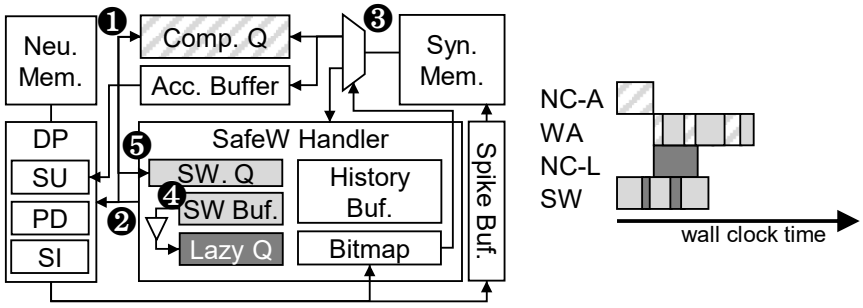
focus only on how the composite queue handles the events for $n1$. Figure 4.7a shows the progress of the event scheduling for $n1$ from timestep $t0$ to $t3$, while Figure 4.7b illustrates the contents of the composite queue at each timestep. At $t0$ (Figure 4.7–(t0)), $n1$ receives two input spikes arriving at $t1$ and $t2$. NeuroEngine schedules the two new in-events by updating the in-event bitmap and inserting the in-events into the in-event queue (marked with the black text in the figure). Then, at $t1$ (Figure 4.7–(t1)), NeuroEngine performs neuron computation for the scheduled in-event for $n1$ and predicts the firing time ($t6$). However, NeuroEngine does not schedule the new out-event at $t6$ because the in-event bitmap indicates that it is a dead event due to the in-event scheduled at $t2$. Next, at $t2$ (Figure 4.7–(t2)), $n1$ performs neuron computation again and predicts the next firing time ($t5$). In this case, there is no in-event scheduled at an earlier timestep than the predicted time, thus NeuroEngine schedules the new out-event at $t5$ by updating the out-event time vector and out-event queue. Lastly, at $t3$ (Figure 4.7–(t3)), an input spike arriving at $t4$ is scheduled as an in-event. At the same time, NeuroEngine detects that the out-event at $t5$ becomes a dead-event by checking the out-event time vector. Therefore, NeuroEngine deletes the corresponding entries in the out-event queue and out-event time vector.

4.3.3 Lazy Update

The event-driven simulator suffers from the low parallelization opportunity induced by the resource contention when scheduling the events. Figure 4.8a illustrates the resource contention in the baseline architecture with the composite queue. Two types of operations exist in the baseline: neuron computation (NC) and weight accumulation (WA). First, the simulator sequentially pops the events from the composite queue. The simulator performs NC to update the states (SU) and predict an out-event (PD). Then, the simulator accesses the out-event queue in the composite queue to store the out-event. If the neuron fires a spike, the simulator stores it in the spike buffer. After NC, the simulator performs WA by iterating over the spikes in the spike buffer,



(a) Baseline composite queue-based architecture



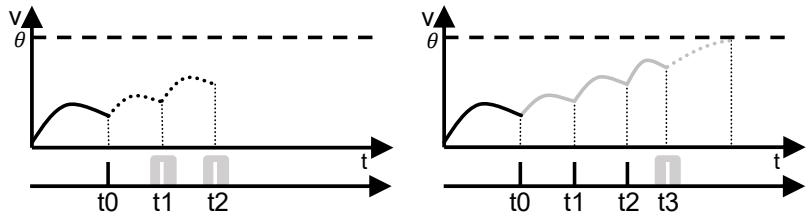
(b) NeuroEngine architecture

Figure 4.8: The working example of the event-driven simulator with and without the lazy update.

searching for the destination neurons using the synapse memory (Syn. Mem.), and accumulating the weights to the buffer (Acc. Buffer). At the same time, the simulator accesses the out-event queue to delete a dead event if necessary and schedule an in-event. When concurrently processing NC and WA, the out-event queue causes a resource contention. As the out-event queue is based on a priority queue, the event schedulings are serialized, resulting in low parallelism.

To maximize parallelism, we propose a novel method called *lazy update*. The idea is to detect the neurons that would not fire a spike after neuron computation and bypass event scheduling for the computations. Then, NeuroEngine defers the computations until the neuron is expected to fire a spike. When flushing the deferred computations, NeuroEngine parallelizes them with weight accumulation as they do not incur event scheduling; thereby, reducing the overall latency.

The key observation is that computations of a neuron can be deferred as long as the neuron does not fire a spike. We refer to such neurons as lazy neurons and use the



(a) Lazy neuron (b) Active neuron

Figure 4.9: Example of the lazy update with a lazy neuron turning into an active neuron.

term active neurons to indicate the neurons that are likely to fire a spike. Figure 4.9 illustrates an example of a lazy neuron turning into an active neuron and how the computations are deferred in NeuroEngine. In the example, the neuron is not expected to fire a spike even after receiving two events at t_1 and t_2 (Figure 4.9a). Thus, NeuroEngine identifies the neuron as a lazy neuron and defers the computations. On the other hand, the neuron is expected to fire a spike after an incoming spike at t_3 (Figure 4.9b); therefore, the neuron turns into an active neuron, and NeuroEngine no longer defers the computations. Then, NeuroEngine flushes the deferred computations to update the internal states before processing the in-event at t_3. We refer to the whole process as lazy update.

We devise a new concept called *safe weight* to determine if a neuron is a lazy neuron with minimal computational overhead. The safe weight indicates the total amount of weight a neuron withstands before turning into an active neuron. NeuroEngine initializes the safe weight of a neuron when performing neuron computation. If the resulting safe weight is positive, the neuron turns into a lazy neuron, and NeuroEngine starts to defer the computations. When a lazy neuron receives a spike, NeuroEngine updates the safe weight of the neuron instead of scheduling neuron computation. If the updated safe weight becomes negative, the neuron becomes active, and NeuroEngine schedules the neuron.

We define simple functions to initialize and update the safe weight and design a

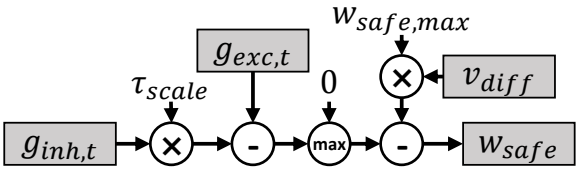


Figure 4.10: Datapath for safe weight initialization.

datapath (Figure 4.10). First, we calculate the maximum voltage a neuron reaches after receiving a spike of weight w. Using the value, we derive how much weight a neuron should receive to fire a spike at the resting voltage and define it as a reference safe weight ($w_{safe,max}$):

$$w_{safe,max} = \theta/(\tau_{vg_{exc}} \cdot (\frac{\tau_v}{\tau_{exc}}^{-\frac{\tau_{vg_{exc}}}{\tau_v}} - \frac{\tau_v}{\tau_{exc}}^{-\frac{\tau_{vg_{exc}}}{\tau_{exc}}})) \tag{4.4}$$

Then, we scale the reference safe weight to take the initial voltage v_t into account. If the current (i.e., g_{exc} and g_{inh}) is not zero, we calculate the effective current (g_{eff}) by scaling g_{inh} and subtract it with g_{exc}. We subtract the effective current from the scaled safe weight to initialize the value. The algorithm is defined as:

$$\begin{aligned} \tau_{scale} &= min(\tau_{inh}/\tau_{exc}, 1) \\ g_{eff} &= max(g_{exc,t} - g_{inh,t} \cdot \tau_{scale}, 0) \\ v_{diff} &= \theta - max(v_t, 0) \\ w_{safe} &= w_{safe,max} \cdot v_{diff} - max(g_t, 0) \end{aligned} \tag{4.5}$$

where τ_{scale} indicates the scaling factor for the inhibitory current and w_{safe} indicates the safe weight. To update the safe weight, we simply subtract the weight of the spike from the safe weight. If the spike is from an inhibitory neuron, we make a scaling factor using the decay constant of the currents and scale the weight before updating the safe weight:

$$\begin{aligned} w_{safe} &= w_{safe} - w_{spike,type} \text{ if } type = exc \\ w_{safe} &= w_{safe} + w_{spike,type} \cdot \tau_{scale} \text{ if } type = inh \end{aligned} \tag{4.6}$$

where $w_{spike,type}$ indicates the weight of the incoming spike. Note that we make a

conservative calculation of the safe weight value not to affect the functionality of the simulation.

To implement lazy update, we define four different operations and add additional datapaths, memory modules, and controllers called a safe weight handler (`SafeW Handler`) (Figure 4.9b). The four operations include: neuron computation for active neurons (NC-A, ❶) and lazy neurons (NC-L, ❷), weight accumulation (WA, ❸), and safe weight update (SW, ❹) for lazy neurons. NC-A in NeuroEngine is similar to the baseline (Figure 4.9a), but with additional computation to initialize the safe weight (SI). If the initialized safe weight is positive, the neuron becomes a lazy neuron and skips the out-event scheduling. Then, NeuroEngine flips the bitmap (`Bitmap`) in the scheduler to indicate that the neuron is a lazy neuron (❶). NC-L (i.e., flushing the deferred computations), on the other hand, does not require prediction and safe weight initialization. Instead, NeuroEngine updates the states for the neurons in the lazy queue (`Lazy Q`) which stores the neurons that turn into an active neuron as a result of the SW (❷).

In WA, NeuroEngine checks the bitmap of the destination neurons. If the bitmap indicates that the neuron is an active neuron, NeuroEngine schedules the event using the composite queue. On the other hand, NeuroEngine bypasses the composite queue and accumulates the weight to the history buffer (`History Buffer`) and schedule a safe weight update event to the safe weight update queue (`SW Q`) if the destination neuron is a lazy neuron. The history buffer is essentially an extended version of the accumulation buffer with a time index to store the accumulated weight with time. Also, the safe weight update queue handles the safe weight update events using the same structure as the in-event queue (❸). Lastly, SW updates the safe weight of the lazy neurons in the safe weight update queue. If the safe weight overflows and becomes negative, the neuron turns into an active neuron, and the bitmap is flipped. Then, the handler pushes the neuron to the Lazy Q and flush the lazy queue in NC-L (❹).

Then, we organize the operations in a way that does not incur resource contention

while maximizing parallelism. The example Gantt chart (Figure 4.8) illustrates how we organize the operations. First, NeuroEngine concurrently performs NC-A and SW as the two operations do not incur resource contention. The other two operations(i.e., WA and NC-L) are blocked as NC-A occupies the state update datapaths and the composite queue. When NC-A finishes, NeuroEngine starts WA and NC-L operations by iterating over the spike buffer and the Lazy Q in parallel. After NeuroEngine completes WA, NC-L, and SW, NeuroEngine moves on to the next timestep. By maximizing the parallelism, NeuroEngine reduces the overall latency compared to the baseline.

There are two corner cases in the lazy update. If a neuron turns into a lazy neuron after NC-A, the remaining events for the neuron in the composite queue should update the safe weight instead of triggering the neuron computation. NeuroEngine detects such events by checking the bitmap after it pops an event from the composite queue and pushes the event to the safe weight queue (❺). Also, if a neuron turns into an active neuron after SW, the remaining events in the safe weight queue should perform neuron computation instead of updating the safe weight. Thus, NeuroEngine checks the bitmap before updating the safe weight and perform neuron computation if the target neuron is an active neuron. Such cases incur stall cycles in the simulation; however, the overhead is minimal.

4.4 Implementation

4.4.1 NeuroEngine Hardware

To build an end-to-end simulation system, we implement NeuroEngine architecture on top of the scalable Network-on-Chip (NoC) similar to the existing brain simulation systems [5, 26]. The NoC supports up to 32 cores using the 2D mesh interconnection network. Each NeuroEngine core consists of neuron and synapse memory, 32-bit fixed-point datapaths, the composite queue to handle the events, and accumulation buffers to store the accumulated weights and history. A single core can process up to

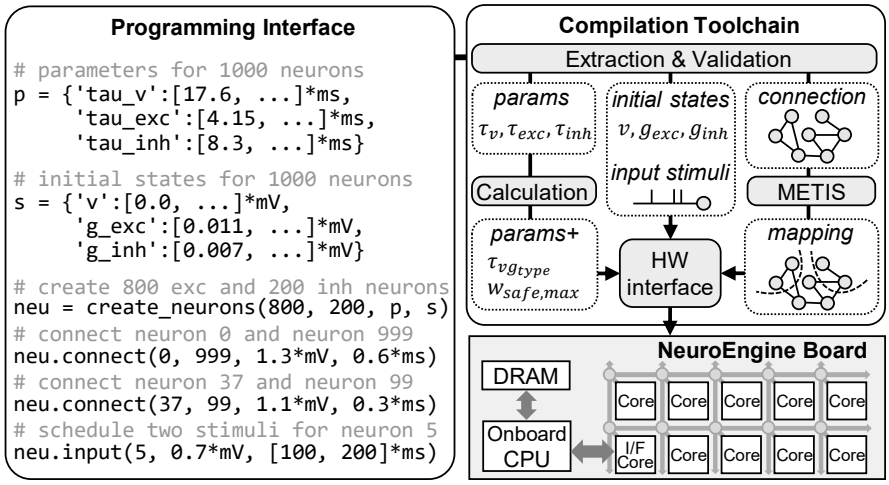


Figure 4.11: Programming interface and compilation toolchain.

512 neurons and 262,144 connections using a 0.60-MB per-core SRAM, which serves as neuron and synapse memory and an additional 0.03-MB per-core SRAM for event queues and history buffers. For the weight accumulation unit, we adopt a memory architecture which supports up to eight synaptic delays as in the existing architecture [26]. NeuroEngine communicates with the host via an onboard CPU and interface core. The onboard CPU supervises the operation of the NeuroEngine chip, and the interface core (I/F Core) in the chip acts as an interface between the onboard CPU and NeuroEngine cores. Also, multiple NeuroEngine chips can be connected together for larger-scale simulations.

4.4.2 Software Stack

Programming interface

We implement a programming interface and compilation toolchain to deploy and evaluate brain simulations on NeuroEngine (Figure 4.11). The programming interface is designed to describe arbitrary network topologies. Using the provided API, users can create neurons, connect them with synapses, and set the network parameters (e.g.,

synaptic weights, delays, θ, τ_v, and τ_g) and initialize the states. Following are the essential APIs our interface provides:

```
create_neurons(exc_neu, inh_neu, params, init_states)
connect(src_neu_id, dst_neu_id, weight, delay)
input(neu_id, weight, timing)
```

`create_neurons` generates a group of biexponential neurons and initializes them with `params` and `init_states`. The number of the excitatory and inhibitory neurons are specified by `exc_neu` and `inh_neu`. `connect` creates a synapse between the two neurons with the given `weight` and `delay`. Lastly, using `input`, the user can inject a series of input stimuli into a neuron at specific timesteps. We also provide additional APIs to mitigate the programming burden in network construction such as `random_connect` which connects a group of neurons with a given probability. An example code for the network creation is shown in Figure 4.11.

Compilation toolchain

Using the program, our compiler checks if the network parameters violate the hardware constraints (i.e., maximum synaptic delay and the size of the network). Then, the compiler calculates additional simulation parameters to enable event-driven computations and our optimization schemes (i.e., $\tau_{vg_{type}}$ and $w_{safe,max}$) using the user-provided parameters and Equation 4.2 and 4.4. Next, the compiler extracts the connectivity of the network in the form of an adjacency matrix. Then we partition the matrix to distribute the neurons to the physical cores in a way that minimizes inter-core connections using an open-source partitioning program called Metis [69]. After that, the toolchain sends all parameters and mapping information to NeuroEngine Board through the PCIe interface. Lastly, the onboard CPU sends the data to NeuroEngine's interface-core via an AXI interface, which then transfers the data to the dedicated target cores. We verify

Table 4.1: Chip area of Time and NeuroEngine.

Accelerator	Component	Area [mm^2]
Time	Neuron Memory & Datapath	4.14
	Synapse Mem.	151.20
	Accumulation Buf.	51.56
	Router + Link	15.31
	Total	222.21
NeuroEngine	Neuron Memory & Datapath	9.47
	Synapse Mem.	151.20
	Accumulation & History Buf.	68.72
	Router + Link	16.14
	Event Queue	7.38
	Total	252.91

Table 4.2: Evaluated brain simulation benchmarks.

Name	Neurons	Synapses
Brette [18]	10,000	2,000,000
Brunel [19]	5,000	2,500,000
Vogels [136]	4,000	320,000
Potjans [99]	8,000	3,000,000
WTA [32]	3,984	3,814,400
TNN [117]	10,656	86,400

the operation of our full system by deploying NeuroEngine on the Xilinx VCU1525 FPGA board and running brain simulations with the toolchain.

4.5 Evaluation

4.5.1 Experimental Setup

To evaluate NeuroEngine, we compare our 32-core NeuroEngine system against the hardware-based time-driven simulator. For the baseline design, we carefully implement a simulator based on existing time-driven architecture, Loihi [26]. We use the

term Time to indicate the Loihi-like baseline with the time-driven execution mechanism for brevity. To measure the energy efficiency and area of NeuroEngine and Time, we write Register-Transfer-Level (RTL) code in Verilog and synthesize it using Samsung 65-nm technology. The SRAM and NoC costs are measured using CACTI 7 [9] and Orion 2.0 [68], respectively. The synthesis results in each simulator operate at 500 MHz. Table 4.1 shows the area consumption after the synthesis. NeuroEngine greatly improves the performance with only 13.8% area overheads.

We collect six different benchmarks from the previous studies using two-stage neuron models (Table 4.2). The benchmarks are of various scales with different numbers of neurons and synapses. The three benchmarks, Brette, Vogels, and Brunel, are variations of a random balanced network that is widely used to exploit mechanisms of the brain. Potjans is a biologically realistic network modeled using anatomical data. Each benchmark is configured to use a timestep of 0.1 ms and to run for 10,000 timesteps. We also use WTA [32] and TNN [117] which perform image classification or clustering workloads. WTA is the most basic type of network to perform classifications using the biological neural network [33]. When porting WTA, we replace the neuron model with biexponential model and achieve 91.87% accuracy on the MNIST dataset which is slightly lower than the reported accuracy (91.9%). TNN adopts a new computational paradigm using two-stage neurons proposed in recent publications [117, 119]. When porting the network to simulators, we reduce the time precision of TNN to 3-bit following the recent findings [88, 119]. We also replace the neuron model with the biexponential model.

4.5.2 Efficiency of the Hardware-based Solutions

First, we evaluate the efficiencies of the hardware-based solution over the software-based counterparts. For evaluation, we profile the latencies of the CPU- (Intel Xeon E5-2650 v4 CPU) and GPU-based (NVIDIA Titan X (Pascal)) solutions and compare them against Time on three benchmarks. For evaluation, we use NEST to sim-

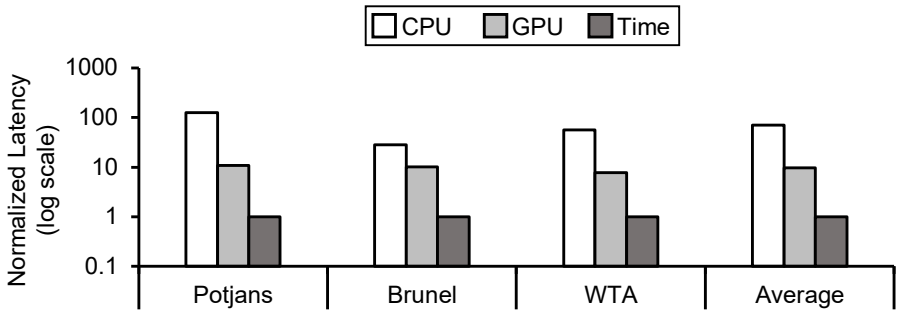


Figure 4.12: The simulation latency profiling of CPU, GPU, and `Time`.

Table 4.3: Area and power consumption of the datapath.

	Time	**Event-Naïve**	**Event-DP**
Power [mW]	12.02	83.37	60.80
Area [mm^2]	0.02	0.17	0.10

ulate the brain on the CPUs and GeNN to simulate on the GPUs. The profile result indicates the efficiencies of HW-based solutions over the SW-based solutions (Figure 4.12). `CPU`, `GPU`, and `Time` in the Figure indicates the CPU, the GPU, and the time-driven baseline, respectively. `CPU` suffers from an average of 70.27× higher latency overhead compared to `Time`. `GPU` mitigates the overhead to some extent using a high-throughput processor, but the latency overhead is still 9.66× higher. To that end, the simulations should rely on ASIC-based platforms.

4.5.3 Low Event-Driven Computation Costs

For the remaining evaluation sections (Section 4.5.3– 4.5.6), we use three variants of event-driven baselines in addition to the time-driven simulator (`Time`): a baseline event-driven simulator (`Event-Naïve`), an event-driven simulator with optimized datapath (`Event-DP`), and an event-driven simulator equipped with both optimized datapath and the composite queue (`Event-CQ`). `Event-Naïve` and `Event-DP` use the PHQ [60] as its event scheduler.

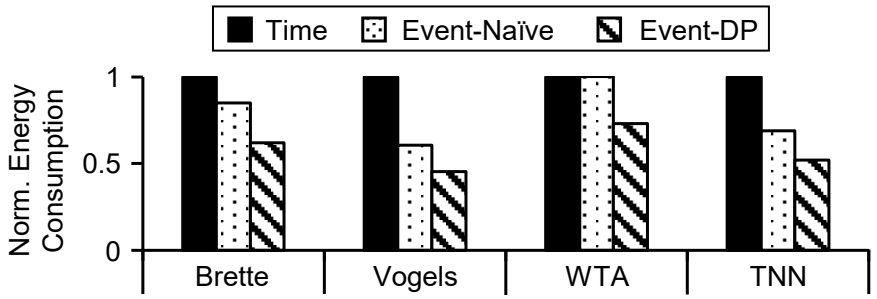


Figure 4.13: Total energy consumption of the datapath.

To evaluate the efficiency of the datapath, we compare the computation costs of our low-cost datapath against two baselines: `Time` and `Event-Naïve`. To identify the bit precision of the datapaths of `Event-Naïve`, we profile the benchmarks. The profile results indicate that one of our benchmarks, `WTA`, requires up to 12 bits to fully represent the time intervals and results in 40-bit fixed-point datapaths, while the optimized datapath requires only 32 bits.

Table 4.3 shows the area and power consumption of each datapath and Figure 4.13 illustrates the total energy consumption of the datapath normalized to `Time` on four benchmarks. The result indicates that the event-driven execution mechanism complicates the datapath to some degree; however, it executes the datapath far less than the time-driven counterpart to achieve higher energy efficiency. `Event-Naïve` suffers from 6.94× larger datapath power consumption and 8.47× larger area consumption compared to `Time`. `Event-DP` achieves 1.37× lower datapath energy consumption compared to `Event-Naïve`. Although `Event-DP` suffers from high datapath power consumption, it reduces the energy consumed by 42% on average with a smaller number of computations. Compared to `Event-Naïve`, `Event-DP` increases the energy efficiency of datapath by 1.35×. The gap between the energy efficiency (1.35×) and power consumption (1.37×) of datapath is due to the slight increase in the number of computations from sub-intervals and the approximate divider (Section 4.3.1).

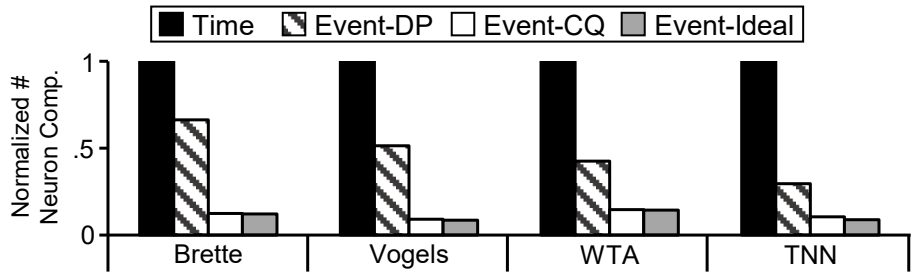


Figure 4.14: Reduced number of neuron computations.

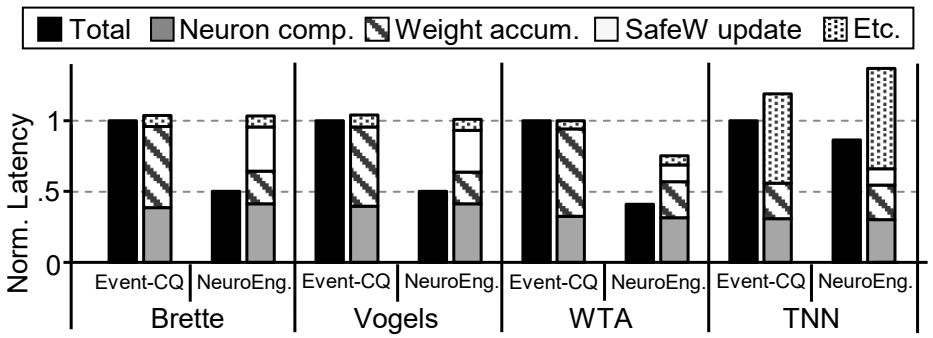


Figure 4.15: Reduced latency with increased parallelism.

4.5.4 Eliminated Redundant Event Scheduling

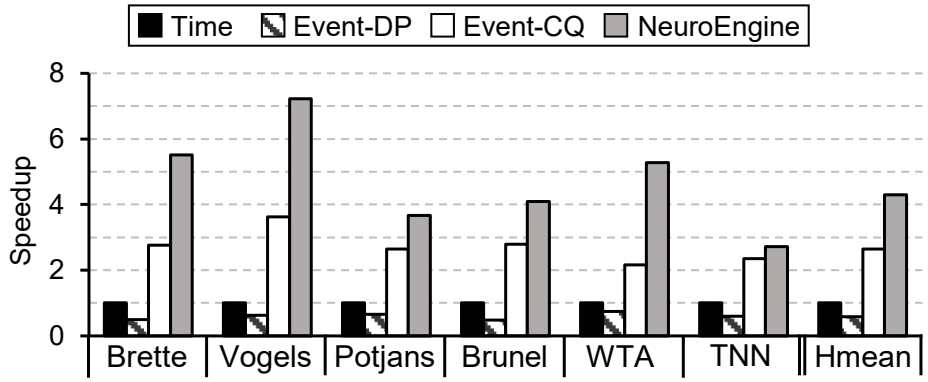
To examine the effectiveness of the composite queue (Section 4.3.2), we profile the total number of neuron computations in the simulation (Figure 4.14). `Event-Ideal` indicates the ideal number of computations excluding duplicate or dead events. The profiling results show that the composite queue properly avoids the scheduling of redundant events. Compared to `Time`, `Event-DP` reduces the number of computations by 53% on average. `Event-CQ` further decreases the number of computations by 76% in comparison to `Event-DP` by removing redundant computations. Remarkably, `Event-CQ` achieves near-ideal computation reduction with only 6% overhead due to the approximation in the datapath (Section 4.3.1).

4.5.5 Speedup Using Lazy Update

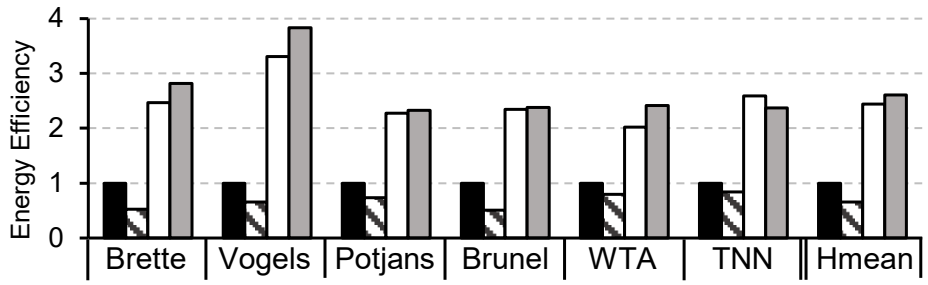
In this experiment, we evaluate the parallelization opportunities introduced with lazy update (Section 4.3.3). For evaluation, we profile the active cycles of neuron computation datapath, weight accumulation unit, and safe weight handler (Figure 4.15). The `Etc` represents the routing and synchronization overhead. We examine the degree of parallelism (i.e., DOP) by dividing the sum of active cycles of each component by the overall latency. Compared to `Event-CQ` which achieves only 1.06 DOP on average, `NeuroEngine` achieves 1.80 DOP. With the high parallelization opportunity, `NeuroEngine` achieves 1.76× speedup over `Event-CQ`. We also observe that lazy update further reduce the overall latency by reducing the weight accumulation overhead for two reasons. First, the weight accumulation for lazy neurons bypass the composite queue and do not suffer from slow priority queue accesses. Second, lazy update performs neuron computation for neurons that are about to fire a spike; therefore, the spikes are generated in a shorter time interval and fully utilize the pipeline stages for weight accumulation. Note that `TNN` introduces a large load imbalance between the cores after the feedforward inhibition [117]; therefore, both `NeuroEngine` and `Event-CQ` suffer from large load imbalance (`Etc`) overhead.

4.5.6 Fast & Efficient Event-Driven Simulations

In this experiment, we evaluate the overall performance benefits of `NeuroEngine`. We employ two evaluation metrics: simulation speed and energy consumption. For comparison, we measure the latency and the energy consumption of the hardware with the six benchmarks (Table 4.2). Our results indicate the significant performance and energy efficiency improvements of `NeuroEngine` (Figure 4.16). Using the composite queue, `Event-CQ` achieves 2.65× harmonic mean speedup over `Time` while `Event-DP` is slower compared to `Time`. Using the lazy update, `NeuroEngine` improves the performance and energy efficiency, achieving 4.30× and 2.60× harmonic mean speedup and energy efficiency over `Time`.



(a) Speedup normalized to the time-driven simulator



(b) Energy efficiency normalized to the time-driven simulator

Figure 4.16: Improvements in the simulation efficiency of NeuroEngine over time-driven and event-driven baselines.

4.5.7 Iso-area Comparisons

We conduct an iso-area comparison of `NeuroEngine` as the increased complexity in the datapath and the memory units consume a larger area (Table 4.1). For fairness, we increase the number of datapaths with additional memory ports to increase parallelism in `Time`. The modified `Time` baseline consumes 253.02 mm^2 which is slightly larger than `NeuroEngine`. Then, we evaluate the modified version under the same benchmarks and compare it against `NeuroEngine`. Figure 4.17 illustrates the normalized speedup and energy efficiency of `NeuroEngine` over the iso-area `Time`. The result indicates that NeuroEngine still achieves 2.62× harmonic mean speedup and 2.38× energy efficiency over the iso-area baseline.

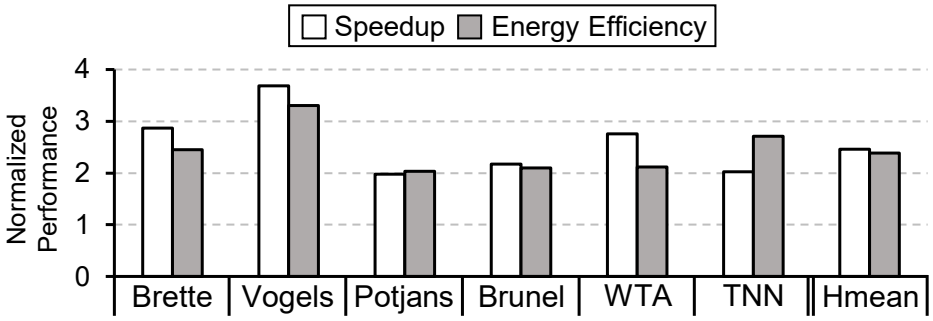


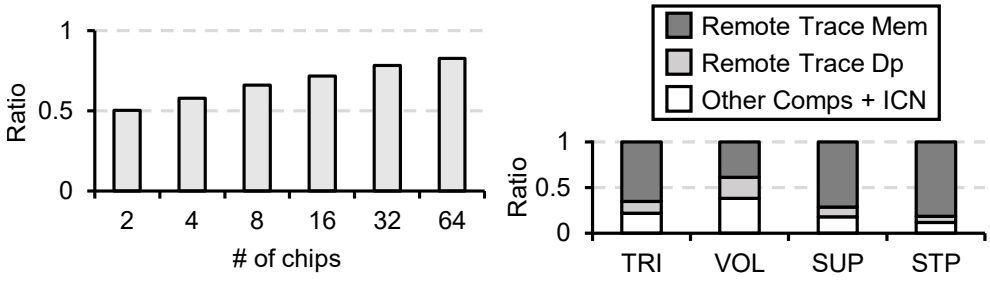
Figure 4.17: The speedup and energy efficiency of `NeuroEngine` over the iso-area `Time` baseline.

Chapter 5

NeuroSync: A Scalable and Accurate Brain Simulator Using Safe and Efficient Speculation

5.1 Motivation

We perform two different experiments to identify the inefficiencies of the time-precise simulators (Section 2.3.2): 1) slow simulation speed due to underutilization and 2) energy inefficiency due to remote trace update (Figure 5.1). The details of the baseline simulator and the benchmarks are discussed in Section 5.3.1 and Section 5.5.1, respectively. To ensure accurate spike transfer, the time-precise simulator suffers from



(a) Ratio of the underutilized cycles at various scales for `TRI` learning rule

(b) Energy consumption breakdown at 64-chip scale with different learning rules

Figure 5.1: Profile result on the (a) underutilized cycles and (b) energy breakdown of the time-precise simulator using `Izhikevich` benchmark.

excessive underutilization as the simulator scales. Especially, the spike transfer latency from the source core to the destination core increases with scale. Therefore, the cores waste more time waiting for the spike transfer to finish. Figure 5.1a shows the ratio of the underutilized cycles waiting for the spike transfer in the time-precise simulator. We scale the simulator and benchmark to evaluate the ratio at different scales. The result shows that the ratio of the underutilized cycles increases with scale, reaching up to 79.18% at a 64-chip scale.

Also, the time-precise simulator suffers from excessive energy consumption to access the memory for remote trace updates. The reason for the overhead is twofold. First, each core should keep and update remote traces of all the remote neurons. As the local neurons are connected to multiple remote neurons (ranging from 100 to 1,000 in typical networks), the number of remote traces in a core is larger than that of the other states. Second, the traces are updated for each timestep even in the absence of spikes, demanding frequent memory accesses. Figure 5.1b shows the energy breakdown of the 64-chip time-precise simulator with different learning rules. `Remote Trace Mem` and `Remote Trace Dp` indicate the energy consumed on the remote trace memory and the remote trace update datapath, respectively. `Other Comps` indicates the energy consumed to update the other states, including the internal states, weights, and local traces. Also, `ICN` refers to the energy consumed on the on-chip and off-chip interconnection. The result shows that most of the energy consumption stems from remote trace memory. Especially, it takes up to 81.33% on `STP` benchmark, which requires multiple traces to update a synapse. There exists a mapping technique to mitigate the overhead; however, it exhibits limited improvement [74].

5.2 Key Ideas

In this section, we propose NeuroSync, a novel architecture to enable both accurate and speculative brain simulations. The key idea is to speculatively change the simulation

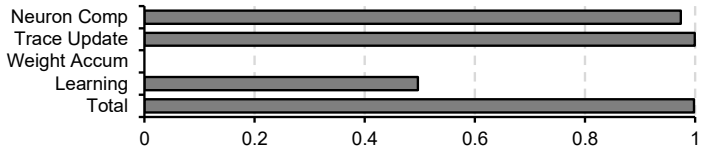


Figure 5.2: Profile result on the ratio of the computations that can be speculatively executed without waiting for the spikes from other cores.

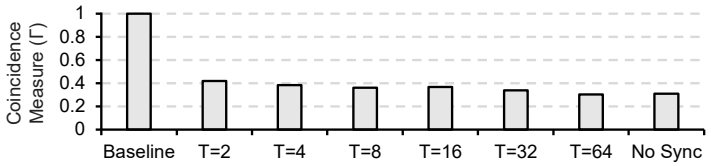


Figure 5.3: Coincidence measure of the time-precise simulator and speculative simulators with relaxed synchronization.

order to maximize the computational efficiency and guarantee simulation accuracy. We first analyze the brain simulation workloads to identify that the time-precise simulation is overly conservative while naively relaxing the synchronization would result in simulation inaccuracy (Section 5.2.1). Based on the observation, we present Safe Speculation, the key mechanism to enable both speculative and accurate simulations (Section 5.2.1). We also present an optimal dataflow based on the new data reuse opportunity in the speculative brain simulation; to temporally reuse the same data across timesteps (Section 5.2.1). However, we observe that the speculative simulation suffers from high overhead due to the rollback and recovery procedure (Section 5.2.2). Accordingly, we propose a speculation-optimal learning simulation methodology, Deferred Learning, which minimizes the overhead without affecting the functionality (Section 5.2.2).

5.2.1 Speculative Brain Simulation

Observations

To begin with, we present our observation that the simulator can execute most of the computations, assuming that the cores do not receive spikes from other cores. We profile the ratio of the computations that a core can speculatively execute. Figure 5.2 illustrates the results of the four simulation stages on `Izhikevich-TRI` benchmark; `total` indicates the sum of all computations. The results show that a core can execute most of the computations without waiting for the spikes from other cores. More than 97% of the computations can be executed in the two stages as a neuron rarely receives or fires a spike within a single timestep. On the other hand, the simulator fails to speculatively execute most of the computations related to the weight accumulation and learning stage. The reason is that they are triggered only when a neuron fires a spike. However, the trace update stage contributes a large amount of the total computations; thus, most of the computations can be speculatively executed. As a result, we conclude that most of the computations are not vulnerable to belated spike arrivals and the time-precise simulation is overly conservative.

Although most of the computations can be speculatively executed, we observe that relaxing the synchronization critically affects the functionality of the simulations. We compare the spiking activity of the time-precise baseline against speculative simulators with relaxed synchronization. For speculative simulators, we take two different strategies: *periodic synchronization* and *no synchronization*. Using the strategies, the cores speculatively proceed the timestep without synchronization. Instead, they synchronize only at the end of each pre-defined period (i.e., periodic synchronization) or do not synchronize at all (i.e., no synchronization).

As a key comparison metric, we employ the coincidence measure (Γ) between reference spike trains (from time-precise simulators) and target spike trains (from specu-

lative simulators) defined in previous publications [67, 71]. Γ is defined as:

$$\Gamma = \frac{N_{coinc} - \langle N_{coinc} \rangle}{0.5 \cdot N \cdot (N_{data} + N_{model})}$$

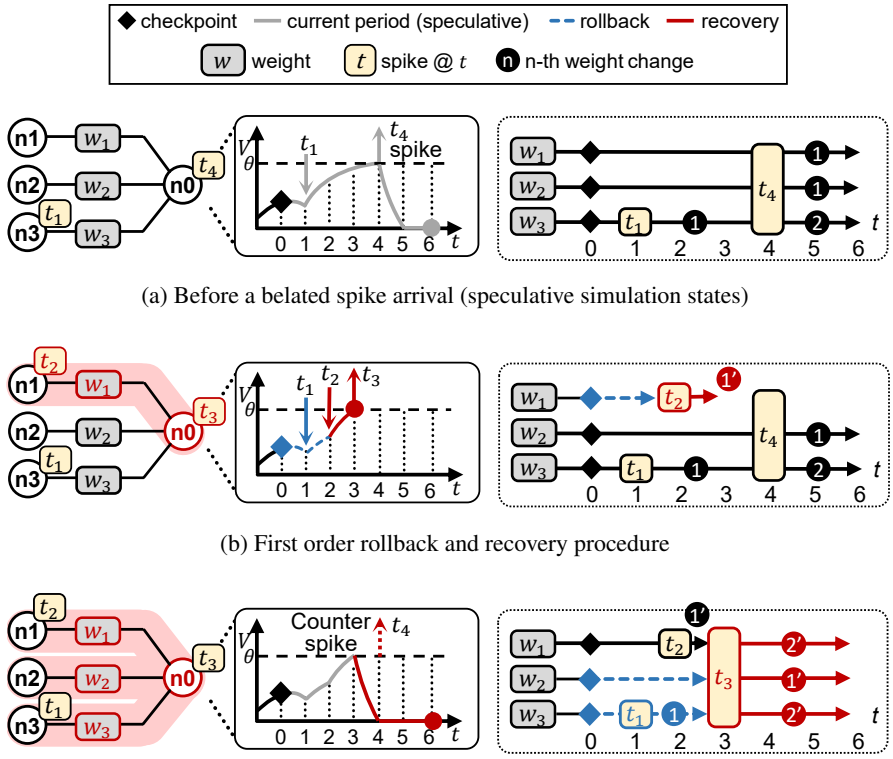
where N_{coinc} indicates the number of spike sets in the reference and target spike trains that coincide within a pre-defined precision (i.e., 2 ms). $\langle N_{coinc} \rangle$ is the number of coincidences at a random chance. Also, N_{data} and N_{model} refer to the number of spikes in the reference and target spike trains, respectively. Lastly, N is used to normalize Γ to a maximum of one. To summarize, a high Γ value indicates a high correlation between two spike trains, and thus, a high similarity between two simulators.

Figure 5.3 illustrates the evaluation results. We use `Izhikevich-TRI` benchmark for four-chip-scale and simulate the network over 200 ms. The duration was chosen after a typical duration between subsequent inputs (i.e., duration between subsequent saccades) in biological networks [117]. `Baseline` indicates the time-precise baseline and `T=2`$\sim$`64` indicate simulators with the periodic synchronization with a period of 2 to 64. Also, `No Sync` indicates a speculative simulator with no synchronization. The result clearly shows that relaxing the synchronization greatly affects the spiking activity of the neurons with a large correlation gap between the simulators. Even with a small synchronization period of two, the speculative simulator exhibits a small Γ value of 0.42. The gap widens as the synchronization period increases with Γ value of 0.31 with no synchronization.

From the observation, we conclude that despite the small fraction of the computations affected by the belated spike arrivals (Figure 5.2), it greatly affects the spiking activity (Figure 5.3).

Safe Speculation

Motivated by the observation, we propose Safe Speculation, a speculative and accurate brain simulation mechanism. We aim to take advantage of the speculative simulations while achieving the same functionality as the time-precise baseline. Using Safe



(a) Before a belated spike arrival (speculative simulation states)

(b) First order rollback and recovery procedure

(c) Second order rollback and recovery procedure

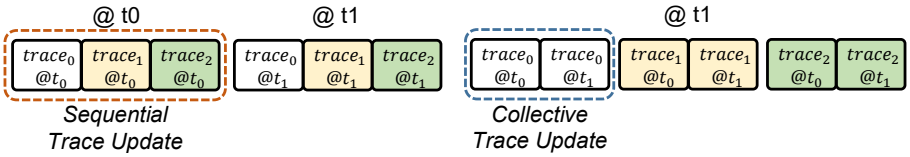
Figure 5.4: Mechanism of the rollback and recovery procedure in Safe Speculation. The figure illustrates the voltage changes of the destination neuron (n_0) and the weight changes of the fan-in synapses.

Speculation, each core speculatively advances the timesteps within a predefined period without synchronization. By speculatively advancing the timesteps, the cores are fully utilized without waiting for other cores to finish. To ensure accurate simulation, the core saves clean copies of the simulation states during speculation as checkpoints. When a mis-speculation is detected, the core rolls back the simulation states using the checkpoints and recovers the states. After all the computations and communications, including the rollback and recovery finish, the cores globally synchronize and advance to the next period.

Figure 5.4 illustrates an example of Safe Speculation. First, the simulator specu-

latively advances the timesteps for a pre-defined period, from t_0 to t_6 (Figure 5.4a). For simplicity, we do not include the trace changes in the figure. During speculation, a neuron (n_0) receives a spike at t_1 and fires a spike at t_4. The core evaluates the neuronal states and traces for each timestep and evaluates the weight changes when either the source or destination neuron fires a spike (❶–❷ indicates n-th weight change due to the spikes). The core saves the clean copies of the simulation states (marked with diamond) as checkpoints when the states change. For example, it saves the clean copy of the internal state and the weight w_3 at t_1 and saves the copy of the weight w_1 and w_2 at t_4. Upon receiving a belated spike that should have arrived at t_2 from a source neuron (n_1), Safe Speculation performs first-order rollback and recovery (Figure 5.4b). In the first-order rollback and recovery, the core rolls back the states to t_0 using the checkpoints. It rolls back the 1) internal states and local traces of the destination neuron (n_0), 2) remote traces of the source neuron (n_1), and 3) the weight of the fan-out synapse of the source neuron ($n_1 - n_0$). Then, the core recomputes the states to t_2 as a rollback procedure (marked with a blue line in the figure). Finally, it recovers the states using the belated spike at t_2 (marked with red line). Therefore, the internal states are updated and w_1 changes to a new value due to the spike (❶′).

However, during the recovery, the simulator potentially suffers from additional mis-speculations; the spiking activity of the destination neuron changes during recovery. For example, the destination neuron (n_0) fires a spike at t_3 instead of t_4 when recovering the internal states (Figure 5.4b). In this case, the computations using the mis-speculated spike t_4 should be canceled. Therefore, the core triggers a second-order rollback and recovery, which rolls back the computations related to the mis-speculated spike (Figure 5.4c). The core rolls back post-learning for the fan-in synapses ($w_1 - w_3$) of the destination neuron (n_0) at t_4 and recomputes learning at t_3. Also, the core generates and transfers a *counter spike* to the destination cores to inform the cores to cancel the mis-speculated spike at t_4.



(a) Baseline trace update

(b) Collective Trace Update

Figure 5.5: Trace update stage of the baseline and Collective Trace Update.

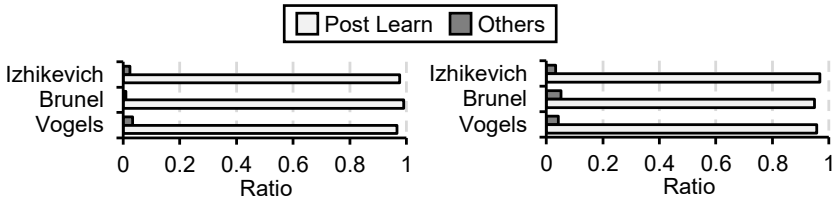
Collective Trace Update

Aside from the performance benefits, we identify that the speculation introduces a new data reuse opportunity: to reuse the data across different timesteps. Figure 5.5 shows the trace update procedure of the baseline simulator and our new dataflow. In time-precise simulations, the simulator updates all the traces (i.e., remote traces) before advancing to the next timestep (Figure 5.5a). The simulator sequentially accesses all the traces, and thus, it suffers from a limited temporal data reuse opportunity. Therefore, we adopt *Collective Trace Update* which collectively update the same traces across different timesteps (Figure 5.5b). Instead of updating the traces at each timestep, it collectively updates the traces at each synchronization period assuming that the traces are not used for learning within the period. If the speculation fails and learning demands up-to-date traces within the period, the simulator updates the required traces on-demand.

5.2.2 Optimal Speculative Simulation

Observations

As a next step, we evaluate the area and performance overhead of Safe Speculation and Collective Trace Update by conducting two different experiments. First, we evaluate the additional on-chip memory (i.e., area overhead) to keep the checkpoints. To do so, we profile the maximum amount of clean copies each core saves (i.e., checkpoints) over the whole simulation period. Second, we evaluate the performance overhead by profiling the computations spent on the rollback and recovery. Similarly, we profile the

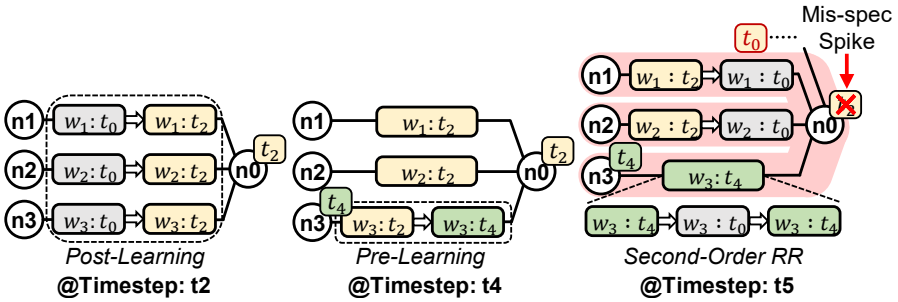


(a) Area overhead (b) Latency overhead

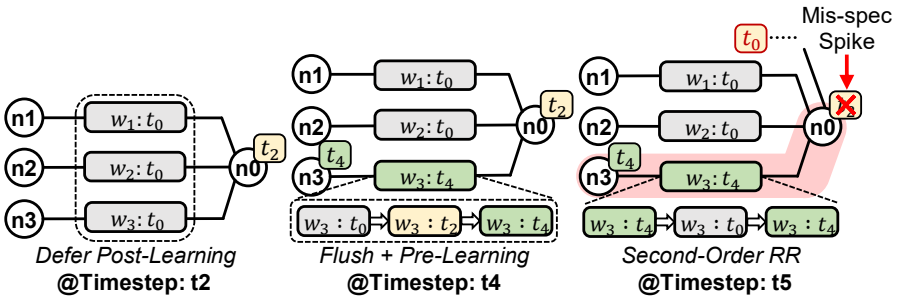
Figure 5.6: Area and latency overhead of the rollback and recovery due to post-learning (`Post Learn`) and other types of computations (`Others`). The results are normalized to the total overhead.

maximum overhead among the cores as the overall latency is bound to the tail latency. For both cases, we classify the overhead into different categories according to which type of computation incurs the overhead.

Figure 5.6 shows the profile results on three different benchmarks; `Post Learn` indicates the overhead induced by post-learning, and `Others` indicates the overhead due to the other types of computations (e.g., pre-learning). For the checkpoint profile results, `Post Learn` includes the overhead induced by 1) remote traces that are updated on-demand to support post-learning and 2) updated weights due to post-learning. For the latency profile results, `Post Learn` indicates the latency overhead to roll back post-learning in the second-order rollback and recovery (Section 5.2.1). We normalize the data to the maximum amount of checkpoints (Figure 5.6a) and the maximum latency overhead among the cores (Figure 5.6b). The results indicate that the post-learning-related overhead becomes a bottleneck in terms of area and latency; `Post-Learn` incurs more than 95% of the total overhead in both cases. The reason is that only a single core simulates post-learning for all the fan-in synapses triggered by a spike; on the other hand, other computations such as pre-learning for the fan-out synapses are spread across different cores. Due to the sparse nature of the spikes, post-learning incurs significant load imbalance among the cores and critically affects the worst-case area and latency overhead. Note that the total amount of checkpoints for pre-learning and post-learning are the same; however, the cores do not share on-chip



(a) Baseline mechanism



(b) Deferred Learning mechanism

Figure 5.7: Learning and the second-order rollback and recovery procedure of the baseline and the optimized mechanism using Deferred Learning.

memory, and thus post-learning contributes more to the area overhead.

Deferred Learning

To minimize the excessive post-learning overhead in the speculative simulations, we propose a new simulation methodology, Deferred Learning: *to defer post-learning until a source neuron fires a spike*. The key insight is that post-learning can be deferred without affecting the simulation results. Unlike pre-learning, which is immediately followed by weight accumulation, post-learning is not followed by weight accumulation until the source neuron fires a spike. As the updated weight affects the other computations (e.g., neuron computation) only after weight accumulation, the simulator can safely defer post-learning.

By adopting Deferred Learning on top of Safe Speculation, the simulator reduces

the number of speculative post-learning; thus, the simulator significantly reduces the post-learning-related overhead (Section 5.2.2). First, a core does not need to keep the post-learning-related dirty copies when the learning is deferred; therefore, it consumes significantly less area. Second, the simulator no longer suffers from the excessive latency overhead to iterate and roll back all the fan-in synapses upon a second-order rollback and recovery.

Figure 5.7 illustrates an example of how Deferred Learning differs from the baseline learning mechanism. The timing information on neurons (e.g., t_2 in Figure 5.7a @t_2 for n_0) refers to the spike time of the neurons. Also, the tuples for the synapses (e.g., w_1:t_0) indicate learning with the weight and spike time that triggered learning. Figure 5.7a illustrates the baseline mechanism. In the example, the baseline simulator performs 1) post-learning for fan-in synapses due to the output spike at t_2, 2) pre-learning for the fan-out synapse due to the incoming spike at t_4, and 3) second-order rollback and recovery (RR) due to the mis-speculated spike at t_5. During the rollback and recovery at t_5, the simulator iterates over all the synapses to roll back post-learning at t_2 and pre-learning at t_4. Therefore, it suffers from high area overhead to save the clean copies of the weights and latency overhead to roll back the weights.

On the other hand, Deferred Learning benefits from the reduced area and latency overhead by deferring post-learning until the source neuron fires a spike (Figure 5.7b). At t_2, the destination neuron (n_0) fires a spike; however, the simulator simply defers post-learning. As a result, the simulator does not save the clean copies for the fan-in synapses. When a source neuron (n_3) fires a spike at t_4, the simulator flushes the deferred post-learning for the synapse (w_3:t_2) and then performs pre-learning for the synapse (w_3:t_4). Upon a second-order rollback and recovery at t_5, the simulator rolls back and recovers only the fan-in synapses whose deferred computations have been flushed (i.e., w_3:t_4). For the other two fan-in synapses, the simulator simply discards deferred post-learning. Therefore, Deferred Learning significantly reduces the excessive overhead to iterate over all the fan-in synapses upon a mis-speculation.

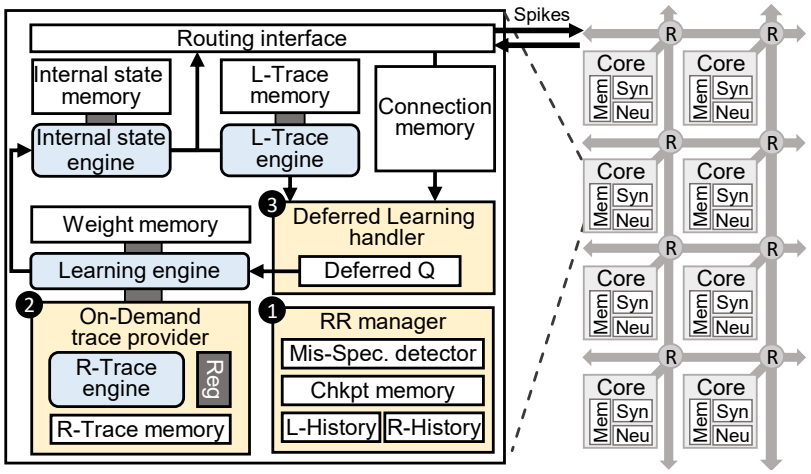


Figure 5.8: Overview of the NeuroSync architecture.

5.3 NeuroSync Architecture

In this section, we design NeuroSync to implement the three key ideas presented in Section 5.2. We add three different modules on top of the existing time-precise simulator [8, 26, 72]: 1) Rollback & Recovery (RR) manager, 2) On-Demand trace provider, and 3) Deferred Learning handler. Figure 5.8 illustrates the overview of the NeuroSync architecture. We first introduce the working mechanism of the baseline time-precise simulator (Section 5.3.1). Then, we design RR manager which detects mis-speculations and orchestrates rollback and recovery procedures for fast and accurate speculative simulations (Section 5.3.2, Figure 5.8–❶). Next, we design On-Demand trace provider to enable Collective Trace Update for higher energy efficiency (Section 5.3.3, Figure 5.8–❷). Finally, we design Deferred Learning handler to effectively defer and flush post-learning (Section 5.3.4, Figure 5.8–❸). We also release the functional simulation framework for NeuroSync.

5.3.1 Baseline Simulator

As the baseline, we adopt an end-to-end time-precise simulator presented in [8]. The baseline simulator is essentially a combination of: 1) a time-precise simulator similar to the state-of-the-art neuromorphic processor, Loihi [26], 2) a flexible digital neuron [72] capable of supporting 12 neuron models including Leaky Integrate-and-Fire (LIF) [121] and Izhikevich [61] model, and 3) a flexible learning engine [8] to support 17 biologically plausible learning rules including Long-Term, Short-Term, and Homeostatic weight changes.

The baseline simulator keeps four modules (colored in blue in Figure 5.8) and corresponding memories to support time-precise simulations. A core iterates over the local neurons to update internal states using the Internal state engine. The simulator can reconfigure the engine to simulate the target neuron model [72]. At the same time, L-Trace engine updates the local traces in parallel. If a neuron fires a spike, it transfers the spikes to the destination cores. It also triggers post-learning for the fan-in synapses using Learning engine. To update the weight, it uses Weight memory, which keeps the list of weight, delay, and learning-related metadata. Learning engine applies different learning rules depending on the microcode. Then, the cores synchronize to ensure that all the generated spikes are properly delivered. Next, R-Trace engine updates the traces. We configure R-Trace engine to process up to eight remote traces in parallel. Finally, Learning engine performs pre-learning for the fan-out synapses in a similar way to post-learning, and performs weight accumulation.

Each core processes up to 128 neurons, 32,768 synapses, and 4,096 remote traces and rely on on-chip SRAM to store the neuronal and synaptic states. To simulate a larger scale network, 64 cores are connected together using a 2D mesh topology in a single chip, and 64 chips are connected together in a single board. The scale is chosen after the single-board scale of the existing simulation systems adopting multichip designs [10, 28, 29, 37]. The chips are connected using 2 GBps low-power wireline transceivers [105]. For inter-chip communications, we adopt a bandwidth-efficient

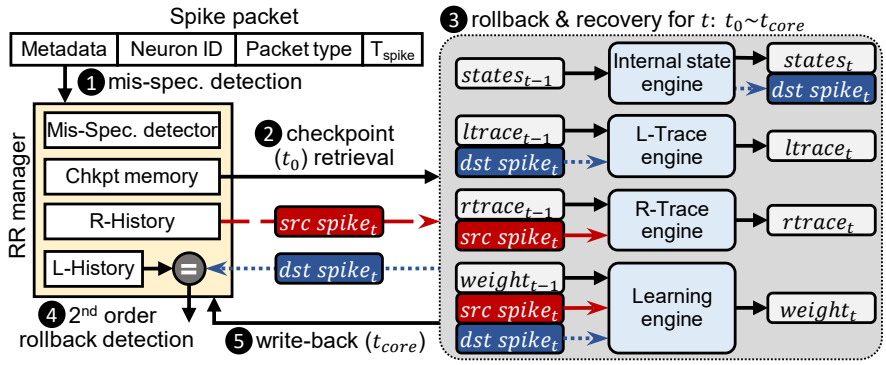


Figure 5.9: Working mechanism of RR manager.

multicast-based routing [138] in SpiNNaker. Also, we use a point-to-point routing similar to the one adopted in Loihi [26] for inter-core communications.

5.3.2 RR manager

As the first step, we design RR manager to support Safe Speculation. It manages the checkpoints and the spiking history of the neurons and detects mis-speculations to perform rollback and recovery. Figure 5.9 shows the mechanism of RR manager. First, when a state changes during the speculation and generates a dirty copy, RR manager saves the clean copy to the checkpoint memory. Also, it saves the spiking history of the remote and local neurons to the history buffer (R-History and L-History) when a neuron receives or fires a spike.

To properly detect mis-speculations, RR manager extends the spike packet by embedding the modulo of the spike time and speculation period (t_{spike}) and adds new packet types (e.g., counter spike). Upon receiving a spike, RR manager compares t_{spike} against the current simulation time of the core (t_{core}). If t_{spike} is earlier than t_{core}, RR manager triggers a first-order rollback and recovery (❶). Then, it retrieves checkpoints of the mis-speculated states from memory: 1) the remote traces of the source neuron ($rtrace_{t_0}$), 2) weights of the fan-out synapses of the source neuron ($weight_{t_0}$), 3) the local traces ($ltrace_{t_0}$), and the internal states ($states_{t_0}$) of the destination neurons (❷).

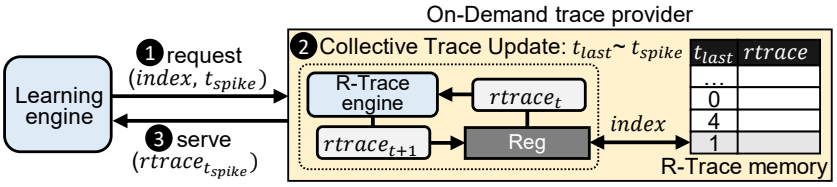


Figure 5.10: Working mechanism of On-Demand trace provider.

Next, RR manager rolls back and recovers the states back to t_{core} using the checkpoints and the spike history of the remote neurons ($src\ spike_t$) from R-History (❸). During recovery, RR manager checks whether the spiking activity of the destination neuron ($dst\ spike_t$) changes using L-History. If the spiking activity changes during recovery, RR manager triggers a second-order rollback and recovery (❹). After recovering the states, the manager writes back the recovered states to the memory (❺).

5.3.3 On-Demand trace provider

Next, we design On-Demand trace provider to support Collective Trace Update. It extends Remote trace memory and Remote trace engine to update the traces on-demand and at each synchronization period. Figure 5.10 illustrates the internal architecture and the working mechanism of On-Demand trace provider. First, it keeps a time-augmented trace memory to store an additional field (t_{last}), which indicates the last time the trace was updated. When a spike triggers learning, Learning engine requests remote traces of the neuron involved in learning (❶). Next, On-Demand trace provider reads the traces ($rtrace$) and the last updated time (t_{last}) for the index. Using the data, it collectively updates the traces from t_{last} to t_{spike} by reusing the data across different timesteps (❷). After that, it serves the updated traces to the learning engine and writes back the data (❸). Finally, after the synchronization, it iterates over the remote trace memory to update all the remote traces to the synchronization time.

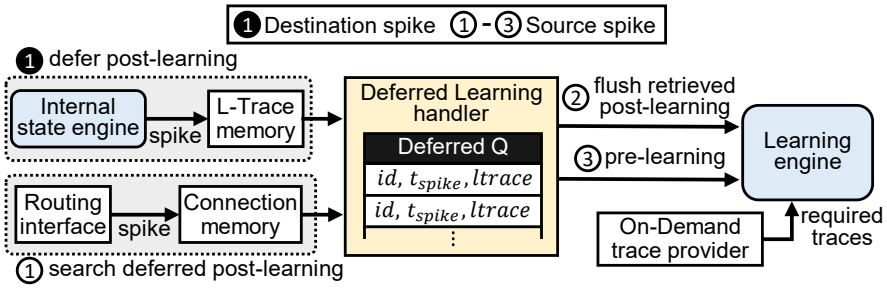


Figure 5.11: Working mechanism of Deferred Learning handler.

5.3.4 Deferred Learning handler

Lastly, we design Deferred Learning handler for Deferred Learning (Section 5.2.2). It defers post-learning for the fan-in synapses and properly flushes them when a source neuron fires a spike. Figure 5.11 illustrates Deferred Learning handler design. When a destination neuron fires a spike, the simulator skips post-learning for the fan-in synapses; instead, it stores the neuron id (id), spike time (t_{spike}), and local traces ($ltrace$) to the deferred queue (Deferred Q) (❶). Upon receiving a new incoming spike from a source neuron, the handler iterates over Deferred Q to search for the deferred learning (①). If a destination neuron connected to the source neuron exists in the queue, the handler flushes the deferred post-learning for the synapse connecting the remote neuron and the destination neuron (②). After that, the simulator performs pre-learning for the fan-out synapses (③).

Deferred Learning handler serves two additional mechanisms to support second-order rollback and recovery and to efficiently manage the queue. First, upon a second-order rollback and recovery, the handler searches for the fan-in synapses that flushed the deferred computations and thus are the target synapses to roll back and recover (e.g., synapse $n_3 - n_0$ in Figure 5.7b). For that, it uses R-History to check whether the source neuron for the fan-in synapse fired a spike. Then, it uses RR manager to roll back and recover the weights of the synapses whose source neuron fired a spike. Second, the handler flushes the deferred computations at the end of each synchronization

period. If Deferred Q becomes too large, the time to search for the deferred computations in the queue incurs a slowdown; therefore, the handler flushes the deferred computations periodically. Note that the deferred computations can be safely flushed after the periodic synchronization.

5.4 Simulation Front-Ends

To deploy brain simulations on NeuroSync, we design a custom software stack. First, we implement a programming interface by modifying Brian2 [122] to specify the target network using custom APIs. The provided APIs include: 1) `create_neurons`: create a population of neurons using the initial states, a target neuron model, and the corresponding parameters, and 2) `create_conns`: connect a source and a destination neuron with the provided weight, delay, a target learning rule, and corresponding parameters. There also exist other APIs to support the input stimulus, pseudo random generations, and simulation result monitors. Then, the software stack constructs the network to extract simulation parameters such as initial simulation states. Next, it uses a graph partitioning tool [69] to map the networks to the hardware in a way that reduces the inter-core connections and the number of per-core remote traces. The software stack also generates routing tables for the inter-chip communications in a way that minimizes the number of entries and packet hops [89]. Finally, the software stack sends the network parameters and the connection data to the NeuroSync cores, where they program the memory according to the data.

5.5 Evaluation

5.5.1 Experimental Setup

For evaluation, we compare a 64-chip brain simulator using NeuroSync against a 64-chip baseline time-precise simulator (Section 5.3.1). In this section, for brevity, we use

Table 5.1: Evaluated simulation benchmarks.

Network (Neu. Model)	Izhikevich [61] (Izhi.)				Brunel [19] (CUB)				Vogels [135] (COBE)			
Learning	Tri	Vol	Sup	STP	Tri	Vol	Sup	STP	Tri	Vol	Sup	STP
# Neurons	20,480				20,480				30,000			
# Synapses	20,480,000				20,480,000				18,000,000			

the term `NeuroSync` and `Time-Precise` to indicate the 64-chip NeuroSync and the 64-chip time-precise baseline, respectively. Also, we use the term `Speculative` to indicate a baseline speculative simulator without Deferred Learning support. To obtain the area-, power-, and latency-related parameters for each simulator, we write the RTL code in Verilog. We thoroughly verify the functional correctness of the implementation by comparing the neuronal and the synaptic changes of the design against those of the software-based simulator [122]. Then, we synthesize the code using Synopsys Design Compiler with Samsung 28-nm standard cell library and memory compiler. Also, we use Synopsys IC Compiler II using the same technology node for place and route. The off-chip transceiver costs are evaluated based on the previous literature on the low-power transceiver design [105]. We assume 80 ns off-chip link latency as in the previous literature with a similar chip scale [22]. Lastly, the synchronization period is set to 16 from our empirical analysis that it was sufficient to reduce the idle cycles (Figure 5.12).

We adopt three representative biological networks used to evaluate FlexLearn [8] with different neuron models. Along with the networks, we also utilize four representative learning rule combinations adopted in FlexLearn; therefore, we evaluate our simulator on twelve different benchmarks with different neuron models and learning rules. Then, we scale the networks to evaluate the performance and the energy efficiency of the target system scale. Table 5.1 summarizes the characteristics of the benchmarks. The learning rules include, `TRI`: a combination of triplet [96], multiplicative, and exponential [44], `VOL`: a combination of momentary [16], additive [120], and

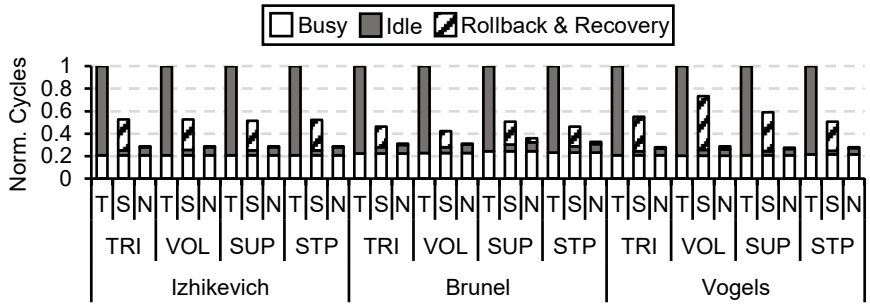


Figure 5.12: Breakdown of the simulation cycles on three different baseline simulators (Normalized to the total cycles of `Time-Precise`).

linear decay [16], `SUP`: a combination of suppression [41] and multiplicative [131], and `STP`: a combination of all-to-all [44], multiplicative, and separate [4].

5.5.2 Results

Reduced Underutilization Using NeuroSync: In this experiment, we evaluate the improved simulation efficiency of `Speculative` and `NeuroSync`. We profile the cycles of each simulator and classify them into different categories depending on whether the simulator is in busy, idle, or in rollback and recovery state. Our results clearly show that the speculative simulation benefits from the reduced underutilization. Also, `NeuroSync` maximizes the benefits of the speculative simulation by reducing the rollback and recovery overhead. Figure 5.12 shows the profile results; `T`, `S`, and `N` refer to `Time-Precise`, `Speculative`, and `NeuroSync`, respectively. Compared to `Time-Precise`, `Speculative` achieves higher speedup by reducing the idle cycles. However, it suffers from high rollback and recovery overhead which occupies 49.24% of the cycles on average. `NeuroSync` further reduces the total simulation latency by consuming up to 32.31× fewer cycles on rollback and recovery. Note that `Speculative` consumes slightly fewer cycles in the idle state than `NeuroSync` due to the overlap between the idle and the rollback and recovery state.

Reduced Memory Accesses Using Collective Trace Update: Aimed at minimizing

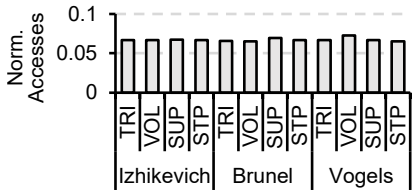


Figure 5.13: Relative remote trace memory access of NeuroSync normalized to that of Time-Precise

Figure 5.14: Total amount of dirty copies of NeuroSync normalized to that of Speculative

the excessive energy consumed on the remote trace update, Collective Trace Update reuses the trace data across different timesteps. To evaluate the effectiveness of the data reuse opportunity, we profile the number of remote trace memory accesses. Figure 5.13 shows the number of remote trace memory accesses of NeuroSync normalized to Time-Precise. The result shows that NeuroSync significantly reduces the number of memory accesses using Collective Trace Update. On average, NeuroSync accesses remote trace memory 14.89× less compared to Time-Precise.

Minimal Area Overhead of NeuroSync: Deferred Learning reduces the dirty copies generated in Speculative. We examine the area benefits of Deferred Learning by evaluating the maximum dirty copy generated over the simulation in Speculative and NeuroSync. Figure 5.14 shows the maximum amount of dirty copies generated in NeuroSync normalized to that of Speculative. The result shows that Deferred Learning minimizes the overhead by generating significantly fewer dirty copies compared to Speculative. Overall, it generates 37.52× fewer dirty copies on average compared to Speculative. NeuroSync generates slightly more dirty copies using Vogels-Vol as it fires a spike more frequently than other benchmarks; however, it still generates 7.73× fewer dirty copies over Speculative. By deferring post-learning, NeuroSync effectively reduces the required checkpoints for the rollback and recovery.

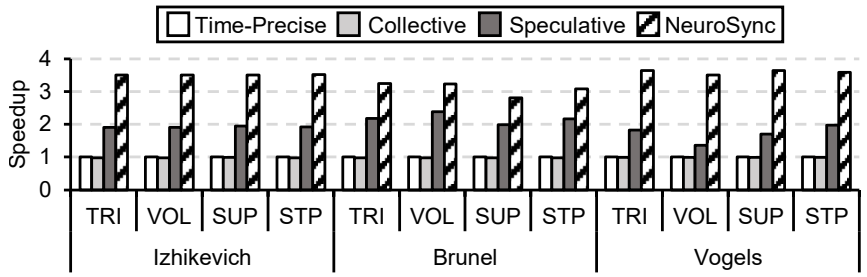
By reducing the dirty copies over Speculative, NeuroSync benefits from the

Table 5.2: Chip area comparison (mm^2).

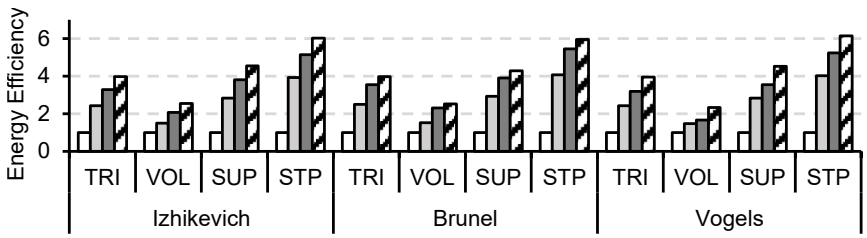
Component	Time-Precise	Speculative	NeuroSync
Neuron	3.07	4.67	4.67
Synpase	36.59	49.53	37.00
Trace	12.72	24.14	17.45
ICN	9.50	9.63	9.54
Total	61.88	87.97	68.66

reduced area overhead. Table 5.2 illustrates the single-chip area of Time-Precise, Speculative, and NeuroSync. Neuron, Synapse, and Trace indicate the area of the memory modules and the datapaths for each state variable. ICN indicates the interconnection network, including the router, off-chip interface, routing table, and link to support core-to-core and chip-to-chip communications. The results show that Speculative incurs 42.16% additional area compared to Time-Precise. The overhead mainly stems from Synapse and Trace to store the dirty copies related to the post-learning. On the other hand, NeuroSync incurs only 10.96% area overhead by significantly reducing the dirty copies generated by post-learning. Also, both Speculative and NeuroSync consume slightly higher ICN overhead due to the slightly larger spike packet size and the increased link length with larger footprint.

Overall Efficiency of NeuroSync: In this experiment, we evaluate the overall speedup and energy efficiency of NeuroSync. To analyze the impact of each idea, we evaluate the speedup and energy efficiency by applying each scheme one by one to the Time-Precise baseline. For that purpose, we implement an additional baseline with only collective trace update (Collective). Figure 5.15 shows the speedup and the energy efficiency improvement of the baseline simulators over Time-Precise. Collective achieves 2.38× harmonic mean energy efficiency with reduced memory accesses, and Speculative achieves 1.90× and 3.17× harmonic mean speedup and energy efficiency over Time-Precise. Note that the improved energy efficiency compared to Collective stems from the reduced static energy consumption with



(a) Speedup normalized to `Time-Precise`.



(b) Energy efficiency normalized to `Time-Precise`

Figure 5.15: Improvements in the brain simulation efficiency of `NeuroSync` over `Time-Precise` and `Speculative`.

reduced latency. `NeuroSync` further improves the speedup and energy efficiency by 1.73× and 1.19× over `Speculative` by reducing the number of rollback and recovery. Overall, `NeuroSync` achieves 3.37× speedup and 3.81× energy efficiency over `Time-Precise`.

Comparison Against CPU: In this experiment, we compare the performance of NeuroSync against that of the CPU-based solution. For evaluation, we profile the brain simulation latency on Intel Xeon E5-2630 v4 CPU (24-core, 2.2 GHz) by using NEST [45] as a simulation framework. We break down the performance into busy and idle cycles and normalize the latency to the busy cycles of `NeuroSync`. Figure 5.16 illustrates the profile results. The result shows that CPU-based solution is 299.04× slower on average compared to `NeuroSync` due to its slow underlying processor compared to the hardware-based solutions. Increasing the number of cores or adopting multi-node simulation would relieve the performance overhead to some extent. However, it would

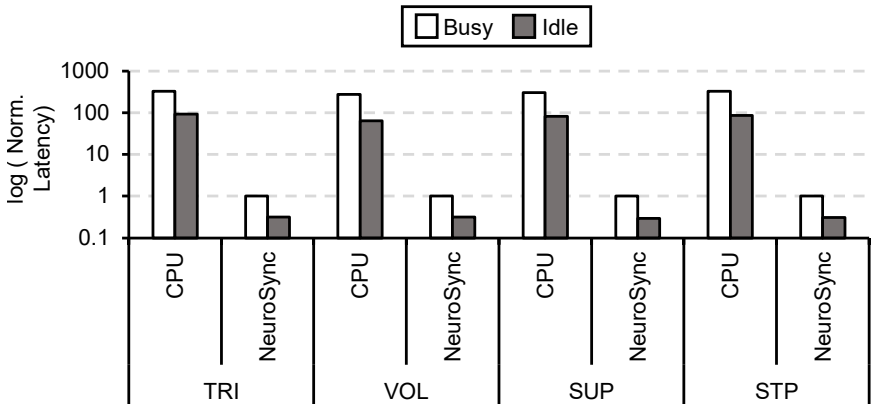


Figure 5.16: Latency of the CPU-based solution and NeuroSync normalized to the busy cycles of NeuroSync.

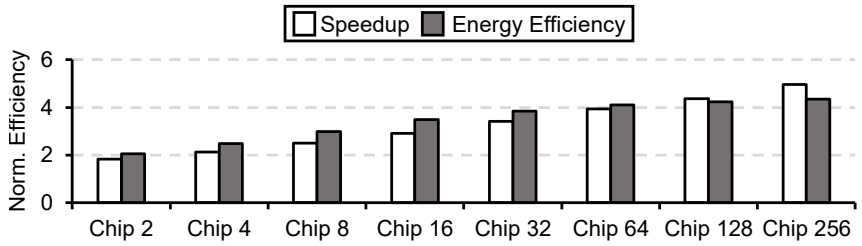


Figure 5.17: Speedup and energy efficiency of NeuroSync normalized to those of Time-Precise at various chip scales.

incur higher idle time which is 62.32× slower than the overall latency of NeuroSync even at the current scale.

Scalability of NeuroSync: Lastly, we evaluate the scalability of NeuroSync. We set Izhikevich-TRI as the benchmark and scale the network from 2-chip scale to 256-chip scale. For each scale, the system simulates $\sqrt{2}\times$ more neurons and $2\times$ more synapses. Then, we evaluate the speedup and the energy efficiency of NeuroSync over Time-Precise at each scale. Figure 5.17 shows the evaluation results. The results show that NeuroSync is extremely scalable compared to Time-Precise using its speculative simulation methodology. NeuroSync achieves higher speedup due to the increased spike transfer latency among the cores. Also, NeuroSync achieves

higher energy efficiency due to the reduced static power consumption with higher speed. Especially at the 256-chip scale, the simulator achieves 4.97× and 4.35× higher speedup and energy efficiency compared to `Time-Precise`.

Chapter 6

A Flexible, Efficient, and Scalable Simulation System

The main purpose of the FlexBrain project was to design a flexible, efficient, and scalable brain simulation accelerator. Therefore, we need to integrate FlexLearn, NeuroEngine, and NeuroSync into a single system to meet the three goals. As the three architectures optimize different steps in the overall brain simulation procedure, we can easily combine them after minor architectural modifications as follows.

FlexLearn + NeuroEngine: The event-driven neuron computation proposed in NeuroEngine orthogonally applies to FlexLearn as most of the learning rules are based on the spiking activity of the neurons rather than the internal states of the neurons. However, some sub-rules utilize the internal states, and thus, are incompatible with the event-driven execution mechanism, which evaluates the states only when a neuron receives or fires a spike. For example, the `Average Voltage` sub-rule in `HB` and the `Voltage Dependent` sub-rule for `ADH` require an averaged voltage of the destination neuron (1.6 and 5.2 in Table 3.2). Also, two sub-rules (1.5 and 3.1 in Table 3.2) involve the voltage value of the neuron when the neuron receives a spike. Therefore, they are incompatible with lazy update, which updates the states only when a neuron fires a spike. As a result, we should apply the time-driven execution mechanisms or partially apply the event-driven execution mechanisms (i.e., without lazy update) for such sub-rules when combining the two architectures.

FlexLearn + NeuroEngine + NeuroSync: Also, we can extend NeuroSync to support the flexible datapath and the event-driven execution mechanisms proposed in FlexLearn and NeuroEngine. First, we should extend the checkpoint memory to save the events stored in the composite queue structure. Accordingly, the RR manager saves the checkpoint of the in-events that are scheduled before the next synchronization period and the out-events stored in the out-event queue. Upon a mis-speculation, the composite queue structure should restore the event scheduler using the checkpoint and recovers the computations. If the mis-speculation results in generating a mis-speculated spike, NeuroSync should generate a counter-spike which not only recovers the weight accumulation at the destination core but also removes the in-event for the spike. Note that, NeuroSync is already integrated with FlexLearn to flexibly support different learning rules. Therefore, the combined architecture results in a flexible, efficient, and scalable acceleration system which was the optimal goal of the FlexBrain project.

Chapter 7

Related Work

Representative Software-Based Simulators: Brain simulation requires accurate evaluations of networks with different topologies, neuron behaviors, and learning rules. For higher applicability, existing simulators rely on general-purpose processors such as CPUs and GPUs. For example, NEURON [54], NEST [45], GENESIS [15], Brian [48, 122] and Auryn [142] provide a simulation framework to support CPU-based brain simulations. GPU-based simulators such as CARLsim [87], GeNN [139], NeMo [34], and BSIM [104] are also used to accelerate the simulation speed. To improve simulation speeds, the simulators often utilize multiple machines to simulate large-scale biological neural networks. For instance, a simulation of 900 million neurons and 9 trillion synapses has been demonstrated by running C2 [7], a CPU-based cortical simulator developed by IBM Research, on top of 147,465 CPUs [6]. Using programmable processors, they provide accurate and versatile simulation frameworks; however, they suffer from severe computation and memory requirements of the brain simulation and thus suffer from slow simulation speed [8, 72, 127, 142].

Representative Hardware-Based Simulators: Aimed at improving the simulation efficiency, custom-designed hardware-based simulators (e.g., TrueNorth [5, 82, 108], ROLLS [103], Loihi [26], ODIN [38–40]) employ digital circuits implementing the learning rules and the neurons of the human brain. These accelerators achieve fast sim-

ulation speeds and higher energy efficiency over general-purpose processors at the cost of flexibility. For instance, Loihi [26] implements a limited set of learning rules which can be defined in a sum-of-products form taking as input spike counts and per-neuron traces. Some prior studies, aimed at achieving even higher energy efficiency, propose to employ analog circuits. Example analog-based accelerators include Neurogrid [11] and BrainScaleS project [110–112].

Event-Driven Simulators: To minimize the computational overhead, some simulators adopt event-driven simulation methodology. NEVESIM [95], ED-LUT [106], PUNNETS [77], and MVASpike [17] are the CPU-based simulators supporting event-driven computations. Among them, ED-LUT supports two-stage neurons using a LUT table. Also, the NEVESIM, PUNNETS, and MVASpike support two-stage neurons using a linear approximation for the prediction. To improve the performance over the CPU-based approach, some simulators adopt FPGAs or ASICs for the event-driven simulation. But, none of the existing works effectively handles the two-stage neurons and incoming spikes with synaptic delays. Minitaur [92] and PEASE [107] are hardware-based event-driven simulators; however, they only support one-stage neurons. Also, other FPGA-based simulators supporting simplified two-stage neurons using the SHQ [21, 113] do not effectively handle incoming spikes with synaptic delays.

Imprecise Simulators: Some simulators sacrifice accuracy for higher efficiency. To minimize the underutilization of the time-precise simulator, NeuroGrid [11] and BrainScaleS [35, 84] eliminate the global synchronization procedure. One of the SpiNNaker [94] libraries allow the cores to proceed the simulation without synchronization [140]. As these works do not properly perform rollback and recovery upon misspeculations, they incur inaccurate simulation results. TrueNorth [5,28] adopts a fixed-cycle synchronization where the cores proceed to the next timestep after waiting for only fixed cycles. By avoiding the synchronization procedure, the simulators benefit from simple design choices and higher simulation efficiency. However, the simulators do not perform rollback and recovery to ensure accurate simulations. Therefore, their

applicability is highly limited for the emerging brain simulation workloads, emphasizing the importance of spike times [62, 117]. Other simulators improve the simulation efficiency by limiting the key operations required to simulate the brain. Flexon [72] and proposes a flexible digital neuron to simulate a wide range of neuron behaviors. Also, the work by Smith [115] designs a digital circuit to simulate neuron behaviors. However, these works do not support an end-to-end simulation and thus should be augmented to other processors.

Brain Simulation Infrastructures: Brain simulations using the underlying hardware require software front-ends interfacing with the hardware. A typical workflow of a brain simulation is as follows. First, researchers specify the properties of their target biological neural networks (e.g., topology) using a hardware-independent description language such as PyNN [27] and NeuroML [20, 47]. Then, given the description, a neural network compiler (e.g., LCompiler [73, 74], a neuromorphic compiler by Ji et al. [66]) compiles the description and produces executable code for the underlying hardware or instruction set architecture (e.g., NISA [52]). During the compilation, the compiler may apply optimizations to improve performance; for instance, LCompiler for Intel Loihi optimizes neuron-to-core mapping to minimize energy consumption. After that, the produced executable runs on the underlying hardware. When the simulation outcomes are validated, the researcher can upload their brain simulation configurations to ModelDB [55], the open database for sharing latest findings in neuronal and synaptic dynamics.

Computing Like the Human Brain: Depending on how information is encoded, the abstraction of the human brain can be classified into two types: Rate-based Neural Networks (RNNs) and Temporal Neural Networks (TNNs) [116, 117]. Our work focuses on RNNs which use sequences of spikes from a *single* synapse to encode and convey information [16, 24, 32]. As RNNs use multiple spikes from a single synapse to communicate and compute, the highly complex interactions between the spikes need to be considered (e.g., homeostasis [143]). On the other hand, TNNs encode information

in the spatio-temporal patterns between the spikes from *multiple* synapses [75, 126]. With TNNs, the complex interactions needed by RNNs are no longer necessary, allowing much more simpler abstractions to be used [13]. Using the space-time computing capability of TNNs, prior studies demonstrate that various brain-like computations are possible with TNNs [70, 76, 81, 101]. For instance, a recent study by Smith [118] utilizes TNNs to implement a generalized race logic. As RNNs and TNNs both demand learning rules to adjust the synaptic weights, FlexLearn can equally benefit both RNNs and TNNs. Note that both RNNs and TNNs are active research topics, and there is yet no consensus on which is more suitable to describe the computations of the brain.

Chapter 8

Conclusion

To better understand the working mechanism of the brain, neuroscientists heavily rely on brain simulations. They strive to explore diverse biological phenomena to demystify the brain and thus impose three key requirements on the simulation framework. First, it should be flexible to support diverse neuron models and learning rules which describe different phenomena. Second, it should be fast and efficient to explore how the dynamics change over a long period of time (e.g., months or years). Lastly, it should be scalable to simulate large-scale networks such as the anatomical data of the animal brain.

Conventional simulation frameworks are usually implemented in software to cope with different network configurations. They utilize CPUs to support diverse neuron models and learning rules at the cost of simulation speed and energy efficiency. The GPU-based frameworks are also programmable with highly parallel simulations of neuronal dynamics. However, they are not sufficient to simulate complex dynamics with slow simulation speed and energy inefficiency.

The simulation frameworks with hardware-based accelerators improve the speed and efficiency to some extent. Unfortunately, they are still sub-optimal when simulating large-scale networks over long time intervals. First, the accelerators are inflexible so they can simulate only a limited set of neurons and learning rules. Second, the ex-

isting works still suffer from the intense computational overhead to update the internal state of the neurons. Lastly, the existing works suffer from underutilization and excessive memory access overhead as the simulator scales.

The dissertation proposed three novel architectures to resolve the three challenges of the existing hardware-based accelerators. First, we presented FlexLearn, a highly flexible and efficient on-chip learning architecture for brain simulations. FlexLearn achieves high flexibility by using the datapaths implementing the sub-rules as the basic building blocks, and high efficiency by integrating the datapaths in an efficient manner. We then integrated FlexLearn with Flexon, the state-of-the-art digital neuron, to implement an end-to-end brain simulation accelerator. The resulting FlexLearn accelerator realizes fast and highly efficient brain simulations by achieving high flexibility.

Second, we presented NeuroEngine, which enables fast and efficient advanced brain-inspired computing using the event-driven simulation system. The key idea is to optimize the key components in event-driven simulation ranging from the datapaths to the event scheduling units. We further improve the simulation by proposing a novel method called lazy update to increase the parallelism. The resulting design, NeuroEngine, achieves significant performance improvement and energy efficiency compared to both state-of-the-art time-driven and naive event-driven simulators.

Lastly, we presented NeuroSync, a scalable and accurate brain simulation system for large-scale brain simulations. NeuroSync speculatively changes the simulation order for higher performance and energy efficiency while guaranteeing simulation accuracy. To implement the idea, we proposed a rollback and recovery mechanism to correct the mis-speculated states and a new dataflow to maximize energy efficiency. We further adopted a novel speculation-optimal learning simulation methodology to enable speculative simulations at a low cost. Using the proposed schemes, NeuroSync, achieves significant performance improvement and energy efficiency compared to the state-of-the-art brain simulation systems with minimal area overhead.

Bibliography

[1] "Intel® Xeon® Processor E5-2650 v4," https://ark.intel.com/products/91767/Intel-Xeon-Processor-E5-2650-v4-30M-Cache-2-20-GHz-.

[2] "NVIDIA TITAN X Graphics Card for VR Gaming — NVIDIA GeForce," https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/.

[3] "Profiler User's Guide," https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[4] L. F. Abbott, J. A. Varela, K. Sen, and S. B. Nelson, "Synaptic Depression and Cortical Gain Control," *Science*, vol. 275, no. 5297, pp. 221–224, 1997.

[5] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, 2015.

[6] R. Ananthanarayanan, S. K. Esser, H. D. Simon, and D. S. Modha, "The Cat is Out of the Bag: Cortical Simulations with 10^9 Neurons, 10^{13} Synapses," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009, pp. 1–12.

[7] R. Ananthanarayanan and D. S. Modha, "Anatomy of a Cortical Simulator," in *Proceedings of the 2007 ACM/IEEE Conference on High Performance Networking and Computing (SC)*, 2007, pp. 1–12.

[8] E. Baek, H. Lee, Y. Kim, and J. Kim, "FlexLearn: fast and highly efficient brain simulations using flexible on-chip learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 304–318.

[9] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[10] F. Barchi, G. Urgese, A. Siino, S. Di Cataldo, E. Macii, and A. Acquaviva, "Flexible on-line reconfiguration of multi-core neuromorphic platforms," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 2, pp. 915–927, 2019.

[11] 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.

[12] G.-q. Bi and M.-m. Poo, "Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type," *Journal of neuroscience*, vol. 18, no. 24, pp. 10 464–10 472, 1998.

[13] O. Bichler, D. Querlioz, S. J. Thorpe, J.-P. Bourgoin, and C. Gamrat, "Extraction of temporally correlated features from dynamic vision sensors with spike-timing-dependent plasticity," *Neural Networks*, vol. 32, pp. 339–348, 2012.

[14] J. M. Bower and D. Beeman, *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System*, 2nd ed. Springer-Verlag New York, 1998.

[15] ——, *The book of GENESIS: exploring realistic neural models with the GEneral NEural SImulation System*. Springer Science & Business Media, 2012.

[16] J. M. Brader, W. Senn, and S. Fusi, "Learning Real-World Stimuli in a Neural Network with Spike-Driven Synaptic Dynamics," *Neural Computation*, vol. 19, no. 11, pp. 2881–2912, 2007.

[17] R. Brette, "Exact simulation of integrate-and-fire models with synaptic conductances," *Neural Computation*, vol. 18, no. 8, pp. 2004–2027, 2006.

[18] R. Brette, M. R. T. Carnevale, M. Hines, D. Beeman, J. M. Bower, M. Diesmann, A. Morrison, P. H. Goodman, J. Frederick C. Harris, M. Zirpe, T. Natschläger, D. Pecevski, B. Ermentrout, M. Djurfeldt, A. Lansner, O. Rochel, T. Vieville, E. Muller, A. P. Davison, S. E. Boustani, and A. Destexhe, "Simulation of networks of spiking neurons: A review of tools and strategies," *Journal of Computational Neuroscience*, vol. 23, pp. 349–398, 2007.

[19] N. Brunel, "Dynamics of Sparsely Connected Networks of Excitatory and Inhibitory Spiking Neurons," *Journal of Computational Neuroscience*, vol. 8, no. 3, pp. 183–208, 2000.

[20] R. C. Cannon, P. Gleeson, S. Crook, G. Ganapathy, B. Marin, E. Piasini, and R. A. Silver, "LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2," *Frontiers in Neuroinformatics*, vol. 8, p. 79, 2014.

[21] L.-C. Caron, M. D'Haene, F. Mailhot, B. Schrauwen, and J. Rouat, "Event management for large scale event-driven digital hardware spiking neural networks," *Neural networks*, vol. 45, pp. 83–93, 2013.

[22] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 609–622.

[23] K. Cheung, S. R. Schultz, and W. Luk, "NeuroFlow: A General Purpose Spiking Neural Network Simulation Platform using Customizable Processors," *Frontiers in Neuroscience*, vol. 9, p. 516, 2016.

[24] C. Clopath and W. Gerstner, "Voltage and Spike Timing Interact in STDP – A Unified Model," *Frontiers in Synaptic Neuroscience*, vol. 2, p. 25, 2010.

[25] M. Davies, "Benchmarks for progress in neuromorphic computing," *Nature Machine Intelligence*, vol. 1, no. 9, pp. 386–388, 2019.

[26] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, Guruguhanathan, Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[27] A. P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, "PyNN: a common interface for neuronal network simulators," *Frontiers in Neuroinformatics*, vol. 2, p. 11, 2009.

[28] M. V. DeBole, B. Taba, A. Amir, F. Akopyan, A. Andreopoulos, W. P. Risk, J. Kusnitz, C. O. Otero, T. K. Nayak, R. Appuswamy, P. J. Carlson, A. S. Cassidy, P. Datta, S. K. Esser, G. J. Garreau, K. L. Holland, S. Lekuch, M. Mastro, J. McKinstry, C. di Nolfo, B. Paulovicks, J. Sawada, K. Schleupen, B. G. Shaw, J. L. Klamo, M. D. Flickner, J. V. Arthur, and D. S. Modha, "TrueNorth: Accelerating from zero to 64 million neurons in 10 years," *Computer*, vol. 52, no. 5, pp. 20–29, 2019.

[29] L. Deng, G. Wang, G. Li, S. Li, L. Liang, M. Zhu, Y. Wu, Z. Yang, Z. Zou, J. Pei, Z. Wu, X. Hu, Y. Ding, W. He, Y. Xie, and L. Shi, "Tianjic: A unified and scalable chip bridging spike-based and continuous neural computation," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 8, pp. 2228–2246, 2020.

[30] M. D'Haene, B. Schrauwen, J. Van Campenhout, and D. Stroobandt, "Accelerating event-driven simulation of spiking neurons with multiple synaptic time constants," *Neural computation*, vol. 21, no. 4, pp. 1068–1099, 2009.

[31] J. J. DiCarlo, D. Zoccolan, and N. C. Rust, "How does the brain solve visual object recognition?" *Neuron*, vol. 73, no. 3, pp. 415–434, 2012.

[32] P. U. Diehl and M. Cook, "Unsupervised learning of digit recognition using spike-timing-dependent plasticity," *Frontiers in Computational Neuroscience*, vol. 9, p. 99, 2015.

[33] Z. Du, D. D. B.-D. Rubin, Y. Chen, L. He, T. Chen, L. Zhang, C. Wu, and O. Temam, "Neuromorphic Accelerators: A Comparison Between Neuroscience and Machine-Learning Approaches," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015, pp. 494–507.

[34] A. K. Fidjeland and M. P. Shanahan, "Accelerated Simulation of Spiking Neural Networks Using GPUs," in *2010 International Joint Conference on Neural Networks (IJCNN)*, 2010, pp. 1–8.

[35] J. Fieres, J. Schemmel, and K. Meier, "Realizing biological spiking network models in a configurable wafer-scale hardware system," in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, 2008, pp. 969–976.

[36] G. A. Fonseca Guerra and S. B. Furber, "Using stochastic spiking neural networks on spinnaker to solve constraint satisfaction problems," *Frontiers in neuroscience*, vol. 11, p. 714, 2017.

[37] E. P. Frady, G. Orchard, D. Florey, N. Imam, R. Liu, J. Mishra, J. Tse, A. Wild, F. T. Sommer, and M. Davies, "Neuromorphic Nearest Neighbor Search Using Intel's Pohoiki Springs," in *Proceedings of the Neuro-inspired Computational Elements Workshop (NICE*, 2020, pp. 1–10.

[38] C. Frenkel, G. Indiveri, J.-D. Legat, and D. Bol, "A Fully-Synthesized 20-Gate Digital Spike-Based Synapse with Embedded Online Learning," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[39] C. Frenkel, M. Lefebvre, J.-D. Legat, and D. Bol, "A 0.086-mm^2 12.7-pJ/SOP 64k-Synapse 256-Neuron Online-Learning Digital Spiking Neuromorphic Processor in 28nm CMOS," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 11, pp. 145–158, 2018.

[40] C. Frenkel, J.-D. Legat, and D. Bol, "A Compact Phenomenological Digital Neuron Implementing the 20 Izhikevich Behaviors," in *2017 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, 2017, pp. 1–4.

[41] R. C. Froemke and Y. Dan, "Spike-timing-dependent synaptic modification induced by natural spike trains," *Nature*, vol. 416, no. 6879, pp. 433–438, 2002.

[42] G. Fuhrmann, I. Segev, H. Markram, and M. Tsodyks, "Coding of Temporal Information by Activity-Dependent Synapses," *Journal of Neurophysiology*, vol. 87, pp. 140—-148, 2002.

[43] J. A. Garrido, R. R. Carrillo, N. R. Luque, and E. Ros, "Event and time driven hybrid simulation of spiking neural networks," in *Proc. International Work-Conference on Artificial Neural Networks (IWANN)*. Springer, 2011, pp. 554–561.

[44] W. Gerstner and W. M. Kistler, *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

[45] M.-O. Gewaltig and M. Diesmann, "NEST (NEural Simulation Tool)," *Scholarpedia*, vol. 2, no. 4, p. 1430, 2007.

[46] M. Gilson and T. Fukai, "Stability versus Neuronal Specialization for STDP: Long-Tail Weight Distributions Solve the Dilemma," *PLoS One*, vol. 6, no. 10, p. e25339, 2011.

[47] P. Gleeson, S. Crook, R. C. Cannon, M. L. Hines, G. O. Billings, M. Farinella, T. M. Morse, A. P. Davison, S. Ray, U. S. Bhalla, S. R. Barnes, Y. D. Dimitrova, and R. A. Silver, "NeuroML: A Language for Describing Data Driven Models of Neurons and Networks with a High Degree of Biological Detail," *PLoS Computational Biology*, vol. 6, no. 6, p. e1000815, 2010.

[48] D. F. M. Goodman and R. Brette, "The Brian simulator," *Frontiers in Neuroscience*, vol. 3, no. 2, pp. 192– 197, 2009.

[49] A. Gupta, Y. Wang, and H. Markram, "Organizing Principles for a Diversity of GABAergic Interneurons and Synapses in the Neocortex," *Science*, vol. 287, no. 5451, pp. 273–278, 2000.

[50] W. Gurstner, *Neuronal Dynamics: From Single Neurons To Networks And Models Of Cognition*. Cambridge University Press, 2014.

[51] R. Gütig, R. Aharonov, S. Rotter, and H. Sompolinsky, "Learning Input Correlations through Nonlinear Temporally Asymmetric Hebbian Plasticity," *Journal of Neuroscience*, vol. 23, no. 9, pp. 3697–3714, 2003.

[52] A. Hashmi, A. Nere, J. J. Thomas, and M. Lipasti, "A Case for Neuromorphic ISAs," in *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 145–158.

[53] D. O. Hebb, *The Organization of Behavior*. Wiley & Sons, 1949.

[54] M. L. Hines and N. T. Carnevale, "NEURON: A Tool for Neuroscientists," *The Neuroscientist*, vol. 7, no. 2, pp. 123–135, 2001.

[55] M. L. Hines, T. Morse, M. Migliore, N. T. Carnevale, and G. M. Shepherd, "ModelDB: a Database to Support Computational Neuroscience," *Journal of Computational Neuroscience*, vol. 17, no. 1, pp. 7–11, 2004.

[56] M. Hines and N. Carnevale, "The NEURON Simulation Environment," *Neural Computation*, vol. 9, no. 6, pp. 1179–1209, 1997.

[57] N. Hiratani and T. Fukai, "Interplay between short-and long-term plasticity in cell-assembly formation," *PloS one*, vol. 9, no. 7, p. e101535, 2014.

[58] R. V. Hoang, D. Tanna, L. C. J. Bray, S. M. Dascalu, and F. C. H. Jr., "A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling," *Frontiers in Neuroinformatics*, vol. 7, p. 19, 2013.

[59] J. R. Hughes, "Post-Tetanic Potentiation," *Physiological Reviews*, vol. 38, no. 1, pp. 91–113, 1958.

[60] A. Ioannou and M. G. H. Katevenis, "Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks," *IEEE/ACM Transactions on Networking*, vol. 15, no. 2, pp. 450–461, 2007.

[61] E. M. Izhikevich, "Simple Model of Spiking Neurons," *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

[62] ——, "Polychronization: Computation with Spikes," *Neural Computation*, vol. 18, no. 2, pp. 245–282, 2006.

[63] ——, "Solving the distal reward problem through linkage of stdp and dopamine signaling," *Cerebral cortex*, vol. 17, no. 10, pp. 2443–2452, 2007.

[64] E. M. Izhikevich and N. S. Desai, "Relating STDP to BCM," *Neural Computation*, vol. 15, no. 7, pp. 1511–1523, 2003.

[65] E. M. Izhikevich and G. M. Edelman, "Large-scale model of mammalian thalamocortical systems," *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, vol. 105, no. 9, pp. 3593–3598, 2008.

[66] Y. Ji, Y. Zhang, W. Chen, and Y. Xie, "Bridging the Gap Between Neural Networks and Neuromorphic Hardware with A Neural Network Compiler," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018, pp. 448–460.

[67] R. Jolivet, R. Kobayashi, A. Rauch, R. Naud, S. Shinomoto, and W. Gerstner, "A benchmark test for a quantitative assessment of simple neuron models," *Journal of neuroscience methods*, vol. 169, no. 2, pp. 417–424, 2008.

[68] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration," in *2009 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2009, pp. 423–428.

[69] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[70] S. R. Kheradpisheh, M. Ganjtabesh, and T. Masquelier, "Bio-inspired unsupervised learning of visual features leads to robust invariant object recognition," *Neurocomputing*, vol. 205, pp. 382–392, 2016.

[71] W. M. Kistler, W. Gerstner, and J. L. v. Hemmen, "Reduction of the hodgkin-huxley equations to a single-variable threshold model," *Neural computation*, vol. 9, no. 5, pp. 1015–1045, 1997.

[72] D. Lee, G. Lee, D. Kwon, S. Lee, Y. Kim, and J. Kim, "Flexon: A Flexible Digital Neuron for Efficient Spiking Neural Network Simulations," in *2018*

ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), 2018, pp. 275–288.

[73] C.-K. Lin, A. Wild, G. N. Chinya, Y. Cao, M. Davies, D. M. Lavery, and H. Wang, "Programming Spiking Neural Networks on Intel's Loihi," *Computer*, vol. 51, no. 3, pp. 52–61, 2018.

[74] C.-K. Lin, A. Wild, G. N. Chinya, T.-H. Lin, M. Davies, and H. Wang, "Mapping Spiking Neural Networks onto a Manycore Neuromorphic Hardware," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018, pp. 78–89.

[75] W. Maass, "Noisy Spiking Neurons with Temporal Coding have more Computational Power than Sigmoidal Neurons," pp. 211–217, 1997.

[76] W. Maass, T. Natschläger, and H. Markram, "Real-Time Computing Without Stable States: A New Framework for Neural Computation Based on Perturbations," *Neural Computation*, vol. 14, no. 11, pp. 2531–2560, 2002.

[77] T. Makino, "A Discrete-Event Neural Network Simulator for General Neuron Models," *Neural Computing & Applications*, vol. 11, no. 3-4, pp. 210–223, 2003.

[78] H. Markram, J. Lübke, M. Frotscher, and B. Sakmann, "Regulation of synaptic efficacy by coincidence of postsynaptic aps and epsps," *Science*, vol. 275, no. 5297, pp. 213–215, 1997.

[79] H. Markram, Y. Wang, and M. Tsodyks, "Differential signaling via the same axon of neocortical pyramidal neurons," *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, vol. 95, no. 9, pp. 5323–5328, 1998.

[80] S. J. Martin, P. D. Grimwood, and R. G. M. Morris, "Synaptic Plasticity and Memory: An Evaluation of the Hypothesis," *Annual Review of Neuroscience*, vol. 23, no. 1, pp. 649–711, 2000.

[81] T. Masquelier and S. J. Thorpe, "Unsupervised Learning of Visual Features through Spike Timing Dependent Plasticity," *PLoS Computational Biology*, vol. 3, no. 2, p. e31, 2007.

[82] 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, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[83] M. Mikaitis, G. P. Garcia, J. C. Knight, and S. B. Furber, "Neuromodulated Synaptic Plasticity on the SpiNNaker Neuromorphic System," *Frontiers in Neuroscience*, vol. 27, p. 105, 2018.

[84] S. Millner, A. Grübl, K. Meier, J. Schemmel, and M.-O. Schwartz, "A VLSI implementation of the adaptive exponential integrate-and-fire neuron model," in *Advances in Neural Information Processing Systems 23*, 2010, pp. 1642–1650.

[85] A. Morrison, M. Diesmann, and W. Gerstner, "Phenomenological models of synaptic plasticity based on spike timing," *Biological Cybernetics*, vol. 98, no. 6, pp. 459–478, 2008.

[86] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to understand large caches," Tech. Rep., 2009.

[87] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. V. Veidenbaum, "A configurable simulation environment for the efficient simulation of

large-scale spiking neural networks on graphics processors," *Neural Networks*, vol. 22, no. 5-6, pp. 791–800, 2009.

[88] H. Nair, J. P. Shen, and J. E. Smith, "Direct CMOS Implementation of Neuromorphic Temporal Neural Networks for Sensory Processing," 2020, arXiv preprint arXiv:2009.00457.

[89] J. Navaridas, M. Luján, L. A. Plana, S. Temple, and S. B. Furber, "Spinnaker: Enhanced multicast routing," *Parallel Computing*, vol. 45, pp. 49–66, 2015.

[90] F. Naveros, J. A. Garrido, R. R. Carrillo, E. Ros, and N. R. Luque, "Event- and time-driven techniques using parallel CPU-GPU co-processing for spiking neural networks," *Frontiers in Neuroinformatics*, vol. 11, p. 7, 2017.

[91] E. O. Neftci, H. Mostafa, and F. Zenke, "Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks," *IEEE Signal Processing Magazine*, vol. 36, pp. 61–63, 2019.

[92] D. Neil and S.-C. Liu, "Minitaur, an event-driven FPGA-based spiking network accelerator, author=Neil, Daniel and Liu, Shih-Chii," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2621–2628, 2014.

[93] A. Nere, A. Hashmi, M. Lipasti, and G. Tononi, "Bridging the Semantic Gap: Emulating Biological Neuronal Behaviors with Simple Digital Neurons," in *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 472–483.

[94] E. Painkras, L. A. Plana, J. Garside, S. Temple, S. Davidson, J. Pepper, D. Clark, C. Patterson, and S. Furber, "SpiNNaker: A Multi-Core System-on-Chip for Massively-Parallel Neural Net Simulation," in *Proceedings of the IEEE 2012 Custom Integrated Circuits Conference (CICC)*, 2012, pp. 1–4.

[95] D. Pecevski, D. Kappel, and Z. Jonke, "NEVESIM: event-driven neural simulation framework with a Python interface," *Frontiers in neuroinformatics*, vol. 8, p. 70, 2014.

[96] J.-P. Pfister and W. Gerstner, "Beyond Pair-Based STDP: a Phenomenological Rule for Spike Triplet and Frequency Effects," in *Advances in Neural Information Processing Systems 18*, 2006, pp. 1083—1090.

[97] L. A. Plana, D. Clark, S. Davidson, S. Furber, J. Garside, E. Painkras, J. Pepper, S. Temple, and J. Bainbridge, "SpiNNaker: Design and Implementation of a GALS Multicore System-on-Chip," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 7, no. 4, pp. 1–18, 2011.

[98] F. J. Ponulak and J. J. Hopfield, "Rapid, parallel path planning by propagating wavefronts of spiking neural activity," *Frontiers in computational neuroscience*, vol. 7, p. 98, 2013.

[99] T. C. Potjans and M. Diesmann, "The Cell-Type Specific Cortical Microcircuit: Relating Structure and Activity in a Full-Scale Spiking Network Model," *Cerebral Cortex*, vol. 24, no. 3, pp. 785–806, 2014.

[100] E. Prati, "Atomic scale nanoelectronics for quantum neuromorphic devices: comparing different materials," *International Journal of Nanotechnology*, vol. 13, no. 7, pp. 509–523, 2016.

[101] D. Probst, W. Maass, H. Markram, and M.-O. Gewaltig, "Liquid Computing in a Simplified Model of Cortical Layer IV: Learning to Balance a Ball," in *Proceedings of the 22nd International Conference on Artificial Neural Networks (ICANN)*, 2012, pp. 209–216.

[102] G. qiang Bi and M. ming Poo, "Synaptic Modification by Correlated Activity: Hebb's Postulate Revisited," *Annual Review of Neuroscience*, vol. 24, no. 1, pp. 139–166, 2001.

[103] N. Qiao, H. Mostafa, F. Corradi, M. Osswald, F. Stefanini, D. Sumislawska, and G. Indiveri, "A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses," *Frontiers in Neuroscience*, vol. 9, p. 141, 2015.

[104] P. Qu, Y. Zhang, X. Fei, and W. Zheng, "High Performance Simulation of Spiking Neural Network on GPGPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2510–2523, 2020.

[105] A. Ramachandran, A. Natarajan, and T. Anand, "29.4 a 16gb/s 3.6pj/b wireline transceiver with phase domain equalization scheme: Integrated pulse width modulation (ipwm) in 65nm cmos," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 488–489.

[106] E. Ros, R. Carrillo, E. M. Ortigosa, B. Barbour, and R. Agís, "Event-driven simulation scheme for spiking neural networks using lookup tables to characterize neuronal dynamics," *Neural Computation*, vol. 18, no. 12, pp. 2959–2993, 2006.

[107] A. Roy, S. Venkataramani, N. Gala, S. Sen, K. Veezhinathan, and A. Raghunathan, "A programmable event-driven architecture for evaluating spiking neural networks," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2017, pp. 1–6.

[108] J. Sawada, F. Akopyan, A. S. Cassidy, B. Taba, M. V. Debole, P. Datta, R. Alvarez-Icaza, A. Amir, J. V. Arthur, A. Andreopoulos, R. Appuswamy, H. Baier, D. Barch, D. J. Berg, C. di Nolfo, S. K. Esser, M. Flickner, T. A. Horvath, B. L. Jackson, J. Kusnitz, S. Lekuch, M. Mastro, T. Melano, P. A. Merolla, S. E. Millman, T. K. Nayak, N. Pass, H. E. Penner, W. P. Risk, K. Schleupen, B. Shaw, H. Wu, B. Giera, A. T. Moody, N. Mundhenk, B. C. V. Essen, E. X. Wang, D. P. Widemann, Q. Wu, W. E. Murphy, J. K. Infantolino, J. A. Ross, D. R. Shires, M. M. Vindiola, R. Namburu, and D. S. Modha, "TrueNorth

Ecosystem for Brain-Inspired Computing: Scalable Systems, Software, and Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016, pp. 130–141.

[109] L. K. Scheffer, C. S. Xu, M. Januszewski, Z. Lu, S.-y. Takemura, K. J. Hayworth, G. B. Huang, K. Shinomiya, J. Maitlin-Shepard, S. Berg, J. Clements, P. M. Hubbard, W. T. Katz, L. Umayam, T. Zhao, D. Ackerman, T. Blakely, J. Bogovic, T. Dolafi, D. Kainmueller, T. Kawase, K. A. Khairy, L. Leavitt, P. H. Li, L. Lindsey, N. Neubarth, D. J. Olbris, H. Otsuna, E. T. Trautman, M. Ito, A. S. Bates, J. Goldammer, T. Wolff, R. Svirskas, P. Schlegel, E. Neace, C. J. Knecht, C. X. Alvarado, D. A. Bailey, S. Ballinger, J. A. Borycz, B. S. Canino, N. Cheatham, M. Cook, M. Dreher, O. Duclos, B. Eubanks, K. Fairbanks, S. Finley, N. Forknall, A. Francis, G. P. Hopkins, E. M. Joyce, S. Kim, N. A. Kirk, J. Kovalyak, S. A. Lauchie, A. Lohff, C. Maldonado, E. A. Manley, S. McLin, C. Mooney, M. Ndama, O. Ogundeyi, N. Okeoma, C. Ordish, N. Padilla, C. M. Patrick, T. Paterson, E. E. Phillips, E. M. Phillips, N. Rampally, C. Ribeiro, M. K. Robertson, J. T. Rymer, S. M. Ryan, M. Sammons, A. K. Scott, A. L. Scott, A. Shinomiya, C. Smith, K. Smith, N. L. Smith, M. A. Sobeski, A. Suleiman, J. Swift, S. Takemura, I. Talebi, D. Tarnogorska, E. Tenshaw, T. Tokhi, J. J. Walsh, T. Yang, J. A. Horne, F. Li, R. Parekh, P. K. Rivlin, V. Jayaraman, M. Costa, G. S. Jefferis, K. Ito, S. Saalfeld, R. George, I. A. Meinertzhagen, G. M. Rubin, H. F. Hess, V. Jain, and S. M. Plaza, "A connectome and analysis of the adult drosophila central brain," *Elife*, vol. 9, p. e57443, 2020.

[110] J. Schemmel, D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner, "A Wafer-Scale Neuromorphic Hardware System for Large-Scale Neural Modeling," in *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2010, pp. 1947–1950.

[111] J. Schemmel, J. Fieres, and K. Meier, "Wafer-scale Integration of Analog Neural Networks," in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, 2008, pp. 431–438.

[112] J. Schemmel, A. Grübl, K. Meier, and E. Mueller, "Implementing Synaptic Plasticity in a VLSI Spiking Neural Network Model," in *2006 IEEE International Joint Conference on Neural Networks (IJCNN)*, 2006, pp. 1–6.

[113] G. Séguin-Godin, F. Mailhot, and J. Rouat, "Efficient event-driven approach using synchrony processing for hardware spiking neural networks," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2015, pp. 2696–2699.

[114] S. B. Shrestha and G. Orchard, "Slayer: Spike layer error reassignment in time," in *Advances in Neural Information Processing Systems 31*, 2018, pp. 1419–1428.

[115] J. E. Smith, "Efficient Digital Neurons for Large Scale Cortical Architectures," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, 2014, pp. 229–240.

[116] ——, "Research Agenda: Spacetime Computation and the Neocortex," *IEEE Micro*, vol. 37, no. 1, pp. 8–14, 2017.

[117] ——, *Space-Time Computing with Temporal Neural Networks*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2017.

[118] ——, "Space-Time Algebra: A Model for Neocortical Computation," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 289–300.

[119] ——, "A Neuromorphic Paradigm for Online Unsupervised Clustering," 2020, arXiv preprint arXiv:2005.04170.

[120] S. Song, K. D. Miller, and L. F. Abbott, "Competitive Hebbian learning through spike-timing-dependent synaptic plasticity," *Nature Neuroscience*, vol. 3, no. 9, pp. 919–926, 2000.

[121] R. B. Stein, "A Theoretical Analysis of Neuronal Variability," *Biophysical Journal*, vol. 5, no. 2, pp. 173–194, 1965.

[122] M. Stimberg, R. Brette, and D. F. Goodman, "Brian 2, an intuitive and efficient neural simulator," *Elife*, vol. 8, p. e47314, 2019.

[123] J. E. Stine, I. Castellanos, M. Wood, F. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, "FreePDK: An Open-Source Variation-Aware Design Kit," in *2007 IEEE International Conference on Microelectronic Systems Education (MSE)*, 2007, pp. 173–174.

[124] D. J. Surmeier and R. Foehring, "A mechanism for homeostatic plasticity," *Nature Neuroscience*, vol. 7, no. 7, pp. 691–692, 2004.

[125] P. T. P. Tang, T.-H. Lin, and M. Davies, "Sparse coding by spiking neural networks: Convergence theory and computational results," 2017, arXiv preprint arXiv:1705.05475.

[126] S. J. Thorpe and M. Imbert, "Biological Constraints on Connectionist Modelling," *Connectionism in Perspective*, pp. 63–92, 1989.

[127] R. A. Tikidji-Hamburyan, V. Narayana, Z. Bozkus, and T. A. El-Ghazawi, "Software for brain network simulations: a comparative study," *Frontiers in neuroinformatics*, vol. 11, p. 46, 2017.

[128] M. Tsodyks, A. Uziel, and H. Markram, "Synchrony Generation in Recurrent Networks with Frequency-Dependent Synapses," *Journal of Neuroscience*, vol. 20, no. 1, p. RC50, 2000.

[129] M. Tsodyks and S. Wu, "Short-term synaptic plasticity," *Scholarpedia*, vol. 8, no. 10, p. 3153, 2013.

[130] T. Tsumoto, "Long-term potentiation and long-term depression in the neocortex," *Progress in neurobiology*, vol. 39, no. 2, pp. 209–228, 1992.

[131] G. G. Turrigiano, K. R. Leslie, N. S. Desai, L. C. Rutherford, and S. B. Nelson, "Activity-dependent scaling of quantal amplitude in neocortical neurons," *Nature*, vol. 391, no. 6670, pp. 892–896, 1998.

[132] G. G. Turrigiano and S. B. Nelson, "Homeostatic plasticity in the developing nervous system," *Nature Reviews Neuroscience*, vol. 5, no. 2, pp. 97–107, 2004.

[133] J. Uy, M. Ridding, S. Hillier, P. Thompson, and T. Miles, "Does induction of plastic change in motor cortex improve leg function after stroke?" *Neurology*, vol. 61, no. 7, pp. 982–984, 2003.

[134] M. C. W. van Rossum, G. Q. Bi, and G. G. Turrigiano, "Stable Hebbian Learning from Spike Timing-Dependent Plasticity," *The Journal of Neuroscience*, vol. 20, no. 23, pp. 8812–8821, 2000.

[135] T. P. Vogels, H. Sprekeler, F. Zenke, C. Clopath, and W. Gerstner, "Inhibitory Plasticity Balances Excitation and Inhibition in Sensory Pathways and Memory Networks," *Science*, vol. 334, no. 6062, pp. 1569–1573, 2011.

[136] T. P. Vogels and L. F. Abbott, "Signal propagation and logic gating in networks of integrate-and-fire neurons," *Journal of neuroscience*, vol. 25, no. 46, pp. 10 786–10 795, 2005.

[137] T. Willhalm, R. Dementiev, and P. Fay, "Intel® Performance Counter Monitor - A better way to measure CPU utilization," http://www.intel.com/software/pcm.

[138] J. Wu and S. Furber, "A multicast routing scheme for a universal spiking neural network architecture," *The Computer Journal*, vol. 53, no. 3, pp. 280–288, 2010.

[139] E. Yavuz, J. Turner, and T. Nowotny, "GeNN: a code generation framework for accelerated brain simulations," *Scientific Reports*, vol. 6, no. 1, pp. 1–14, 2016.

[140] A. R. Young, M. E. Dean, J. S. Plank, and G. S. Rose, "A review of spiking neuromorphic hardware communication systems," *IEEE Access*, vol. 7, pp. 135 606–135 620, 2019.

[141] F. Zenke, E. J. Agnes, and W. Gerstner, "Diverse synaptic plasticity mechanisms orchestrated to form and retrieve memories in spiking neural networks," *Nature communications*, vol. 6, p. 6922, 2015.

[142] F. Zenke and W. Gerstner, "Limits to high-speed simulations of spiking neural networks using general-purpose computers," *Frontiers in Neuroinformatics*, vol. 8, p. 76, 2014.

[143] F. Zenke, G. Hennequin, and W. Gerstner, "Synaptic Plasticity in Neural Networks Needs Homeostasis with a Fast Rate Detector," *PLOS Computational Biology*, vol. 9, no. 11, p. e1003330, 2013.

초 록

두뇌의 연산 능력을 이해하고 활용하기 위해 신경과학자들은 시간에 따른 상태 변화를 예측하는 두뇌 시뮬레이션에 의존한다. 두뇌에서 나타나는 다양한 행동 양식을 탐색하고 신경과학에 크게 기여하기 위해서, 신경과학자들은 대규모 시뮬레이션을 유연하고 효율적인 시뮬레이션 프레임워크를 필요로 한다. 하지만, 기존 프레임워크는 소프트웨어 기반으로 시뮬레이션하여 너무 느리고 비효율적이다. 하드웨어 기반 가속기들은 이와 같은 비효율성을 완화하지만, 시뮬레이션의 유연성이 낮고 매 시간 간격마다 필요로하는 고정적으로 발생하는 연산 및 동기화에 의해 최적의 시뮬레이션 환경을 제공하지 못한다. 본 학위논문은 유연성, 효율성, 확장성을 극대화한 두뇌 시뮬레이션 프레임워크를 위해 설계한 아키텍처인 FlexLearn, NeuroEngine, NeuroSync를 소개한다. 첫째로 FlexLearn은 17가지 대표적인 하위 학습 방법을 지원하는 회로를 설계하고, 하위 학습 방법을 조합하여 타겟하는 학습 모델을 시뮬레이션할 수 있게 만들었다. 다음 NeuroEngine은 매 시간 간격마다 뉴런의 상태 변화를 계산하지 않고 아닌 특정 이벤트가 발생하는 경우에만 상태 변화를 계산하여 시뮬레이션 효율성을 극대화한다. 마지막으로 NeuroSync는 매 시간 간격마다 시뮬레이션을 동기화하는 과정에서 발생하는 오버헤드를 최소화한다.

주요어: 두뇌 시뮬레이션, 뉴로모픽 컴퓨터, 스파이킹 뉴럴 네트워크, 가속 시스템
학번: 2018-24440