DFSynthesizer: Dataflow-based Synthesis of Spiking Neural Networks to Neuromorphic Hardware

SHIHAO SONG, HARRY CHONG, ADARSHA BALAJI, ANUP DAS,
JAMES SHACKLEFORD, and NAGARAJAN KANDASAMY, Drexel University, USA

Spiking Neural Networks (SNNs) are an emerging computation model that uses event-driven activation and bio-inspired learning algorithms. SNN-based machine learning programs are typically executed on tile-based neuromorphic hardware platforms, where each tile consists of a computation unit called a crossbar, which maps neurons and synapses of the program. However, synthesizing such programs on an off-the-shelf neuromorphic hardware is challenging. This is because of the inherent resource and latency limitations of the hardware, which impact both model performance, e.g., accuracy, and hardware performance, e.g., throughput. We propose DFSynthesizer, an end-to-end framework for synthesizing SNN-based machine learning programs to neuromorphic hardware. The proposed framework works in four steps. First, it analyzes a machine learning program and generates SNN workload using representative data. Second, it partitions the SNN workload and generates clusters that fit on crossbars of the target neuromorphic hardware. Third, it exploits the rich semantics of the Synchronous Dataflow Graph (SDFG) to represent a clustered SNN program, allowing for performance analysis in terms of key hardware constraints such as number of crossbars, dimension of each crossbar, buffer space on tiles, and tile communication bandwidth. Finally, it uses a novel scheduling algorithm to execute clusters on crossbars of the hardware, guaranteeing hardware performance. We evaluate DFSynthesizer with 10 commonly used machine learning programs. Our results demonstrate that DFSynthesizer provides a much tighter performance guarantee compared to current mapping approaches.

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

Additional Key Words and Phrases: Neuromorphic computing, Synchronous Dataflow Graph (SDFG), machine learning, Spiking Neural Networks (SNN), compiler, mapping

ACM Reference format:
Shihao Song, Harry Chong, Adarsha Balaji, Anup Das, James Shackleford, and Nagarajan Kandasamy. 2022. DFSynthesizer: Dataflow-based Synthesis of Spiking Neural Networks to Neuromorphic Hardware. *ACM Trans. Embedd. Comput. Syst.* 21, 3, Article 27 (May 2022), 35 pages.
https://doi.org/10.1145/3479156

This work is supported by (1) the US DOE CAREER Award DE-SC0022014 (Architecting the Hardware-Software Interface for Neuromorphic Computers), (2) the National Science Foundation Award CCF-1937419 (RTML: Small: Design of System Software to Facilitate Real-Time Neuromorphic Computing), and (3) the National Science Foundation Faculty Early Career Development Award CCF-1942697 (CAREER: Facilitating Dependable Neuromorphic Computing: Vision, Architecture, and Impact on Programmability).
Authors' address: S. Song, H. Chong, A. Balaji, A. Das, J. Shackleford, and N. Kandasamy, Drexel University, 3141 Chestnut Street, Philadelphia, PA, 19104; emails: ss3695@dragons.drexel.edu, hjc39@dragons.drexel.edu, ab3586@drexel.edu, anup.das@drexel.edu, jas64@drexel.edu, nk78@drexel.edu.

1539-9087/2022/05-ART27 $15.00
https://doi.org/10.1145/3479156

1 INTRODUCTION

The **Spiking Neural Network (SNN)** is an emerging computing model that uses spike-based computations and bio-inspired learning algorithms [71]. In an SNN, pre-synaptic neurons communicate information encoded in spike trains to post-synaptic neurons, via synapses (see Figure 1). Performance, e.g., accuracy of an SNN model, is assessed in terms of the **inter-spike interval (ISI)**, which is defined as the inverse of the mean firing rate of the neurons.

SNNs are typically executed on neuromorphic hardware platforms such as DYNAP-SE [73], TrueNorth [47], and Loihi [45]. These hardware platforms are designed as a tile-based architecture with a shared, hierarchical interconnect to facilitate inter-tile communication (see Figure 2) [25]. Each tile consists of a crossbar for mapping neurons and synapses, and input and output buffer space for communicating spikes over the interconnect. A crossbar is a 2D organization of horizontal and vertical wires, where the horizontal wires are connected to pre-synaptic neurons while the vertical wires are connected to post-synaptic neurons. **Non-Volatile Memory (NVM)** cells are placed at the crosspoints of each crossbar to implement storage of synaptic weights [24, 72].[1]

Energy consumed by neuromorphic hardware can be several orders of magnitude lower than a conventional machine learning accelerator such as Eyeriss [26]. This is due to low-power VLSI implementation of analog neurons [62], low-power and high-density NVM-based synaptic storage [24], and distributed computing and storage architecture using crossbars. Given these advantages, a neuromorphic hardware can implement machine learning tasks for power-constrained platforms such as embedded systems and edge nodes of the **Internet of Things (IoT)** [5].

Unlike conventional von Neumann computing systems, where CPUs compute by exchanging data centrally from the main memory, synthesizing, i.e., compiling and mapping a machine learning program on a neuromorphic hardware, is challenging. This is because in a neuromorphic hardware, computation units (i.e., the neurons) and storage units (i.e., the synapses) are distributed within the hardware as crossbars. It is therefore important to properly partition a large SNN model such that it can be mapped efficiently to the underlying resources. Additionally, each crossbar also presents limitations on how many pre-synaptic connections are allowed per post-synaptic neuron, and how much buffer space is available to send and receive spikes over the interconnect. These hardware limitations impact both model accuracy and hardware performance such as throughput, latency, and energy consumption.

We develop **DFSynthesizer**, a systematic and end-to-end framework to analyze and map machine learning programs to state-of-the-art neuromorphic hardware, while guaranteeing performance. Following are our key **contributions**:[2]

- **Contribution 1.** We present an approach to analyze machine learning programs and generate SNN workload using representative data. Our framework allows workload generation with only a modest impact on model performance.
- **Contribution 2.** We present an approach to decompose and partition complex SNN workloads and generate clusters of neurons and synapses such that each cluster can fit onto the resources of a crossbar in the hardware.
- **Contribution 3.** We exploit the rich semantics of **Synchronous Dataflow Graphs (SDFGs)** [69] to represent clustered SNN programs. This allows for the SNN's performance, e.g., throughput, to be estimated on the hardware as a function of key properties such as number of crossbars, dimension of crossbars, buffer space on tiles, and tile communication bandwidth.

[1] Beyond neuromorphic computing, NVMs are also used as main memory for conventional computing using shared-memory computers [85, 86, 88–90].

[2] Contributions 2, 3, and 4 appeared in our prior work [83]. This work introduces contributions 1, 5, and 6.

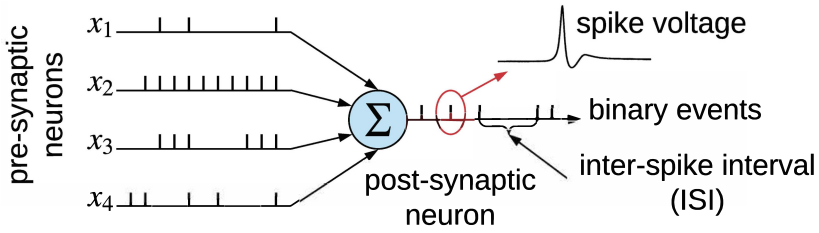

Fig. 1. Integration of spike trains at the post-synaptic neuron from four pre-synaptic neurons in a Spiking Neural Network (SNN). Each spike is a voltage waveform of time duration to the order of ms.

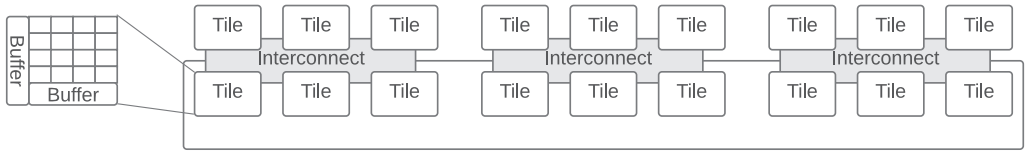

Fig. 2. A tile-based neuromorphic architecture [25], which is representative of many neuromorphic platforms such as DYNAP-SE [73], TrueNorth [47], and Loihi [45].

- **Contribution 4.** We develop a novel scheduling algorithm based on Self-Timed Execution for executing clusters on crossbars of a neuromorphic hardware, providing performance guarantee in scenarios with dynamic resource availability.
- **Contribution 5.** We propose a design-space exploration framework incorporating DFSynthesizer that allows the Pareto space of different SNN mappings to hardware to be explored while considering other hardware metrics such as energy, latency, and reliability.
- **Contribution 6.** We evaluate DFSynthesizer using 10 machine learning programs that are representative of the three most commonly used neural network classes—**convolutional neural network (CNN)**, **multi-layer perceptron (MLP)**, and **recurrent neural network (RNN)**.

2 SCOPE AND HIGH-LEVEL OVERVIEW OF DFSYNTHESIZER

DFSynthesizer is developed for supervised machine learning approaches, where a machine learning model is first trained using representative data from the field. Machine learning inference refers to generating output from the trained model by feeding live data. To improve energy efficiency, the inference is performed on a neuromorphic hardware. Once deployed on the hardware, the model is expected to perform inference in real time on a continuous basis from data collected using sensors.[3] Therefore, a key performance metric for neuromorphic hardware performing real-time inference is throughput, defined as the number of frames processed per unit time, where a frame is defined as an individual image (for image-based models) or a window of time-series data.[4]

Figure 3 illustrates the proposed end-to-end framework of DFSynthesizer, which synthesizes, i.e., compiles, and maps a machine learning program to a neuromorphic hardware in four steps. First, it analyzes a machine learning program written in a high-level language such as Python and C/C++ to generate SNN workload (Section 3). Second, it compiles SNN workloads to an

[3]Camera sensors are used for image classification models, e.g., LeNet, AlexNet, and VGG16, while electrocardiogram sensors are used for heart rate classification and estimation models. See our evaluation setup in Section 7.
[4]By maximizing the throughput, DFSynthesizer minimizes the time to process individual frames using the neuromorphic inference hardware, which makes DFSynthesizer applicable to both real-time and non-real-time applications.

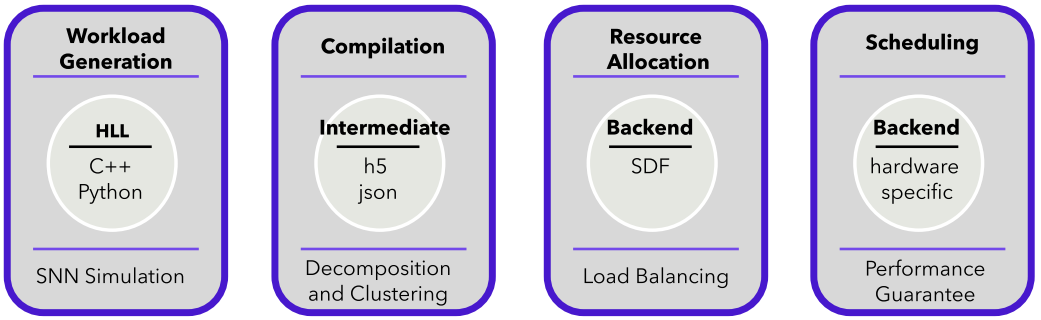

Fig. 3. High-level overview of DFSynthesizer. A machine learning program is analyzed and mapped to the hardware using the proposed four-step methodology.

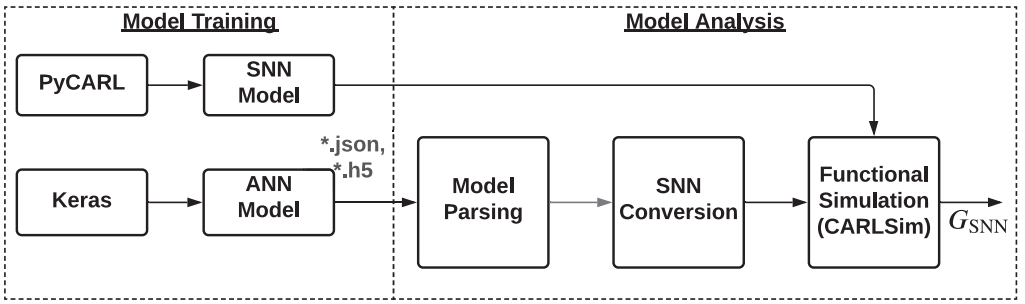

Fig. 4. Workflow of the workload generation step of DFSynthesizer.

intermediate representation format (h5 and json), performing spatial decomposition and clustering to fit onto the resources of a crossbar (Section 4). Third, it uses **Synchronous Dataflow Graph (SDF)** to represent clustered SNN (in XML representation), allocating resources to the clusters considering hardware resource constraints (Section 5). Finally, it schedules the SDF representation of a clustered SNN to the hardware crossbars, guaranteeing performance (Section 6).

3 PROGRAM ANALYSIS AND WORKLOAD GENERATION

In this step, a machine learning program is analyzed to generate its workload. In the following, we discuss the steps involved in the workload generation.

3.1 Workflow for Workload Generation

Figure 4 summarizes the workflow of the workload generation step of DFSynthesizer, where a machine learning program is analyzed to generate its workload, which is then used to map the application to a neuromorphic hardware.

DFSynthesizer can incorporate both **Artificial Neural Networks (ANNs)** and SNNs in its workflow. At a high level, the proposed workflow consists of a model training component followed by model analysis. In the following, we elaborate on these components.

3.2 Model Training

3.2.1 Training Artificial Neural Networks. DFSynthesizer's frontend is integrated with Keras [57], which is used to define a model and train it on a database. Keras utilizes the Tensorflow backend [1]. DFSynthesizer also supports other frameworks such as PyTorch [76]. To demonstrate

the capabilities of DFSynthesizer, we evaluate it with three CNN architectures: (1) LeNet [68], trained on the MNIST handwritten digit dataset [49]; (2) AlexNet [66], trained on the ImageNet dataset [48]; and (3) VGGNet [82], trained on the ImageNet dataset. These models are derived from the MLPerf [78] dataset and instantiated in Keras. We use a Lambda workstation with two GPUs (see our evaluation setup in Section 7) to train these models.

3.2.2 Training Spiking Neural Networks. DFSynthesizer's frontend supports training SNN models using PyCARL [7], a Python frontend to CARLsim [28]. CARLsim facilitates SNN simulations using CPUs and multi-GPUs. PyCARL is designed to integrate with PyNN [46], which provides a common frontend to different SNN simulators with various degrees of neurobiological details. We use CARLsim for model training. CARLsim's support for built-in biologically realistic neuron, synapse, current, and emerging learning models and continuous integration and testing make it an easy-to-use and powerful simulator of biologically plausible SNN models. DFSynthesizer can also utilize other SNN simulators such as Brian [55], NEST [51], and NEURON [59] for model training.

3.3 Model Analysis

3.3.1 Model Parsing and Conversion. Unfortunately, ANN models cannot be executed directly on event-driven neuromorphic hardware platforms such as DYNAP-SE [73], TrueNorth [47], and Loihi [45]. Recently, many tools have been proposed to convert ANN operations to SNNs. Examples include Nengo [19], N2D2 [22], and SNNToolBox [79]. A common limitation of these toolboxes is that they are open-loop converters, meaning that the conversion is performed considering performance degradation only. In our prior work [8], we have proposed a closed-loop conversion mechanism, where the conversion of analog operations to spiking equivalent is performed considering the energy consumption on hardware. These conversion steps are briefly discussed below.[5]

(1) *ReLU Activation Functions:* This is implemented as the approximate firing rate of a **leaky integrate and fire (LIF)** neuron.
(2) *Bias:* A bias is represented as a constant input current to a neuron, the value of which is proportional to the bias of the neuron in the corresponding analog model.
(3) *Weight Normalization:* This is achieved by setting a factor λ to control the firing rate of spiking neurons.
(4) *Softmax:* To implement softmax, an external Poisson spike generator is used to generate spikes proportional to the weighted sum accumulated at each neuron.
(5) *Max and Average Pooling:* To implement max pooling, the neuron that fires first is considered to be the winning neuron, and therefore, its responses are forwarded to the next layer, suppressing the responses from other neurons in the pooling function. To implement average pooling, the average firing rate (obtained from total spike count) of the pooling neurons is forwarded to the next layer.

We have extended our framework with the following new functionalities to allow for the conversion of CNN architectures such as LeNet, AlexNet, and VGGNet to their spiking counterparts.

(1) *1-D Convolution:* The 1-D convolution is implemented to extract patterns from inputs in a single spatial dimension. A $1 \times n$ filter, called a kernel, slides over the input while computing the element-wise dot-product between the input and the kernel at each step.

[5]The conversion framework was introduced in [8] for converting the CNN-based HeartClass application to its equivalent SNN representation. We used this application to evaluate DFSynthesizer. Additionally, we have extended the conversion framework to add other key functionalities such as Layer Flattening, Concatenation, Binary Weight Activation, and Non-Zero Biases. These new functionalities allowed the conversion framework to convert state-of-the-art CNN architectures such as LeNet, AlexNet, and VGG16, which are used to evaluate DFSynthesizer.

Table 1. Accuracy Impact Due to Conversion of Three State-of-the-art CNN Models to Their SNN Equivalent

Application	Top-1 Accuracy (%)		Application	Top-1 Accuracy (%)		Application	Top-1 Accuracy (%)	
	Original	SNN		Original	SNN		Original	SNN
LeNet	94.98%	94.08%	AlexNet	74.1%	71.7%	VGG16	93.56%	91.62%

The original accuracy numbers are obtained by simulating these architectures in Keras [57] with Tensorflow backend [1]. The converted accuracy numbers reported in the columns marked "SNN" are obtained from CARLsim [28]. We use a multi-GPU machine to simulate these architectures using both Keras and CARLsim. See our evaluation framework in Section 7.

(2) *Residual Connections:* Residual connections are implemented to convert the residual block used in CNN models such as ResNet. Typically, the residual connection connects the input of the residual block directly to the output neurons of the block, with a synaptic weight of 1. This allows for the input to be directly propagated to the output of the residual block while skipping the operations performed within the block.
(3) *Flattening:* The flatten operation converts the 2-D output of the final pooling operation into a 1-D array. This allows for the output of the pooling operation to be fed as individual features into the decision-making regarding fully connected layers of the CNN model.
(4) *Concatenation:* The concatenation operation, also known as a merging operation, is used as a channel-wise integration of the features extracted from two or more layers into a single output.

Table 1 reports the accuracy impact due to the SNN conversion of three state-of-the-art supervised CNN models. These accuracy numbers are obtained from CARLsim [28], which allows functional simulation and performance estimation of SNN-based applications. We use these three converted CNN models to evaluate DFSynthesizer (see Section 7).

3.3.2 Workload Generation. The SNN model (or the converted ANN model) is analyzed in CARLsim to generate the following information:

- *Spike Data:* the exact spike times of all neurons in the SNN model. We let $spk(i)$ represent a list of spike times of the i^{th} neuron in the model.
- *Weight Data:* the synaptic strength of all synapses in the SNN model. We let $w(i, j)$ represent the synaptic weight of the connection between the i^{th} and j^{th} neurons in the SNN model.

The spike and weight data of a trained SNN form the **SNN workload**. Formally, an SNN workload is defined as follows.

Definition 1 (SNN Workload). An SNN Workload $\mathbf{G_{SNN}} = (\mathbf{N}, \mathbf{S}, \mathbf{W})$ is a directed graph consisting of a finite set N of neurons, a set S of spikes, and a set W of synapses between the neurons.

4 PROGRAM COMPILATION AND PERFORMANCE ESTIMATION

In this step, DFSynthesizer clusters a given machine learning model to map onto the crossbars of a neuromorphic hardware. To do so, we first introduce the system architecture and then discuss the clustering step needed to map applications to this architecture.

4.1 System Architecture

Figure 5 illustrates our system architecture. DFSynthesizer is designed for crossbar-based neuromorphic hardware designs as shown in Figure 2. This is representative of many recent neuromorphic designs [3, 25, 56, 61]. A machine learning model (ANN or SNN) is first analyzed to generate

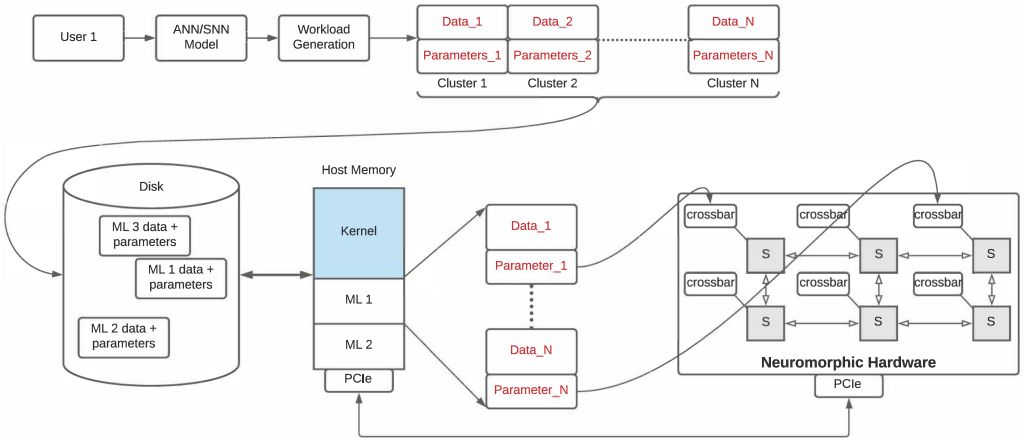

Fig. 5. Our system architecture, integrating a neuromorphic hardware. DFSynthesizer is designed for crossbar-based neuromorphic hardware [3, 25, 56, 61]. This is representative of many recent neuromorphic designs. To evaluate DFSynthesizer, we have configured our evaluation setup to model the DYNAP-SE hardware [73].

its workload (Section 3). This workload is then partitioned to generate clusters, where each cluster consists of a fraction of the neurons and synapses of the original machine learning model. The cluster workload is stored in a disk along with other machine learning workloads. To execute a specific workload on the neuromorphic hardware, it is first loaded into the host memory and then the clusters are programmed on to the crossbars of the hardware via the PCIe interface.[6]

In the remainder of this section, we describe the workload compilation step of DFSynthesizer, which consists of the following two design components: Workload Decomposition and Workload Clustering. We conclude this section by providing a dataflow modeling approach for clustered workloads and performance estimation using such model.

4.2 Workload Decomposition

We note that each $N \times N$ crossbar in a neuromorphic hardware can accommodate up to N pre-synaptic connections per post-synaptic neuron, with typical value of N set between 128 (in DYNAP-SE) and 256 (in TrueNorth). Figure 6 illustrates an example of mapping (a) one 4-input, (b) one 3-input, and (c) two 2-input neurons on a 4×4 crossbar. Unfortunately, neurons with more than four pre-synaptic connections per post-synaptic neuron cannot be mapped to the crossbar. In fact, in many complex machine learning models such as AlexNet and VGG16, the number of pre-synaptic connections per post-synaptic neuron is much higher than 128. Therefore, these neurons cannot be mapped to a 128×128 crossbar in DYNAP-SE.

To address the above limitation, we have previously proposed a spatial decomposition technique that exploits the firing principle of LIF neurons, decomposing each neuron with many pre-synaptic connections into a sequence of homogeneous **fanin-of-two (FIT)** neural units [14].

Figure 7 illustrates the spatial decomposition using a small example of a three-input neuron shown in Figure 7(a). We consider the mapping of this neuron to 2×2 crossbars. Since each crossbar can accommodate a maximum of two pre-synaptic connections per neuron, the example

[6]Although we illustrate the crossbars to be interconnected in a mesh-based architecture such as **Networks-on-Chip (NoC)** [20], DFSynthesizer can work with other interconnect types such as Segmented Bus [17].

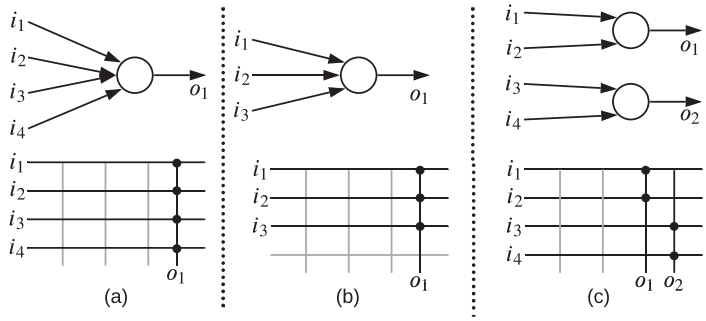

Fig. 6. Example mapping of (a) one 4-input, (b) one 3-input, and (c) two 2-input neurons on a 4×4 crossbar.

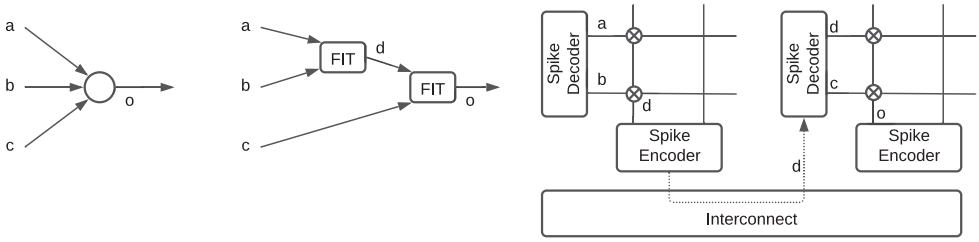

Fig. 7. Illustrating the decomposition of a three-input neuron (a) to a sequence of FIT neural units (b). The mapping of the FIT units to two 2×2 crossbars is shown in (c).

three-input neuron cannot be mapped to the crossbar directly. The most common solution is to eliminate a synaptic connection, which may lead to accuracy loss. Figure 7(b) illustrates the decomposition mechanism, where the three-input neuron is implemented using two FIT neural units connected in sequence as shown in Figure 7(b). Each FIT unit is similar to a two-input neuron and it exploits the leaky integrated behavior in hardware to maintain the functional equivalence between Figures 7(a) and 7(b).

For the sake of completeness, Figure 7(c) illustrates the mapping of the decomposed neuron utilizing two 2×2 crossbars. The functionality of the FIT neural units is implemented using the NVM cells of the two crossbars.

To describe the decomposition algorithm, we introduce the following notations. Let $n_i^1, n_i^2, \ldots, n_i^{m_i}$ be the m_i pre-synaptic connections of the neuron N_i. Let $F_i^1, F_i^2, \ldots, F_i^{m_i-1}$ be the $(m_i - 1)$ FIT neural units that are generated by spatially decomposing this neuron. The input of unit F_i^j denoted as $In(F_i^j)$ can be represented as

$$In(F_i^j) = \begin{cases} \{n_i^1, n_i^2\} & \text{for j = 1} \\ \{n_i^{j+1}, Out(F_i^{j-1})\} & \text{otherwise} \end{cases} \quad \forall j \in \{1, 2, \ldots, m_i - 1\}, \tag{1}$$

where $Out(F_i^j)$ is the output of the unit F_i^j. When decomposing a neuron, we note that the first FIT unit uses two of the original inputs of the original neuron. Subsequently, all other FIT units use one of the original inputs and the output of the preceding FIT units as shown in Figure 7(b).

Formally, a decomposed SNN graph is defined as follows.

Definition 2 (Decomposed SNN Graph). A decomposed SNN graph $\mathbf{G_{DSNN}} = (\mathbf{F}, \mathbf{L})$ is a directed graph consisting of a finite set $\mathbf{F}$ of FIT neural units and a finite set $\mathbf{L}$ of links between these units.

Algorithm 1 shows the pseudo-code of the spatial decomposition technique, which performs the graph transformation $G_{SNN} \rightarrow G_{DSNN}$. For each neuron N_i (line 1), a set of inputs to this neuron is obtained (line 2). The first FIT unit is formed using two inputs (line 3). This is in accordance with Equation (1) and Figure 7(b). The FIT unit is inserted into the decomposed graph G_{DSNN} (line 4). The algorithm then creates the other FIT units iteratively (lines 5–8) using Equation (1) and stores those units in G_{DSNN}. Finally, the graph G_{DSNN} is returned (line 10).

The overall complexity of this algorithm is calculated as follows. The outer for loop (lines 1–9) is executed for the neurons in the original graph G_{SNN}, i.e., for $|N|$ times. Within each iteration, the algorithm creates a total of $(|In(N_i)| - 1)$ FIT units, where $In(N_i)$ is the set of inputs of neuron N_i. Therefore, the algorithmic complexity is

$$\text{Complexity} = O\left(\sum_{i=1}^{|N|}\left(|In(N_i| - 1\right)\right) \approx O\left(|W|\right). \tag{2}$$

In deriving the final expression, we note that the input connections of all the neurons in the graph G_{SNN} are the edges W in the graph.

ALGORITHM 1: Spatial Decomposition of SNN Graph G_{SNN}

Input: $G_{SNN} = (\mathbf{N}, \mathbf{W})$
Output: $G_{DSNN} = (\mathbf{F}, \mathbf{L})$

1 **for** $N_i \in \mathbf{N}$ **do** /* for each node of G_{SNN} */
2 $\{n_i^1, n_i^2, \ldots, n_i^{m_i}\} = \texttt{In}(N_i)$; /* input links of N_i */
3 Create node F_i^1 with $\texttt{In}(F_i^1) = \{n_1, n_2\}$; /* first FIT unit */
4 G_{DSNN}.insert(F_i^1); /* insert the FIT neural unit u_1^i in G_{DSNN} */
5 **for** $j = 2; j < m_i; j++$ **do** /* remaining FIT units */
6 Create node F_i^j with $\texttt{in}(F_i^j) = \{n_i^{j+1}, F_i^{j-1}\}$;
7 G_{DSNN}.insert(F_i^j);
8 **end**
9 **end**
10 Return G_{DSNN}

4.3 Workload Clustering

The decomposed SNN graph is clustered such that each cluster is able to fit onto a crossbar. Figure 8 illustrates the concept using an example of a decomposed SNN graph shown in (❶). The nodes are the FIT neural units and the links are the synaptic connections. The number on a link represents the average number of spikes communicated between the source and destination FIT units for the representative training data. We consider the mapping of this decomposed SNN graph to a hardware with 2×2 crossbars. Since a crossbar in this hardware can only accommodate a maximum of two pre-synaptic connections, we partition the graph of (❶) into two partitions (shown in two different colors) in (❷). These partitions can then be mapped to the two crossbars as shown in (❸), with an average of eight spikes communicated between the crossbars due to the mapping of the link between neuron $\boldsymbol{d}$ and $\boldsymbol{e}$ on the shared interconnect of the hardware. Finally, the two clusters generated from the SNN graph are shown in (❹) along with the inter-cluster communication.

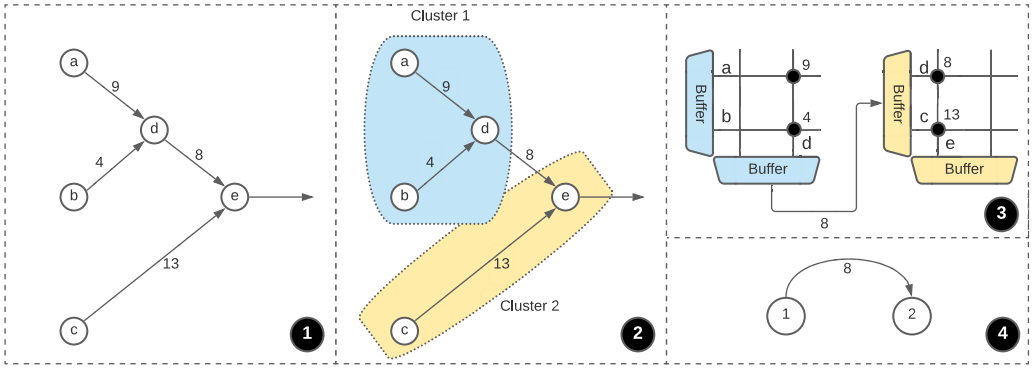

Fig. 8. Illustration of SNN graph clustering. (❶) is the original decomposed SNN graph with FIT neural units shown as the nodes and average spikes communicated between them shown on the links. (❷) shows the partitioning of this graph. (❸) shows the mapping of the partitions to the two crossbars. (❹) shows the two clusters generated from the SNN graph of (❶) considering the constraints of the crossbar.

Formally, a clustered SNN graph is defined as follows.

Definition 3 (Clustered SNN Graph). A clustered SNN graph $G_{CSNN} = (\mathbf{A}, \mathbf{C})$ is a directed graph consisting of a finite set $\mathbf{A}$ of clusters and a finite set $\mathbf{C}$ of connections between these clusters.

Recently, different approaches have been proposed for clustering SNNs. Examples include SpiNeMap [11] for energy minimization and NEUTRAMS [63] for performance. See Section 9 for a comprehensive overview of other state-of-the-art SNN clustering approaches.

We formulate SNN clustering as a graph transformation problem and introduce an efficient algorithm to improve resource utilization. This objective is essential to provide a tighter guarantee on performance of SNNs in hardware as we demonstrate in Section 8.

The graph transformation $G_{DSNN} \rightarrow G_{CSNN}$ is a classical graph partitioning problem [65] and has been applied in many contexts, including task mapping on multiprocessor systems [38]. We propose a greedy approach to pack the FIT neural units and synapses of the decomposed SNN graph G_{DSNN} into clusters, improving cluster resource utilization. Algorithm 2 provides the pseudo-code of the clustering algorithm. For each node of the unrolled graph, the algorithm tries to see if the node can be merged into one of the existing clusters (line 3), before creating a new one (lines 4–8). In this algorithm, clusters in G_{CSNN} are sorted in descending order of neuron and synapse utilization (line 12), so that the heavily utilized clusters are first considered for packing neurons and synapses, further improving their utilization.

4.4 Dataflow Modeling of Clustered Workload

We model a clustered SNN as an SDFG for predictable performance analysis [69]. SDFGs are commonly used to model streaming applications that are implemented on a multi-processor system-on-chip [94]. These graphs are used to analyze a system in terms of key performance properties such as throughput, execution time, communication bandwidth, and buffer requirements [97]. Nodes of an SDFG are called *actors*. Each node is a cluster of the clustered SNN graph $G_{CSNN} = (\mathbf{A}, \mathbf{C})$. Actors are computed by reading *tokens*, i.e., 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

ALGORITHM 2: Utilization-aware SNN Clustering

Input: G_{DSNN} = (**F**, **L**)
Output: G_{CSNN} = (**A**, **C**)
G_{CSNN} = {} and `cluster_list` = {};
foreach $F_i \in \boldsymbol{F}$ **do**
  find $C_j \in$ `cluster_list` such that F_i can be packed in C_j while improving neuron and synapse utilization of C_j;
  if $C_j = \emptyset$ **then**
    Create new cluster C_{new};
    Assign F_i and its synaptic connections to C_{new};
    G_{CSNN}.push(C_{new});
  end
  else
    Assign F_i and its synaptic connections to C_j;
  end
  sort G_{CSNN} in descending order of neuron and synapse utilizations;
end

crossbar. Edges in the graph are called *channels,* and they represent dependencies among actors. An actor is said to be *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 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, an actor is defined as follows.

Definition 4 (Actor). An actor $\boldsymbol{a}_i$ is a tuple $(I_i, O_i, \tau_i, \mu_i)$ consisting of a set I_i ($\subseteq Ports$) of input ports and a set O_i ($\subseteq Ports$) of output ports with $I_i \cap O_i = \emptyset$, τ_i is the execution time of $\boldsymbol{a}_i$, and μ_i is its state space, i.e., buffer space needed for communicating spikes on all of its channels.

The source of channel $ch_i^j \in C$ is an output port of actor $\boldsymbol{a}_i$; the destination is an input port of actor $\boldsymbol{a}_j$. 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 $\boldsymbol{a}_i$ are denoted by $InC(\boldsymbol{a}_i)$ and $OutC(\boldsymbol{a}_i)$, 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 article. The following definitions are introduced to formulate throughput.

Definition 5 (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.

For 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, $RptV = [1111111]$.

4.5 Cyclic Dependency and Deadlock Avoidance

The clustering approach may lead to cyclic dependency among actors. Figure 9(a) illustrates a simple feedforward network of three neurons (A, B, and C). Figure 9(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 9(a), there is a cyclic dependency between the two actors: `actor_1`→`actor_2`→`actor_1`. SDF graphs allow representing such cyclic dependency among actors, justifying our choice of using them for modeling clustered SNNs.

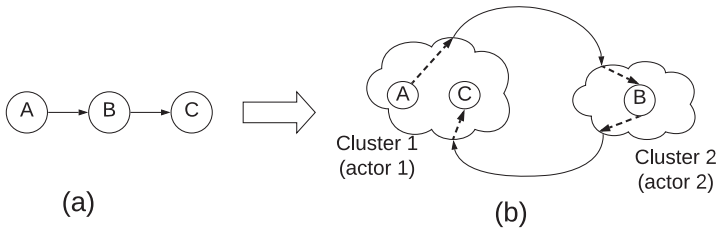

Fig. 9. An example cycle generated during clustering of SNNs.

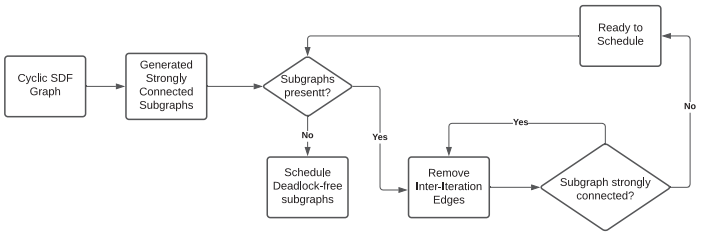

Fig. 10. Cycle breaking for deadlock avoidance of cyclic SDF graphs [18].

However, the presence of cycles complicates the scheduling problem because cyclic dependencies can lead to deadlocks. To address this, a cyclic SDF graph is decomposed into hierarchies of acyclic subgraphs. To describe this, we introduce the following definition.

Definition 6 (Strongly Connected Subgraph). A subgraph Z of a directed (cyclic or acyclic) graph is called a strongly connected subgraph, iff for every pair of vertices a and b of Z, there is a path from a to b and a path from b to a.

Figure 10 shows the flowchart for *cycle breaking*, also known as *sub-independence partitioning*, which is the process of decomposition of strongly connected SDF graphs into hierarchies of acyclic graphs. This is roughly based on the **Loose Interdependence Algorithms Framework (LIAF)** [18]. A cyclic SDF graph is first decomposed into a series of strongly connected subgraphs $Z_1, Z_2, \ldots, Z_N$. For each strongly connected subgraph Z_i, the LIAF algorithm tries to break cycles by properly removing edges that have sufficient delays. Let $Z_i(V_i, E_i)$ be the strongly connected subgraph of the SDF Graph. An edge $e_j \in E_i$ can be removed if it has enough initial tokens to satisfy the consumption requirements of its sink actor for a complete iteration of Z_i and scheduling Z_i without e_j does not lead to deadlock. The edge e_j is called *inter-iteration edge*. The inter-iteration edge removal is performed iteratively until the new subgraph with the inter-iteration edges removed is no longer a strongly connected subgraph (i.e., it becomes a *loosely connected subgraph*). The subgraph is pushed into a ready list for scheduling purposes. The algorithm is repeated for all the strongly connected subgraphs. At the end, all deadlock-free subgraphs are scheduled.

4.6 Performance Estimation

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 [29, 58, 107]. 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

$$a \oplus b = \max(a, b) \text{ and } a \otimes b = a + b. \tag{3}$$

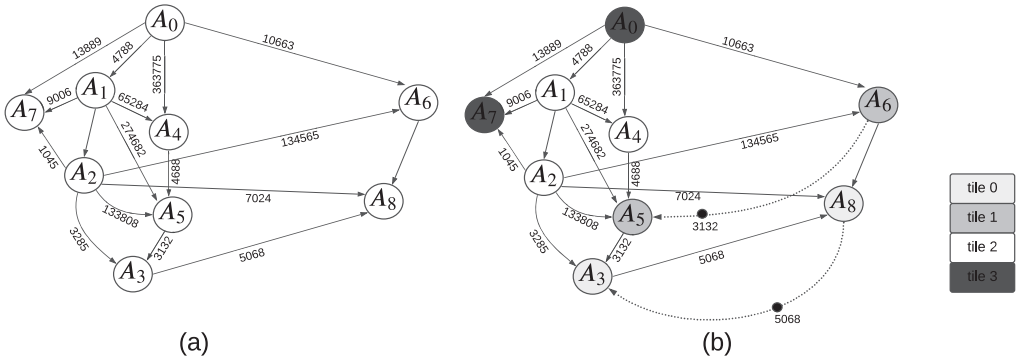

Fig. 11. (a) An example of SDFG obtained from clustering of the EdgeDet application [28]. (b) Mapping of the SDFG to a neuromorphic hardware with four tiles.

The identity element $\mathbb{0}$ for the addition $\oplus$ is $-\infty$ in linear algebra, i.e., $a \oplus \mathbb{0} = a$. The identity element $\mathbb{1}$ for the multiplication $\otimes$ is 0 in linear algebra, i.e., $a \otimes \mathbb{1} = a$.

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 in linear algebra and then use standard analysis techniques for Max-Plus Algebra to estimate timing performance. We use the running example of the SDFG in Figure 11(a), which is obtained by clustering EdgeDet [28], an application used to evaluate DFSynthesizer (see Section 7). The clustering is performed considering 1024 × 1024 crossbars.[7] The firing end times of all nine actors in the k^{th} iteration (in linear algebra) are

$$
\begin{aligned}
t_0(k) &\geq t_0(k-1) + \tau_0 & t_5(k) &\geq \max\Big[t_2(k), t_1(k), t_4(k)\Big] + \tau_5 \\
t_1(k) &\geq t_0(k) + \tau_1 & t_6(k) &\geq \max\Big[t_2(k), t_0(k)\Big] + \tau_6 \\
t_2(k) &\geq t_1(k) + \tau_2 & t_7(k) &\geq \max\Big[t_1(k), t_0(k)\Big] + \tau_7 \\
t_3(k) &\geq \max\Big[t_2(k), t_5(k)\Big] + \tau_3 & t_8(k) &\geq \max\Big[t_2(k), t_3(k), t_6(k)\Big] + \tau_8. \\
t_4(k) &\geq \max\Big[t_1(k), t_0(k)\Big] + \tau_4
\end{aligned}
\tag{4}
$$

Observe that the firing end time of actor A_0 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

$$
\mathbf{t_k} = \oplus\mathbf{T} \otimes \mathbf{t_{k-1}}, \tag{5}
$$

where $\mathbf{T}$ is a matrix in $\mathbb{R}_{\max}^{8\times 8}$ that captures the actor execution times τ_n and $\mathbf{t_k} = \{t_0(k), t_1(k), \ldots, t_8(k)\}$. The following definitions are introduced to estimate latency.

Definition 7 (Digraph). The digraph $\Gamma(T)$ of an $n \times n$ 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., $A = \{1, 2, \ldots n\}$, and E is the set of connected ordered arcs between vertices, i.e., $E = \{(i, j) \mid T_{i,j} \neq -\infty\}$.

[7]We evaluate DFSynthesizer primarily for DYNAP-SE neuromorphic hardware with 128 × 128 crossbars [73]. Here we configure 1024 × 1024 crossbars to generate fewer clusters from EdgeDet for illustration purposes.

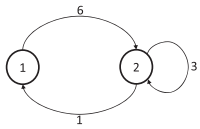

Fig. 12. An example digraph of $T = \left[\begin{smallmatrix} -\infty & 6 \\ 1 & 3 \end{smallmatrix}\right]$.

To give an example, the matrix $T = \left[\begin{smallmatrix} -\infty & 6 \\ 1 & 3 \end{smallmatrix}\right]$ corresponds to the digraph shown in Figure 12.

Definition 8 (Walk). A walk w in digraph $\Gamma(T)$ is the sequence of arcs (x_1, x_2) $(x_2, x_3) \ldots (x_{k-1}, x_k)$; the 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

$$|w|_T = T_{x_1 x_2} + \cdots T_{x_{k-1} x_k}. \tag{6}$$

Definition 9 (Cycle). A cycle c in digraph $\Gamma(T)$ is the walk $(x_1, x_2)(x_2, x_3) \ldots (x_{k-1}, x_k)$, such that $x_k = x_1$.

Definition 10 (Maximum Cycle Mean). The maximum cycle mean, $\rho_{\max}(T)$, is the maximum of the weight-to-length ratio of all cycles c in $\Gamma(T)$, i.e.,

$$\rho_{\max}(T) = \max_{\forall c \text{ in } \Gamma(T)} \frac{|c|_T}{|c|} = \max_{k>1} \max_{x_1, \ldots, x_{k-1}} \frac{T_{x_1 x_2} + \cdots T_{x_{k-1} x_k}}{k-1}. \tag{7}$$

In this article, **performance of an SNN is defined in terms of throughput** of the equivalent SDFG, measured as the inverse of its *maximum cycle mean* (Equation (7)), i.e.,

$$\text{Performance (throughput)} = \frac{1}{\rho_{\max}(T)}. \tag{8}$$

In Equation (8), the performance is computed using the worst-case execution time of an actor on a crossbar. This is obtained from the propagation delay of current through the synaptic elements in the crossbar. As shown in many recent works [99, 100, 102], the current propagation delay within a crossbar depends on the specific synaptic elements that are being activated in the crossbar. This is due to the difference in the amount of parasitic components on the bitlines and wordlines of a crossbar along the different current paths. For performance guarantee purposes, we assume the worst-case propagation delay in the crossbar and use the same to represent the execution time of actors on the crossbars of a neuromorphic hardware.

The performance metric defined in Equation (8) provides the maximum throughput, considering only the worst-case execution time of actors. However, a neuromorphic hardware introduces constraints such as limited buffer space on the crossbars and non-zero latency on the interconnect, which can lower the throughput significantly. Therefore,

$$\text{Throughput}\Big|_{SNN} \leq \text{Throughput}\Big|_{\max} = \frac{1}{\rho_{\max}(T)}. \tag{9}$$

In this work, we show that performance is impacted by

(1) how hardware resources are allocated to actors of a clustered SNN (Section 5) and
(2) how actors mapped to the same crossbar are time-multiplexed and scheduled (Section 6).

We seek to find the lower bound on performance ($\text{Throughput}|_{\text{bound}}$) such that

$$\text{Throughput}|_{\text{bound}} \leq \text{Throughput}|_{SNN} \leq \text{Throughput}|_{\text{max}}. \tag{10}$$

By making $\text{Throughput}|_{\text{bound}}$ close to $\text{Throughput}|_{\text{max}}$, we provide a tighter bound on performance.

5 RESOURCE ALLOCATION AND HARDWARE MAPPING

The performance obtained using Equation (7) defines the maximum throughput obtained when the clustered SNN is mapped to a hardware with infinite resources, i.e., a hardware with as many crossbars as the number of actors (clusters) in the clustered SNN graph. Additionally, each crossbar is assumed to have sufficient buffer space to send and receive spikes over the shared interconnect. However, state-of-the-art neuromorphic hardware platforms present the following three critical limitations. First, the number of crossbars in a neuromorphic hardware is limited. Therefore, the available crossbars need to be time-multiplexed among the clusters of an SNN. Second, the input and output buffer space on each crossbar are limited. Therefore, no more than one cluster can be executed on a crossbar concurrently. Third, the communication bandwidth of each tile is limited. Therefore, only a few spikes can be sent or received from the interconnect at once. Formally, a neuromorphic hardware is defined as follows.

Definition 11 (Neuromorphic Hardware Graph). A neuromorphic hardware graph $G_{NH} = (\mathbf{T}, \mathbf{I})$ is a directed graph consisting of a finite set $\mathbf{T}$ of tiles and a finite set $\mathbf{I}$ of interconnect links.

Each tile consists of a crossbar to map neurons and synapses, and input and output buffers to receive and send tokens (spikes) over the interconnect, respectively. A tile T_i is a tuple $\langle N, inB_i, outB_i \rangle$, where N_i is the dimension of the crossbar on the tile—i.e., the tile T_i can accommodate N_i pre-synaptic neurons, N_i post-synaptic neurons, and N_i^2 synaptic connections; inB_i is the input buffer size on the tile; and $outB_i$ is its output buffer size. Each interconnect link is bidirectional, representing two-way communication between the source and destination tiles with a fixed bandwidth BW.

The mapping $\mathcal{M} : G_{CSNN} \rightarrow G_{NH}$ is specified by matrix $(m_{ij}) \in \{0, 1\}^{|\mathbf{A}| \times |\mathbf{T}|}$, where m_{ij} is defined as

$$m_{ij} = \begin{cases} 1 & \text{if actor } A_i \in \mathbf{A} \text{ is mapped to tile } T_j \in \mathbf{T} \\ 0 & \text{otherwise.} \end{cases} \tag{11}$$

The mapping constraint is that a cluster can be mapped to only one tile, i.e.,

$$\sum_j m_{ij} = 1 \; \forall i. \tag{12}$$

The throughput of the clustered SNN graph G_{CSNN} on the neuromorphic hardware G_{NH} for mapping $\mathcal{M}$ is computed as

$$\tau_{\mathcal{M}} = \texttt{DFSynthesizer}(G_{CSNN}, G_{NH}, \mathcal{M}), \tag{13}$$

where `DFSynthesizer` is the extended Max-Plus formulation of Equation (7) incorporating platform constraints. The following three steps describe `DFSynthesizer`. Without loss of generality, we use Equation (14) as a running mapping example, where the nine actors of Figure 11 are mapped to four tiles:

$$\begin{aligned} \textbf{tile\_0} &: A_3, A_8, \textbf{tile\_2} : A_1, A_2, A_4 \\ \textbf{tile\_1} &: A_5, A_6 \; \textbf{tile\_3} : A_0, A_7. \end{aligned} \tag{14}$$

The mapping corresponding to Equation (14) is therefore $\mathcal{M} = \begin{bmatrix} 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\ 0 & \mathbf{1} & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \end{bmatrix}^T$.

5.1 Step 1: Modeling Limited Buffer Sizes of Crossbars

Limited input and output buffer sizes of a tile are modeled as back-edges with initial tokens indicating the buffer size available on the tile. This is illustrated in Figure 11(b) with the back-edge from A_8 to A_3, both of which are mapped to tile 0. When an actor generates spikes on a channel, the available size reduces; when the receiving actor consumes the spike, the available buffer is released. In the example, before A_3 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 5,068 spikes per firing, 5,068 tokens from the back-edge are consumed, indicating reservation of the buffer spaces. On the consumption side, when A_8 is executed, it frees 5,068 buffer spaces, indicated by a release of these 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 tile from firing simultaneously, the output buffer space is claimed at the start of execution and released only at the end of firing.

5.2 Step 2: Actor Ordering on Crossbars

The number of crossbars in a neuromorphic hardware is limited. 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 its 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 performance guarantee, i.e., throughput. For this, we apply our Max-Plus Algebra formulation (Equation (7)) on the SDFG of Figure 11(b). This is our *static-order schedule* and is constructed at *design time*.

5.3 Step 3: Actor Execution on Crossbars

Once the static-order schedule is constructed for all tiles of the hardware, we use a self-timed execution strategy [74] to execute these actors at runtime. Here, 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 runtime, ready actors are inserted into a list and fired in the same order previously determined during design time.

5.4 Mapping Exploration

Sections 5.1 through 5.3 extend the Max-Plus formulation to incorporate platform constraints. Using these constraints and the new formulation, one can estimate the throughput of a clustered SNN on a neuromorphic hardware for a specific actor-to-tile mapping. In the following, we explain the mapping scenario where the number of tiles in the hardware is less than the number of actors in the clustered SNN. Therefore, each tile needs to be time-multiplexed between multiple actors.

Figure 13 conceptually illustrates the mapping exploration using DFSynthesizer compared to state-of-the-art solutions and the selection of lower bound on throughput. ❶ represents the throughput obtained using SpiNeMap [11], which optimizes energy consumption for a hardware platform where the number of tiles is higher than the number of actors. When SpiNeMap is applied

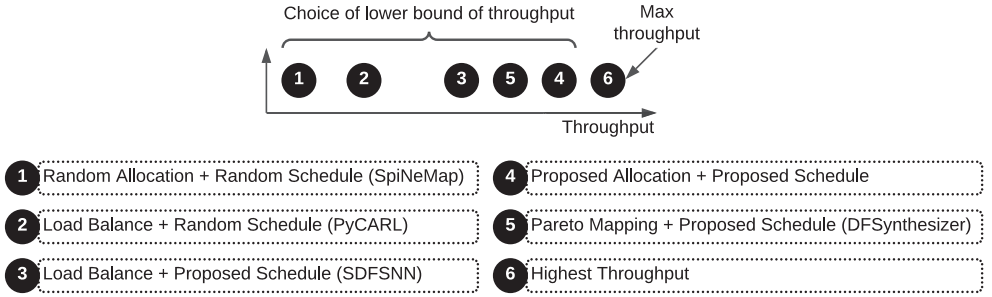

Fig. 13. Different mapping explorations and choices for the lower bound of throughput (see Equation (10)).

to the case where the tiles need to be time-multiplexed, it randomly distributes the actors to the tiles and schedules them arbitrarily, without considering throughput. Therefore, the throughput represented by ❶ (SpiNeMap) is significantly lower than the maximum throughput (i.e., the upper bound) represented using ❻. Therefore, the throughput variation is $T_{❻} - T_{❶}$.

In Figure 13, ❷ represents the throughput obtained using a solution such as PyCARL [7], which balances the load on each tile for a scenario where actors need to be time-multiplexed on the tiles. However, the actors mapped to a tile are scheduled in an arbitrary order without considering throughput. By balancing the tile load, PyCARL reduces the number of clusters mapped per tile, which improves throughput. Therefore, the throughput represented by ❷ is higher than ❶, but lower than the maximum throughput ❻. Therefore, the throughput variation is $T_{❻} - T_{❷}$.

In Figure 13, ❸ represents the throughput obtained using our previous work SDFSNN [83], which first balances the load of each tile by distributing the actors evenly, and then uses a dataflow approach to schedule the actors on each tile, improving throughput. The throughput represented by ❸ is therefore higher than both ❶ and ❷, but lower than the maximum throughput ❻. Therefore, the throughput variation is $T_{❻} - T_{❸}$.

In Figure 13, ❹ represents the throughput obtained using a mapping exploration framework, which explores a combination of actor-to-tile mapping and dataflow-based scheduling of actors on each tile to maximize the throughput. This throughput is higher than ❶–❸, and is closer to the maximum throughput ❻. Finally, ❺ represents the throughput obtained using an actor-to-tile mapping that jointly optimizes energy and throughput and uses dataflow-based scheduling of actors on each tile to further improve the throughput. Since this solution takes energy into consideration in the mapping step, the throughput can be somewhat lower than ❹ as illustrated in the figure. In Section 8, we evaluate all these approaches and show that ❺ is still higher than ❶–❸.

To conclude, the design-space exploration of DFSynthesizer can generate mappings representing two minimum throughput solutions: ❹ and ❺. Although the maximum throughput remains the same for DFSynthesizer and other state-of-the-art approaches, the minimum throughput of DFSynthesizer (i.e, ❺) is higher than the minimum throughput obtained using all state-of-the-art mapping solutions (i.e., ❶–❸). Therefore, the difference between maximum and minimum throughput is the least in DFSynthesizer compared to all state-of-the-art solutions, meaning that DFSynthesizer provides a stricter performance guarantee, which is critical for real-time systems. We now describe DFSynthesizer.

We integrate the extended Max-Plus formulation inside a design-space exploration framework to obtain cluster mappings that are Pareto optimal in terms of hardware metrics such as throughput, latency, energy, and reliability. In the following, we describe our mapping explorations

considering energy and throughput. Such formulations can be trivially extended to consider other metrics.

The energy consumption $E_{\mathcal{M}}$ of the mapping $\mathcal{M}$ is measured considering the number of spikes that are generated inside each tile and the number of spikes that are routed on the interconnect [101]. The energy parameters are reported in Table 3. Using these parameters, the energy consumption is

$$E_{\mathcal{M}} = E_{spk} + E_{\text{comm}}, \tag{15}$$

where E_{spk} is the energy consumed in generating the spikes and propagating the spike current via the synapses, and E_{comm} is the energy consumed in communicating spikes via the shared interconnect, where $S(T_i)$ is the number of spikes generated inside tile $T_i \in T$ and $S(I_{i,j})$ is the number of spikes communicated on the link $I_{i,j}$ between tiles T_i and T_j in the hardware.

Our objective is to maximize throughput of a given machine learning model on hardware (Equation (7)) and minimize the hardware energy consumption (Equation (15)). We formulate a joint metric $\lambda = E/\tau$ and minimize it during our mapping explorations. To this end, we propose an iterative approach, which explores different mapping alternatives, satisfying the cluster mapping constraint (Equation (12)). For each mapping alternative, we evaluate throughput and energy consumption. Finally, Pareto-optimal mappings are retained and returned.

Algorithm 3 provides the pseudo-code of our proposed mapping exploration. We start by randomly distributing clusters to the tiles (line 3). We evaluate throughput and energy consumption of this mapping and compute the joint metric λ (lines 4–5). For each cluster, we do the following. We move the cluster from its current tile to every other tile and recalculate λ (lines 6–10). If λ reduces, the new mapping is retained (lines 11–13), and the algorithm proceeds to analyze the next cluster. In this way, a local minimum is reached, starting from the initial random allocation of clusters. We re-execute the algorithm η times, starting with a different random allocation of the clusters each time. In this way, many mappings are explored. Finally, mappings that are Pareto optimal in terms of throughput and energy consumption are retained.

ALGORITHM 3: Mapping of the Clustered Graph G_{cl}

Input: $G_{cl} = (C, A)$, $G_{nh} = (T, I)$
Output: $\mathcal{M}_{\max}$

1 $\mathbb{M} = \{\}$; /* This set holds all the mappings */
2 **for** $r = 0; r < \eta; r$++ **do** /* Run for η times */
3 Allocate clusters randomly to tiles. Call this mapping $\mathcal{M}$;
4 Calculate $\tau_{\mathcal{M}}$ using (7) and energy consumption $E_{\mathcal{M}}$ using (15);
5 Calculate the joint metric $\lambda = \tau_{\mathcal{M}} \cdot E_{\mathcal{M}}$;
6 **for** $C_i \in C$ **do** /* For each cluster in the graph G_{cl} */
7 T_{C_i} = GetTileOfCluster($\mathcal{M}$, C_i); /* Get the tile to which the cluster C_i is mapped in the mapping $\mathcal{M}$ */
8 **for** $T_j \in T \setminus T_{C_i}$ **do** /* Move the cluster to every other tile */
9 $\mathcal{M}_j$ = MoveClusterToTile($\mathcal{M}$, C_i, T_j); /* Update the mapping to reflect the movement of cluster C_i to tile T_j */
10 Calculate $\tau_{\mathcal{M}_j}$, $E_{\mathcal{M}_j}$, and λ_j;
11 **if** $\lambda_j < \lambda$ **then** /* If the joint metric improves */
12 $\mathcal{M} = \mathcal{M}_j$; /* Retain the new mapping */
13 **end**
14 **end**
15 $\mathbb{M}$.insert($\mathcal{M}$)
16 **end**
17 $\mathbb{M}_{PO}$ = ParetoOptimization($\mathbb{M}$); /* Retain only the Pareto-Optimal Mappings */
18 Return $\mathcal{M}_{\max}$, the mapping with minimum execution time.

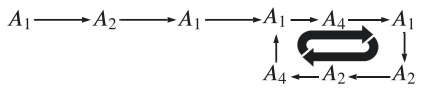

Fig. 14. Self-timed execution consisting of transient phase followed by periodic phase.

The complexity of this algorithm is as follows. The unit function `GetTileofCluster` is essentially an `argmax` function with a complexity of $O(|T|)$. The unit function `MoveClusterToTile` is an update of matrix and can be performed in $O(1)$. Therefore, the complexity of the algorithm is $\eta \times |C| \times |T|$. Here, η is a user-defined parameter and controls the compilation time with a tradeoff on the solution quality, i.e., execution time and energy consumption of the application on hardware.

6 SCHEDULING AND PERFORMANCE GUARANTEE

Self-timed execution is widely used to schedule SDFGs [54]. Static schedules are constructed using worst-case actor execution times determined during design time. Actor ordering on each tile is retained while discarding the timing information. At runtime, actors are fired while maintaining the same order as determined during design time. In this regard, the following lemmas are stated [35, 38, 54].

Lemma 1. *For a consistent and strongly connected SDFG, the self-timed execution consists of a transient phase followed by a periodic phase.*

Lemma 2. *For a consistent and strongly connected SDFG, the throughput of an actor is given by the average firing of the actor per unit time in the periodic phase of the self-timed execution.*

Figure 14 shows an example self-timed execution of three actors, A_1, A_2 and A_4, of Figure 11(b) on tile 2.

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. For such dynamic scenarios, SDFG schedules must be constructed for every allocation scenario. If the runtime schedule is different from that used for analysis at design time, the throughput obtained will be significantly different than what is guaranteed at design time. There are therefore two approaches to generating runtime schedules.

- Store the actor mapping and scheduling for all resource allocation scenarios and for all applications from design time (*storage-based* solution).
- Construct the schedule at runtime based on the mappings stored from the design time (*construction-based* solution).

The former is associated with high storage overhead and the latter with longer execution time. Both storage and schedule construction time are crucial for machine learning systems deployed in resource- and power-constrained environments. Therefore, we propose a modification of the self-timed execution scheduling as follows. First, we construct the static-order schedule for all actors of an SNN on a single tile at design time. This is achieved using the Max-Plus Algebra formulation of Equation (7). Next, we discard the exact timing information, retaining only the actor firing orders for runtime use. At runtime, we first construct the cluster mapping to tiles (Section 5.4), considering the available tiles. Next, we use the single-tile static-order schedule to **derive** the actor schedules on each tile, without having to construct them from scratch.

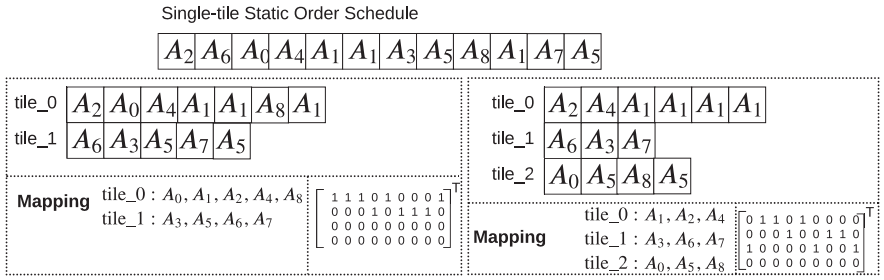

Fig. 15. Schedules constructed from the same single-tile static order schedule using two and three tiles, respectively.

Figure 15 illustrates the construction of per-tile schedules for an SNN application with nine actors, and with two different mappings of actors to tiles from 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 are the same as those on the single tile. Since tile schedules are not constructed from scratch, the schedule construction time is much lower.

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. See Section 8 for performance evaluation.

7 EVALUATION METHODOLOGY

We conduct all simulations on a Lambda workstation, which has AMD Threadripper 3960X with 24 cores, 128 MB cache, 128 GB RAM, and 2 RTX3090 GPUs. Keras [57] and CARLsim [28] use the two GPUs to accelerate model training and SNN function simulation, respectively.

Figure 16 illustrates our evaluation setup using the cycle-accurate NeuroXplorer [15] framework. This framework is validated extensively against the DYNAP-SE neuromorphic hardware [7, 8, 11, 41, 44] and can model the architecture of other neuromorphic hardware platforms such as Loihi [45] and TrueNorth [47]. NeuroXplorer can simulate multi-compartment neuron models and nine-parameter Izhikevich and LIF spiking neuron models. Additionally, NeuroXplorer can model NVM synapses such as **Phase Change Memory (PCM)** and **Oxide-based Resistive Random Access Memory (OxRRAM)**. NeuroXplorer also models the spike delay on the shared interconnect as well as the delay in propagating spikes through the synapses of a crossbar [15]. The mapping and scheduling results obtained using DFSynthesizer are used in NeuroXplorer to estimate energy, accuracy, and throughput.

7.1 Evaluated Applications

We evaluate 10 machine learning programs that are representative of the three most commonly used neural network classes: CNN, MLP, and RNN. These applications are (1) LeNet-based handwritten digit recognition with 28×28 images of handwritten digits from the MNIST dataset; (2) AlexNet for ImageNet classification; (3) VGG16, also for ImageNet classification; (4) ECG-based heart beat classification (HeartClass) [8, 32] using **electrocardiogram (ECG)** data; (5) image smoothing (ImgSmooth) [28] on 64×64 images; (6) edge detection (EdgeDet) [28] on 64×64 images using difference-of-Gaussian; (7) MLP-based handwritten digit recognition

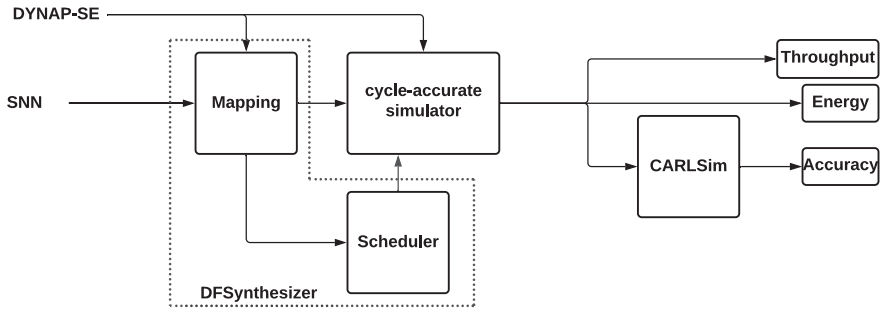

Fig. 16. Our evaluation setup based on NeuroXplorer [15].

Table 2. Applications Used to Evaluate DFSynthesizer

Class	Applications	Dataset	Synapses	Neurons	Topology	Top-1 Accuracy (%)
CNN	LeNet	MNIST	282,936	20,602	CNN	85.1%
	AlexNet	ImageNet	38,730,222	230,443	CNN	69.8%
	VGG16	ImageNet	99,080,704	554,059	CNN	90.7 %
	HeartClass [8]	Physionet	1,049,249	153,730	CNN	63.7%
MLP	ImgSmooth [28]	CARLsim	9,025	4,096	FeedForward (4096, 1024)	100%
	EdgeDet [28]	CARLsim	114,057	6,120	FeedForward (4096, 1024, 1024, 1024)	100%
	DigitRecogMLP	MNIST	79,400	884	FeedForward (784, 100, 10)	91.6%
RNN	HeartEstm [41]	Physionet	66,406	166	Recurrent Reservoir	100%
	VisualPursuit [64]	[64]	163,880	205	Recurrent Reservoir	47.3%
	DigitRecogSTDP [50]	MNIST	11,442	567	Recurrent Reservoir	83.6%

(DigitRecogMLP) [50] using the MNIST database; (8) heart rate estimation (HeartEstm) [41] using ECG data; (9) RNN-based predictive visual pursuit (VisualPursuit) [64]; and (10) recurrent digit recognition (DigitRecogSTDP) [50]. To demonstrate the potential of DFSynthesizer, we consider a real-time neuromorphic system, where these machine learning programs are executed continuously in a streaming fashion. Therefore, by optimizing throughput, DFSynthesizer improves real-time performance.

Table 2 summarizes the topology, the number of neurons and synapses of these applications, and their baseline accuracy on the DYNAP-SE neuromorphic hardware using the SpiNeMap [11] mapping framework. As reported in many recent works [7, 11, 44], spike latency on the shared interconnect of a neuromorphic hardware can lead to ISI distortion and spike disorder. Since the performance of an SNN is a function of ISI, such non-idealities can lead to accuracy loss. Therefore, the accuracy of the three CNN architectures, LeNet, AlexNet, and VGG16, in Table 2 is somewhat lower than that reported via functional simulation in Table 1.

7.2 Hardware Parameters

We model the DYNAP-SE neuromorphic hardware [73] with 1,024 tiles organized in a 32×32 mesh. Each tile has one 128×128 crossbar. To test the scalability of DFSynthesizer, we also evaluate other crossbar configurations, e.g., 256×256, 512×512, and 1024×1024. Table 3 reports the relevant hardware parameters.

The additional overhead in time multiplexing the tiles among multiple crossbars is incorporated in computing the throughput using NeuroXplorer. Specifically, once the cluster mapping to tiles are generated using DFSynthesizer, the synaptic weights of all clusters mapped to a tile are pre-loaded into the tile's local memory (see our system architecture in Figure 5). In this way, DFSynthesizer reduces the overhead of transferring synaptic weights at runtime from the shared main memory. Additionally, since the loading of clusters (context switching) in crossbars happens concurrently from their respective private memory, the time-multiplexing overhead is minimal.

Table 3. Major Simulation Parameters Extracted from [73]

Neuron technology	28nm FD-SOI
Synapse technology	HfO_2 -based OxRAM
Supply voltage	1.0V
Energy per spike	50pJ at 30Hz spike frequency
Energy per routing	147pJ
Switch bandwidth	1.8G. Events/s

7.3 Evaluated Metrics

We evaluate the following performance metrics:

- **Performance.** This is the throughput of each application on the hardware.
- **Resource Utilization.** This is the neuron, synapse, buffer, connection, and input and output bandwidth utilization on the hardware for each application.
- **Energy Consumption.** This is the energy consumed on the hardware for each application. This is the total energy consumed to generate spikes on each tile and communicate spikes between tiles via the shared interconnect.
- **Cluster Connection.** This is the average degree of the SDFG as a percentage of the total number of nodes, obtained using the clustering technique for each application.
- **Spike Communication.** This is the total number of spikes communicated on the shared interconnect of the neuromorphic hardware.
- **Synthesis Time.** This is the time to compile and map each application on the hardware.

7.4 Evaluated Approaches

We evaluate the following approaches.

- **SpiNeMap [11].** This approach first partitions an SNN into clusters of neurons and synapses by incorporating its workload. The objective is to minimize inter-cluster communication. Clusters are then mapped to tiles while minimizing spike communication on the shared interconnect and reducing energy consumption. When mapping SNNs to neuromorphic hardware with fewer tiles than the number of actors, (1) SpiNeMap allocates actors to tiles randomly and (2) SpiNeMap schedules the actors on each tile arbitrarily. Therefore, SpiNeMap does not consider throughput.
- **PyCARL [7].** This approach maps neurons and synapses to tiles of a neuromorphic hardware, balancing the number of neurons and synapses on each tile. PyCARL does not incorporate SNN workload, i.e., spikes generated by neurons in the SNN. Therefore, some tiles may end up communicating more spikes than others; i.e., those tiles become the energy bottleneck.
- **SDFSNN [83].** This approach uses the load-balancing mapping of PyCARL to allocate actors to tiles. It uses dataflow scheduling to improve the throughput.
- **DFSynthesizer.** The proposed approach first clusters an SNN, considering its workload. The objective is to improve cluster utilization. This is done by first decomposing the SNN into homogeneous neural units with fanin-of-two. The clusters are then mapped to tiles, jointly optimizing throughput and energy consumption. DFSynthesizer uses dataflow-based scheduling of actors to tiles to further improve the throughput.

8 RESULTS AND DISCUSSIONS

8.1 Throughput

Figure 17 reports the throughput on DYNAP-SE for the evaluated approaches, for each application normalized to SpiNeMap. For reference, we have reported the maximum throughput in frames per

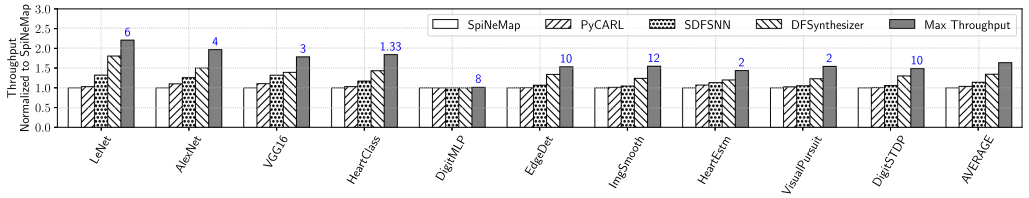

Fig. 17. Throughput on DYNAP-SE for each evaluated application normalized to SpiNeMap. The throughput in frames per second is reported for the maximum throughput approach for each application assuming unlimited hardware resources.

second obtained with unlimited hardware resources for each application. For image-based applications (LeNet, AlexNet, VGGNet, EdgeDet, ImgSmooth, and DigitSTDP), a frame corresponds to an individual image. For other time-series applications (HeartClass, HeartEstm, and VisualPursuit), a frame corresponds to a window of 500ms. We make the following four key observations.

First, although the number of neurons and synapses of larger applications such as AlexNet and VGG16 is significantly higher than LeNet, the throughput of LeNet on a hardware with unlimited resources,[8] i.e., without time-multiplexing of crossbars, is only 1.5× higher than AlexNet and 2× higher than VGG16. This is because with no time-multiplexing of crossbars, computations in a machine learning program take place concurrently on the crossbars, the basic philosophy of distributed computing, which is enabled using neuromorphic platforms. Therefore, the overhead due to time-multiplexing of crossbars is no longer the throughput bottleneck. Rather, the bottleneck shifts to spike delay between the clusters. Additionally, in our framework we cluster machine learning programs to minimize inter-cluster spikes. Therefore, even though Alexnet has a significantly higher number of neurons and synapses than LeNet, its number of inter-cluster spikes is not significantly higher. The throughput of AlexNet is only 33% lower than LeNet. Similarly, VGG16, which has higher inter-cluster spikes than AlexNet, has 25% lower throughput.

Second, the throughput obtained using SpiNeMap is the least because SpiNeMap does not guarantee throughput during actor-to-tile mapping and actor scheduling on tiles. The throughput of PyCARL is on average 4% higher than SpiNeMap. This is because PyCARL balances the load on the tiles, and therefore, the average number of actors mapped to each tile is lower than SpiNeMap, which results in higher throughput. The throughput of SDFSNN is on average 9.7% higher than PyCARL. This improvement is because of the use of dataflow-based scheduling, which maximizes the throughput. DFSynthesizer improves throughput by an average of 17% compared to SDFSNN. This improvement is because unlike SDFSNN, which maps actors to tiles balancing the tile load without considering the throughput, DFSynthesizer performs throughput- and energy-aware mapping of actors to tiles and then uses dataflow-based scheduling to further improve the throughput. We have analyzed such throughput differences in Section 5.4.

Third, the throughput using DFSynthesizer is only 16% lower on average than the maximum throughput obtained with unlimited hardware resources. Finally, the throughput of DigitMLP is a very small application. All the techniques generate the same number of clusters for this application, resulting in similar throughput.

8.2 Workload Energy

Figure 18 reports the workload energy estimated on DYNAP-SE of the evaluated approaches for each application normalized to SpiNeMap. For reference, we have reported the workload energy

[8]In the context of this work, unlimited resources refer to a neuromorphic hardware that has at least the same number of crossbars as there are clusters in the machine learning program.

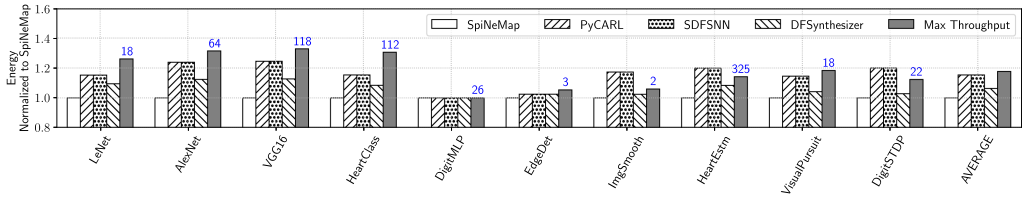

Fig. 18. Workload energy on DYNAP-SE for each evaluated application normalized to SpiNeMap. The workload energy in μJ is reported for the maximum throughput approach for each application assuming unlimited hardware resources.

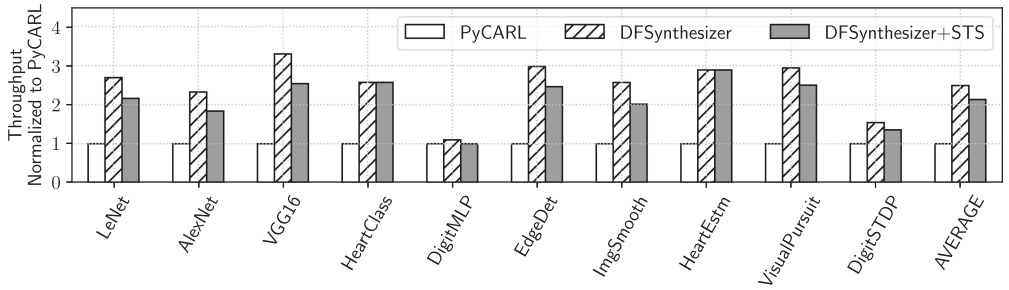

Fig. 19. Throughput normalized to PyCARL.

in μJ obtained using the maximum throughput approach, which assumes unlimited hardware resources. We make the following observation.

The energy consumption of SpiNeMap is the least because this approach partitions SNNs into clusters to explicitly minimize the number of inter-cluster spikes. Therefore, when the clusters are mapped to hardware, the energy consumption on the shared interconnect is reduced.[9] Second, the energy consumption of PyCARL is on average 15% higher than SpiNeMap. This is because PyCARL balances the tile load without incorporating energy consumption. Therefore, clusters with high volume of spike communication between them may get placed on different tiles, increasing the communication energy. SpiNeMap places those tiles on the same tile, lowering the communication energy. The energy consumption of SDFSNN is the same as PyCARL because the cluster-to-tile mapping of these two approaches is the same. SDFSNN gains over PyCARL in terms of throughput due to its dataflow-based cluster scheduling on tiles. We analyzed this in Section 8.1. The energy consumption of DFSynthesizer is lower than SDFSNN by an average of 8%. This reduction is due to the cluster-to-tile mapping of DFSynthesizer, which incorporates energy consumption.

8.3 Scheduling

Figure 19 reports the throughput of each of our applications for our proposed approach normalized to PyCARL. We compare throughput obtained using DFSynthesizer where schedules are independently constructed for each tile against the throughput obtained using our proposed single-tile-based schedule (DFSynthesizer+STS). We make the following three observations.

First, throughput obtained from a single-tile static-order schedule is on average 15% lower than the case when schedules are constructed independently—that is, by using DFSynthesizer. This

[9]The mapping exploration only impacts the communication energy on the shared interconnect. The spike generation energy remains the same for all approaches.

Table 4. Resource Utilization on DYNAP-SE

Application	Utilization (%)				
	Tile	Buffer	Connections	Bandwidth	
				Input	Output
LeNet	100	87.8	37.5	20.34	20.34
AlexNet	100	91.8	46.87	17.09	17.09
VGG16	100	94.2	15.62	6.51	6.51
HeartClass	100	79.1	25	9.76	9.76
DigitMLP	81.25	9.67	46.87	22.78	22.78
EdgeDet	87.5	11.23	68.75	22.78	22.78
ImgSmooth	87.5	8.39	37.5	17.08	17.08
HeartEstm	96.87	9.61	62.5	4.7	4.7
VisualPursuit	90.12	21.2	25.04	12.11	16.6
DigitSTDP	89.33	20.13	22.19	11.94	11.7

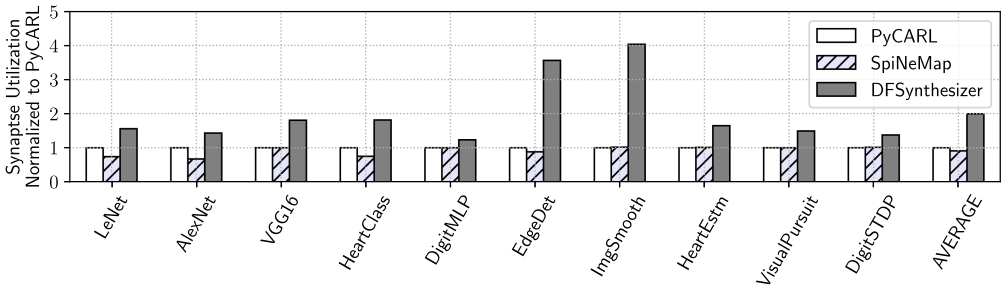

Fig. 20. Average synapse utilization on tiles for each evaluated application normalized to PyCARL.

verifies our Lemma 2. Second, for some applications such as HeartEstm and HeratClass, throughput obtained using DFSynthesizer+STS is exactly the same as that obtained using DFSynthesizer. Third, throughput using DFSynthesizer+STS is still higher than PyCARL by an average of 41%.

8.4 Resource Utilization

Table 4 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 is 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 DFSynthesizer can be used to design neuromorphic hardware while considering key hardware parameters such as number of tiles, buffer space, connections, and IO bandwidth.

To give more insight on the utilization within each tile, Figure 20 reports the average synapse utilization on tiles of the evaluated approaches for each application normalized to PyCARL. We make the following two key observations.

First, the synapse utilization on tiles using SpiNeMap is the least of all three evaluated approaches. This is because SpiNeMap produces the highest number of clusters (Section 8.5) and therefore, the average number of synapses per cluster is the least. Subsequently, when these clusters are mapped to tiles, the average synapse utilization on tiles reduces. Second, DFSynthesizer generates fewer clusters than both SpiNeMap and PyCARL due to its dense packing of synapses using Algorithm 2. Therefore, the average number of synapses per cluster is higher, which increases

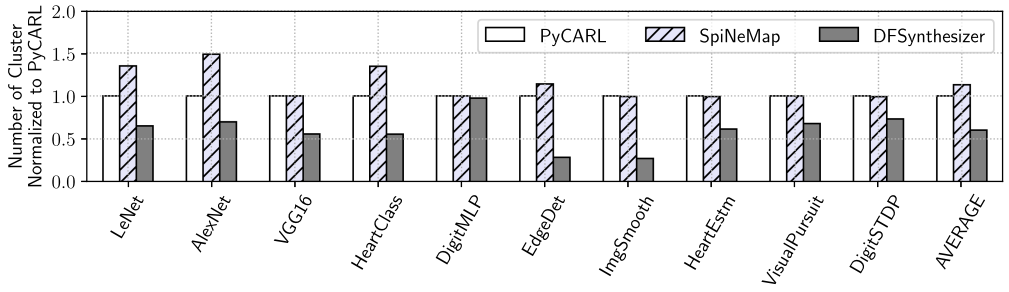

Fig. 21. Number of clusters for each evaluated application normalized to PyCARL.

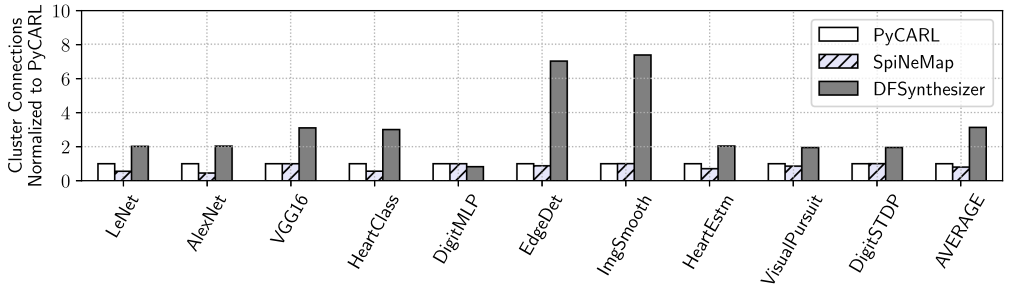

Fig. 22. Cluster connections for each evaluated application normalized to PyCARL.

synapse utilization on tiles when the clusters are mapped to tiles. On average, the average synapse utilization of DFSynthesizer is 2× higher than PyCARL and 2.2× higher than SpiNeMap.

8.5 Number of Clusters

Figure 21 reports the total number of clusters of the evaluated approaches for each application normalized to PyCARL. We make the following two key observations.

First, the number of clusters of SpiNeMap is the highest of all three evaluated approaches. This is because SpiNeMap minimizes the number of inter-cluster communications during clustering of an SNN. Therefore, neurons that spike the most are placed within individual clusters along with their fanins. Since SpiNeMap does not consider cluster utilization, it results in creating more clusters than PyCARL. Second, DFSynthesizer clusters an SNN to maximize the resource utilization on each tile. Therefore, the number of clusters generated by DFSynthesizer is the lowest. Overall, the number of clusters of DFSynthesizer is 41% lower than SpiNeMap and 47% lower than PyCARL. The lower number of clusters, the lower is the size of hardware needed to achieve the highest throughput (Section 8.1). Therefore, DFSynthesizer reduces the hardware requirement for machine learning applications.

8.6 Cluster Connections

Figure 22 reports the cluster connections of the evaluated approaches for each application normalized to PyCARL. We make the following two key observations.

First, the number of inter-cluster connections of SpiNeMap is the least of all three evaluated approaches. This is because SpiNeMap minimizes the number of inter-cluster communication while clustering an SNN, which indirectly reduces the cluster connectivity. Second, DFSynthesizer clusters an SNN to maximize the resource utilization on each tile. Therefore, the number of connections

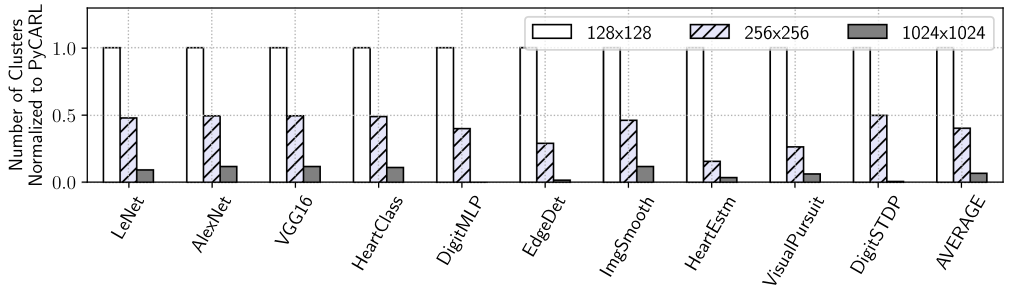

Fig. 23. Number of clusters generated using DFSynthesizer for 128 × 128, 256 × 256, and 1024 × 1204 crossbars, normalized to the configuration of DYNAP-SE with 128 × 128 crossbars.

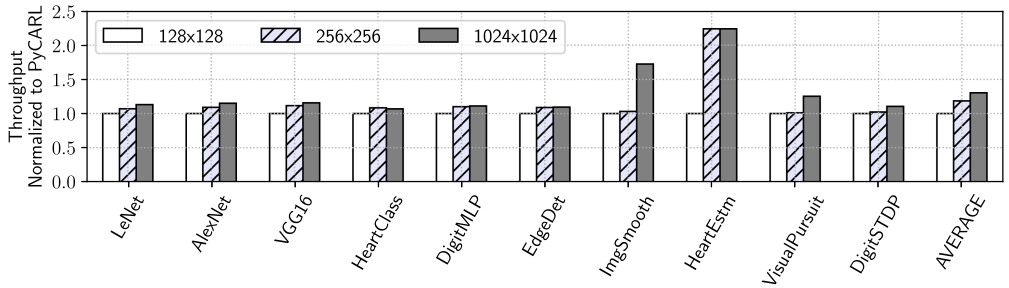

Fig. 24. Throughput achieved using DFSynthesizer for 128 × 128, 256 × 256, and 1024 × 1204 crossbars, normalized to throughput on DYNAP-SE with 128 × 128 crossbars.

between the clusters is higher in DFSynthesizer because of the higher number of post-synaptic neurons mapped to each cluster. Overall, the average cluster connections of DFSynthesizer is 3.1× higher than SpiNeMap and 3.9× higher than PyCARL.

8.7 Architecture Exploration

Figure 23 reports the number of clusters generated using DFSynthesizer for neuromorphic hardware with 128 × 128, 256 × 256, and 1024 × 1024 crossbars, normalized to a DYNAP-SE configuration with 128 × 128 crossbars. We observe that the number of clusters generated using DFSynthesizer reduces by 60% and 92% when the size of a crossbar increases to 256 × 256 and 1024 × 1024, respectively.

Fewer number of clusters increases throughput. To illustrate this, Figure 24 reports the throughput using DFSynthesizer for different crossbar sizes normalized to throughput on DYNAP-SE with four 128 × 128 crossbars. We make the following two observations.

First, throughput increases by 18% and 30% when using 256 × 256 and 1024 × 1024 crossbars, respectively. This improvement is because with larger-size crossbars, there are fewer clusters generated by DFSynthesizer (Figure 23). Therefore, the number of clusters per tile reduces, which reduces the bottleneck of time-multiplexing clusters on tiles. This increases throughput. Second, for applications such as DigitMLP, EdgeDet, and HeartEstm, there is no throughput improvement when the crossbar size increased from 512 × 512 to 1024 × 1024. This is because for these applications, a 256 × 256 crossbar configuration is sufficient to achieve the highest throughput. For all other applications, the throughput increases by 11% when going from 256 × 256 to 1024 × 1024 crossbars.

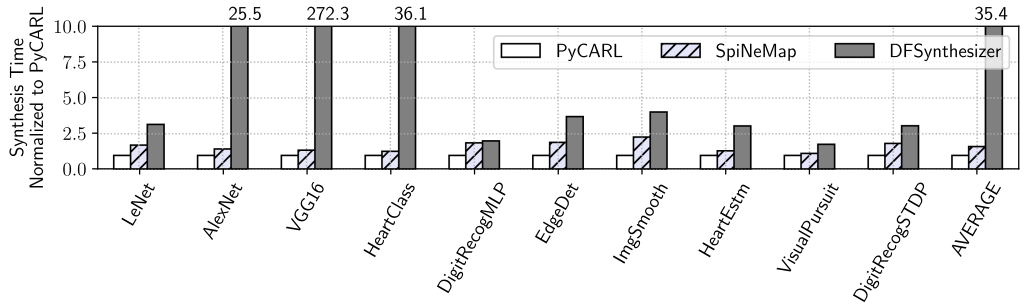

Fig. 25. Synthesis time for each application normalized to PyCARL.

8.8 Synthesis Time

Figure 25 reports the synthesis time on DYNAP-SE for the evaluated approaches, for each application normalized to PyCARL. We make the following three key observations.

First, the synthesis time of SpiNeMap is on average 61.6% higher than PyCARL. The higher synthesis time of SpiNeMap is due to the analysis it performs with the workload to obtain the minimum energy mapping. Second, the synthesis time of DFSynthesizer is the highest. On average, the synthesis time of DFSynthesizer is 35× higher than PyCARL and 25× higher than SpiNeMap. This higher synthesis time is due to (1) DFSynthesizer's mapping explorations using Algorithm 3 and (2) DFSynthesizer's SDFG analysis mechanism using the proposed Max Plus formulation. Third, the synthesis time of DFSynthesizer increases with model complexity. The synthesis time of DFSynthesizer is higher than PyCARL by 3.1× for LeNet, 25.5× for AlexNet, and 272.3× for VGG16.

8.9 Model Quality

DFSynthesizer does not alter synaptic connections. Therefore, the model quality, e.g., accuracy, is not impacted by the analysis technique of DFSynthesizer. The only impact DFSynthesizer introduces is in converting CNNs. The accuracy impact is reported in Table 1. For all other applications, DFSynthesizer's accuracy is the same as the baseline accuracy reported in Table 2.

9 RELATED WORKS

Recently, many approaches were proposed to map machine learning workloads to neuromorphic hardware. Corelet [2] is used to map SNNs to TrueNorth [47]. PACMAN [53] is used to map SNNs to SpiNNaker [52]. PyNN [7] is used to map SNNs on Loihi [45], BrainScaleS [80], and Neurogrid [21] by balancing the load on each tile. PyCARL [7] is used to map SNNs to DYNAP-SE [73]. The primary objective of these approaches is to balance the workload on each tile by distributing the neurons and synapses evenly.

Beyond load balancing, recent techniques have also explored other objectives. PSOPART [44] is used to map SNNs to neuromorphic hardware, reducing the energy consumption on the shared interconnect. SpiNeMap [11] performs energy-aware clustering of SNNs and then maps the clusters to tiles, reducing the communication energy. DecomposeSNN [14] decomposes an SNN to improve the cluster utilization. There are also performance-oriented SNN mapping approaches such as [10, 12, 16, 83], energy-aware SNN mapping approaches such as [101], circuit aging-aware SNN mapping approaches such as [13, 67, 84, 87, 91], endurance-aware SNN mapping approaches such as [93, 99, 102], and thermal-aware SNN mapping approaches such as [100]. These approaches are evaluated with emerging SNN-based applications [8, 32, 41, 50, 64, 75], which we also use to evaluate DFSynthesizer.

There are also other mapping approaches such as [4, 70, 77, 103–106]. We compared DFSynthesizer against PyCARL and SpiNeMap, and found it to perform significantly better.

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 [30, 43, 81, 81, 96, 98, 108]. Bonfietti et al. evaluated mappings of SDFG to a multiprocessor system, maximizing the throughput [23]. Stemmer et al. propose to use probabilistic analysis to allocate and schedule SDFGs on multiprocessor systems [95]. Das et al. evaluated the fault-tolerant mapping of SDFGs to multiprocessor systems [31, 33, 35–40, 42]. Recently, SDFG-based analysis was also proposed for analyzing machine learning applications [6, 9, 27, 34, 60, 92]. However, none of these approaches address application analysis with limited hardware resources, both at design time and at runtime.

10 CONCLUSIONS

We introduce DFSynthesizer for predictable synthesis 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 map machine learning applications to a resource-constrained hardware. DFSynthesizer makes five key contributions. First, we present an approach to analyze machine learning programs and generate SNN workload using representative data. Second, we present an approach to decompose and partition complex SNN workloads to generate clusters of neurons and synapses such that each cluster can fit onto a crossbar of the hardware. Third, we exploit the rich semantics of SDFGs to represent clustered SNN programs. This allows for the SNN's performance, e.g., throughput, to be estimated on the hardware as a function of key properties such as number of crossbars, dimension of crossbars, buffer space on tiles, and tile communication bandwidth. Fourth, we develop a novel scheduling algorithm based on Self-Timed Execution for executing clusters on crossbars of a neuromorphic hardware, providing performance guarantee in scenarios with dynamic resource availability. Five, we propose a design-space exploration framework incorporating DFSynthesizer that allows the Pareto space of different SNN mappings to hardware to be explored while considering other hardware metrics such as energy, latency, and reliability.

We evaluate DFSynthesizer using 10 machine learning programs that are representative of the three most commonly used neural network classes: CNN, MLP, and RNN. Our results demonstrate that DFSynthesizer provides a much tighter performance guarantee compared to current practices.

APPENDIX

A CONVERTING ANALOG OPERATIONS TO SPIKING EQUIVALENT

In this section, we briefly elaborate how an analog operation such as **Rectified Linear Unit (ReLU)** is implemented using SNN. The output Y of a ReLU activation function is given by

$$Y = \max 0, \sum_i w_i * x_i, \tag{16}$$

where w_i is the weight and x_i is the activation on the i^{th} synapse of the neuron. To map the ReLU activation function, we consider a particular type of spiking neuron model known as an **Integrate and Fire (IF)** neuron model. The IF spiking neuron's transfer function can be represented as

$$v_m(t+1) = v_m(t) + \sum_i w_i * x_i(t), \tag{17}$$

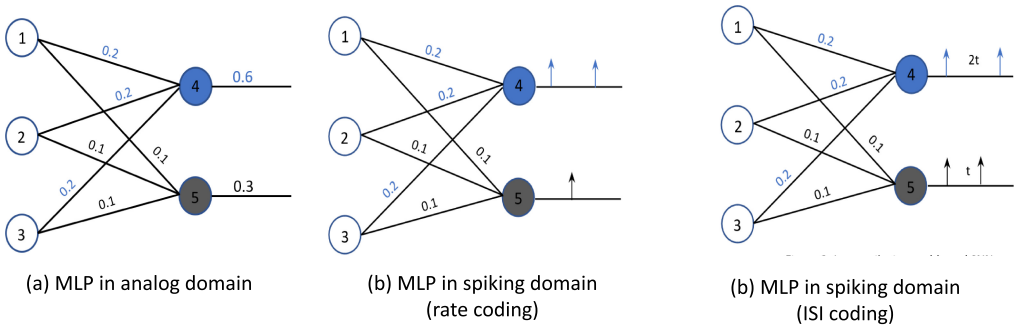

Fig. 26. Example of converting an analog MLP to its spiking equivalent.

where $v_m(t)$ is the membrane potential of the IF neuron at time t, w_i is the weight, and $x_i(t)$ is the activation on the i^{th} synapse of the neuron at time t. The IF spiking neuron integrates incoming spikes (X_i) and generates an output spike (Y_{spike}) when the membrane potential (v_m) exceeds the threshold voltage (v_{th}) of the IF neuron. Therefore, by ensuring that the output spiking rate Y_{spike} is proportional to the ReLU activation Y, i.e., $Y_{\text{spike}} \propto Y$, we accurately convert the ReLU activation to the spike-based model. To further illustrate this, we consider the MLP of Figure 26(a) and its SNN conversion using rate-based encoding (Figure 26(b)) and ISI encoding (Figure 26(c)).

In Figure 26(a), neurons 1, 2, and 3 are the input neurons and neurons 4 and 5 are the output neurons. To keep the model simple, let us consider the case where the activations of the input neurons 1, 2, and 3 are equal to 1. Using Equation (16), we know that the output of neurons 4 and 5 are 0.6 and 0.3, respectively. Figures 26(b) and 26(c) show the mapped SNN model using rate-based and inter-spike interval encoding schemes, respectively. In the rate-based model in Figure 26(b), the rate of spikes generated is expected to be proportional to the output of neurons 4 and 5 in the MLP. In the case of the ISI-based SNN model, the inter-spike interval of the spikes generated by neurons 4 and 5 is expected to be proportional to the output generated in the MLP, as shown in Figure 26(c).

We note that non-linear activation functions such as sigmoid and tanh cannot be accurately mapped to a spike-based model. This can be attributed to the transfer function of a biological spiking neuron (neuron response curve) closely resembling a ReLU and not sigmoid and tanh activation functions. While approximate implementations of the sigmoid and tanh operators using spiking neurons can be found in the literature, they induce significant inaccuracies into the conversion process and require more resources (neurons) to implement. The tanh activation function, for instance, generates output values ranging between −1.0 and 1.0. In order to represent the tanh function in a spike-based model, both excitatory and inhibitory spiking neurons will be required to represent the positive and negative output values, respectively. This will require doubling the number of spiking neurons needed to represent the tanh activation function.

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)*.

[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)*.

[3] Aayush Ankit, Abhronil Sengupta, and Kaushik Roy. 2017. TraNNsformer: Neural network transformation for memristive crossbar based neuromorphic system design. In *International Conference on Computer-Aided Design (ICCAD)*.

[4] Aayush Ankit, Abhronil Sengupta, and Kaushik Roy. 2018. Neuromorphic computing across the stack: Devices, circuits and architectures. In *International Workshop on Signal Processing Systems (SIPS)*.
[5] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A survey. *Computer Networks*. 54, 15 (2010), 2787–2805.
[6] 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 (IPDPS) Workshops*.
[7] 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)*.
[8] 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.
[9] Adarsha Balaji and Anup Das. 2019. A framework for the analysis of throughput-constraints of SNNs on neuromorphic hardware. In *IEEE Annual Symposium on VLSI (ISVLSI)*.
[10] Adarsha Balaji and Anup Das. 2020. Compiling spiking neural networks to mitigate neuromorphic hardware constraints. In *International Green and Sustainable Computing Conference (IGSC) Workshops*.
[11] 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. 2020. Mapping spiking neural networks to neuromorphic hardware. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 1 (2020), 76–86.
[12] Adarsha Balaji, Thibaut Marty, Anup Das, and Francky Catthoor. 2020. Run-time mapping of spiking neural networks to neuromorphic hardware. *Journal of Signal Processing Systems* 92, 11 (2020), 1293–1302.
[13] 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. *Computer Architecture Letters* 18, 2 (2019), 149–152.
[14] Adarsha Balaji, Shihao Song, Anup Das, Jeffrey Krichmar, Nikil Dutt, James Shackleford, Nagarajan Kandasamy, and Francky Catthoor. 2020. Enabling resource-aware mapping of spiking neural networks via spatial decomposition. *Embedded Systems Letters* 13, 3 (2020), 142–145.
[15] Adarsha Balaji, Shihao Song, Twisha Titirsha, Anup Das, Jeffrey Krichmar, Nikil Dutt, James Shackleford, Nagarajan Kandasamy, and Francky Catthoor. 2021. NeuroXplorer 1.0: An extensible framework for architectural exploration with spiking neural networks. In *International Conference on Neuromorphic Systems (ICONS)*.
[16] Adarsha Balaji, Salim Ullah, Anup Das, and Akash Kumar. 2019. Design methodology for embedded approximate artificial neural networks. In *Great Lakes Symposium on VLSI (GLSVLSI)*.
[17] Adarsha Balaji, Yuefeng Wu, Anup Das, Francky Catthoor, and Siebren Schaafsma. 2019. Exploration of segmented bus as scalable global interconnect for neuromorphic computing. In *Great Lakes Symposium on VLSI (GLSVLSI)*.
[18] Shuvra S. Battacharyya, Praveen K. Murthy, and Edward A. Lee. 1996. Loose interdependence algorithms. In *Software Synthesis from Dataflow Graphs*.
[19] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence C. Stewart, Daniel Rasmussen, Xuan Choo, Aaron Voelker, and Chris Eliasmith. 2014. Nengo: A python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*.
[20] Luca Benini and Giovanni De Micheli. 2002. Networks on chip: A new paradigm for systems on chip design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.
[21] 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.
[22] O. Bichler, D. Briand, V. Gacoin, and B. Bertelone. 2017. N2D2: Neural network design & deployment. https://github.com/CEA-LIST/N2D2.
[23] 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.
[24] Geoffrey W. Burr, Robert M. Shelby, et al. 2017. Neuromorphic computing using non-volatile memory. *Advances in Physics: X* 2, 1 (2017), 89–124.
[25] Francky Catthoor, Srinjoy Mitra, Anup Das, and Siebren Schaafsma. 2018. Very large-scale neuromorphic systems for biological signal processing. In *CMOS Circuits for Biological Sensing and Processing*.
[26] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2016), 127–138.
[27] 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.

[28] 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)*.

[29] Jason Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *Design Automation Conference (DAC)*.

[30] 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)*.

[31] Anup Das, Bashir M. Al-Hashimi, and Geoff V. Merrett. 2016. Adaptive and hierarchical runtime manager for energy-aware thermal management of embedded systems. *ACM Transactions on Embedded Computing Systems* 15, 2 (2016), 1–25.

[32] Anup Das, Francky Catthoor, and Siebren Schaafsma. 2018. Heartbeat classification in wearables using multi-layer perceptron and time-frequency joint distribution of ECG. In *International conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*.

[33] Anup Das and Akash Kumar. 2012. Fault-aware task re-mapping for throughput constrained multimedia applications on NoC-based MPSoCs. In *International Workshop on Rapid System Prototyping (RSP)*.

[34] Anup Das and Akash Kumar. 2018. Dataflow-based mapping of spiking neural networks on neuromorphic hardware. In *Great Lakes Symposium on VLSI (GLSVLSI)*.

[35] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. 2012. Energy-aware communication and remapping of tasks for reliable multimedia multiprocessor systems. In *International Conference on Parallel and Distributed Systems (ICPADS)*.

[36] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. 2013. Aging-aware hardware-software task partitioning for reliable reconfigurable multiprocessor systems. In *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*.

[37] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. 2013. Communication and migration energy aware design space exploration for multicore systems with intermittent faults. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[38] 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.

[39] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. 2014. Energy-aware task mapping and scheduling for reliable embedded computing systems. *ACM Transactions on Embedded Computing Systems* 13, 2s (2014), 1–27.

[40] 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.

[41] A. Das, P. Pradhapan, W. Groenendaal, P. Adiraju, R. T. Rajan, F. Catthoor, S. Schaafsma, J. L. Krichmar, N. Dutt, and C. Van Hoof. 2018. Unsupervised heart-rate estimation in wearables with Liquid states and a probabilistic readout. *Neural Networks* 99 (2018), 134–147.

[42] Anup Das, Amit Kumar Singh, and Akash Kumar. 2013. Energy-aware dynamic reconfiguration of communication-centric applications for reliable MPSoCs. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*.

[43] Anup Das, Matthew J. Walker, Andreas Hansson, Bashir M. Al-Hashimi, and Geoff V. Merrett. 2015. Hardware-software interaction for run-time power optimization: A case study of embedded Linux on multicore smartphones. In *International Symposium on Low Power Electronics and Design (ISLPED)*.

[44] 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)*.

[45] 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.

[46] 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.

[47] 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.

[48] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

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

[50] Peter U. Diehl and Matthew Cook. 2015. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in Computational Neuroscience* 9 (2015).
[51] Jochen M. Eppler, Moritz Helias, Eilif Muller, Markus Diesmann, and Marc-Oliver Gewaltig. 2009. PyNEST: A convenient interface to the NEST simulator. *Frontiers in Neuroinformatics* 2 (2009), 12.
[52] Steve B. Furber, Francesco Galluppi, Steve Temple, and Luis A. Plana. 2014. The SpiNNaker project. *Proc. IEEE* 102, 5 (2014), 652–665.
[53] 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.
[54] Amir Hossein Ghamarian, Marc C. W. Geilen, Sander Stuijk, Twan Basten, Bart D. Theelen, Mohammad Reza Mousavi, Arno J. M. Moonen, and Marco J. G. Bekooij. 2006. Throughput analysis of synchronous data flow graphs. In *International Conference on Application of Concurrency to System Design (ACSD)*.
[55] Dan F. M. Goodman and Romain Brette. 2009. The brian simulator. *Frontiers in Neuroscience* 3 (2009), 26.
[56] Roshan Gopalakrishnan, Yansong Chua, Pengfei Sun, Ashish Jith Sreejith Kumar, and Arindam Basu. 2020. HFNet: A CNN architecture co-designed for neuromorphic hardware with a crossbar array of synapses. *Frontiers in Neuroscience* 14 (2020).
[57] Antonio Gulli and Sujit Pal. 2017. *Deep Learning with Keras*.
[58] 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.
[59] Michael L. Hines and Nicholas T. Carnevale. 1997. The NEURON simulation environment. *Neural Computation*. 9, 6 (1997), 1179–1209.
[60] Hyesun Hong, Hyunok Oh, and Soonhoi Ha. 2017. Hierarchical dataflow modeling of iterative applications. In *Design Automation Conference (DAC)*.
[61] Miao Hu, John Paul Strachan, Zhiyong Li, Emmanuelle M. Grafals, Noraica Davila, Catherine Graves, Sity Lam, Ning Ge, Jianhua Joshua Yang, and R. Stanley Williams. 2016. Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication. In *Design Automation Conference (DAC)*.
[62] Giacomo Indiveri. 2003. A low-power adaptive integrate-and-fire neuron circuit. In *IEEE International Symposium on Circuits and Systems (ISCAS)*.
[63] 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)*.
[64] Hirak J. Kashyap, Georgios Detorakis, Nikil Dutt, Jeffrey L. Krichmar, and Emre Neftci. 2018. A recurrent neural network based model of predictive smooth pursuit eye movement in primates. In *International Joint Conference on Neural Networks (IJCNN)*.
[65] Brian W. Kernighan and Shen Lin. 1970. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal* 49, 2 (1970), 291–307.
[66] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems* 25 (2012), 1097–1105.
[67] Shamik Kundu, Kanad Basu, Mehdi Sadi, Twisha Titirsha, Shihao Song, Anup Das, and Ujjwal Guin. 2021. Special session: Reliability analysis for ML/AI hardware. In *IEEE VLSI Test Symposium (VTS)*.
[68] Yann LeCun et al. 2015. LeNet-5, convolutional neural networks. http://yann.lecun.com/exdb/lenet.
[69] E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.
[70] 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.
[71] Wolfgang Maass. 1997. Networks of spiking neurons: The third generation of neural network models. *Neural Networks* 10, 9 (1997), 1659–1671.
[72] A. Mallik, D. Garbin, A. Fantini, D. Rodopoulos, R. Degraeve, J. Stuijt, A. K. 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*.
[73] Saber Moradi, Ning Qiao, Fabio Stefanini, and Giacomo Indiveri. 2017. A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs). *IEEE Transactions on Biomedical Circuits and Systems* 12, 1 (2017), 106–122.
[74] Orlando M. Moreira and Marco J. G. Bekooij. 2007. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing* 2007 (2007), 1–14.
[75] Ethan J. Moyer, Anup Das, et al. 2020. Machine learning applications to DNA subsequence and restriction site analysis. In *IEEE Signal Processing in Medicine and Biology Symposium*.

[76] 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. *Neural Information Processing Systems* 32 (2019).

[77] 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)*.

[78] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2020. Mlperf inference benchmark. In *International Symposium on Computer Architecture (ISCA)*.

[79] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, and Michael Pfeiffer. 2016. Theory and tools for the conversion of analog to spiking convolutional neural networks. *arXiv*.

[80] 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 *IEEE International Symposium on Circuits and Systems (ISCAS)*.

[81] Rishad A. Shafik, Anup Das, Sheng Yang, Geoff Merrett, and Bashir M. Al-Hashimi. 2015. Adaptive energy minimization of openMP parallel applications on many-core systems. In *Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA)/Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (DITAM)*.

[82] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv*.

[83] Shihao Song, Adarsha Balaji, Anup Das, Nagarajan Kandasamy, and James Shackleford. 2020. Compiling spiking neural networks to neuromorphic hardware. In *International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[84] Shihao Song and Anup Das. 2020. A case for lifetime reliability-aware neuromorphic computing. In *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*.

[85] Shihao Song and Anup Das. 2020. Design methodologies for reliable and energy-efficient PCM systems. In *International Green and Sustainable Computing Conference (IGSC) Workshops*.

[86] Shihao Song, Anup Das, and Nagarajan Kandasamy. 2020. Exploiting inter- and intra-memory asymmetries for data mapping in hybrid tiered-memories. In *International Symposium on Memory Management (ISMM)*.

[87] Shihao Song, Anup Das, and Nagarajan Kandasamy. 2020. Improving dependability of neuromorphic computing with non-volatile memory. In *European Dependable Computing Conference*.

[88] Shihao Song, Anup Das, Onur Mutlu, and Nagarajan Kandasamy. 2019. Enabling and exploiting partition-level parallelism (PALP) in phase change memories. *ACM Transactions on Embedded Computing Systems* 18, 5s (2019), 1–25.

[89] Shihao Song, Anup Das, Onur Mutlu, and Nagarajan Kandasamy. 2020. Improving phase change memory performance with data content aware access. In *International Symposium on Memory Management (ISMM)*.

[90] Shihao Song, Anup Das, Onur Mutlu, and Nagarajan Kandasamy. 2021. Aging-aware request scheduling for non-volatile main memory. In *Asia and South Pacific Design Automation Conference (ASPDAC)*.

[91] Shihao Song, Jui Hanamshet, Adarsha Balaji, Anup Das, Jeff Krichmar, Nikil Dutt, Nagarajan Kandasamy, and Francky Catthoor. 2021. Dynamic reliability management in neuromorphic computing. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 17, 4 (2021), 1–27.

[92] Shihao Song, Ankita Paul, Lakshmi Varshika Mirtinti, Anup Das, and Nagarajan Kandasamy. 2021. A design flow for mapping spiking neural networks to many-core neuromorphic hardware. In *International Conference on Computer-Aided Design (ICCAD)*.

[93] Shihao Song, Twisha Titirsha, and Anup Das. 2021. Improving inference lifetime of neuromorphic systems via intelligent synapse mapping. In *International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*.

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

[95] 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 (ECRTS)*.

[96] Sander Stuijk, Twan Basten, M. C. W. Geilen, and Henk Corporaal. 2007. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Design Automation Conference (DAC)*.

[97] 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)*.

[98] Sander Stuijk, Marc Geilen, and Twan Basten. 2006. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Design Automation Conference (DAC)*.

[99] Twisha Titirsha and Anup Das. 2020. Reliability-performance trade-offs in neuromorphic computing. In *International Green and Sustainable Computing Conference (IGSC) Workshops*.
[100] Twisha Titirsha and Anup Das. 2020. Thermal-aware compilation of spiking neural networks to neuromorphic hardware. In *Languages and Compilers for Parallel Computing (LCPC) Workshop*.
[101] Twisha Titirsha, Shihao Song, Adarsha Balaji, and Anup Das. 2021. On the role of system software in energy management of neuromorphic computing. In *ACM International Conference on Computing Frontiers*.
[102] Twisha Titirsha, Shihao Song, Anup Das, Jeffrey Krichmar, Nikil Dutt, Nagarajan Kandasamy, and Francky Catthoor. 2021. Endurance-aware mapping of spiking neural networks to neuromorphic hardware. *IEEE Transactions on Parallel and Distributed Systems* 33, 2 (2021), 288–301.
[103] 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)*.
[104] 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 (TETCI)* 2, 5 (2018), 345–358.
[105] Qiangfei Xia and J. Joshua Yang. 2019. Memristive crossbar arrays for brain-inspired computing. *Nature Materials* 18, 4 (2019), 309.
[106] 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.
[107] Zhiru Zhang and Bin Liu. 2013. SDC-based modulo scheduling for pipeline synthesis. In *International Conference on Computer-Aided Design (ICCAD)*.
[108] Xue-Yang Zhu, Marc Geilen, Twan Basten, and Sander Stuijk. 2012. Static rate-optimal scheduling of multirate DSP algorithms via retiming and unfolding. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Received November 2020; revised June 2021; accepted August 2021