Compiling Spiking Neural Networks to Neuromorphic Hardware

Shihao Song, Adarsha Balaji, Anup Das, Nagarajan Kandasamy, and James Shackleford
{shihao.song,ab3586,anup.das,nk78,jas64}@drexel.edu
Electrical and Computer Engineering, Drexel University
Philadelphia, PA

Abstract

Machine learning applications that are implemented with spike-based computation model, e.g., Spiking Neural Network (SNN), have a great potential to lower the energy consumption when executed on a neuromorphic hardware. However, compiling and mapping an SNN to the hardware is challenging, especially when compute and storage resources of the hardware (viz. crossbars) need to be shared among the neurons and synapses of the SNN. We propose an approach to analyze and compile SNNs on resource-constrained neuromorphic hardware, providing guarantees on key performance metrics such as execution time and throughput. Our approach makes the following three key contributions. First, we propose a greedy technique to partition an SNN into clusters of neurons and synapses such that each cluster can fit on to the resources of a crossbar. Second, we exploit the rich semantics and expressiveness of Synchronous Dataflow Graphs (SDFGs) to represent a clustered SNN and analyze its performance using Max-Plus Algebra, considering the available compute and storage capacities, buffer sizes, and communication bandwidth. Third, we propose a self-timed execution-based fast technique to compile and admit SNN-based applications to a neuromorphic hardware at run-time, adapting dynamically to the available resources on the hardware. We evaluate our approach with standard SNN-based applications and demonstrate a significant performance improvement compared to current practices.

CCS Concepts: • **Hardware** → **Neural systems**; **Emerging languages and compilers**; **Emerging tools and methodologies**; • **Computer systems organization** → **Data flow architectures**; **Neural networks**.

LCTES '20, June 16, 2020, London, United Kingdom
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7094-3/20/06…$15.00
https://doi.org/10.1145/3372799.3394364

Keywords: neuromorphic computing, data flow, machine learning, spiking neural network

ACM Reference Format:
Shihao Song, Adarsha Balaji, Anup Das, Nagarajan Kandasamy, and James Shackleford. 2020. Compiling Spiking Neural Networks to Neuromorphic Hardware. In *Proceedings of the 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '20), June 16, 2020, London, United Kingdom.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3372799.3394364

1 Introduction

Machine learning tasks implemented with spike model [15] and brain-inspired learning algorithms [21], e.g., Spiking Neural Network (SNN) [47], have a great potential to lower the energy consumption when they are executed on a neuromorphic hardware such as DYNAP-SE [51], TrueNorth [30], Neurogrid [10], SpiNNaker [34], and Loihi [28]. This makes SNNs attractive for implementing machine learning applications in resource and power-constrained environments, ones where sensor and edge devices of the Internet-of-Things (IoT) [37] typically operate. A neuromorphic hardware consists of computation units called crossbars, communicating with each other using an interconnect. A crossbar can accommodate a fixed number of neurons and synapses.

Executing a program on a hardware involves several steps: compilation, resource allocation, and run-time mapping. Although apparent for mainstream computers, these steps are challenging and not very well defined when executing an SNN-based machine learning application on a neuromorphic hardware. This is because a neuromorphic hardware implements accumulation-based alternate computing, where neural computations and synaptic storage are co-located inside each crossbar and distributed in the hardware. This is different from a conventional computer where CPUs compute by exchanging data centrally from the memory.

Prior research efforts such as [5, 8, 9, 27, 40, 41, 46] have *only* addressed design-time analysis of an application with *unlimited* hardware resources, e.g., arbitrarily large crossbars and many interconnected crossbars as needed to accommodate all neurons and synapses of the application. While these efforts are still relevant when designing the hardware, they cannot provide a realistic guarantee of performance when executing these applications on an off-the-shelf neuromorphic hardware. This is because prior efforts fail to answer how to *share* compute and storage resources of the hardware to guarantee performance when not all neurons and synapses of an SNN can fit on the hardware at once. Table 1, shown in Section 6, lists the number of neurons and synapses in standard machine learning applications, which are on the order of thousands of neurons and hundreds of thousands

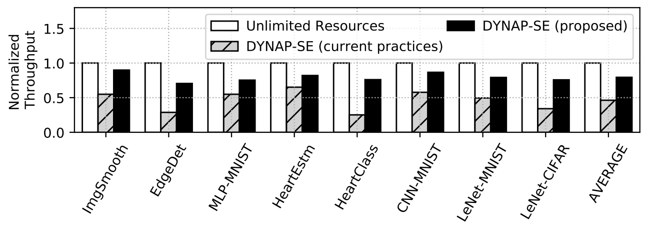

Figure 1. Performance impact due to limited resources.

Figure 1 illustrates the throughput impact due to limited resources on DYNAP-SE for the evaluated applications (see Section 6). We observe that throughput obtained on DYNAP-SE using current practices is on average 64% lower than throughput analyzed using unlimited resources. *Our objective is to reduce this performance gap when compiling SNN-based applications on neuromorphic hardware with limited resources.* The figure also plots our approach, which achieves an average 78% higher throughput than current practices.

A second limitation of existing approaches is that they do not address run-time aspects, i.e., how to *compile and admit* machine learning applications to the hardware in the *least possible time* based on the available resources.

To address these limitations, we propose a systematic and predictable approach to compile and map SNN-based machine learning applications on resource-constrained neuromorphic hardware, providing performance guarantee.

Contributions: Following are our key contributions.

- We propose to partition an SNN into clusters of neurons and synapses, where each cluster can fit on to the resources of a crossbar in the hardware. We pose this as a bin-packing problem and propose a greedy strategy to solve it, maximizing crossbar utilization.
- We exploit the rich semantics and expressiveness of Synchronous Data Flow Graphs (SDFGs) to represent a clustered SNN and use Max-Plus Algebra to analyze its performance, e.g., throughput.
- We model resource constraints such as limited crossbars, input and output buffer sizes, and communication channel bandwidth into the SDFG representation. We extend the Max-Plus Algebra and use Self-Timed Execution to construct static-order schedules to estimate performance of this hardware-aware SDFG.
- We exploit a property of Self-Timed Scheduling to derive the schedule for each tile at run-time, starting from a single static-order schedule, without having to construct these schedules from scratch. This reduces the time to compile and admit a machine learning application to the hardware at run time and adapt dynamically to the available hardware resources.

Figure 2 shows a high-level workflow of our proposed approach. The colored boxes in this figure are our key contributions. This workflow incorporates both Artificial Neural Network (ANN)-based applications written in a high-level language such as PyTorch [54] or Tensorflow [1], and SNN-based applications written in PyCARL [5]. In the former scenario, analog computations of the trained ANN-based model is first converted into spiking domain using the N2D2 (Neural Network Design & Deployment) tool [11], an open-source framework for Deep Neural Network (DNN) simulation and full SNN-based applications building. Using this tool we have previously demonstrated conversion of the heart-rate classification application [6], with less than 5% accuracy loss. Once an SNN-based application is available, we simulate the model in CARLsim [18] to record the number of spikes for each neuron in the model when excited with the training input. This spike information is then used in the partitioning step of our workflow to generate a clustered model, which is then used for analyzing performance on hardware.

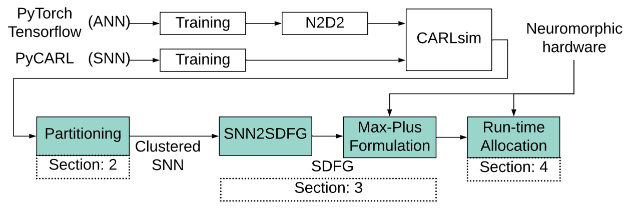

Figure 2. Our proposed approach.

We evaluate performance and scalability of our approach using standard SNN-based applications. Our results, detailed in Section 7, demonstrate a significant performance improvement compared to standard practices.

2 Crossbar-Aware Clustering of SNNs

2.1 Introduction to Spiking Neural Networks

An SNN is a computation model with spiking neurons and synapses. Neurons communicate with each other by sending short impulses of infinitesimally small duration, called spikes, via synapses. Spiking neurons can be organized into feedforward topology, which consists of one input layer, one or more hidden layers, and one output layer (e.g., DNN [44]). Spiking neurons can also be organized in a recurrent topology [48]. SNN-based machine learning applications, especially those that are deployed on sensor and edge devices of an IoT, typically operate on streaming data, i.e., these applications are

iterative in nature. For these applications, real-time performance is measured in terms of **throughput**. We formulate throughput in Section 3.2.

2.2 Crossbar Resource Constraints

A typical neuromorphic hardware (see Figure 7) consists of crossbars, which are interconnected using an interconnection fabric. A crossbar implements neuron dynamics and facilitates synaptic storage. Therefore, each neuron and synapse of an SNN must be mapped to one of these crossbars.

In terms of constraints, a crossbar can accommodate only a fixed number of synapses per neuron. This is illustrated in Figure 3 with three examples using a small 4×4 crossbar. In Figure 3(a), the crossbar implements a single 4-input neuron. In this example, 5 out of 8 (62.5%) input and output (IO) ports are utilized, and 4 out of 16 (25%) crosspoints are utilized. In Figure 3(b), the crossbar implements one 3-input neuron; the IO and crosspoint utilization are 50% and 18.75%, respectively. Finally, in Figure 3(c), the crossbar implements two 2-input neurons, resulting in IO and crosspoint utilization of 75% and 25%, respectively. Clearly, utilization varies based on how neurons and synapses of an SNN are mapped to a crossbar.

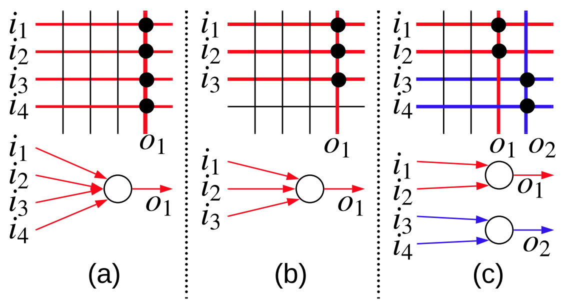

Figure 3. Mapping of neurons & synapses to a 4x4 crossbar.

The SNN of a machine learning application can have many neurons with many synapses per neuron. Take the example of LeNet [43], a state-of-the-art convolutional neural network (CNN) to classify handwritten digits (Figure 4). This application has 4,634 neurons and 1,029,286 synapses, much beyond what a single crossbar can accommodate. To map such a large SNN to the hardware, the SNN needs to be partitioned into clusters of neurons and synapses, where each cluster can fit on to the resources of a crossbar in the hardware. We discuss how to form clusters from an SNN in Sec. 2.3 and how to share crossbars among clusters in Sec. 3.

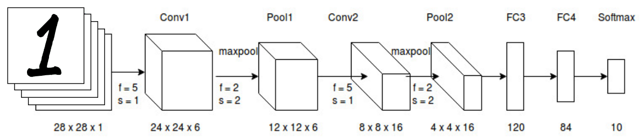

Figure 4. The LeNet Cnvolutional Neural Network used for handwritten digit classification.

2.3 SNN Partitioning

The SNN partitioning problem is a classic bin-packing problem and we propose a greedy strategy to solve this. Algorithm 1 shows the pseudo-code of this clustering algorithm. We first sort (in ascending order) neurons based on each neuron's fanin synapses and store them in a list (`neuron_list`). For each neuron in this sorted list, we check to see if this neuron can be merged in one of the existing clusters in the `cluster_list`. A neuron can be merged in a cluster if the total number of IOs, crosspoints, and buffer usage of the cluster after merging the neuron can still fit on a crossbar of the hardware. This is to ensure that the clustered SNN is deadlock-free when executed on the hardware. If the neuron can be merged, we assign the neuron and its fanin synapses to the cluster. Otherwise, we form a new cluster. The `cluster_list` is sorted in descending order of utilization so that the less utilized clusters can be used for merging neurons with higher fanin.

Algorithm 1: Crossbar-aware SNN partitioning.

```
neuron_list = sort neurons of the SNN based on their fanin synapses;
clusters_list = {};
foreach $n \in$ neuron_list do
    find $C \in$ cluster_list such that $n$ can be merged in $C$;     /* A
      neuron can be merged to a cluster if the IO, bandwidth,
      and buffer constraints of the cluster post merging are
      not violated. */
    if $C = \emptyset$ then
        cluster_list.push($C$);
    end
    else
        Assign $n$ to $C$;
    end
    sort cluster_list in descending order of IO and crosspoint
      utilizations;
end
check for consistency, connectivity, and deadlock in the clustered SNN;
```

2.4 Analyzing Inter-cluster Communication

After partitioning, we analyze the inter-cluster communication, i.e., the number of spikes that are expected between these clusters when an SNN model is deployed in the field on a neuromorphic hardware. We use the spike information collected during CARLsim-based SNN simulation (Fig. 2) to compute the number of spikes between each cluster pair using the neuron-to-cluster mapping obtained from Algorithm 1. Next, we describe how to analyze this clustered SNN.

3 Dataflow Modeling of SNN Clusters

We model a clustered SNN as a Synchronous Data Flow Graph (SDFG) for predictable performance analysis.

3.1 Operational Semantics of SDF Graphs

Synchronous Data Flow Graphs (SDFGs, see [45]) are commonly used to model streaming applications that are implemented on a multi-processor system-on-chip [63]. Both pipelined streaming and cyclic dependencies between tasks can be easily modeled in SDFGs. These graphs are used to analyze a system in terms of throughput and other performance properties, e.g. execution time and buffer requirements [66].

Nodes of an SDFG are called *actors*. Each node is a cluster of the SNN. Actors are computed by reading *tokens* (spikes) from their input ports and writing the results of the computation as tokens on the output ports. The number of tokens produced or consumed in one execution of an actor is called the *port rate*. They represent the number of spikes per unit time at the input and output of different clusters in the SNN. Port rates are visualized as annotations on edges. Actor execution is also called *firing*, and it requires a fixed amount of time to execute on a crossbar. Edges in the graph are called *channels* and they represent dependencies among actors.

Figure 5 shows the example of an SDFG constructed using our SNN2SDF tool [58] for the LeNet CNN model used in handwritten digit classification [43]. There are 7 actors and 13 channels in this graph. For instance, `actor_4` has two outgoing channels. The channel going to `actor_6` has a port rate of 2 spikes per unit time, and the one going to `actor_1` has a port rate of 3 spikes per unit time. From Fig. 5 we also see that there are cycles in the graph. Such cycles may arise during the partitioning step. Figure 6(a) illustrates a simple feedforward network of 3 neurons (A, B, & C). Figure 6(b) illustrates a scenario where neurons A and C are placed in cluster 1 (actor 1) and neuron B in cluster 2 (actor 2) during partitioning. Due to the connectivity of the neurons in Figure 6(a), there is a cyclic dependencency between the two actors: $\underline{\texttt{actor\_1}\rightarrow\texttt{actor\_2}\rightarrow\texttt{actor\_1}}$. Therefore, Directed Acyclic Graphs (DAGs) *cannot* be used to represent and analyze clustered SNNs. This justifies our choice of using SDFGs.

An actor is called *ready* when it has sufficient input tokens on all its input channels and sufficient buffer space on all its output channels; an actor can only fire when it is ready. A channel may also contain an *initial token*, shown as annotation. For instance, the channel between `actor_0` and `actor_6` in the figure has 1 initial token. A set $Ports$ of ports is assumed, and with each port $p \in Ports$, a finite rate $Rate(p) \in \mathbb{N} \setminus \{0\}$ is associated. Formally,

Definition 1. *(Actor) An actor $\mathbf{a}_i$ is a tuple $(I_i, O_i, \tau_i, \mu_i)$ consisting of a set I_i ($\subseteq Ports$) of input ports, a set O_i ($\subseteq Ports$) of output ports with $I_i \cap O_i = \emptyset$, τ_i is the execution time of $\mathbf{a}_i$ and μ_i is its state space, i.e., buffer space needed for communicating spikes on all of its channels.*

Definition 2. *(SDFG) An SDFG is a directed graph $G_{app} = (A, C)$ consisting of a finite set A of actors and a finite set $C \subseteq Ports^2$ of channels. The source of channel $ch_i^j \in C$ is an output port of actor $\mathbf{a}_i$, the destination is an input port of actor $\mathbf{a}_j$. All ports of all actors are connected to precisely one channel, and all channels are connected to ports of some actors. The source and the destination port of channel ch_i^j are denoted by $SrcP(ch_i^j)$ and $DstP(ch_i^j)$ respectively. Channels connected to the input and output ports of an actor $\mathbf{a}_i$ are denoted by $InC(\mathbf{a}_i)$ and $OutC(\mathbf{a}_i)$ respectively.*

Before an actor $\boldsymbol{a}_i$ starts its firing, it requires $Rate(q_i)$ tokens from all $(p, q_i) \in InC(\boldsymbol{a}_i)$. When the actor completes execution, it produces $Rate(p_i)$ tokens on every $(p_i, q) \in OutC(\boldsymbol{a}_i)$. One important property of an SDFG is *throughput*, which is defined as the inverse of its long-term period. A period is the average time needed for one iteration of the SDFG. An iteration is defined as the minimum non-zero execution such that the original state of the SDFG is obtained. This is the performance parameter used in this paper. Following definitions are introduced to formulate throughput.

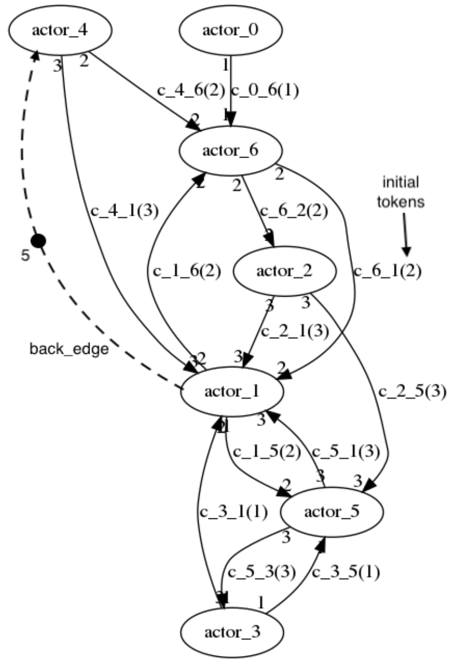

Figure 5. SDFG representation of the LeNet CNN model used for handwritten digit classification [43].

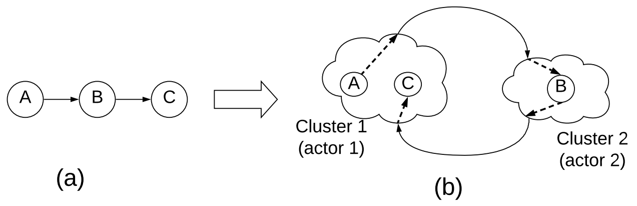

Figure 6. An example cycle generated due to partitioning.

Definition 3. *(Repetition Vector) The Repetition Vector* RptV *of an SDFG is defined as the vector specifying the number of times actors in the SDFG are executed in one iteration.*

In the SDFG representation of a clustered SNN, all spikes generated on a channel are consumed by the destination actor. This means that all actors are fired exactly once during one iteration of the application. So, $RptV = [1111111]$. Furthermore, by incorporating constraints of a crossbar during the partitioning, we ensure that the SDFG generated from the clustered SNN is consistent, connected, and deadlock free.

3.2 Computing Performance on Infinite Resources

We present an approach to compute the application period of an SDFG by analyzing its maximum cycle mean (MCM) and assuming infinite hardware resources. For this, we use Max-Plus Algebra [19, 38, 72]. The key difference of our approach with these prior approaches is the incorporation of resource constraints, which we describe next. The Max-Plus semiring $\mathbb{R}_{\max}$ is the set $\mathbb{R} \cup \{-\infty\}$ defined with two basic operations $\oplus$ and $\otimes$, which are related to linear algebra as

$$a \oplus b = \max(a, b) \text{ and } a \otimes b = a + b \tag{1}$$

To use Max-Plus Algebra to analyze an SDFG, it is customary to express the time at which an actor fires in terms of preceding firings and then use standard analysis techniques for Max-Plus Algebra to estimate timing performance. For the SDFG in Figure 5, firing end time of all 7 actors in the k^{th} iteration (in linear algebra) are

$$
\begin{aligned}
&t_0(k) \geq t_0(k-1) + \tau_0 \qquad (2)\\
&t_1(k) = \max\Big[t_3(k-1), t_5(k-1), t_2(k), t_4(k), t_6(k)\Big] + \tau_1\\
&t_2(k) = t_6(k) + \tau_2\\
&t_3(k) = t_5(k) + \tau_3\\
&t_4(k) \geq t_4(k-1) + \tau_4\\
&t_5(k) = \max\Big[t_3(k-1), t_1(k), t_2(k)\Big] + \tau_5\\
&t_6(k) = \max\Big[t_1(k-1), t_0(k), t_4(k)\Big] + \tau_6
\end{aligned}
$$

Observe that the firing end time of actor `actor_i` in the k^{th} iteration is after its firing end time in the $(k-1)^{\text{th}}$ iteration. Furthermore, the production and consumption rates are the same for every channel in the SDFG. Using previously introduced Max-Plus semantics, firing end times for every actor in the SDFG can be expressed as

$$t_{\mathfrak{n}}(k) = \oplus t_{\mathfrak{m}}(k-1) \otimes \tau_{\mathfrak{n}}, \forall \mathfrak{m} \in Pre(\mathfrak{n}) \tag{3}$$

With a simple transformation of variables, the above sum-of-product equation can be rewritten as

$$\mathbf{t_k} = \mathbf{T} \cdot \mathbf{t_{k-1}} \tag{4}$$

where T captures execution times $\tau_{\mathfrak{n}}$. The following definitions are introduced to estimate latency.

Definition 4. *(Digraph) The digraph (T) of a $n \times n$ matrix T with entries defined in $\mathbb{R}_{max}$ is the tuple $\langle A, E \rangle$, where A is the set of vertices, i.e., $A = \{1, 2, \cdots n\}$ and E is the set of connected ordered arcs between vertices i.e., $E = \{(i, j) \mid T_{i,j} \neq -\infty\}$.*

Definition 5. *(Walk) A walk w in digraph (T) is the sequence of arcs $(x_1, x_2)(x_2, x_3) \cdots (x_{k-1}, x_k)$; head of an arc in the sequence is either the start vertex of the walk or tail vertex of a preceding arc; and the tail vertex of an arc in the sequence is either the end vertex of the walk or head vertex of a succeeding arc. Weight of the walk is given by*

$$|w|_T = T_{x_1x_2} + \cdots T_{x_{k-1}x_k} \tag{5}$$

Definition 6. *(Cycle) A cycle c in digraph (T) is the walk $(x_1, x_2)(x_2, x_3) \cdots (x_{k-1}, x_k)$, such that $x_k = x_1$.*

Definition 7. *(Maximum Cycle Mean) The maximum cycle mean, $\rho_{max}(T)$ is the maximum of the weight-to-length ratio of all cycles c in (T) i.e.,*

$$\rho_{max}(T) = \max_{\forall c \text{ in } (T)} \frac{|c|_T}{|c|} = \max_{k \geq 1} \max_{x_1, \cdots, x_k} \frac{T_{x_1x_2} + \cdots T_{x_kx_1}}{k} \tag{6}$$

In this paper, **performance of an SNN is defined in terms of throughput** of the equivalent SDFG, measured as the inverse of its *maximum cycle mean* (Equation 6).

4 Hardware-Aware Performance Analysis

We now extend the Max-Plus formulation to analyze performance of an SNN on a resource-constrained hardware.

4.1 Platform Description

Performance of an SNN, computed using Equation 6, gives the maximum period possible with infinite hardware resources in terms of crossbars, buffer sizes, and communication bandwidth. For off-the-shelf neuromorphic hardware, however, these resources are limited. Figure 7 shows a typical tile-based neuromorphic hardware, where tiles are connected via an interconnection. Each tile consists of a crossbar (C), input and output buffers, and a network interface (NI). A crossbar is a two dimensional organization of horizontal and vertical electrodes. At every cross-point, there is a Oxide-based Resistive RAM (OxRAM) [36] for synaptic storage.

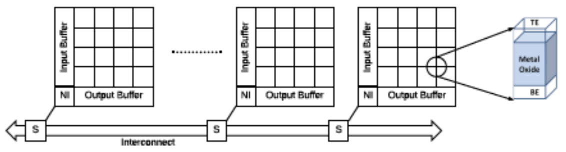

Figure 7. An example tile-based neuromorphic hardware. Each tile consists of a crossbar, input and output buffers, and a network interface (NI).

4.2 Binding Actors (Clusters) to Tiles

Similar in vein to PYCARL [5], we use a load balancing strategy to bind clusters of an SNN to the tiles of the hardware. We first formulate the load of a tile as follows:

$$load(tile) = a * \text{crossbar} + b * \text{buffer} + c * \text{connection} + d * \text{bandwidth} \tag{7}$$

where a, b, c and d are user-defined constants used to prioritize different hardware resources on a tile. Next, we propose a greedy approach to balance the load on each tile. For this, we first distribute the clusters evenly to the tiles and calculate the standard deviation of tile loads. For every cluster pair that is bound to two different tiles, we swap the clusters to see if the standard deviation reduces. If it reduces, we retain this new binding and continue analyzing other cluster pairs.

4.3 Executing a Cluster on a Crossbar

A cluster is executed by placing its neurons and synapses on to the crossbar of a tile. Figure 8 illustrates this execution mechanism. Synaptic weights w_1 and w_2 are programmed into OxRAM cells P1 and P2, respectively. The output spike voltages, v_1 from N1 and v_2 from N2, inject current into the crossbar, which is obtained by multiplying a pre-synaptic neuron's output spike voltage with the OxRAM cell's conductance at the cross-point of the pre- and post-synaptic neurons (following Ohm's law). Current summations along columns are performed in parallel using Kirchhoff's current law, and implement the sums $\sum_j w_i v_i$, needed for forward

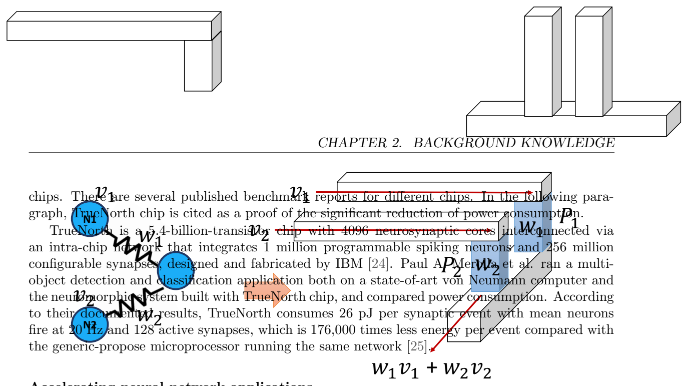

Figure 8. Executing a cluster on a crossbar.

Although a crossbar implements analog computations, spikes at the output are converted into digital packets before communicating on the interconnect. We use the Address Event Representation (AER) protocol [13]. Figure 9 shows an example explaining the principles behind AER. Here, four neurons in a crossbar spikes at time 3, 0, 1 and 2 time units, respectively. The encoder encodes these four spikes in order to be communicated on the interconnect. As can be clearly seen from this figure, a spike is encoded uniquely with its source and time of spike. Therefore, each token in the SDFG is simply a spike packet with header encoding the address and time, and zero payload.

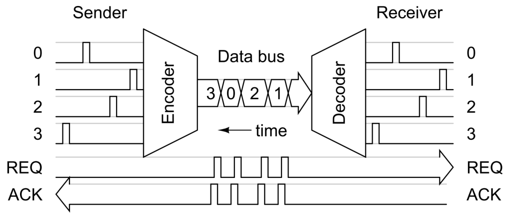

Figure 9. An example AER protocol (adapted from [13]).

4.4 Computing-Resource Aware Performance

To compute performance of an SNN on a resource-constrained neuromorphic hardware, we first construct its hardware-aware SDFG and then compute the maximum cycle mean using the Max-Plus Algebra formulation of Equation 6.

The following three steps describes this approach starting from the actor binding of Section 4.2. Without loss of generality, we use Equation 8 as a running binding example.

$$\textbf{tile\_0}: \text{actor\_3}, \text{actor\_6} \quad \textbf{tile\_2}: \text{actor\_2} \tag{8}$$
$$\textbf{tile\_1}: \text{actor\_2}, \text{actor\_5} \quad \textbf{tile\_3}: \text{actor\_1}, \text{actor\_4}$$

Step 1 (Buffer Modeling): Limited input and output buffer-sizes are modeled as back-edges with initial tokens in the hardware-aware SDFG. The number of tokens on this back-edge indicates the buffer-size available. When an actor generates spikes on a channel, the available size reduces; when the receiving actor consumes the spike, the available buffer is released. Figure 5 shows such an example of a back-edge, where the buffer size of the channel from `actor_4` to `actor_1` is shown as five. Before `actor_4` can be executed, it has to check if enough buffer space is available. This is modeled by requiring tokens from the back-edge to be consumed. Since it produces three tokens per firing, three tokens from the back-edge are consumed, indicating reservation of three buffer spaces. On the consumption side, when `actor_1` is executed, it frees three buffer spaces, indicated by a release of three tokens on the back-edge. We assume *atomic* execution of actors on a crossbar, i.e., a crossbar reads input tokens and produces output tokens in the output buffer for no more than one actor at any given instance of time. To prevent other actors mapped to the same crossbar from being fired simultaneously, the output buffer space is claimed at the start of execution and released only at the end of firing.

Figure 10 shows the final hardware-aware SDFG of LeNet-based handwritten digit classification on a neuromorphic hardware with four tiles. For simplicity of representation, we have omitted the back-edges from the figure.

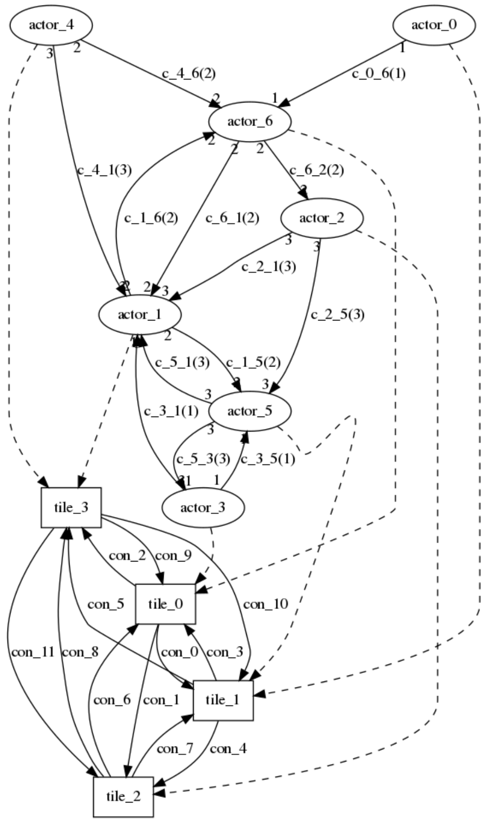

Figure 10. Hardware-aware SDFG of LeNet-based handwritten digit recognition on the neuromorphic hardware of Fig. 7.

Step 2 (Actor Ordering): The number of crossbars in a neuromorphic hardware is limited and therefore they may have to be shared between actors of an SNN. However, on a tile, only one instance of an actor can be executing at the same moment in time. We use time-division multiple-access (TDMA) to allocate time slices to actors mapped to the same tile. During the allocated time slice, an actor is executed on the crossbar of the tile and generates spikes, which are stored in the output buffer for communication on the interconnect. Next, we generate the order in which the actors bound to a tile are fired to provide a guarantee on performance, i.e., throughput. For this, we apply our Max-Plus Algebra formulation (Eq. 6) on the hardware-aware SDFG of Fig. 10. This is our *static-order schedule.* We construct this schedule at *design time.*

Step 3 (Actor Execution): Once the static-order schedule is constructed for all tiles of the hardware, we use self-timed execution strategy [52] for executing these actors at run-time. In this strategy, the exact firing times of actors are discarded, retaining only the assignment and ordering of actors on each tile as obtained from the design-time analysis (step 2). At run time, ready actors are inserted in a list and fired in the same order as determined from design time.

5 Run-time Resource Management

A modern neuromorphic hardware is expected to execute many SNN applications simultaneously. When a new application is to be admitted to a hardware, which is currently running other applications, the incoming application needs to be compiled and mapped to the hardware within a short time window, based on resources currently available on the hardware. Furthermore, when an existing application finishes execution, its hardware resources are freed, meaning that such resources can now be allocated to other running applications to improve their performance. Clearly, a dynamic compilation strategy is needed to address them.

We observe that over 75% of the total compilation time of an SNN application is due to the time consumed in constructing the static-order schedule for each tile of the neuromorphic hardware (see Section 7.3). To address this, we exploit the basic property of Max-Plus Algebra and self-timed scheduling, which is expressed as the following lemma.

Lemma 1. *If the schedule of actors on a single-tile system is used to derive the schedule for a multi-tile system by keeping the actor firing order unchanged, the resultant multi-tile schedule is free of deadlocks [12].*

Based on this lemma, we propose the following. First, we construct the static-order schedule for all actors of an SNN on a single tile at design-time. This is achieved using our proposed Max-Plus Algebra formulation of Equation 6. Next, we discard the exact timing information, retaining only the actor firing orders for run-time use. At run-time, we first construct the actor binding to tiles (Section 4.2), considering the available resources. Next, we use the single-tile static-order schedule to fire actors when they are ready. Figure 11 illustrates our run-time methodology.

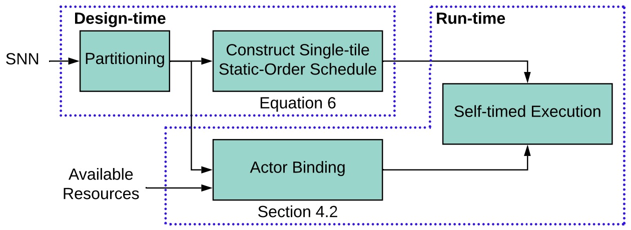

Figure 11. Our approach to run-time resource management.

Figure 12 illustrates the construction of per-tile schedules for an SNN application with seven run-time actors, and with two different binding of actors to tiles and the same single-tile static order schedule. We illustrate two scenarios in this example. In the first scenario (left), the application uses two tiles of the hardware. In the second scenario (right), the application uses three tiles of the hardware. In both scenarios, actor orders on each tile is the same as that on the single-tile. Since tile schedules are not constructed from scratch, the schedule construction time is much lower (see Table 3).

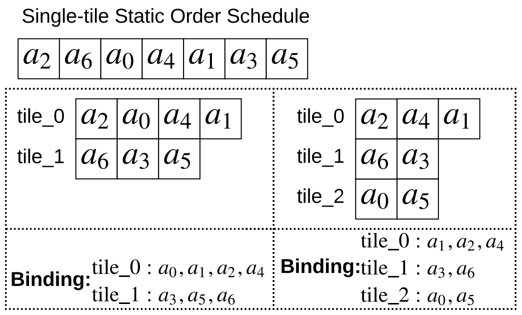
Single-tile Static Order Schedule
a_2 a_6 a_0 a_4 a_1 a_3 a_5
tile_0 a_2 a_0 a_4 a_1
tile_1 a_6 a_3 a_5
tile_0 a_2 a_4 a_1
tile_1 a_6 a_3
tile_2 a_0 a_5
Binding: tile_0 : a_0, a_1, a_2, a_4
tile_1 : a_3, a_5, a_6
Binding: tile_0 : a_1, a_2, a_4
tile_1 : a_3, a_6
tile_2 : a_0, a_5

Figure 12. Schedules constructed from the same single-tile static order schedule using 2 and 3 tiles, respectively.

However, performance obtained using this single-tile schedule can be lower than the maximum performance of a multi-tile schedule constructed independently. As long as this performance deviation is bounded, the actor schedule for any tile can be easily derived from the binding of actors to this tile and a given single-tile static-order schedule. In Section 7.6, we evaluate the performance of this scheduling.

6 Evaluation Methodology

6.1 Hardware Models

We model the DYNAP-SE neuromorphic hardware [51] with the following configurations.

- A tiled array of 4 crossbars, each with 128 input and 128 output neurons. There are 65,536 crosspoints (i.e., OxRAM NVMs) in each crossbar.
- Spikes are digitized and communicated between cores through a mesh routing network using the Address Event Representation (AER) protocol.

- Each synaptic element is an HfO2-Based OxRAM Device. Timing parameters are modeled from [36].

To test scalability of our compilation technique, we also evaluate hardware models with 9 and 16 neuromorphic cores.

6.2 Evaluated Applications

We evaluate eight standard SNN-based machine learning applications: 1) image smoothing (ImgSmooth) [18] on 64×64 images; 2) edge detection (EdgeDet) [18] on 64×64 images using difference-of-Gaussian; 3) multi-layer perceptron (MLP)-based handwritten digit recognition (MLP-MNIST) [32] on 28×28 images of handwritten digits from the MNIST dataset [31]; 4) heart-rate estimation (HeartEstm) using electrocardiogram (ECG) data [26] from the Physionet database [50]; 5) ECG-based heart-beat classification (HeartClass) [6]; 6) handwritten digit classification with standard CNN (CNN-MNIST) [56, 62]; 7) handwritten digit classification with the LeNet CNN (LeNet-MNIST) [56]; and 8) image classification with LeNet CNN (LeNet-CIFAR) [56] with images from the CIFAR dataset [42]. The LeNet CNN model is described in [43]. Table 1 summarizes the topology and the number of neurons, synapses, and spikes of these applications. Image-based applications are iteratively executed on test images.

Applications	Synapses	Neurons	Topology	Spikes	Accuracy
ImgSmooth [18]	136,314	980	FeedForward (4096, 1024)	17,600	100%
EdgeDet [18]	272,628	1,372	FeedForward (4096, 1024, 1024, 1024)	22,780	100%
MLP-MNIST [32]	79,400	984	FeedForward (784, 100, 10)	2,395,300	95.5%
HeartEstm [26]	636,578	6,952	Recurrent	3,002,223	99.2%
HeartClass [6]	2,396,521	24,732	CNN[1]	1,036,485	85.12%
CNN-MNIST [56]	159,553	5,576	CNN[2]	97,585	96.7%
LeNet-MNIST [56]	1,029,286	4,634	CNN[3]	165,997	99.1%
LeNet-CIFAR [56]	2,136,560	18,472	CNN[4]	589,953	84.0%

1. Input(82x82) - [Conv, Pool]*16 - [Conv, Pool]*16 - FC*256 - FC*6
2. Input(24x24) - [Conv, Pool]*16 - FC*150 - FC*10
3. Input(32x32) - [Conv, Pool]*6 - [Conv, Pool]*16 - Conv*120 - FC*84 - FC*10
4. Input(32x32x3) - [Conv, Pool]*6 - [Conv, Pool]*6 - FC*84 - FC*10

Table 1. Applications used to evaluate our approach.

6.3 Evaluated State-of-the-art Techniques

We evaluate the following three approaches.

- SpiNeMap [8] maps SNNs to tiles, minimizing spikes on the interconnect. Clusters on a tile are executed in a random order.
- PYCARL [5] maps SNNs to tiles, balancing tile load. Clusters on a tile are executed in a random order.
- Our proposed approach uses SDFGs to analyze the performance of an SNN on a neuromorphic hardware. Clusters are allocated to tiles based on this analysis. Overall, our approach balances load on each tile and uses static-order schedule to improve throughput.

6.4 Evaluated Metrics

We evaluate the following performance metrics.

- Performance: This is the throughput of each application on the hardware.
- Compilation Time: This is the time to compile and map each application on the hardware.
- Resource utilization: This is the tile, buffer, connection, and input and output bandwidth utilization on the hardware for each application.

7 Results and Discussion

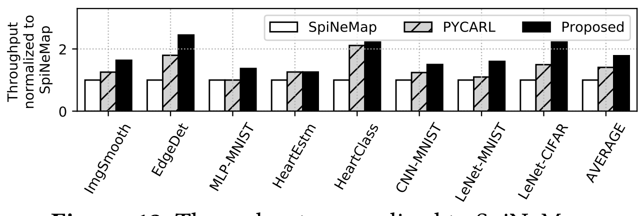

Figure 13. Throughput, normalized to SpiNeMap.

First, throughput obtained using SpiNeMap is the lowest among all the evaluated techniques. This is because SpiNeMap places SNN clusters on tiles to minimize the number of inter-tile spikes. Therefore, some tiles need to execute many SNN clusters. As cluster ordering on a tile is not addressed in SpiNeMap, throughput is significantly low. Second, throughput obtained using PYCARL is better than SpiNeMap by an average of 41%. Although PYCARL also orders cluster execution on a tile randomly, throughput of PYCARL is higher than SpiNeMap. This is due to PYCARL's strategy to balance the load on each tile, resulting in lower number of clusters mapped per tile than SpiNeMap. Third, throughput obtained using our approach is the highest (78% higher than SpiNeMap and 28% higher than PYCARL). This improvement is due to our static-order schedule, which we analyze and construct at design-time for every tile of the hardware to decide the exact order in which clusters mapped to the same tile need to be executed to improve performance.

7.2 Cluster Binding

We reason that balancing the load on the tiles of a hardware is essential to achieving high throughput. Figure 14 reports throughput of each of our application on the DYNAP-SE hardware. We compare our proposed approach against baseline SpiNeMap with random cluster order on each tile and SpiNeMap with static-order schedule on each tile. Throughput results are normalized to SpiNeMap. We make the following two observations. First, throughput of SpiNeMap improves by an average of 39% when static-order scheduling is enabled for each tile of the hardware. Second, our approach improves throughput further by an average of 27%. Although the static-order scheduling remains the same, our proposed approach, which balances the load on each tile improves throughput compared to SpiNeMap.

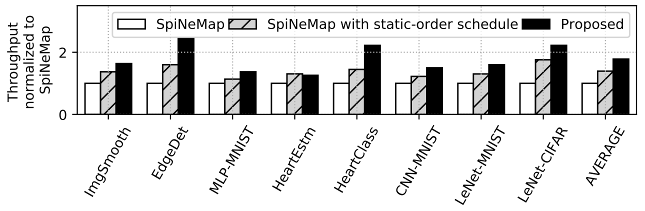

Figure 14. Throughput, normalized to SpiNeMap.

7.3 Compilation Time

Figure 15 reports the fraction of total compilation time of each of our application using our proposed approach for the DYNAP-SE hardware, distributed into time to bind clusters to tiles and the time to construct static-order schedule on each tile. The number on each bar reports the absolute time in ms to compile these applications on the DYNAP-SE hardware. We observe that the time consumed to create static-order schedule on each tile is on average 75% of the total time to compile these applications on the hardware. For some

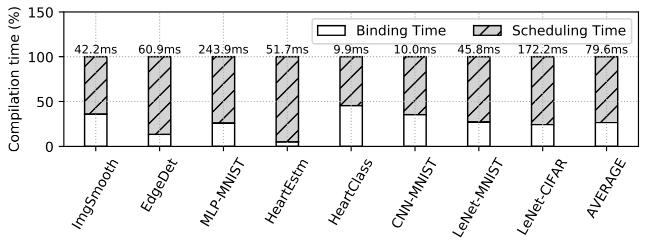

Figure 15. Fraction of total compile time, distributed into binding and scheduling time.

7.4 Resource Utilization

Table 2 reports the utilization of hardware resources (tile resources, buffer size, connections, and input and output bandwidth) on the DYNAP-SE neuromorphic hardware for each application. The average utilization of hardware resources are 92.5% for the crossbar IOs on each tile, 9.0% for buffer space, 42.6% for connections, and 15% for input and output tile bandwidth. Since we perform hardware-aware analysis, resource utilization never exceeds 100%.

These results illustrate that our approach can also be used for designing neuromorphic hardware, not only in terms of number of tiles, but all other resources such as buffer space, connections, and input and output bandwidth.

7.5 Performance Scalability

Figure 16 reports throughput of each of our application for our proposed approach normalized to SpiNeMap. We compare throughput obtained on three hardware models: 4 tiles (baseline), 9 tiles (arranged in a 3×3), and 16 tiles (arranged in a 4×4). We make the following two observations.

Application	Utilization (%)				
	Tile	Buffer	Connections	Bandwidth	
				Input	Output
ImgSmooth	87.5	8.39844	37.5	17.0898	17.0898
EdgeDet	87.5	11.2305	68.75	22.7864	22.7865
MLP-MNIST	81.25	9.375	46.875	22.7865	22.7864
HeartEstm	96.875	9.61914	62.5	4.70197	4.70197

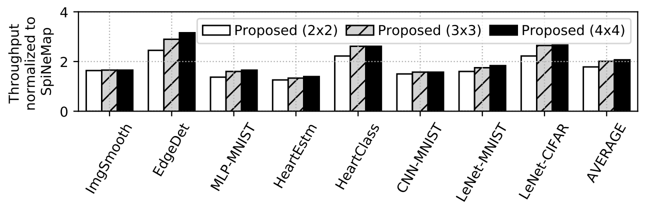

Figure 16. Throughput, normalized to SpiNeMap.

First, throughput generally increases with increasing the number of neuromorphic tiles. With 9 and 16 tiles, the average throughput is higher than the baseline configuration by 11% and 15%, respectively. This improvement is because with more tiles in the hardware, a tile is shared among fewer clusters, which improves throughput. Second, for applications such as ImgSmooth, four tiles are sufficient to map the application. There is therefore no significant improvement in throughput when the number of tiles in the hardware is increased. For other applications such as EdgeDet, throughput increases with increase in the number of tiles.

7.6 Run-time Performance

Figure 17 reports throughput of each of our application

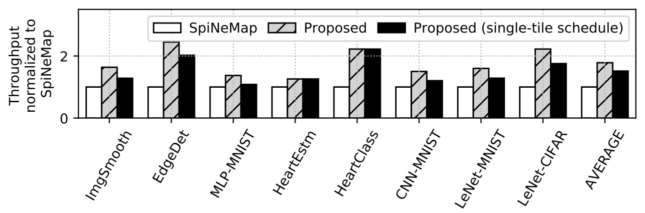

Figure 17. Throughput, normalized to SpiNeMap.

First, throughput obtained at run-time from a single-tile static-order schedule is on average 15% lower than the case when schedules are constructed independently — that is, by using our design-time analysis method. This verifies Lemma 1. Second, for some application such as HeartEstm and HeratClass, throughput obtained at run time is exactly the same as that obtained at design time. Third, throughput at run time is still higher than SpiNeMap by an average of 51.4%.

Table 3 compares the compilation time for each application using our approach at design time against that at run time. On average, the run-time approach achieves an average 67.5% reduction in compilation time. This is due to the reduction of schedule construction overhead using the single-tile static-order schedule along with the self-timed execution approach.

Application	Compilation time (ms)		Application	Compilation time (ms)	
	Design-time	Run-time		Design-time	Run-time
ImgSmooth	42.203	17.3685	EdgeDet	60.872	12.1011
MLP-CNN	243.862	77.0835	HeartEstm	51.73	6.6567
HeartClass	9.939	5.0043	CNN-MNIST	10.005	4.072
LeNet-MNIST	45.829	15.1356	LeNet-CIFAR	172.203	52.635

Table 3. Compilation time at design time vs. at run time.

7.7 Accuracy Impact

Table 1 column 6 reports the model accuracy for each of the evaluated applications obtained on DYNAP-SE. The accuracy is within 5% of the top accuracy reported in literature. We observe that there is no accuracy loss for applications that are built directly in spiking domain (EdgeDet and ImgSmooth). For all other applications converted from the analog domain, accuracy loss is less than 5% of the accuracy reported in literature. This loss is attributed to the N2D2 tool.

8 Related Works

8.1 State-of-the-art Neuromorphic Hardware

In SpiNNaker [34], each ARM9 core can implement multiple neuron functionality, with the local memory serving as the synaptic storage. TrueNorth is a million-neuron digital CMOS chip from IBM [30]. The chip has 4,096 tiles, with each tile hosting 12.75 kilobytes of local SRAM memory to store the synapses. Loihi is a 128-tile neuromorphic chip from Intel, with each tile having 1,024 spiking neurons and 2 Mb of SRAM to store synapses[28]. There are also many other neuromorphic chips such as Neurogrid [10], BrainScaleS [57], Braindrop [53], and ODIN [33]. These architectures are similar to DYNAP-SE [51], which we evaluate.

8.2 Mapping SNNs to Neuromorphic Hardware

Corelet is a proprietary tool from IBM to map SNNs to TrueNorth [2]. PACMAN is used to map SNNs to SpiNNaker [35]. Beyond these hardware-specific tools, there are also general-purpose ones. For instance, PyNN [29] is used to map SNNs on Loihi, BrainScaleS, SpiNNaker, and Neurogrid by balancing the load on each tile. The PSO-based technique developed in Das et al. is used to map SNNs to a hardware, reducing the energy consumption between tiles[27]. SpiNeMap reduces the communication between tiles [8]. PYCARL is proposed to perform hardware-software co-simulation of SNNs [5]. We compare our approach against PYCARL and SpiNeMap, and found it to perform significantly better.

There are also other approaches that use a single large crossbar to map SNNs [3, 46, 68–71].

8.3 Non-volatile Memory

Recently, NVMs are used to lower the energy consumption of von-Neumann computing [59–61] and neuromorphic computing. To this end, Ramasubramanian et al. use STT-MRAM [55], Burr et al. use PCM [16], and Mallik et al. use OxRAM [49] to design neuromorphic tiles.

8.4 Similar Concept in Related Domain

SDFGs are widely used for predictable mapping of applications to multiprocessor systems. Numerous approaches to throughput analysis of SDFGs have been previously proposed [20, 65, 67, 73]. Bonfietti et al. evaluated mappings of SDFG to multiprocessor system, maximizing the throughput [14]. Stemmer et al. propose to use probabilistic analysis to allocate and schedule SDFGs on multiprocessor systems [64]. Das et al. evaluated the fault-tolerant mapping of SDFGs to multiprocessor systems [23–25]. Recently, SDFG-based analysis is also proposed for analyzing machine learning applications [4, 7, 17, 22, 39]. However, none of these approaches address application analysis with limited hardware resources, both at design-time and at run-time.

9 Conclusions

We introduce an approach for predictable compilation of SNN-based applications on state-of-the-art neuromorphic hardware. Prior works have only addressed design-time mapping, considering unlimited resources in the underlying hardware. These approaches present significant limitations when used to compile and execute machine learning applications on a resource-constrained hardware. Our approach makes three contributions. First, we propose a technique to generate neuron and synapse clusters, where each cluster can fit on to the resources of a tile of the hardware. Second, we exploit the rich semantics of SDFG to model SNN clusters and analyze performance on a resource-constrained hardware. Finally, we propose a scheduling approach based on self-timed execution to reduce the time to compile and admit an application to a hardware at run-time, adjusting to dynamic resource availability. We conducted experiments with standard SNN-based applications and demonstrate a significant increase in performance over current practices.

10 Acknowledgments

This work is supported by 1) the National Science Foundation Award CCF-1937419 (RTML: Small: Design of System Software to Facilitate Real-Time Neuromorphic Computing) and 2) the National Science Foundation Faculty Early Career Development Award CCF-1942697 (CAREER: Facilitating Dependable Neuromorphic Computing: Vision, Architecture, and Impact on Programmability).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–283.

[2] Arnon Amir, Pallab Datta, William P Risk, Andrew S Cassidy, Jeffrey A Kusnitz, Steve K Esser, Alexander Andreopoulos, Theodore M Wong, Myron Flickner, Rodrigo Alvarez-Icaza, et al. 2013. Cognitive computing programming paradigm: A corelet language for composing networks of neurosynaptic cores. In *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–10.

[3] Aayush Ankit, Abhronil Sengupta, and Kaushik Roy. 2018. Neuromorphic Computing Across the Stack: Devices, Circuits and Architectures. In *Workshop on Signal Processing Systems*. IEEE, 1–6.

[4] Marco Bacis, Giuseppe Natale, Emanuele Del Sozzo, and Marco Domenico Santambrogio. 2017. A pipelined and scalable dataflow implementation of convolutional neural networks on FPGA. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 90–97.

[5] Adarsha Balaji, Prathyusha Adiraju, Hirak J Kashyap, Anup Das, Jeffrey L Krichmar, Nikil D Dutt, and Francky Catthoor. 2020. PyCARL: A PyNN Interface for Hardware-Software Co-Simulation of Spiking Neural Network. In *International Joint Conference on Neural Networks (IJCNN)*. IEEE.

[6] Adarsha Balaji, Federico Corradi, Anup Das, Sandeep Pande, Siebren Schaafsma, and Francky Catthoor. 2018. Power-accuracy trade-offs for heartbeat classification on neural networks hardware. *Journal of Low Power Electronics (JOLPE)* 14, 4 (2018), 508–519.

[7] Adarsha Balaji and Anup Das. 2019. A Framework for the Analysis of Throughput-Constraints of SNNs on Neuromorphic Hardware. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 193–196.

[8] Adarsha Balaji, Anup Das, Yuefeng Wu, Khanh Huynh, Francesco G Dell'Anna, Giacomo Indiveri, Jeffrey L Krichmar, Nikil D Dutt, Siebren Schaafsma, and Francky Catthoor. 2019. Mapping spiking neural networks to neuromorphic hardware. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 1 (2019), 76–86.

[9] Adarsha Balaji, Shihao Song, Anup Das, Nikil Dutt, Jeff Krichmar, Nagarajan Kandasamy, and Francky Catthoor. 2019. A Framework to Explore Workload-Specific Performance and Lifetime Trade-offs in Neuromorphic Computing. *IEEE Computer Architecture Letters* 18, 2 (2019), 149–152.

[10] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V Arthur, Paul A Merolla, and Kwabena Boahen. 2014. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proc. IEEE* 102, 5 (2014), 699–716.

[11] O Bichler, D Briand, V Gacoin, B Bertelone, T Allenet, and JC Thiele. 2017. N2D2-Neural Network Design & Deployment. *https://github.com/CEA-LIST/N2D2* (2017).

[12] J. Blazewicz. 1976. Scheduling dependent tasks with different arrival times to meet deadlines. In *Proceedings of the International Workshop organized by the Commision of the European Communities on Modelling and Performance Evaluation of Computer Systems*. North-Holland Publishing Co., 57–65.

[13] Kwabena A Boahen. 1998. Communicating neuronal ensembles between neuromorphic chips. In *Neuromorphic systems engineering*. Springer.

[14] Alessio Bonfietti, Michele Lombardi, Michela Milano, and Luca Benini. 2013. Maximum-throughput mapping of SDFGs on multi-core SoC platforms. *J. Parallel and Distrib. Comput.* 73, 10 (2013), 1337–1350.

[15] Romain Brette. 2015. Philosophy of the spike: rate-based vs. spike-based theories of the brain. *Frontiers in Systems Neuroscience* 9 (2015), 151.

[16] Geoffrey W. Burr, Robert M. Shelby, Abu Sebastian, Sangbum Kim, Seyoung Kim, Severin Sidler, Kumar Virwani, Masatoshi Ishii, Pritish Narayanan, Alessandro Fumarola, Lucas L. Sanches, Irem Boybat, Manuel Le Gallo, Kibong Moon, Jiyoo Woo, Hyunsang Hwang, and Yusuf Leblebici. 2017. Neuromorphic computing using non-volatile memory. *Advances in Physics: X* 2, 1 (2017), 89–124.

[17] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2017. Using dataflow to optimize energy efficiency of deep neural network accelerators. *IEEE Micro* 37, 3 (2017), 12–21.

[18] T-S. Chou, H J Kashyap, J Xing, S Listopad, Emily L Rounds, M Beyeler, N Dutt, and J L Krichmar. 2018. CARLsim 4: An open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters. In *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1158–1165.

[19] Jason Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *Design Automation Conference (DAC)*. ACM, 433–438.

[20] Morteza Damavandpeyma, Sander Stuijk, Twan Basten, Marc Geilen, and Henk Corporaal. 2012. Modeling static-order schedules in synchronous dataflow graphs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 775–780.

[21] Yang Dan and Mu-ming Poo. 2004. Spike timing-dependent plasticity of neural circuits. *Neuron* 44, 1 (2004), 23–30.

[22] Anup Das and Akash Kumar. 2018. Dataflow-Based Mapping of Spiking Neural Networks on Neuromorphic Hardware. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 419–422.

[23] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. 2014. Communication and migration energy aware task mapping for reliable multiprocessor systems. *Future Generation Computer Systems* 30 (2014), 216–228.

[24] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. 2014. Energy-aware task mapping and scheduling for reliable embedded computing systems. *ACM Trans. Embedded Comput. Syst.* 13, 2s (2014), 72:1–72:27. https://doi.org/10.1145/2544375.2544392

[25] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. 2015. Reliability and energy-aware mapping and scheduling of multimedia applications on multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems* 27, 3 (2015), 869–884.

[26] Anup Das, Paruthi Pradhapan, Willemijn Groenendaal, Prathyusha Adiraju, Raj Thilak Rajan, Francky Catthoor, Siebren Schaafsma, Jeffrey L Krichmar, Nikil Dutt, and Chris Van Hoof. 2018. Unsupervised heart-rate estimation in wearables with liquid states and a probabilistic readout. *Neural Networks* 99 (2018), 134–147.

[27] Anup Das, Yuefeng Wu, Khanh Huynh, Francesco Dell'Anna, Francky Catthoor, and Siebren Schaafsma. 2018. Mapping of local and global synapses on spiking neuromorphic hardware. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1217–1222.

[28] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. 2018. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 1 (2018), 82–99.

[29] Andrew P Davison, Daniel Brüderle, Jochen M Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. 2009. PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics* 2 (2009), 11.

[30] Michael V DeBole, Brian Taba, Arnon Amir, Filipp Akopyan, Alexander Andreopoulos, William P Risk, Jeff Kusnitz, Carlos Ortega Otero, Tapan K Nayak, Rathinakumar Appuswamy, et al. 2019. TrueNorth: Accelerating from zero to 64 million neurons in 10 years. *Computer* 52, 5 (2019), 20–29.

[31] Li Deng. 2012. The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[32] Peter U Diehl and Matthew Cook. 2015. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in Computational Neuroscience* 9 (2015), 99.

[33] Charlotte Frenkel, Martin Lefebvre, Jean-Didier Legat, and David Bol. 2018. A 0.086-mm^2 12.7-pJ/SOP 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28-nm CMOS. *IEEE Transactions on Biomedical Circuits and Systems* 13, 1 (2018), 145–158.

[34] Steve B Furber, Francesco Galluppi, Steve Temple, and Luis A Plana. 2014. The SpiNNaker project. *Proc. IEEE* 102, 5 (2014), 652–665.

[35] Francesco Galluppi, Xavier Lagorce, Evangelos Stromatias, Michael Pfeiffer, Luis A Plana, Steve B Furber, and Ryad B Benosman. 2015. A framework for plasticity implementation on the SpiNNaker neural architecture. *Frontiers in Neuroscience* 8 (2015), 429.

[36] Daniele Garbin, Elisa Vianello, Olivier Bichler, Quentin Rafhay, Christian Gamrat, Gérard Ghibaudo, Barbara DeSalvo, and Luca Perniola. 2015. HfO 2-based OxRAM devices as synapses for convolutional neural networks. *IEEE Transactions on Electron Devices* 62, 8 (2015), 2494–2501.

[37] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems* 29, 7 (2013), 1645–1660.

[38] Bernd Heidergott, Geert Jan Olsder, and Jacob Van Der Woude. 2014. *Max Plus at work: modeling and analysis of synchronized systems: a course on Max-Plus algebra and its applications.* Princeton University Press.

[39] Hyesun Hong, Hyunok Oh, and Soonhoi Ha. 2017. Hierarchical dataflow modeling of iterative applications. In *Design Automation Conference (DAC).* ACM, 1–6.

[40] Yu Ji, Youhui Zhang, Wenguang Chen, and Yuan Xie. 2018. Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* ACM, 448–460.

[41] Yu Ji, YouHui Zhang, ShuangChen Li, Ping Chi, CiHang Jiang, Peng Qu, Yuan Xie, and WenGuang Chen. 2016. NEUTRAMS: Neural network transformation and co-design under neuromorphic hardware constraints. In *International Symposium on Microarchitecture (MICRO).* ACM, 1–13.

[42] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2014. The CIFAR-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html* 55 (2014).

[43] Yann LeCun et al. 2015. LeNet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet* 20 (2015), 5.

[44] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436.

[45] E.A. Lee and D.G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[46] Matthew Kay Fei Lee, Yingnan Cui, Thannirmalai Somu, Tao Luo, Jun Zhou, Wai Teng Tang, Weng-Fai Wong, and Rick Siow Mong Goh. 2019. A system-level simulator for RRAM-based neuromorphic computing chips. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 4 (2019), 64.

[47] Wolfgang Maass. 1997. Networks of spiking neurons: the third generation of neural network models. *Neural Networks* 10, 9 (1997).

[48] Wolfgang Maass, Thomas Natschläger, and Henry Markram. 2002. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation* 14 (2002), 2531–2560.

[49] A Mallik, D Garbin, A Fantini, D Rodopoulos, R Degraeve, J Stuijt, AK Das, S Schaafsma, P Debacker, G Donadio, et al. 2017. Design-technology co-optimization for OxRRAM-based synaptic processing unit. In *Symposium on VLSI Technology.* IEEE, T178–T179.

[50] George B Moody, Roger G Mark, and Ary L Goldberger. 2001. PhysioNet: a web-based resource for the study of physiologic signals. *IEEE Engineering in Medicine and Biology Magazine* 20, 3 (2001), 70–75.

[51] S. Moradi, N. Qiao, F. Stefanini, and G. Indiveri. 2018. A Scalable Multicore Architecture With Heterogeneous Memory Structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs). *IEEE Transactions on Biomedical Circuits and Systems* 12, 1 (2018), 106–122.

[52] Orlando M Moreira and Marco JG Bekooij. 2007. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing* 2007 (2007), 1–14.

[53] Alexander Neckar, Sam Fok, Ben V Benjamin, Terrence C Stewart, Nick N Oza, Aaron R Voelker, Chris Eliasmith, Rajit Manohar, and Kwabena Boahen. 2018. Braindrop: A mixed-signal neuromorphic architecture with a dynamical systems-based programming model. *Proc. IEEE* 107, 1 (2018), 144–164.

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS).* 8024–8035.

[55] Shankar Ganesh Ramasubramanian, Rangharajan Venkatesan, Mrigank Sharad, Kaushik Roy, and Anand Raghunathan. 2014. SPINDLE: SPINtronic deep learning engine for large-scale neuromorphic computing. In *International Symposium on Low Power Electronics and Design (ISLPED).* ACM, 15–20.

[56] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2019. Mlperf inference benchmark. *arXiv preprint arXiv:1911.02549* (2019).

[57] Johannes Schemmel, Andreas Grübl, Stephan Hartmann, Alexander Kononov, Christian Mayr, Karlheinz Meier, Sebastian Millner, Johannes Partzsch, Stefan Schiefer, Stefan Scholze, et al. 2012. Live demonstration: A scaled-down version of the brainscales wafer-scale neuromorphic system. In *International Symposium on Circuits and Systems (ISCAS).* IEEE, 702–702.

[58] Shihao Song, Adarsha Balaji, Anup Das, Nagarajan Kandasamy, and James Shackleford. 2020. *DFSynthesizer: A tool for data-flow based analysis of SNNs on neuromorphic hardware.* Retrieved April, 2020 from https://github.com/drexel-DISCO/DFSynthesizer

[59] Shihao Song, Anup Das, and Nagarajan Kandasamy. 2020. Exploiting Inter- and Intra-Memory Asymmetries for Data Mapping in Hybrid Tiered-Memories. In *ACM International Symposium on Memory Management (ISMM).*

[60] Shihao Song, Anup Das, Onur Mutlu, and Nagarajan Kandasamy. 2019. Enabling and Exploiting Partition-Level Parallelism (PALP) in Phase Change Memories. *ACM Transactions in Embedded Computing (TECS)* 5s (2019), 1–25.

[61] Shihao Song, Anup Das, Onur Mutlu, and Nagarajan Kandasamy. 2020. Improving Phase Change Memory Performance with Data Content Aware Access. In *ACM International Symposium on Memory Management (ISMM).*

[62] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. 2014. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806* (2014).

[63] S. Sriram and S.S. Bhattacharyya. 2000. *Embedded Multiprocessors; Scheduling and Synchronization.* Marcel Dekker.

[64] Ralf Stemmer, Hai-Dang Vu, Kim Grüttner, Sébastien Le Nours, Wolfgang Nebel, and Sébastien Pillement. 2020. Towards Probabilistic Timing Analysis for SDFGs on Tile Based Heterogeneous MPSoCs. In *Euromicro Conference on Real-Time Systems.*

[65] Sander Stuijk, Twan Basten, MCW Geilen, and Henk Corporaal. 2007. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Design Automation Conference (DAC).* ACM, 777–782.

[66] S. Stuijk, M. Geilen, and T. Basten. 2006. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow

graphs. In *Design Automation Conference (DAC)*. ACM, 899–904.

[67] Sander Stuijk, Marc Geilen, and Twan Basten. 2006. Exploring tradeoffs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Design Automation Conference (DAC)*. ACM, 899–904.

[68] Wei Wen, Chi-Ruo Wu, Xiaofang Hu, Beiye Liu, Tsung-Yi Ho, Xin Li, and Yiran Chen. 2015. An EDA framework for large scale hybrid neuromorphic computing systems. In *Design Automation Conference (DAC)*. ACM, 1–6.

[69] Parami Wijesinghe, Aayush Ankit, Abhronil Sengupta, and Kaushik Roy. 2018. An all-memristor deep spiking neural computing system: A step toward realizing the low-power stochastic brain. *IEEE Transactions on Emerging Topics in Computational Intelligence* 2, 5 (2018), 345–358.

[70] Qiangfei Xia and J Joshua Yang. 2019. Memristive crossbar arrays for brain-inspired computing. *Nature Materials* 18, 4 (2019), 309.

[71] Xinjiang Zhang, Anping Huang, Qi Hu, Zhisong Xiao, and Paul K Chu. 2018. Neuromorphic computing with memristor crossbar. *Physica Status Solidi (a)* 215, 13 (2018), 1700875.

[72] Zhiru Zhang and Bin Liu. 2013. SDC-based modulo scheduling for pipeline synthesis. In *International Conference on Computer-Aided Design (ICCAD)*. IEEE, 211–218.

[73] Xue-Yang Zhu, Marc Geilen, Twan Basten, and Sander Stuijk. 2012. Static rate-optimal scheduling of multirate DSP algorithms via retiming and unfolding. In *Real Time and Embedded Technology and Applications Symposium*. IEEE, 109–118.