General Purpose Computation with Spiking Neural Networks: Programming, Design Principles, and Patterns

John V. Monaco
Naval Postgraduate School
Monterey, California

Ryad B. Benosman
University of Pittsburgh
Pittsburgh, Pennsylvania

ABSTRACT

Neuromorphic computer architectures utilize an event-based paradigm to operate in low-power environments and achieve high-throughput with massive parallelism. The integrate-and-fire (IF) neuron model has become a greatest common denominator among many emerging architectures built from materials with neuron-like response properties. There is recent interest in using these architectures as general purpose computing devices, especially for problems that are computationally demanding such as constraint satisfaction, integer factorization, and numerical analysis. However, programming a spiking neural network (SNN) to perform these kinds of tasks remains a difficult problem. There is currently a lack of overarching principles to guide this kind of development, and specific solutions generally remain unportable between different architectures. We identify some of the challenges to using neuromorphic computers for general purpose computing. To address these challenges, we introduce four design principles that aim to facilitate good design on a SNN architecture. We then describe several patterns that solve recurring neuromorphic design challenges, including serial execution on a parallel architecture and the implementation of structured memory in a SNN.

KEYWORDS

neuromorphic computing, software design, time intervals

ACM Reference Format:
John V. Monaco and Ryad B. Benosman. 2020. General Purpose Computation with Spiking Neural Networks: Programming, Design Principles, and Patterns. In *Neuro-inspired Computational Elements Workshop (NICE '20), March 17–20, 2020, Heidelberg, Germany.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3381755.3398698

1 INTRODUCTION

Greater interaction between computational science and neuroscience promises to aid in the development of low-power brain-inspired architectures [17]. It is apparent that humans are capable of both pattern recognition and symbolic processing tasks, but it is not yet clear how to efficiently integrate the two on a single architecture. Spiking neural networks (SNN), are capable of universal computation [16]. This model exhibits properties of both digital computation, through spike event rates, and analog computation, through the precise timing between spikes [25]. SNNs have become a focal point for near-future neuromorphic architectures, as there exist a variety of materials shown to emulate the behavior of the spiking neuron. Yet, a hurdle remains in programming such devices to perform general purpose computing tasks, such as control flow and memory management.

There are predominantly two ways to perform general purpose computation with neural networks. The first is a top-down approach whereby the parameters of a neural network are modified through a training procedure (e.g., backpropagation) using input/output examples. The network itself might be trained to perform some specific task, such as sort a list [10], or execute a sequence of instructions [20]. This approach necessitates the existence of exemplars and relies on the ability to generalize through learning, e.g., the ability to solve symbolic processing tasks not seen during training.

The second approach to general purpose computing is bottom-up, whereby the network is gradually constructed from primitive components until the desired behavior is achieved. This approach is akin to software development, in which an algorithm represented in a high-level programming language is translated into an equivalent neural network. Parts of this process may be automated, such as through compilation in order to abstract low-level architectural details away from the programmer [22, 23].

This work is concerned with the bottom-up approach to programming a spiking neural network. This process is analogous to writing software and shares some of the same goals as software engineering: a good SNN design should be maintainable, extensible, and consistent. However, there are currently a lack of design principles to facilitate good design when programming a SNN. In addition, using a SNN for general purpose computing carries challenges that are inherent to programming parallel architectures, notably synchronization and the ability to enforce serial execution when one operation takes precedence over another.

In this paper, we highlight these challenges and propose four overarching design principles that may facilitate future software development on neuromorphic architectures. These include principles of encapsulation, composition, input precedence, and spike parity. We then describe several patterns that solve recurring problems in SNN design, such as ordered execution and memory management.

2 BACKGROUND

2.1 Neuron Model

In this work, we use a simplified version of the neuron model in Lagorce and Benosman [15] shown in Figure 1. Spikes can be transmitted along three different synapse types: V-synapses modify the membrane potential directly by w_e, while g_e-synapses and gated g_f-synapses modify the membrane potential respectively

NICE '20, March 17–20, 2020, Heidelberg, Germany

ACM ISBN 978-1-4503-7718-8/20/03…$15.00
https://doi.org/10.1145/3381755.3398698

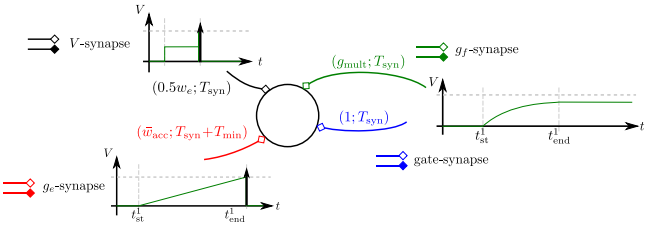

Figure 1: Neuron model with multiple synaptic dynamic kernels as introduced in Lagorce and Benosman [15] .

with linear and exponential kernels. Neurons emit a spike event when the membrane potential reaches a predefined threshold V_t, which is the same for all neurons. The internal variable g_e can be viewed as a modifiable leak that adjusts according to spikes transmitted over g_e-synapses. Each synapse is configured with a propagation delay, which is the time it takes a spike event to reach the postsynaptic neuron. By default, this is $T_{\text{syn}} = 1\,\text{ms}$, and we avoid synaptic connections less than T_{syn}. This model is compatible with the IF model [1], as described in Section 5.2, thus remains portable across a variety of neuromorphic architectures.

We simulate the above neuron model using Brian [9] with Euler integration and time step $dt = T_{\text{neu}} = 1\,\mu\text{s}$ where T_{neu} is the time it takes for a neuron to emit a spike. The networks in this paper utilize only V- and g_e-synapses with weights chosen from a small set of different values. This approach is amenable to neuromorphic architectures in which synapse weights are either shared or limited in resolution [3]. Generally, synapse weights are proportional to the magnitude required to either invoke or suppress a spike. For convenience, we define two V-synapse weights: excitatory weight $w_e = V_t$ and inhibitory weight $w_i = -V_t$. The weight w_e is needed to induce a postsynaptic spike from the resting state. Additionally, let $w_{\text{acc}} = \frac{\tau_m V_t}{T_{\text{max}}}$ be the weight needed for a spike along a g_e-synapse to induce a spike after time T_{max}, starting from a resting state.

2.2 Value Representation

Values are encoded as the precise time interval between two spikes, and neural networks are constructed to operate on the temporally encoded values. The encoding function $f(\cdot)$ and decoding function $f^{-1}(\cdot)$ are given by

$$\Delta t = f(x) = T_{\text{min}} + xT_{\text{cod}} \quad (1)$$

$$x = f^{-1}(\Delta t) = \frac{\Delta t - T_{\text{min}}}{T_{\text{cod}}} \quad (2)$$

where Δt is a time interval between two spikes, $x \in [0, 1]$, T_{min} is a minimum interval necessary for encoding 0, and T_{cod} is the range over which values can be encoded.

The maximum number of unique values that can be represented over the finite interval $T_{\text{max}} = T_{\text{min}} + T_{\text{cod}}$ is $\frac{T_{\text{cod}}}{dt}$, limited by the time step dt. Reasoning is as follows: consider $\Delta t_1 = f(x_1)$ and $\Delta t_2 = f(x_2)$ with $|\Delta t_1 - \Delta t_2| < dt$; then Δt_1 and Δt_2 are equal to exactly the same number of time steps in the simulator.

2.3 Design Challenges

We motivate the need for SNN programming design principles and patterns by identifying two different challenges: the first due to inherent differences between neuromorphic architectures and the von Neumann architecture, and the second due to neuromorphic architectural diversity.

While much effort has focused on the use of neuromorphic computers as deep neural network accelerators [5, 7], the same devices have been proposed as platforms for high performance computing (HPC) [2, 12] and shown to be capable of both speeding up and reducing energy requirements of non-machine learning tasks [21, 26]. This requires the ability to manipulate symbols, notably: storing and retrieving values from memory, binding variables, mechanisms for control flow, and evaluating mathematical functions. Within a SNN, these capabilities may be programmed in a way analogous to the way conventional software is written [15]. However, the word *programming* is somewhat of a misnomer. By definition, an algorithm is a *sequence of instructions*, and programming largely involves the implementation of algorithms. In contrast, all parts of a SNN execute simultaneously and there is no innate ordering of instructions at the hardware level. Thus, the implementation of sequential algorithms on massively parallel machines presents one of the main challenges to programming SNNs. We introduce some patterns that provide general mechanisms for control flow within a SNN to help address this challenge.

Emerging neuromorphic architectures (e.g., [3, 4, 8], see [13] for a survey) are diverse in terms of neuron model (e.g., dynamics, refractory period), topological constraints (e.g., maximum fan-in/fan-out), noise, tolerance to component failure, and parameter constraints (e.g., binary-valued synapse weights). This diversity presents a challenge for development. Solutions tend to remain platform-specific, adapting to the constraints of a particular architecture. Because of this diversity there is also a lack of tools to facilitate development, such automated compilation, static error checking, and debugging environments. We aim to address this challenge by describing principles that apply broadly to SNN programming and patterns that are compatible with the IF neuron model which can be simulated on a range of architectures.

3 DESIGN PRINCIPLES

A *design principle* is an overarching concept meant to facilitate good design [18]. In software engineering, design principles aim to mitigate the symptoms that plague bad design, including rigidity (code that is difficult to modify), fragility (code that breaks when modified), immobility (code that is not reusable), and viscosity (designs that are difficult to preserve). Principles such as *single responsibility* and *interface segregation* form the basis of object-oriented (OO) software design.

The development of OO software design was biologically inspired, taking into account the way cells communicate by passing messages [19]. In this light, some software engineering principles naturally extend to SNN design. For example, the single responsibility principle states that a class should encapsulate a behavior or responsibility within a larger program. Applied to SNN development, networks may be constructed such that each network

encapsulates a function or behavior. This would likely make it easier to reason about a larger network, as individual sub-networks could be tested and debugged in isolation. However not all OO design principles can be applied to SNN design. For example, polymorphism, which provisions a single interface to different types, has no direct application within a SNN due to the lack of a type system. It is therefore necessary to identify foundational principles that guide SNN design.

We describe four principles, some inspired by OO design, that aim to facilitate SNN designs that are understandable and maintainable. These include:

Principle of encapsulation: neurons should form networks that implement a single behavior, publicly exposing only a few input and output neurons that define the network's application programming interface (API).

Principle of composition: communication between encapsulated networks should occur only through API neurons. Complex behavior is achieved by connecting networks together, leading to a hierarchical structure.

Principle of input precedence: each network implements a state machine in which only a subset of inputs are active at any given time. Neurons must know how this machine transitions between states in order to properly use the API.

Principle of spike parity: temporally encoded values traverse a network in spike pairs, and networks largely transition between states depending on the number of spikes received; therefore neurons must know how many spikes to send.

In the rest of this section, we describe these design principles in detail and provide examples of their use.

3.1 Principle of Encapsulation

A challenge in SNN programming is scaling up networks to handle increasingly complex and diverse tasks. As networks grow in size, so too does the difficulty in reasoning about, modifying, and debugging networks. This motivates the *principle of encapsulation*, which states that each network should implement only a single behavior or function and expose this behavior through designated input and output neurons that form the network's API. A network should communicate with other neurons only through it's API, internalizing the structure that implements a particular behavior. Encapsulation is much like that in OO design, which specifies that a class should encapsulate a single responsibility.

Forcing network communication through an API creates a bottleneck in terms of programming effort. Once a given behavior has been implemented, debugged, and encapsulated, it can be reused in larger networks. Developers can focus on connecting interfaces and building hierarchically without having to reason about each network's behavior. In this way, encapsulation enables composition (described in the next section), and these two principles go hand-in-hand.

There are primarily four different kinds of networks depending on the role of the network and encapsulated behavior:

Functional networks encapsulate primitive functions, such as logic, arithmetic, and comparison functions. These networks are primarily stateless, in that they should return to some resting state after being used.

Connecting networks form the glue between various parts of a larger SNN. These include converters, synchronizers, and spatiotemporal mappings. For example, interfacing with a digital computer might require converting temporally encoded values to a binary spike pattern.

Routing networks form the basis of control flow within a SNN. These include networks to perform branching, condition checking, and other ways of selectively activating different parts of a network. Routing networks largely control the flow of information through a network.

Memory networks act as registers and form the basis of symbol manipulation within a SNN. These include networks that store primitive scalar values, such as volatile and persistent memory, and complex data structures built from primitives, such as lists and sets.

As an example of encapsulation, and to introduce the notation used in subsequent figures, Figure 2 (left) shows a network that encapsulates volatile memory. The API of this network includes two input neurons, *store* and *recall*, and one output neuron, *output*. In Figure 2, input neurons are blue, output neurons are red, and synapse type (V vs g_e) and weight are given by line style and color. Spike propagation delays are denoted by small circles along each synapse: ●=T_{syn} and ○=T_{neu}.

A scalar value is stored by sending two spikes to *store* such that the time interval between spikes encodes the value to be stored. The first spike to *store* causes the potential of *acc* to decrease from just below the neuron's spiking threshold, and the second spike stops this decrease. The stored value can later be recalled with a single spike to *recall*, which causes the potential of *acc* to increase until the threshold is reached, after which it spikes. The result is *output* emits two spikes separated by exactly the same time interval that was stored. The volatile memory network is re-writable, in that storing a value will clear any value that was previously stored. However, the network is volatile because a stored value can be recalled only once, after which the network returns to a resting state. A network that implements persistent memory is described in Section 4.2.

3.2 Principle of Composition

One of the central goals of OO programming is to enable scalable designs by building complex programs from simple pieces. This approach was biologically inspired, first described by Alan Kay who noted that individual cells perform relatively simple and atomic operations yet work together to achieve complex behavior [19].

Like OO design, programming SNN networks should focus on creating small, self-contained modules and then connecting these together to achieve something more complex. The *principle of composition* states that encapsulated networks should be connected to form larger networks, which may then again encapsulate some behavior. Complex behavior is achieved through recursive applications of composition, and this approach leads to networks with a hierarchical structure.

Figure 2 (right) demonstrates this principle. Consider a network that stores a 2-tuple, an ordered data structure containing two scalar values. Rather than building this network from individual neurons, the volatile memory network is reused shown by networks

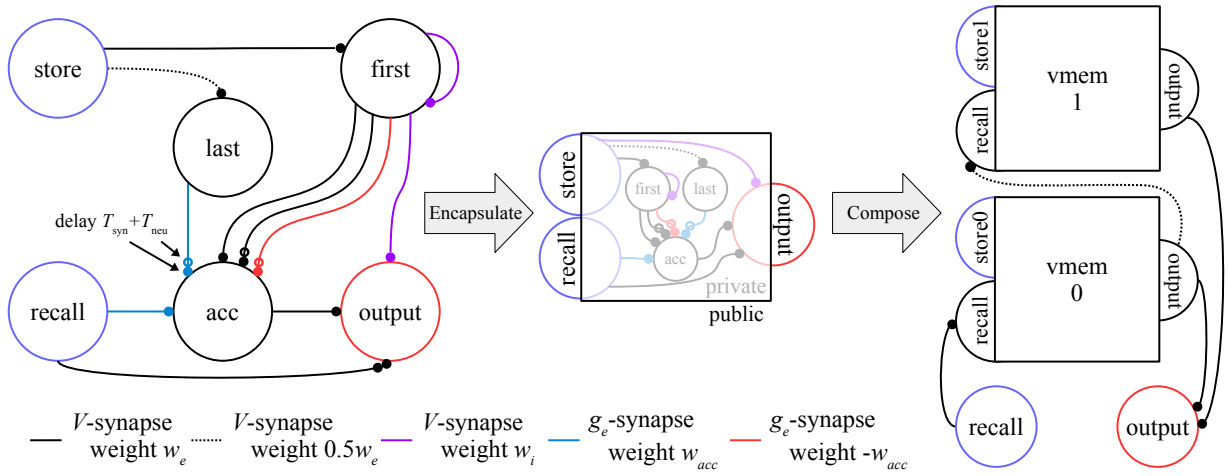

Figure 2: Principles of encapsulation and composition using the volatile memory network as an example.

vmem0 and *vmem1*. The network structure focuses on the ordering of values, abstracting away the required memory components. The 2-tuple network contains three input neurons, *store0*, *store1*, and *recall*. Values are stored in each of the volatile memory networks and later recalled in serial with a single spike to *recall*. When values are recalled, *output* will emit 4 spikes: first, the pair of spikes encoding the value in *vmem0* followed by the pair of spikes from *vmem1*.

3.3 Principle of Input Precedence

On a von Neumann architecture, a global clock is used to coordinate the activity of underlying circuits. Synchronous logic enables complex circuits to be built from smaller pieces that are easier to reason about and debug, compared to asynchronous designs which must take into account the stochastic nature of signal propagation. Like asynchronous circuits, spiking neural networks lack a global clock. There is no built in mechanism to synchronize the computations performed by individual neurons, and as a result, networks undergo state transitions as inputs are received or internal changes take effect. The network may be either waiting for input or performing a computation, and in some states it may expect *not* to receive any spikes. Presynaptic neurons that communicate through an API must know to transmit spikes at the appropriate time. This notion is codified by the *input exclusion principle*, which specifies that each network can be summarized by a state machine, and other neurons must have knowledge of this state machine in order to properly communicate through an API.

For example, consider the volatile memory network in Figure 2 (left). This network begins in a resting state in which it expects some value to be stored through spikes to *store*. After the first spike to *store*, another spike to *store* is expected and there must be no spikes to *recall*. Recalling a value before any value is stored, or during a store, can put the network into an undefined state.

This behavior and state descriptions of the volatile memory network are shown in Figure 3 and Table 1, respectively. Figure 3 shows a spike raster plot of the network as a value is stored and then recalled. In state 0, the network waits for a value to be stored; thus only *store* should spike. The first spike to *store* causes the network to transition to state 1, during which it expects another spike to *store*. After the value is stored, the network transitions to state 2 in which that value can be recalled or another value can be stored, which would overwrite the existing value. A spike to *recall* transitions the network to state 3, during which it is expected that neither *store* or *recall* will spike. Finally, the network transitions back to state 0. Attempting to recall a value in state 0 could put the network into an undefined state, as could attempting to recall or store a value while a value is being recalled (state 3).

3.4 Principle of Spike Parity

A temporal encoding implies that values propagate through a network in spike pairs. However, these pairs may not necessarily be unique. For example, consider encoding two different values x_1 and x_2 with $x_1 \neq x_2$. Let $\Delta t_1 = f(x_1)$ and $\Delta t_2 = f(x_2)$. These two intervals may be represented using four spikes at times t_1, t_2, t_3, t_4 where $\Delta t_1 = t_2 - t_1$ and $\Delta t_2 = t_4 - t_3$. This way of transmitting multiple values in *serial* enables each value to be processed in isolation from the other. But if these two values are always transmitted together, a more compact representation would be to let Δt_1 share it's 2nd spike with the first spike of Δt_2. In other words, let $\Delta t_1 = t_2 - t_1$ and $\Delta t_2 = t_3 - t_2$. In this way, the values are *chained* together and can be processed one after the other. Only 3 spikes are needed instead of 4, and both values occupy time $t_3 - t_1$ instead of $t_4 - t_1$, where $t_1 < t_2 < t_3 < t_4$. An even more compact representation is to *superimpose* the two values on top of each other. That is, let both Δt_1 and Δt_2 share their first spike such that $\Delta t_1 = t_2 - t_1$ and $\Delta t_2 = t_3 - t_1$ where $\Delta t_1 < \Delta t_2$.

Besides scalar values, neurons may emit a spike simply to indicate that a network has reached a given state (or to transition a network into a given state). For example, the *store* input neuron of the volatile memory network expects two spikes that encode a scalar value while the *recall* neuron expects only a single spike to transition the network into a "recall" state. The single spike to *recall* can be viewed as a *point* event, or boolean value that encodes a 1 where the absence of a spike encodes a 0.

Table 1: Volatile memory network state descriptions.

State	Description	Allowed Inputs
0	Waiting for input	store
1	Storing value	store
2	Waiting for recall	store, recall
3	Recalling value	

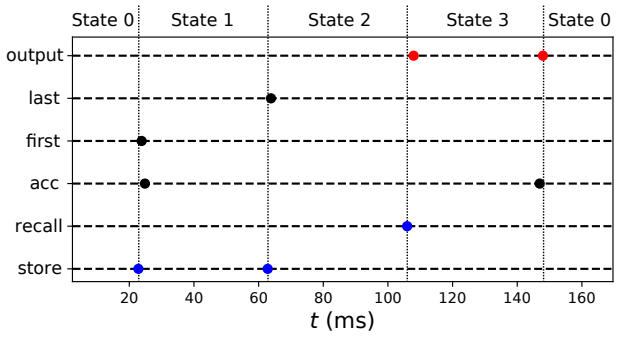

Figure 3: Principle of input precedence. The volatile memory network undergoes state transitions as a value is stored and recalled. States are labeled above the spike raster plot; refer to Table 1 for state descriptions.

The *parity* of a neuron specifies how many spikes it expects to emit and in what format. There are primarily four different ways to represent values that differ in terms of parity. These are shown in Figure 4 and summarized below.

Point: A single spike acts as a flag and encodes a boolean value where 1 is the presence of a spike and 0 is the absence. In this format the intervals between spikes have no meaning.

Serial: A scalar value is encoded by a unique pair of spikes, and multiple encoded values are non-overlapping in time. In this format $2n$ spikes are needed to encode n values, and each value propagates individually in serial order.

Chained: A sequence of scalar values is encoded by sharing the 2nd spike of the nth encoded value with the 1st spike of the $n+1$th encoded value. In this format $n+1$ spikes encode n values, and the total time of the sequence is proportional to the sum of the encoded values.

Superimposed: A set of scalar values is encoded by sharing only the 1st spike of every encoded value. In this format $n+1$ spikes are needed to encode n values, and the total time to encode the set is proportional to the largest interval. The values must be unique and transmit in parallel.

Presynaptic neurons must know which format (point, serial, chained, superimposed) is expected in order to properly communicate with other neurons. Connecting a neuron that encodes values in serial to another neuron that expects chained input would likely result in the network reaching an undefined state. Thus, the *principle of spike parity* states that the format and parity of a presynaptic neuron should match that of a postsynaptic neuron.

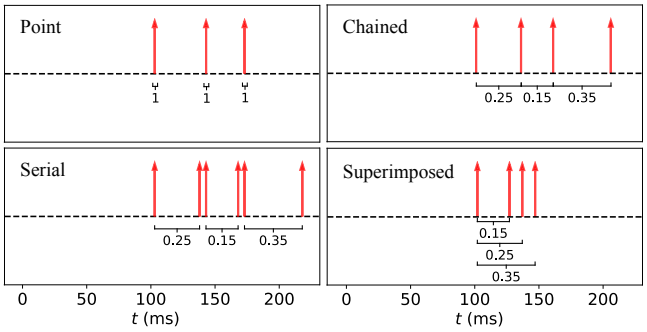

Figure 4: Principle of spike parity. Four different methods to temporally encode multiple values. Each method differs in both format and parity.

4 DESIGN PATTERNS

Whereas a design principle acts as a general guide to good design, a *design pattern* is a solution to a specific recurring problem. We describe patterns that solve two different kinds of problems: routing and memory. Figuring out how to "pass spikes around" a large network constitutes a major design challenge when there is an order of precedence that must be enforced between different parts of the network. We first focus on patterns for passing spikes around to different parts of a network. These include patterns for branching (splitting the path a spike takes) and gating (turning on and off the path a spike takes). Following this, we describe patterns to store and retrieve values from memory.

4.1 Routing Patterns

4.1.1 Branching. On a computer that performs serial execution (e.g., von Neumann and Harvard architectures), selectively executing different parts of a program is achieved by jumping the instruction pointer to different memory locations. Within a SNN, all parts of the network are executed simultaneously. Thus, to selectively activate different parts of the network at different times, a branching mechanism is needed.

The *branching pattern* enables splitting the path a spike follows to activate different parts of a network over time. An instantiation of this pattern is shown in Figure 5 (left), which contains a network with 1 input neuron and 2 output neurons. For spike events $e_0, \ldots, e_n$ from *input*, where e_n is the n+1th spike, the behavior of the branching network is to send even spikes $e_0, e_2, \ldots, e_{2k}$ to *first* and odd spikes $e_1, e_3, \ldots, e_{2k+1}$ to *last*. The V-synapse with weight w_e from *input* to *first* causes *first* to spike, followed by a spike along the inhibitory recurrent connection on *first* to prevent spiking on the next input. Likewise, the V-synapse with weight $0.5w_e$ from *input* to *last* induces spiking on every other input. The branching pattern is characterized by recurrent inhibition to the *first* neuron and delayed activation of the *last* neuron. This pattern is replicated in many of the networks to follow and represents one of the basic mechanisms of flow control in a parallel architecture.

4.1.2 Gating. Selectively enabling spikes to flow within a network is another mechanism by which serial execution can be achieved. In the *gating pattern*, spikes may propagate along a path depending

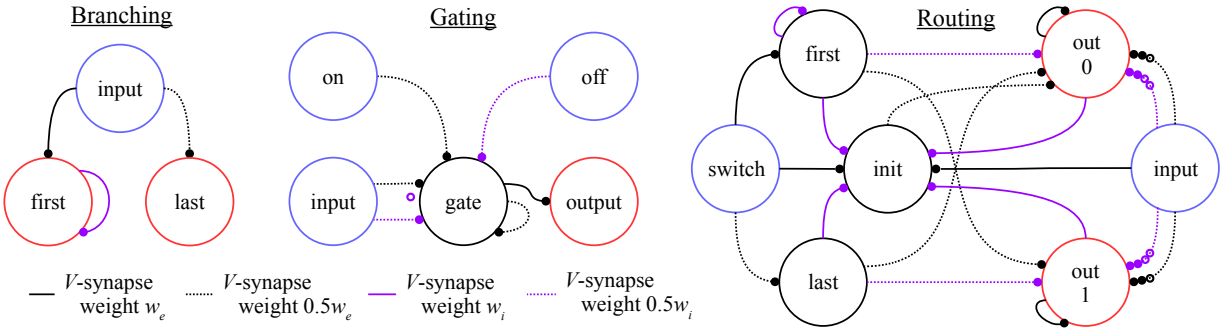

Figure 5: Branching (left), gating (center), and routing (right) networks.

on the state of the network. Shown in Figure 5 (center) is a gate network that selectively enables spikes to pass through from *input* to *output*. This behavior is achieved depending on the state of the *gate* neuron: if the membrane potential of *gate* is $0.5V_t$, i.e., in an *on* state, then a spike from *input* causes *gate* to spike, also causing *output* to spike. Otherwise, any input spikes are not propagated to the output. Whereas spike flow in the branching pattern depends on the number of spikes received, flow in the gating pattern depends on whether the gate is in an on/off state.

The *input* neuron, connected to *gate*, quickly excites and then inhibits *gate* over V-synapses with weights $0.5w_e$ and $0.5w_i$. The inhibitory connection from *input* is delayed by an extra T_{neu} to immediately suppress the effect of the excitatory connection.

The gate pattern is characterized by an excitatory/inhibitory synapse pair with the inhibitory synapse delayed by one time step. When *gate* is off, this ensures that any subsequent spikes to *input* will not cause any output spikes. But when *gate* is on, i.e., has membrane potential set to $0.5V_t$, the first excitatory connection will cause it to spike. The w_e-weighted recurrent connection to *gate* maintains this "on" state, pushing the potential back to $0.5V_t$ with the combined effect from the inhibitory *input* connection.

Incorporating both the branching pattern and gate pattern, the routing network shown in Figure 5 (right) enables switching between two different paths. In this network, spikes flow from *input* to either *out0* or *out1* depending on the state of the network. The network starts in a state that sends spikes to *out0*. After a single spike to *switch*, the network transitions to a state that sends *input* spikes to *out1*. Subsequent spikes to *switch* alternate the state of the network between emitting spikes at one of the two output neurons.

4.2 Memory Patterns

Temporally encoded values are transient by nature because a time interval expires after a finite duration. However, the ability to store and later recall a value from memory is a central component of general purpose computation. In a SNN, long term memory can be achieved by setting the membrane potential of a neuron proportional to some value. This value will remain serialized until it is later recalled, upon which it is converted from space (a physical quantity) back to time.

4.2.1 Scalars. The volatile memory network in Figure 2 (left) stores a value by setting the membrane potential of the *acc* neuron. This network enables recalling a value only once after it is stored. A network capable of long term storage and multiple retrievals would better facilitate general purpose computation within a SNN. Using the *flip-flop pattern*, we show that volatile memory in conjunction with routing can form the basis of a persistent memory store.

The flip-flop pattern is characterized by a bistable network capable of long-term information storage. Figure 6 shows a persistent memory network which is an instantiation of this pattern. Persistence is achieved by a composition of volatile memory networks and routing between the networks, much like a flip-flop circuit, passing the stored value between the two volatile memory networks as it is recalled. This network enables multiple consecutive recall operations, and like the volatile memory network, values can be overwritten after they are stored.

4.2.2 Vectors and Sets. The need to store and recall values from structured memory arises frequently in general purpose computation. We describe two different patterns that implement two commonly used data structures: a vector, which stores an ordered sequence of scalars, and a set, which stores an unordered collection of unique scalars. The implementation of more complex data structures (e.g., a hash table or tree) remains an item for future work.

The *sequential memory pattern* is characterized by a sequence of memory networks connected in serial. Shown in Figure 7 (top) is a sequential memory network encapsulated by an API that enables the caller to store a value to each memory location and recall the values in serial through the *output* neuron. Following the principle of spike parity, this network outputs values in serial format. A spike to *recall* causes the memory at location 0 to output its value, which then recalls the memory at location 1, and so on, such that the values in memory appear in serial at the output.

A central principle of encoding values as time is that multiple values can be *superimposed*, whereby different values are projected onto the same time interval. A single spike marks the beginning of a set of superimposed values, and each subsequent spike marks the end of an interval, i.e., a single value. Superimposition provides a compact representation for multiple values, requiring only $N + 1$ spikes to represent N values and occupying a time interval proportional to the largest value.

Encompassing this principle is the *superimposed memory pattern*, shown in Figure 7 (bottom). Like sequential memory, values can be stored to different locations; however, a recall causes every value to be recalled in parallel and piped to a single output, resulting in the

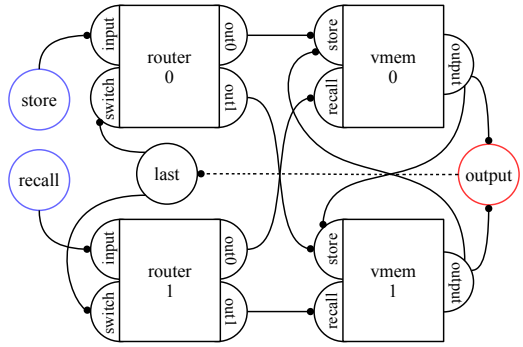

Figure 6: Persistent memory network. A stored value is routed to either *vmem0* or *vmem1* upon recall to achieve persistence, much like a flip-flop circuit.

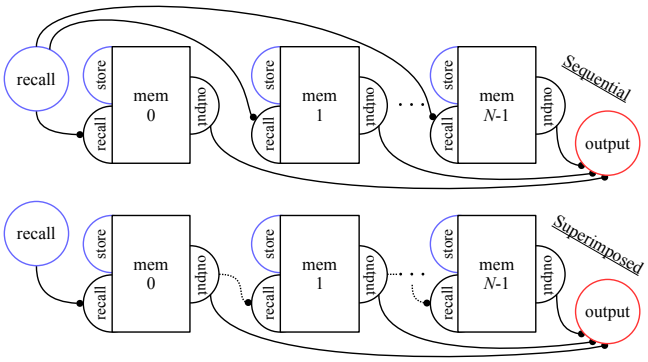

Figure 7: Sequential (top) and superimposed (bottom) memory networks implement vectors and sets, respectively.

superimposition of all values in memory. Because the values are superimposed, two values of equal magnitude in different memory locations will appear as a single value at the output neuron. In this regard, superimposed memory can indicate only set membership and does not disclose the memory location of each value.

5 DISCUSSION

5.1 Related Work

General purpose computation with neural networks (not necessarily spiking) has generally taken two different approaches: top-down and bottom-up. In a top-down approach, network parameters are configured inductively by learning to generalize from a set of exemplars. This approach largely consists of deep neural networks learned through backpropagation.

The neural Turing machine (NTM) [10] is a fully-differentiable network augmented with external memory and attentional process, resembling a von Neumann architecture. In the NTM, a controller that interacts with the world through external inputs and outputs is able to read from and write to the external memory. This model was later extended to the differentiable neural computer (DNC) which addressed some deficiencies of the NTM, specifically the abilities to discover unused memory locations, free previously used memory locations, and better preserve temporal linkage.

In a bottom-up approach, a neural network is configured deductively or derived from a set of axioms. This approach is akin to software development, as the programmer must construct the network by hand or using tools based on deductive principles, such as compilation. The NEural Language (NEL) provides a framework to deterministically configure a neural network that implements a procedure specified in a high-level programming language [22, 23]. Neuron activations encode scalar and boolean values as well as lists and stacks, for which a set of primitive functions are used to read and write to. In the same spirit, JaNNeT (Just an Automatic Neural Network Translator) compiles networks using cellular encoding as an intermediary representation, requiring four different transfer functions and asynchronous global dynamics [11].

Unlike NEL and JaNNeT that require precise neuron activation, vector symbolic architectures (VSAs) encompass a class of models in which symbols are mapped to high-dimensional vectors in a space that provides binding and superposition operations [14]. These operations enable pointer manipulation and form the basis of symbol processing. In the neural engineering framework (NEF), vectors are encoded by spike rate based on neuron tuning curves, and functions are implemented through transformations of encoded vectors between populations of neurons [6]. This approach enables the representation of algorithms atop biologically plausible models, such as with leaky integrate-and-fire (LIF) neurons [24].

5.2 Portability Across Architectures

Although major neuromorphic architectures (e.g., TrueNorth, Loihi, SpiNNaker) differ in terms of architectural constraints and neuron dynamics, most are capable of simulating an IF model. Thus, the IF neuron model is a greatest common denominator across architectures and may serve as a kind of standardized instruction set. Designing networks that are compatible with an IF model will maximize portability across architectures.

In this work, g_e-synapses form the backbone of temporal encoding and many patterns described. While V-synapses are compatible with an IF neuron model, g_e-synapses are not. However, it is possible to simulate g_e-synapses within a IF model using only a small number of V-synapses.

Figure 8 (right) shows a network that simulates the behavior of a g_e-synapse (left) using only V-synapses. The g_e-synapse connecting neurons *pre* and *post* is replaced with auxiliary neurons *aux0* through *auxN-1* and V-synapses. When *pre* spikes, it induces tonic spiking behavior in *aux0*, which continues to spike until *post* spikes. The tonic spiking of *aux0*, connected to post with weight $w_{\text{aux}} = V_t/(T_{\max}/T_{\text{syn}})$, mimics the integration of the linear variable g_e. This occurs until *post* spikes, when an inhibitory connection causes all *aux* neurons to stop spiking as if g_e were reset back to 0. An example of this behavior that mimics a single spike along a g_e-synapse is shown in Figure 9.

In cases where a higher resolution is needed, i.e., several spikes over a g_e-synapse are expected, several *aux* neurons can be placed between *pre* and *post* with staggered activation upon successive spikes to *pre*. The weights to each *aux* neuron follow a harmonic progression to achieve a stepwise activation, where *aux0* starts tonic spiking with the first spike from *pre*, *aux1* starts tonic spiking

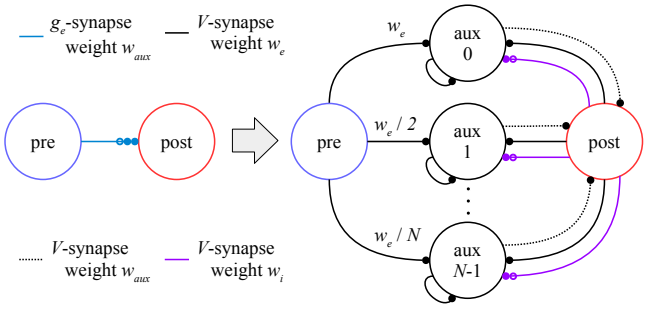

Figure 8: The g_e-simulation network (right) mimics the behavior of a single g_e synapse (left) using only V-synapses.

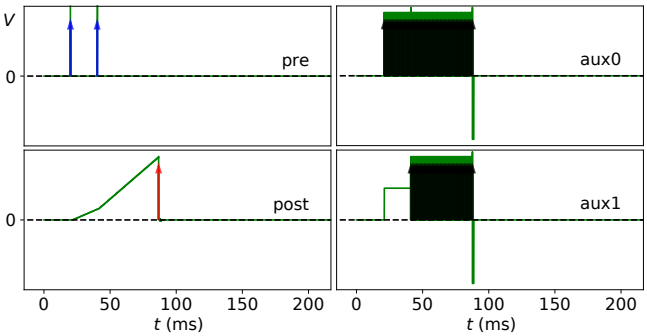

Figure 9: Chronogram of the g_e-simulation network.

with the second spike from *pre*, and so on. This form of control flow is a generalization of the branching pattern.

Note that since an auxiliary neuron is placed between pre and post, only g_e-synapses with at least delay $2T_{\text{syn}} + T_{\text{neu}}$ can be simulated. Connections with delay T_{syn} (which is the default) must first be scaled up to $2T_{\text{syn}} + T_{\text{neu}}$ to obtain a functionally equivalent network. This can be achieved by globally scaling time by $T_{\text{syn}} \equiv 2T_{\text{syn}} + T_{\text{neu}}$. In most cases however, simply increasing the delay of the g_e-synapse alone achieves the desired effect.

6 CONCLUSION

General purpose computation with spiking neural networks requires a paradigm shift from conventional software design as well as addressing practical challenges. We introduced four principles that aim to facilitate good design of SNNs. These principles are inspired by a mix of biological principles and OO software design: encapsulation and composition work in concert to facilitate the building of complex networks from simple pieces; the principle of input precedence specifies that neurons must have temporal knowledge of a network's API as it undergoes state transitions; and the principle of spike parity describes four different ways of encoding values and that spike parity must match between input and output neurons from different networks. In addition, we described several patterns that solve recurring design problems that are specific to SNNs, notably the lack of native control flow and memory structures. Two primary ways in which the routing of spikes can enforce serial execution are through branching and gating. The storage and retrieval of multiple values to memory can be accomplished with persistent memory networks that are connected in either serial or parallel to form vectors and sets, respectively. Together these patterns provide a means of manipulating and passing around symbols in a way that is space efficient compared to learning approaches.

REFERENCES

[1] L.F Abbott. 1999. Lapicque's introduction of the integrate-and-fire model neuron (1907). *Brain Research Bulletin* 50, 5-6 (nov 1999), 303–304.

[2] James B. Aimone. 2019. Neural algorithms and computing beyond Moore's law. *Commun. ACM* 62, 4 (mar 2019), 110–110.

[3] Andrew S Cassidy, Paul Merolla, John V Arthur, Steve K Esser, Bryan Jackson, Rodrigo Alvarez-Icaza, Pallab Datta, Jun Sawada, Theodore M Wong, Vitaly Feldman, et al. 2013. Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*. IEEE, 1–10.

[4] Mike Davies et al. 2018. Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro* 38, 1 (jan 2018), 82–99.

[5] Peter U. Diehl, Guido Zarrella, Andrew Cassidy, Bruno U. Pedroni, and Emre Neftci. 2016. Conversion of artificial recurrent neural networks to spiking neural networks for low-power neuromorphic hardware. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*. IEEE.

[6] Chris Eliasmith and Charles H Anderson. 2004. *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. MIT press.

[7] Steven K. Esser et al. 2016. Convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Sciences* 113, 41 (sep 2016), 11441–11446.

[8] Steve B. Furber, David R. Lester, Luis A. Plana, Jim D. Garside, Eustace Painkras, Steve Temple, and Andrew D. Brown. 2013. Overview of the SpiNNaker System Architecture. *IEEE Trans. Comput.* 62, 12 (dec 2013), 2454–2467.

[9] Dan FM Goodman and Romain Brette. 2008. Brian: a simulator for spiking neural networks in Python. *Frontiers in neuroinformatics* 2 (2008), 5.

[10] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural turing machines. *arXiv preprint arXiv:1410.5401* (2014).

[11] Frédéric Gruau, Jean-Yves Ratajszczak, and Gilles Wiber. 1995. A neural compiler. *Theoretical Computer Science* 141, 1-2 (1995), 1–52.

[12] Kathleen E. Hamilton, Catherine D. Schuman, Steven R. Young, Ryan S. Bennink, Neena Imam, and Travis S. Humble. 2020. Accelerating Scientific Computing in the Post-Moore's Era. *ACM Trans. on Parallel Computing* 7, 1 (apr 2020), 1–31.

[13] Conrad D. James, James B. Aimone, Nadine E. Miner, Craig M. Vineyard, Fredrick H. Rothganger, Kristofor D. Carlson, Samuel A. Mulder, Timothy J. Draelos, Aleksandra Faust, Matthew J. Marinella, John H. Naegle, and Steven J. Plimpton. 2017. A historical survey of algorithms and hardware architectures for neural-inspired and neuromorphic computing applications. *Biologically Inspired Cognitive Architectures* 19 (jan 2017), 49–64.

[14] Pentti Kanerva. 2009. Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors. *Cognitive Computation* 1, 2 (jan 2009), 139–159. https://doi.org/10.1007/s12559-009-9009-8

[15] Xavier Lagorce and Ryad Benosman. 2015. Stick: Spike time interval computational kernel, a framework for general purpose computation using neurons, precise timing, delays, and synchrony. *Neural computation* (2015).

[16] Wolfgang Maass. 1996. Lower bounds for the computational power of networks of spiking neurons. *Neural computation* 8, 1 (1996), 1–40.

[17] Wolfgang Maass, Christos H. Papadimitriou, Santosh Vempala, and Robert Legenstein. 2019. Brain Computation: A Computer Science Perspective. In *Lecture Notes in Computer Science*. Springer International Publishing, 184–199.

[18] Robert C Martin. 2000. Design principles and design patterns. *Object Mentor* 1, 34 (2000), 597.

[19] Stefan L Ram et al. 2003. Dr. Alan Kay on the Meaning of" Object-Oriented Programming". (2003).

[20] Scott Reed and Nando de Freitas. 2016. Neural Programmer-Interpreters. In *International Conference on Learning Representations (ICLR)*.

[21] Catherine D. Schuman, Kathleen Hamilton, Tiffany Mintz, Md Musabbir Adnan, Bon Woong Ku, Sung-Kyu Lim, and Garrett S. Rose. 2019. Shortest Path and Neighborhood Subgraph Extraction on a Spiking Memristive Neuromorphic Implementation. In *Proceedings of the 7th Annual Neuro-inspired Computational Elements Workshop on - NICE '19*. ACM Press.

[22] Hava T Siegelmann. 1994. Neural programming language. In *AAAI*. 877–882.

[23] Hava T Siegelmann. 1996. On nil: The software constructor of neural networks. *Parallel Processing Letters* 6, 04 (1996), 575–582.

[24] Terrence Stewart and Chris Eliasmith. 2011. Neural cognitive modelling: A biologically constrained spiking neuron model of the Tower of Hanoi task. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, Vol. 33.

[25] Frédéric Theunissen and John P Miller. 1995. Temporal encoding in nervous systems: a rigorous definition. *Journal of computational neuroscience* 2, 2 (1995), 149–162.

[26] Stephen J. Verzi, Fredrick Rothganger, Ojas D. Parekh, Tu-Thach Quach, Nadine E. Miner, Craig M. Vineyard, Conrad D. James, and James B. Aimone. 2018. Computing with Spikes: The Advantage of Fine-Grained Timing. *Neural Computation* 30, 10 (oct 2018), 2660–2690.