SNNL: A Programming Language for Spiking Neural Network Development

Qinghui Xing, *Zhejiang University, Hangzhou, 310058, China*
Zirun Li, *Xiamen University, Xiamen, 361005, China*
Ying Li, *Zhejiang University, Hangzhou, 310058, China*
Schahram Dustdar, *TU Wien, Vienna, 1040, Austria*
Xin Du, Gang Pan, and Shuiguang Deng, *Zhejiang University, Hangzhou, 310058, China*

Spiking neural networks (SNNs) are gaining attention for biological plausibility and energy efficiency. Advances in neuromorphic systems—integrating hardware and software tools—accelerate SNN implementation. Yet, deploying SNNs on such platforms remains challenging due to model complexity and system heterogeneity, requiring flexible frameworks. Existing tools (e.g., PyNN and Brian2) show limited expressiveness for neuromorphic applications or poor cross-platform support. This article proposes SNNL, a flexible, domain-specific language for SNN development and deployment on neuromorphic hardware. SNNL decouples neuronal dynamics modeling from network topology specification: equation-based representations handle diverse neuron/synapse models, while hierarchical constructs define complex connectivity patterns. We present a Darwin3-targeted compiler with efficient code generation. Evaluations confirm that SNNL achieves precise neuronal dynamic descriptions and flexible network configurations. This work bridges algorithm-hardware gaps in neuromorphic computing by enhancing programmability. Experimental results have demonstrated the feasibility of SNNL in developing SNNs for neuromorphic systems.

Spiking neural networks (SNNs), 3G neural networks, are characterized by biologically plausible architectures and energy-efficient computation through precise modeling of neuronal/synaptic dynamics. As hierarchical directed graphs (Figure 1), they inherently process spatiotemporal information via neurons interconnected via directed synapses, with node/edge states evolving over time according to predefined dynamics in response to input spikes.[1,2]

Digital Object Identifier 10.1109/MIS.2025.3636128
Date of publication 25 November 2025; date of current version 29 May 2026.

OVERVIEW OF NEUROMORPHIC SYSTEMS

Recent advancements in neuromorphic systems have yielded prominent architectures such as SpiNNaker,[3,4] Loihi,[5,6] TrueNorth,[7] BrainScaleS,[8] and Darwin3.[9] These systems typically adopt homogeneous many-core architectures with 2-D mesh topologies [Figure 1(d)], where distributed cores simulate neuronal dynamics and synaptic interactions. SpiNNaker employs ARM968 cores programmed via C-based toolchains,[3] while Darwin3 introduces a domain-specific instruction set architecture (ISA) optimized for spiking neural computation.[9] This ISA integrates dedicated registers for neuronal state variables and hybrid instructions, merging frequent spike-processing operations, reducing memory footprint, and decoding latency.

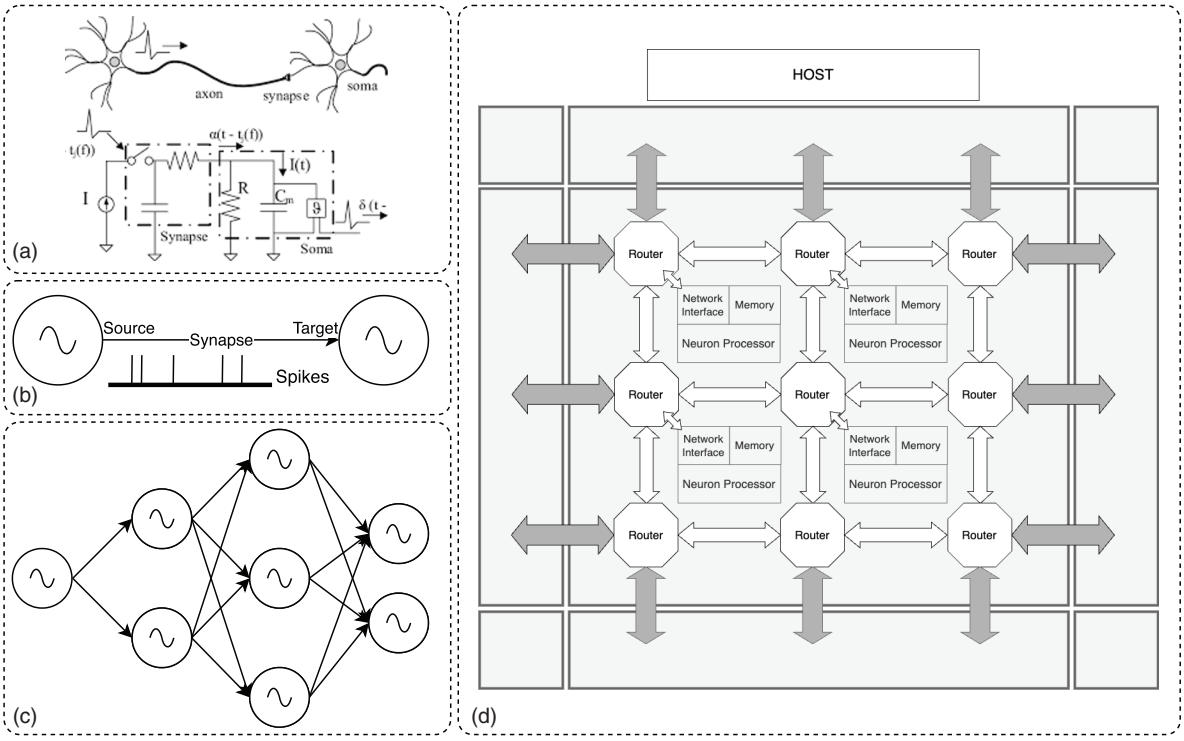

FIGURE 1. (a) Leaky integrate-and-fire neuron. (b) Spike trains are transmitted between neurons. (c) An SNN. (d) Architecture of neuromorphic hardware.

Complementing these specialized instructions, general-purpose arithmetic/logic operations enhance expressiveness for complex dynamics modeling. Applications in diverse domains have employed neuromorphic hardware as back-end support.[10,11,12,13]

Most digital neuromorphic platforms (e.g., SpiNNaker, Loihi, and Darwin3) implement discrete time-step execution. During each interval, cores perform event-driven spike processing: accumulating presynaptic inputs, updating neuronal states based on dynamic equations, and transmitting spikes to target neurons via network-on-chip (NoC) infrastructure. NoC architecture enables scalable intercore communication with reduced latency and improved bandwidth compared to traditional bus/crossbar implementations.

System-level orchestration involves embedded general-purpose processors that manage chip configuration, host interfacing, and data flow control. Host-side runtime environments (e.g., Darwin-S)[14] provide deployment pipelines for SNN applications, handling resource allocation, task scheduling, and hardware–software coordination. These frameworks incorporate runtime libraries with debugging interfaces and data-extraction utilities to support iterative development cycles. Such hierarchical architectures demonstrate the critical balance between specialized neuromorphic compute fabrics and the flexible system management infrastructures required for practical deployment.

SYNTAX OF SNNL

In this section, we present the modeling of SNNs, which is introduced in two independent modules: neural dynamics and network topology (see Figure 2). We first analyze the neural dynamics of SNNs by drawing inspiration from Brian2's definitions for modeling neuronal and synaptic dynamics, specifically tracking the state variables of neurons and synapses over time. Based on this, we employ differential equations as the foundational method for neural dynamics modeling and, in combination with the event-driven characteristics of SNNs, design a set of syntax for modeling the most critical computational units in SNNs: neurons and synapses. Regarding network topology modeling, we conceptualize the SNNs as a weighted directed acyclic graph, where nodes represent neurons that contain dynamic-state information, and edges denote synapses with both weights and dynamic properties. In terms of

network topology design, we focus on neuron assemblies, connectivity clusters between assemblies (referred to as *projections*), and the hierarchical structure of neural network layers. We have developed a highly flexible and user-friendly syntax for constructing network topologies, enabling users to conveniently, quickly, and flexibly model the spatial structure of neuron assemblies, connectivity structure of projections, and other necessary components for network construction.

Neural Dynamics

Neurons and synapses are the fundamental computational units in SNNs, sharing a common computational model. Each computational unit possesses several private-state variables, whose dynamics are influenced by two factors: continuous temporal evolution and discrete changes induced by event signals. Each unit can receive input signals from other units, perform complex computations to update its own state variables, and generate output signals that are transmitted to other units as inputs.

Neurons typically feature a crucial state variable—the membrane potential—that governs the generation of spike signals. When the membrane potential exceeds a certain threshold, the neuron emits a spike signal, transmitting it via synapses to downstream connected neurons. Simultaneously, the membrane potential is reset according to specific rules (e.g., reset to zero or by subtraction) and is incapable of generating another spike signal during a refractory period. Upon receiving spike signals from upstream neurons, synapses undergo a change of state variables, producing a current signal based on its computational rules and transmitting it to downstream neurons. Synapses maintain connection weights between pairs of presynaptic and postsynaptic neurons. In some synaptic plasticity learning rules, synaptic weights are adjusted based on the temporal differences between the spike times of presynaptic and postsynaptic neurons.

As shown in Figure 3, SNNL supports the modeling of neuron dynamics, including state variables, parameters, update rules, thresholds, reset rules, and refractory periods.

The *neuron* keyword specifies the neuron dynamics definition module, where the user defines the name of the neuron model (e.g., lif). The neuron model is divided into several sections, each beginning with a different keyword, which defines different aspects or behaviors of the neuron dynamics model.

The *variables* keyword defines the state variables of the neuron model, such as membrane potential V; membrane conductances m, n, and h; and the activation and inactivation rates of ion channels (α_m, β_m, α_n, β_n, α_h, and β_h). All variables defined in the "variables" section are private to each neuron instance, meaning they are not shared between different neuron instances. The user must define and initialize all necessary state variables for the neuron model in this section.

```
equations_compound
    : UPDATERULES COLON equations_suite;
equations_suite
    : equation NEWLINE
    | NEWLINE INDENT (equation NEWLINE)+ DEDENT;
equation
    : differential
    | assignment_stmt;
differential
    : NAME DIFFERENTIAL ASSIGN expression;

solvers_compound
    : SOLVER COLON solvers_suite;
solvers_suite
    : (x_new_assign_stmt | str) NEWLINE
    | NEWLINE INDENT (str NEWLINE | solver_stmts+) DEDENT;
solver_stmts
    : ( assignment_stmt NEWLINE )* solver_stmt NEWLINE;
solver_stmt
    : X_NEW ASSIGN expression;
```

FIGURE 2. Backus–Naur form of SNNL.

The *parameters* keyword defines the parameters of the neuron model, such as the conductance of ion channels G_k, G_{Na}, and G_L; reversal potentials E_k, E_{Na}, and E_L; and membrane capacitance C_m. All parameters defined in the "parameters" section are shared across all instances of the neuron model. The user must define and initialize all parameters required for the neuron model here.

The *updaterules* keyword defines the update rules for the state variables of the neuron model, including the rules for updating activation and inactivation rates (α_m, β_m, α_n, α_h, and β_h), the probability of ion channel activation and inactivation (m, n, h), and the membrane potential V. The user must specify the update rules in "updaterules" to define the dynamic behavior of the neuron model. The rules can utilize the state variables and parameters defined in the "variables" and "parameters" sections as well as temporary variables.

The *threshold* keyword defines the spike generation condition for the neuron model. When the condition is met, the neuron model emits a spike signal. The final expression in the "threshold" section is treated as the spike condition, which must be a Boolean or integer value representing the spike condition or the spike period.

```
# define excitatory leaky_integrate_and_fire neuron
neuron lif_ex {
    variables:
        v = -65 - 40
        ge
        gi
        theta = 20
        timer = 0

    parameters:
        v_rest_e = -65
        v_reset_e = -65
        v_thresh_e = -52
        refrac_e = 5
        tc_theta = 1e7
        theta_plus = 0.05

    updaterules:
        v' = ((v_rest_e - v) + (I_synE+I_synI)) / 100
        I_synE = ge * (0 - v)
        I_synI = gi * (-100 - v)
        ge' = -ge / 1
        gi' = -gi / 2
        theta' = -theta / tc_theta
        timer' = 100

    solver:
        "euler"

    threshold:
        v > (theta - 20 + v_thresh_e)

    reset:
        v = v_reset_e
        theta = theta + theta_plus
        timer = 0

    refractory:
        timer > refrac_e
}
```

(a)

```
# define stdp synapse
synapse stdp {
    variables:
        w
        pre_trace
        post_trace
    parameters:
        tau_pre = 20
        tau_post = 20
        nu = 0.1

    updaterules:
        pre_trace' = -pre_trace / tau_pre
        post_trace' = -post_trace / tau_post

    prespike:
        ge = ge + w
        gi = gi + w
        pre_trace += 1
        w -= nu * post_trace

    postspike:
        post_trace += 1
        w += nu * pre_trace

}
```

(b)

FIGURE 3. (a) Dynamics modeling of the leaky integrate-and-fire neuron. (b) Dynamics modeling of synapse with spike-timing-dependent plasticity (STDP) learning rules.

The *reset* keyword defines the reset rules for the state variables after a spike, such as the reset rule for the membrane potential *V*. Typically, the "reset" section contains operations to modify state variables to update them after a spike event.

The *refractory* keyword defines the refractory period, i.e., the absolute refractory period after a spike. The user must specify an integer value in "refractory" that represents the refractory period in time-steps, during which the neuron model will not generate spikes.

The synapse model is similar to the neuron dynamics model. As shown in Figure 3, SNNL supports synapse dynamics modeling, including synaptic-state variables, parameters, update rules, and the behavior of the synapse upon receiving a spike signal.

The *prespike* and *postspike* keywords define the update rules for the synaptic-state variables following a spike from the presynaptic and postsynaptic neurons, respectively. The update rules in "prespike" and "postspike" can utilize the state variables and parameters defined in the "variables" and "parameters" sections, respectively, as well as newly defined temporary variables.

Network Topology

An SNN can be regarded as a directed acyclic graph composed of neuron assemblies and projections: connection clusters between neuron assemblies. Neuron

assemblies consist of numerous neurons with identical dynamic models and possess shape information. Synaptic connections between neuron assemblies exhibit various patterns, such as convolutional, fully connected, one to one, and all to one.

In SNNL, the definition of neuron assemblies includes two parts: the neuron dynamic model and the shape of the neuron assembly. Similarly, the definition of projections comprises two parts: connection rules and the synaptic model for each connection. SNNL supports flexible definitions of connection rules and provides user-friendly methods for constructing neuron assemblies and network topologies at the syntax level.

The * operator in SNNL is used to define the neuron model and shape information for neuron clusters. As shown in Figure 4, the left side of the * operator represents the neuron model, while the right side is a list of positive integers that represents the shape of the neuron cluster.

The *connections()* function is used to define the synapse model and connection pattern for connection clusters. As shown in Figure 4, *connections()* takes two arguments, representing the DELTA synapse model and the full connection pattern, respectively.

The *net* keyword is used to define an SNN, where the user specifies the network's name and connects neuron clusters using different connection clusters inside the *net* block. The – and - > operators are used to connect neuron clusters and connection clusters. The left side of – represents the source neuron cluster, while the right side represents the connection cluster. Similarly, the left side of - > represents the connection cluster, and the right side represents the target neuron cluster. By using the – and - > operators, users can easily construct the network topology.

```
# define neuron groups
inputs = lif * (28, 28)
conv2d0 = lif * (24, 24, 6)
pool2d0 = lif * (12, 12, 6)
conv2d1 = lif * (8, 8, 16)
pool2d1 = lif * (4, 4, 16)
dense0 = lif * 120
dense1 = lif * 84
dense2 = lif * 10

# define synapse connections
cconv = connections(delta, CONV2D)
cfull = connections(delta, FULL)

# define network
lenet5 : network
    inputs
    -- cconv((weights_init=uniform((1, 6, 5, 5))),
          (kernel_size=5, out_channel=6)) ->
    conv2d0
    -- cconv((weight_init=0.25),
          (kernel_size=2, strides=2)) ->
    pool2d0
    -- cconv((weights_init=uniform((6, 16, 5, 5))),
          (kernel_size=5, out_channel=16)) ->
    conv2d1
    -- cconv((weight_init=0.25),
          (kernel_size=2, strides=2)) ->
    pool2d1
    -- cfull((weight_init=uniform((256, 120))),
          (unit=120, flatten=True)) ->
    dense0
    -- cfull((weight_init=uniform((120, 84))),
          (unit=84)) ->
    dense1
    -- cfull((weight_init=uniform((84, 10))),
          (unit=10)) ->
    dense2
```

FIGURE 4. An example of network construction.

TURING COMPLETENESS

Turing completeness is a key standard for evaluating the expressive power of a computational system. It requires the system to be capable of simulating the computational process of a Turing machine, thereby achieving universal computational capability. Turing completeness requires that the SNN programming language is theoretically capable of simulating the computational behavior of a Turing machine. This section provides a detailed proof of the Turing completeness of SNNs by examining finite-state representation, tape mapping, and state transition functions.

Finite-State Representation

A Turing machine's set of states is defined as

$$Q = \{q_1, q_2, \ldots, q_n\}$$

where each state q_i represents a distinct stage in the computation process. In contrast, the state of an SNN is characterized by the membrane potentials of a group of neurons, denoted as $v_t \in \mathbb{R}^m$, forming the set

$$V = \{v_t \in \mathbb{R}^m : t \geq 0\}.$$

Given that V is a continuous and infinite set, to establish equivalence with the finite-state set Q of a Turing machine, we discretize V into a finite set $Q' = \{\hat{q}_1, \hat{q}_2, \ldots, \hat{q}_m\}$. Each discrete state $\hat{q}_i$ is defined as

$$\hat{q}_i = \arg\min_{q \in Q'} \| v_t - q \|.$$

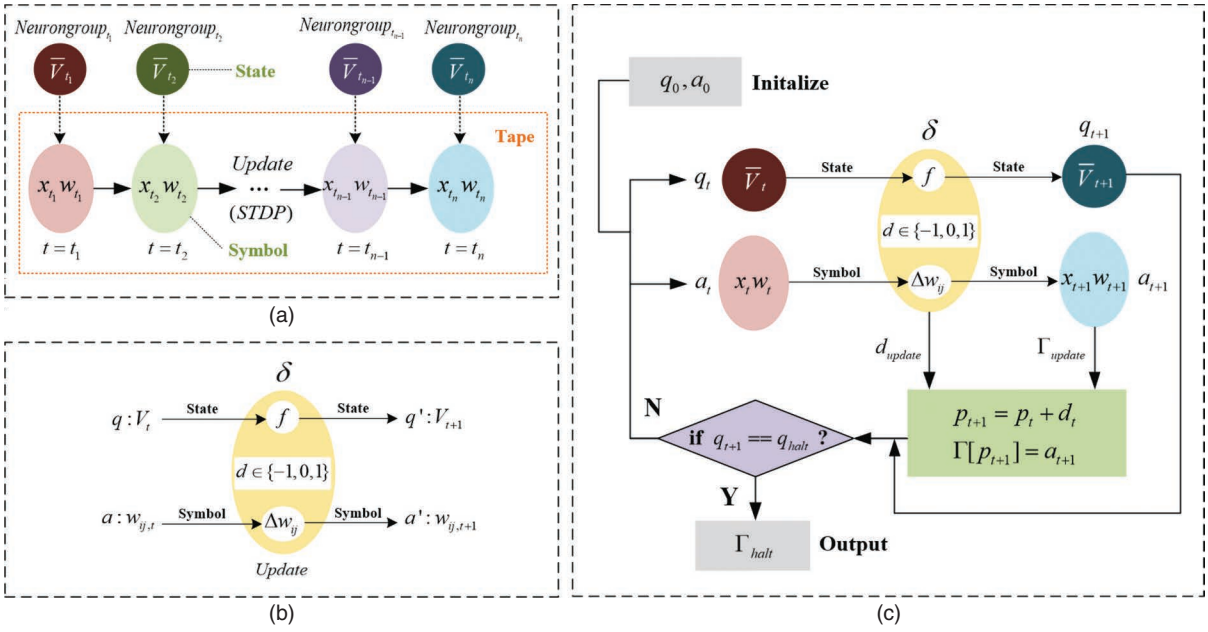

FIGURE 5. (a) Tape mapping. (b) Transfer function. (c) Simulation process of the Turing machine.

Proof of Finite-State Equivalence

To demonstrate the equivalence between Q' and Q, we impose the following condition: for every $q \in Q$, there exists a corresponding $\hat{q}_i \in Q'$ such that

$$\|q - \hat{q}_i\| \leq \epsilon_q$$

where ϵ_q represents the discretization error. Additionally, for any v_t, the distance to its nearest cluster center satisfies

$$\|v_t - \hat{q}_i\| \leq \epsilon_q, \quad \forall v_t \in \text{cluster } \hat{q}_i.$$

By increasing the number of elements in Q' or optimizing the clustering algorithm, ϵ_q can be made arbitrarily small

$$\epsilon_q \to 0, \quad \text{as } |Q'| \to \infty.$$

This ensures that the discrete-state set Q' can approximate the finite-state set Q of the Turing machine, thereby enabling the SNN to represent finite states effectively.

Tape Mapping

The tape of a Turing machine is an infinitely long linear storage medium, with each cell holding a symbol and supporting read and write operations. In an SNN, tape symbols are represented through neuronal activity [see Figure 5(a)].

Symbol Representation and Decoding

In SNNs, symbols are encoded by input spike trains x_t and the synaptic weight matrix W, producing neuronal responses

$$y_t = Wx_t.$$

A decoding function f maps y_t to discrete symbols σ

$$f(y_t) = \sigma_i, \quad \text{if } b_{i-1} \leq y_t < b_i$$

where b_i are partition boundaries, and σ_i are elements of the symbol set.

Synaptic Weight Update

Dynamic adjustment of the synaptic weight matrix W facilitates symbol writing and storage, following Hebbian or spike-timing-dependent plasticity (STDP) learning rules

$$\Delta W_{ij} = \eta \cdot o_i \cdot o_j$$

where η is the learning rate, and o_i and o_j denote the activities of the pre- and postsynaptic neurons, respectively. By appropriately tuning W and b_i, the SNN can accurately perform tape operations, including reading, writing, and storing symbols.

State Transition Function

The state transition function of a Turing machine is defined as

$$\delta(q,\sigma) = (q',\sigma',d)$$

where q is the current state; σ is the current symbol; q' and σ' are the updated state and symbol, respectively; and $d \in \{-1,0,1\}$ indicates the movement direction of the read-write head.

Implementation of State Transitions in SNNs

In SNNs, state transitions are governed by the dynamics of neuronal membrane potentials v_t and the adaptive synaptic weights W. The dynamics are described by

$$v_{t+1} = f(v_t, x_t)$$

where f represents the neuronal dynamic model, and x_t includes encoded information of the current state and symbol.

Symbol Update

Following a state transition, the tape symbol is updated by modifying the synaptic weights to reflect σ'

$$W' = W + \Delta W$$

with ΔW determined by the encoding of the new symbol σ'.

Movement of the Read-Write Head

The direction d of the read-write head's movement is determined by changes in the neuronal membrane potential

$$d = \text{sign}(v_{t+1} - v_t)$$

where $d = -1$ signifies a left move, $d = 0$ indicates no movement, and $d = 1$ denotes a right move. The current position p of the read-write head is updated as

$$p' = p + d.$$

Simulation Process

By integrating finite-state representation, tape mapping, and state transition functions, an SNN can emulate the computational behavior of a Turing machine through the following steps:

1) *Initialization*: Utilize clustering techniques to initialize the SNN's state to the discrete set Q' and load the initial tape symbols into the synaptic weight matrix W.
2) *State computation*: Compute the next state v_{t+1} using the dynamic model f based on the current state v_t and input x_t.
3) *Symbol update*: Adjust the synaptic weights W to update the tape symbol to σ'.
4) *Read-write head movement*: Determine the movement direction d and update the head position p.
5) *Iteration*: Repeat the processes of state update, symbol writing, and head movement until reaching a halt state.

Through theoretical proof and validation, the SNN programming language effectively simulates a Turing machine by approximating finite states through clustering. The simulation error originates primarily from the state discretization process, but its impact on global result consistency is controllable. The SNN language not only accurately simulates the computational processes of a Turing machine but also demonstrates powerful parallel capabilities, providing more efficient solutions for complex computational scenarios.

COMPILER DESIGN

The compiler architecture [see Figure 6(b)]. for this digital subscriber line (DSL) is designed with a layered structure, emphasizing flexibility and extensibility. It includes several key stages:

- *Front end*: The general-purpose program, written in the DSL, is processed in the front end. The primary role of the front end is to analyze the source code and transform it into an intermediate representation (IR). This representation is more abstract than machine code but structured enough for further analysis and transformation.
- *Program segmentation and equation solving*: At this stage, the program is segmented, and any necessary equation solving is performed. This step ensures that complex mathematical or logical relationships are handled efficiently, preparing the code for further synthesis and optimization.
- *Program synthesis*: Following segmentation, the compiler enters a synthesis phase. This is where the source program is translated into a more concrete form, likely as a prelude to generating intermediate code or interacting with the

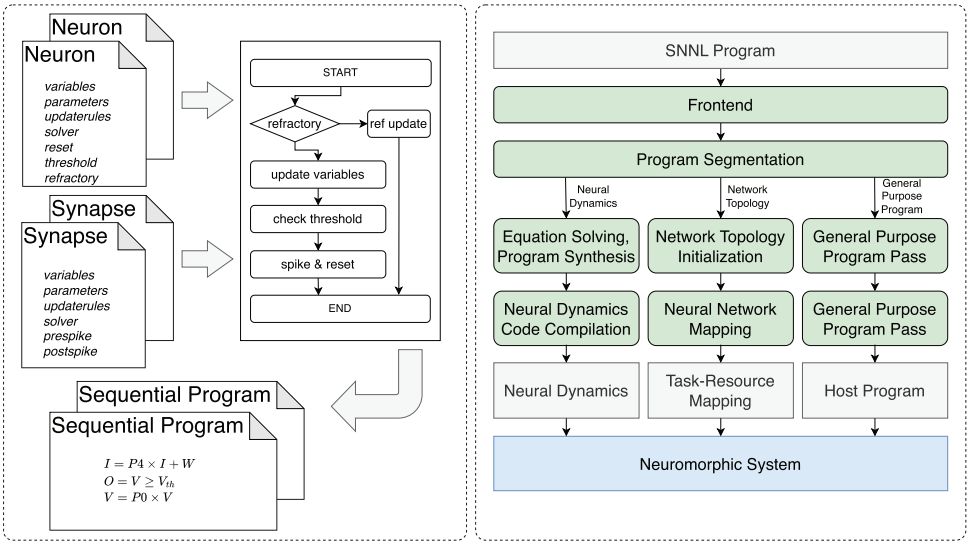

FIGURE 6. (a) Program synthesis. (b) Compiler architecture.

low level virtual machine (LLVM) framework [see Figure 6(a)]

- *LLVM IR and code generation*: The system leverages the LLVM framework, specifically using the LLVM IR as a middle layer. This IR is processed by LLVM's code generation mechanisms, enabling optimization and eventual compilation into target machine code. The back-end phase focuses on transforming the IR into executable target code suitable for specific hardware architectures.
- *Neuromorphic hardware interface*: A notable aspect of this architecture is the interface with neuromorphic hardware. The DSL appears to be tailored to systems that involve neural dynamics and network topology. The compiler has built-in mechanisms for handling neural dynamic code generation, indicating that the language is used in specialized environments such as brain-inspired or neuromorphic computing systems.
- *Neural dynamics and network topology*: During the code generation phase, the compiler takes into account the neural network topology and dynamic behaviors. These components are crucial for translating high-level abstractions into forms that can be efficiently executed on neuromorphic systems, such as those modeled after biological neural networks.
- *Target code and back end*: Finally, the compiler's back end generates the target machine code, utilizing the LLVM back end to produce highly optimized and hardware-specific instructions that can run efficiently on the underlying neuromorphic or general-purpose hardware.

CASE STUDY

SNNL Workflow Experiments

Figure 7 shows a complete workflow from defining neuron models using SNNL to deployment of the model onto a neuromorphic system.

As shown in Figure 7(a), the SNN is composed of three groups of neurons: the input, excitatory, and inhibitory layers. In the input layer, the raw data are encoded into spike trains using Poisson encoding. The excitatory layer is fully connected with the input layer, and the synapses between are trained using STDP rules. Both the excitatory and inhibitory layers have the same amount of neurons. Each neuron in the excitatory layer is uniquely connected to one corresponding neuron in the inhibitory layer. Neurons in the inhibitory layer are reversely connected to every neuron in the excitatory layer except the one from which it receives spikes.

Figure 7(b) gives three code snippets of SNNL to represent the modeling of leaky integrate-and-fire neurons, modeling of synapses with STDP learning rules, and construction of the neural network.

Figure 7(c) depicts the compilation of SNNL. The codes representing the dynamics of neurons and

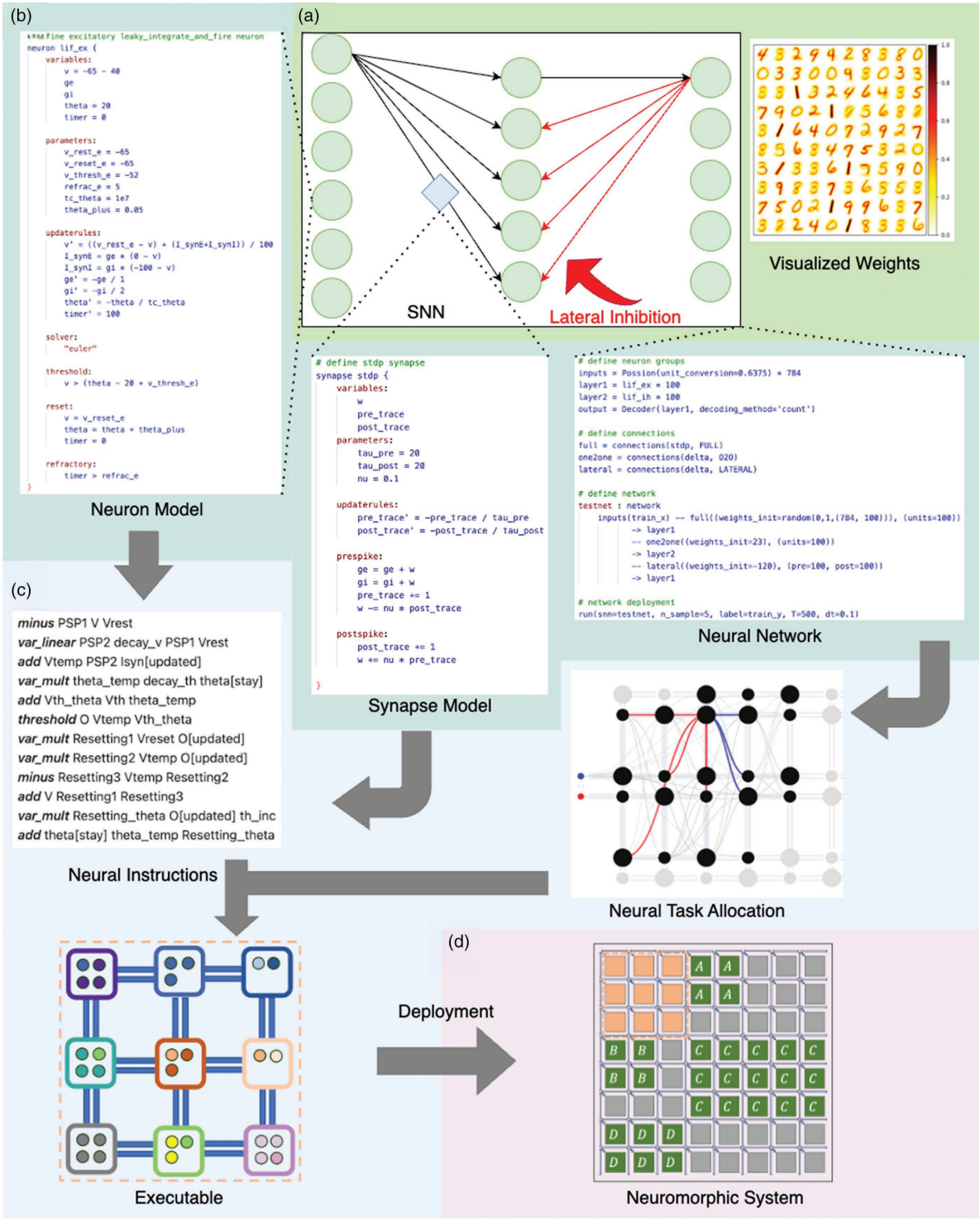

FIGURE 7. Unsupervised learning using STDP. (a) A three-layer SNN and visualized weights between the first two layers. (b) Code snippets using SNNL programming for the aforementioned SNN. (c) Compiled results. (d) The executable file is fed to the neuromorphic system for implementation.

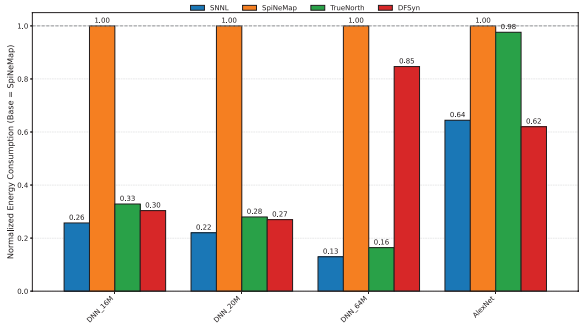

FIGURE 8. Results on energy consumption. Normalized to SpiNeMap.

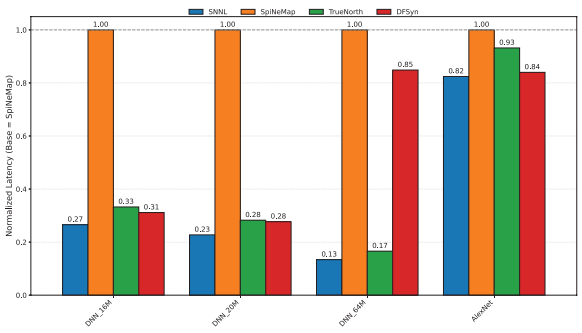

FIGURE 9. Results on latency. Normalized to SpiNeMap.

synapses are compiled into instructions and further into machine codes, which the neuromorphic cores support. The neural network is abstracted as a graph where each node is an entity that comprises a set of computations corresponding to the dynamics of neurons and synapses. As the dynamics of each neuron and synapse can be simulated in parallel, the nodes in the graphs are distributed across different neuromorphic cores for execution. Finally, we obtain the executable file and deploy it onto the neuromorphic system.

Performance Results

To quantitatively evaluate the performance of our proposed SNNL framework, we conducted a comparative analysis against representative SNN programming and mapping frameworks for neuromorphic hardware: SpiNeMap,[15] the core mapping algorithm of TrueNorth,[7] and DFSyn.[16] The evaluation focuses on two critical performance indicators for neuromorphic computing: total energy consumption and inference latency. For a fair comparison, the results for all frameworks are normalized to those of SpiNeMap.

Figure 8 illustrates the normalized energy consumption. The results clearly indicate that SNNL achieves a significant reduction in energy usage across the benchmarks. This efficiency gain is attributed to SNNL's optimizing compiler, which translates the high-level description of neural dynamics into compact, hardware-specific machine code. This process minimizes redundant operations and leverages the energy-efficient features of the target neuromorphic architecture.

The latency results, shown in Figure 9, follow a similar trend. SNNL consistently demonstrates the lowest inference latency compared to the baseline frameworks. This improvement stems from the compiler's ability to generate an optimized execution schedule and perform efficient task-to-core mapping, which reduces both computational overhead and intercore communication delays on the NoC.

Overall, these empirical results confirm that SNNL not only simplifies the development process but also produces highly efficient SNN implementations, outperforming existing frameworks in both energy and speed.

CONCLUSION

This work presented SNNL, a novel programming language designed to streamline the development and deployment of SNNs on neuromorphic hardware. By leveraging a modular approach that separates neural dynamics from network topology, SNNL introduces a flexible and intuitive framework for modeling neuron and synapse dynamics. SNNL supports the construction of complex network topologies, enabling users to define neural assemblies and synaptic connections with ease.

Experimental results demonstrated the effectiveness of SNNL in developing SNN applications for neuromorphic systems, showcasing its capability to handle complex dynamics and network topology. By reducing development complexity and improving accessibility, SNNL has the potential to accelerate the adoption and advancement of SNN-based neuromorphic computing applications.

Future work will focus on expanding the language's capabilities, including enhanced support for hybrid architectures and online learning mechanisms, to further broaden its applicability in neuromorphic research.

REFERENCES

1. J. Perera et al., "Wet-neuromorphic computing: A new paradigm for biological artificial intelligence," *IEEE*

Intell. Syst., vol. 40, no. 3, pp. 39–48, May/Jun. 2025, doi: 10.1109/MIS.2025.3555551.
2. H. Yang, Q. Kong, R. Zhang, and W. Mao, "Efficient spiking variational graph autoencoders for unsupervised graph representation learning tasks," *IEEE Intell. Syst.*, vol. 39, no. 5, pp. 37–46, Sep./Oct. 2024, doi: 10.1109/MIS.2024.3391937.
3. S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The SpiNNaker project," *Proc. IEEE*, vol. 102, no. 5, pp. 652–665, May 2014, doi: 10.1109/JPROC.2014.2304638.
4. S. Höppner et al., "The SpiNNaker 2 processing element architecture for hybrid digital neuromorphic computing," 2021, *arXiv:2103.08392*.
5. M. Davies et al., "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan./Feb. 2018, doi: 10.1109/MM.2018.112130359.
6. G. Orchard et al., "Efficient neuromorphic signal processing with Loihi 2," in *Proc. IEEE Workshop Signal Process. Syst. (SiPS)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 254–259, doi: 10.1109/SiPS52927.2021.00053.
7. M. V. DeBole et al., "TrueNorth: Accelerating from zero to 64 million neurons in 10 years," *Computer*, vol. 52, no. 5, pp. 20–29, May 2019, doi: 10.1109/MC.2019.2903009.
8. C. Pehle et al., "The BrainScaleS-2 accelerated neuromorphic system with hybrid plasticity," *Front. Neurosci.*, vol. 16, Feb. 2022, Art. no. 795876, doi: 10.3389/fnins.2022.795876.
9. D. Ma et al., "Darwin3: A large-scale neuromorphic chip with a novel isa and on-chip learning," *National Sci. Rev.*, vol. 11, no. 5, 2024, Art. no. nwae102, doi: 10.1093/nsr/nwae102.
10. R. K. Chunduri and D. G. Perera, "Neuromorphic sentiment analysis using spiking neural networks," *Sensors*, vol. 23, no. 18, 2023, Art. no. 7701, doi: 10.3390/s23187701. [Online]. Available: https://www.mdpi.com/1424-8220/23/18/7701
11. N. Getty, T. Brettin, D. Jin, R. Stevens, and F. Xia, "Deep medical image analysis with representation learning and neuromorphic computing," *Interface Focus*, vol. 11, no. 1, 2021, Art. no. 20190122, doi: 10.1098/rsfs.2019.0122.
12. A. H. Qahtani et al., "Neuromorphic edge artificial intelligence architecture for real-time surgical decision support: Integrating spiking neural networks with hybrid symbolic-neural reasoning," *J. Adv. Trends Med. Res.*, vol. 2, no. 2, pp. 288–294, 2025, doi: 10.4103/ATMR.ATMR_61_25.
13. C. D. Schuman et al., "A survey of neuromorphic computing and neural networks in hardware," 2017, *arXiv:1705.06963*.
14. S. Deng et al., "Darwin-S: A reference software architecture for brain-inspired computers," *Computer*, vol. 55, no. 5, pp. 51–63, May 2022, doi: 10.1109/MC.2022.3144397.
15. A. Balaji et al., "Mapping spiking neural networks to neuromorphic hardware," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 28, no. 1, pp. 76–86, Jan. 2020, doi: 10.1109/TVLSI.2019.2951493.
16. S. Song, H. Chong, A. Balaji, A. Das, J. Shackleford, and N. Kandasamy, "DFSynthesizer: Dataflow-based synthesis of spiking neural networks to neuromorphic hardware," *ACM Trans. Embedded Comput. Syst.*, vol. 21, no. 3, pp. 1–35, 2022, doi: 10.1145/3479156.

QINGHUI XING is currently pursuing his Ph.D. degree in computer architecture, neuromorphic computing with the School of Computer Science and Technology, Zhejiang University, Hangzhou, 310058, China. His research interests include neuromorphic computing and computer architecture. Xing received his B.S. degree from Northwestern Polytechnical University. He is a co-first author of this article. Contact him at xingqh@zju.edu.cn.

ZIRUN LI is currently pursuing his Ph.D. degree in brain-inspired computing, compiler optimization design with the School of Electronic Science and Engineering, Xiamen University, Xiamen, 361005, China. His research interests include brain-inspired computing and compiler optimization design. Li received his M.S. degree in optical engineering from Huaqiao University. He is a co-first author of this article. Contact him at 36620220156208@stu.xmu.edu.cn.

YING LI is an associate professor with the College of Computer Science, Zhejiang University, Hangzhou, 310058, China. His research interests include service computing, process mining, and compiler technology. Li received his Ph.D. degree in computer science from Zhejiang University. He is the corresponding author of this article. Contact him at cnliying@zju.edu.cn.

SCHAHRAM DUSTDAR is a full professor of computer science (informatics) with a focus on Internet technologies and heads the Distributed Systems Group, TU Wien, Vienna, 1040, Austria. His research interests include distributed systems, Internet of Things, edge computing, edge intelligence, and edge AI. Dustdar received his Ph.D. degree (1992) in business informatics (Wirtschaftsinformatik) from the University of Linz, Austria. He is

a Fellow of IEEE and a member of Academia Europaea. Contact him at dustdar@dsg.tuwien.ac.at.

XIN DU is an assistant professor with the School of Software Technology, Zhejiang University, Hangzhou, 310058, China. His research interests include service computing, distributed systems, and brain-inspired computing. Du received his Ph.D. degree in computer science from Fudan University. Contact him at xindu@zju.edu.cn.

GANG PAN is a professor with the College of Computer Science and Technology, Zhejiang University, Hangzhou, 310058, China. His research interests include pervasive computing, computer vision, and pattern recognition. Pan received his Ph.D. degree in computer science from Zhejiang University. He is a Senior Member of IEEE. Contact him at gpan@zju.edu.cn.

SHUIGUANG DENG is a full professor with the College of Computer Science and Technology, Zhejiang University, Hangzhou, 310058, China. His research interests include edge computing, service computing, and business process management. Deng received his Ph.D. degree in computer science from Zhejiang University. He is a Senior Member of IEEE. Contact him at dengsg@zju.edu.cn.