Real-Time Scheduling of Machine Learning Operations on Heterogeneous Neuromorphic SoC

Anup Das
Electrical and Computer Engineering
Drexel University
Philadelphia, USA
anup.das@drexel.edu

***Abstract*—Neuromorphic Systems-on-Chip (NSoCs) are becoming heterogeneous by integrating general-purpose processors (GPPs) and neural processing units (NPUs) on the same SoC. For embedded systems, an NSoC may need to execute user applications built using a variety of machine learning models. We propose a real-time scheduler, called PRISM, which can schedule machine learning models on a heterogeneous NSoC either individually or concurrently to improve their system performance. PRISM consists of the following four key steps. First, it constructs an interprocessor communication (IPC) graph of a machine learning model from a mapping and a self-timed schedule. Second, it creates a transaction order for the communication actors and embeds this order into the IPC graph. Third, it schedules the graph on an NSoC by overlapping communication with the computation. Finally, it uses a Hill Climbing heuristic to explore the design space of mapping operations on GPPs and NPUs to improve the performance. Unlike existing schedulers which use only the NPUs of an NSoC, PRISM improves performance by enabling batch, pipeline, and operation parallelism via exploiting a platform's heterogeneity. For use-cases with concurrent applications, PRISM uses a heuristic resource sharing strategy and a non-preemptive scheduling to reduce the expected wait time before concurrent operations can be scheduled on contending resources. Our extensive evaluations with 20 machine learning workloads show that PRISM significantly improves the performance per watt for both individual applications and use-cases when compared to state-of-the-art schedulers.**

***Index Terms*—neuromorphic system-on-chip (NSoC), spiking deep convolution neural network (SDCNN), interprocessor communication (IPC), hill climbing, self-timed execution.**

I. Introduction

EVENT-driven neuromorphic devices are a promising solution to accelerate Spiking Deep Convolutional Neural Networks (SDCNNs) [1]–[3]. These are a type of Spiking Neural Networks (SNNs) that replicate the neural architecture of a conventional CNN with two major differences [4]. First, layers of an SDCNN communicate by exchanging spikes instead of tensors. Second, each layer implements the leaky integrate-and-fire function instead of a sigmoidal function (Tanh or ReLU). Due to these differences, converting a CNN to an equivalent SDCNN is not straightforward, and it requires layer-specific optimizations [5]–[10].

Our objective is to schedule SDCNNs on a neuromorphic device that consists of neural processing units (NPUs) to accelerate operations such as matrix multiplication, average/max pooling, batch normalization, layer flattening, residual computation, and concatenation [11]–[15]. These devices are now integrated with general purpose processors (GPPs) and other hardware subsystems on the same system-on-chip (SoC) to implement a complete system. Neuromorphic systems-on-chip (NSoCs) are becoming the key enabler for embedded systems to accelerate machine learning-based user applications by scheduling their operations on NPUs. Scheduling consists of three steps — mapping, which involves assigning operations to different NPUs, ordering, which involves determining the execution order of operations and their communication, and timing, which involves specifying the precise start time of an operation on an NPU [16]–[20].

Table I reports event-driven NSoC platforms (top four rows) and their system software to schedule SDCNN operations.[1] It also reports a few standalone neuromorphic devices and their software (bottom four rows). These devices are not currently part of any SoC but reported here for completeness.

TABLE I
RECENTLY INTRODUCED NSOC PLATFORMS.

NSoC	# NPUs	GPPs	Software
Loihi [24]	128	3 x86 CPUs	LAVA [25]
Akida [26]	80	1 x86 CPU	MetaTF [27]
GrAI [28]	144	2 ARM CPUs	GrAIFlow [28]
SPECK [29]	> 16	1 GPU	CTXCTL [29]
SpiNNaker [30]	144	–	PACMAN [31]
μBrain [32]	64	–	SentryOS [32]
Tianji [33]	6	–	NEUTRAMS [34]
DYNAPs [35]	4	–	SpiNeMap [36]–[38]

Figure 1a shows the architecture of an NSoC with GPPs and NPUs, where NPUs are arranged in an XY mesh. Internally, each NPU consists of circuitries to perform neural computations and memory to store synaptic weights. Unlike a GPP, an NPU does not require frequent memory accesses due to its self-contained memory architecture and therefore, it does not suffer from the memory bandwidth bottleneck. Due to its small hardware area, an NSoC may integrate several NPUs (see Table I) to process a batch of data at once. This is called batch parallelism. A system software exploits this parallelism to improve the throughput. Following are the three key limitations of existing schedulers that motivate this work.

First, existing schedulers do not exploit all forms of parallelism that exist in an NSoC. For instance, an NSoC can sup-

[1]There are also tensor-based NSoCs such as Qualcomm's Snapdragon [21] and Nvidia's Jetson AGX Xavier [22]. These NSoCs are evaluated in [23].

port pipeline parallelism, where the processing of operations of subsequent input data can be overlapped in time by utilizing the platform's heterogeneous computing resources. It can also support operation parallelism, where independent operations acting on the same input data can be scheduled concurrently on multiple computing resources. Figure 3 illustrates the difference between these three forms of parallelism.

Second, machine learning is an evolving area of research and new models are introduced to the community on a regular basis. We analyze five models – SqueezeNet [39], InceptionV3 [40], U-Net [41], BERT [42], and ConvLSTM [43], for their support on existing NSoCs. Table II summarizes these results. While all NSoCs support SqueezeNet, none of them support U-Net, BERT, and ConvLSTM. On the other hand, GrAI is the only hardware that supports InceptionV3. Existing schedulers cannot map a machine learning model on an NSoC if *any* of its operations is not supported on the NSoC's NPU.

TABLE II
ANALYSIS OF EXISTING NSOC PLATFORMS.

NSoC	SqueezeNet	InceptionV3	U-Net	BERT	ConvLSTM	Custom Model
Loihi	√	×	×	×	×	×
Akida	√	×	×	×	×	×
GrAI	√	√	×	×	×	×
SPECK	√	×	×	×	×	×

Third, an NSoC may need to execute different machine learning applications concurrently to satisfy user demands. Imagine a use-case of running social media services (image classification using MobileNet or InceptionV3) while listening to music (audio processing using BERT or ConvLSTM) on a cell phone. Existing schedulers cannot schedule more than one machine learning application at the same time.

We propose PRISM, a performance-oriented real-time scheduler for machine learning operations on a heterogeneous NSoC. Following are our key **contributions**.

1) We propose a mechanism to construct an interprocessor communication graph from a machine learning model using a mapping and a self-timed schedule of its operations on the computing resources. We exploit the expressiveness of a synchronous dataflow graph (SDFG) to represent an IPC graph.
2) We propose a transaction partial order algorithm to create a transaction order for the inter-processor communications of an IPC graph. We embed this transaction order into the graph and schedule it on an NSoC so that communication is overlapped with the computation.
3) We propose a Hill Climbing heuristic to explore the design space of scheduling machine learning operations on the computing resources and create opportunities for batch, pipeline, and operation parallelism via exploiting the platform's heterogeneity.
4) We propose a probabilistic formulation to estimate the performance slowdown due to resource contention. We incorporate this formulation within a use-case mapping framework that uses a heuristic resource sharing strategy and a non-preemptive scheduling to map applications of a use-case on an NSoC. PRISM minimizes the performance slowdown by reducing the expected wait time for operations scheduled on contending resources.

Our extensive evaluations with 20 machine learning workloads and five use-cases show that PRISM significantly improves the performance and performance per watt for both individual applications and multi-application use-cases when compared to state-of-the-art schedulers.

To the best of our knowledge, PRISM is the only scheduler that can schedule machine learning applications (standard or custom) either individually or concurrently by exploiting the heterogeneity of an NSoC platform.

Why Spiking? PRISM primarily deals with trained spiking CNN (i.e., SDCNN inference) models because spiking hardware platforms are energy-efficient due to their event-driven operations. Therefore, these platforms are ideal for embedded systems and other environments where machine learning tasks may need to be performed within an energy budget. Nevertheless, with simple modifications PRISM can also schedule operations of a conventional CNN.

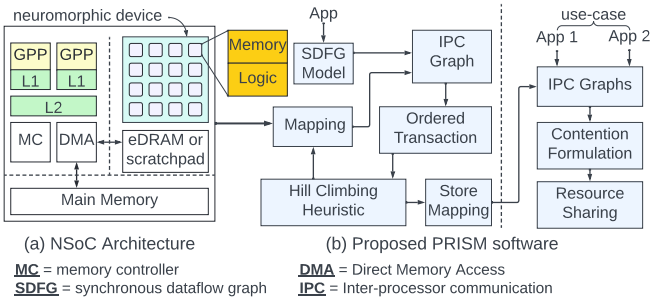


Fig. 1. (a) An NSoC platform with GPPs and NPUs. (b) Proposed scheduler PRISM which schedules both individual applications and use-cases.

Overview of PRISM: Figure 1b shows an overview of PRISM. It has two main components – individual application exploration (left) and use-case mapping strategy (right). For mapping individual applications, PRISM records all Pareto-optimal schedules generated from the proposed Hill Climbing heuristic. It selects a schedule that gives the highest performance for a given energy constraint. For mapping use-cases, PRISM finds the best strategy to share resources amongst the concurrent applications. PRISM improves the quality of experience by minimizing the time from when an application is invoked to when it starts executing.

II. BACKGROUND AND MOTIVATION

To illustrate the different forms of parallelism in an NSoC, Figure 2 shows an embedded platform performing audio and video processing. A compiler compiles a user application into an intermediate representation, which is shown to the right with four operations (opx 1-4). A scheduler schedules these operations on the computing resources of the hardware. On the input front, audio/video frames are first streamed into buffers. Once a batch is ready, the scheduler distributes it into mini-batches and forwards them to the computing resources, one at time. The number of frames in each mini-batch is equal to the number of NPUs so they can be scheduled in parallel.

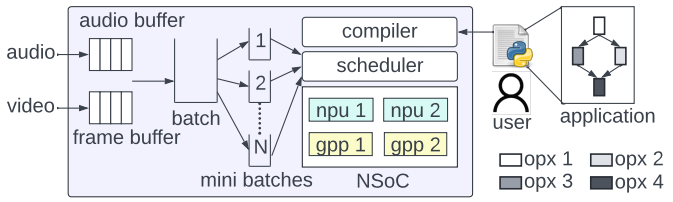


Fig. 2. An embedded platform running a machine learning application.

Figure 3 illustrates two different schedules. In ❶, we show an existing scheduler that uses two NPUs. Each NPU operates on a single input data from a mini-batch and performs all four operations shown in Figure 2 (right). Once done, these NPUs operate on the next mini-batch. This is batch parallelism. Most schedulers exploit this parallelism to improve the performance.

In ❸, we illustrate how PRISM schedules the mini-bathes. We make the following five observations. First, PRISM uses all computing resources (two GPPs and two NPUs). Second, unlike the baseline schedule where each NPU processes all four operations sequentially, here each computing resource processes a specific operation for every input data. Essentially, PRISM creates a four-stage pipeline with NPU 1, GPP 1, GPP 2, and NPU 2, respectively. The processing of a new input can begin when the first pipeline stage (NPU-1) completes its execution. This is pipeline parallelism.

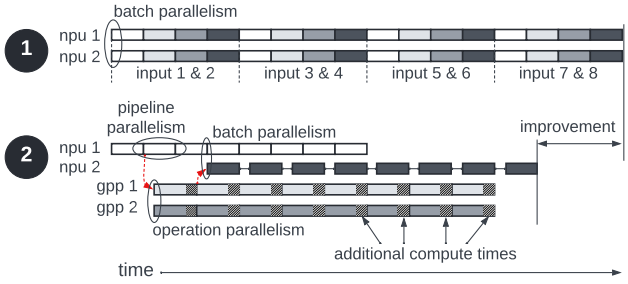


Fig. 3. An example showing the different forms of parallelism in an NSoC.

Third, PRISM schedules operations 2 & 3 on the two GPPs. The figure illustrates additional computation times necessary to process these operations on a GPP compared to an NPU. Fourth, operations 2 & 3 can be scheduled in parallel as they are independent (see Figure 2). PRISM schedules these operations on the two GPPs. This is the operation parallelism, which also contributes to performance improvement.

Finally, data exchanges are necessary between the computing resources. To move data from a GPP to an NPU, PRISM first copies the data from the GPP's cache to the main memory if it is not already there. Next, it initiates a data transfer from the main memory to the memory of a neuromorphic device using the DMA module. Once loaded, it distributes this data to the target NPU using the interconnect. Typically, a neuromorphic device integrates an embedded DRAM (eDRAM) or a scratchpad memory for data storage as shown in Figure 1a. Conversely, to move data from an NPU to a GPP, PRISM first copies the data to the main memory using the DMA module. Thereafter, a GPP loads this data into its cache using its cache placement/replacement policies. These communications are indicated using red arrows. PRISM overlaps data communication with the computation, which further improves the performance. We report 2.2x higher performance compared to state-of-the-art schedulers (Section VI).

A. Synchronous Dataflow Graphs

Synchronous Data Flow Graphs (SDFGs, see [44]) are used to model streaming applications that are implemented on a multi-processor SoC (MPSoCs, see [45]). These graphs are used to analyze a system in terms of execution time [46], throughput [47], buffer requirements [48], energy [49], power [50], temperature [51], and reliability [52].

PRISM models a machine learning application as an SDFG. Nodes of an SDFG are called *actors* and they compute by reading *tokens*. A token is the smallest unit of data communicated between actors. To model an SDCNN as an SDFG, we consider the SDCNN application to be composed of a set of operations (convolution, pooling, batch normalization, etc). We represent each operation as an actor and spikes generated from these operations as tokens. Before an actor starts its execution, it reads tokens from its input ports. After completing its execution, it produces tokens on all of its output ports. The number of tokens produced or consumed in one execution of an actor is called the *port rate*. 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 computing unit. Edges in the graph are called *channels* and they represent dependencies among actors.

An actor is called *ready* when it has sufficient input tokens on all its input channels and sufficient buffer space on all its output channels; an actor can only fire when it is ready.

Definition 1: (ACTOR) *An actor* $\mathbf{a}_i$ *is the tuple* $\langle I_i, O_i, \tau_i, \mu_i \rangle$ *consisting of a set* I_i *(*$\subseteq Ports$*) of input ports, a set* O_i *(*$\subseteq Ports$*) of output ports,* τ_i *is the execution time of* $\mathbf{a}_i$ *and* μ_i *is its state space, which is the memory required to store synaptic weights, and input and output spikes.*

The execution time τ_i of actor $\boldsymbol{a}_i$ is the tuple with its execution time on different computing resources.

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

In our formulation, channels are delayless, i.e., tokens produced in one invocation of an actor are consumed within the iteration. One important performance property of an SDFG is its *throughput*, which is defined as the inverse of its long-term period. A period is the average time needed for one iteration of an SDFG. An iteration is defined as the minimum non-zero execution such that the original state of an SDFG is obtained. This is our performance metric.

B. Performance Estimation of an SDFG

The long-term throughput of an SDFG can be computed by analyzing its maximum cycle mean (MCM). We introduce the following definitions.

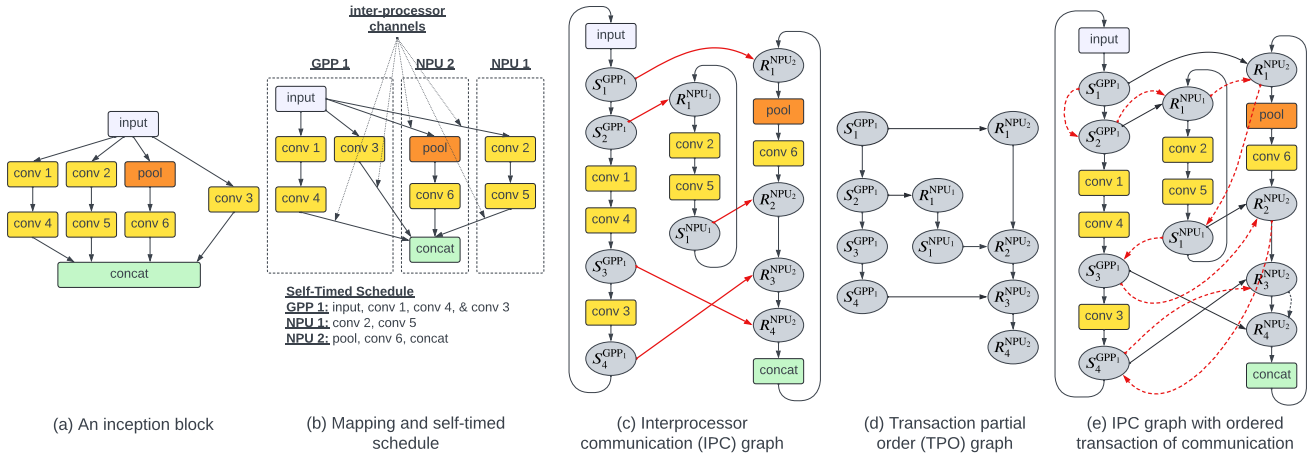


Fig. 4. (a) An inception block. (b) Self-timed schedule of the SDFG representing the inception block. (c) interprocessor communication (IPC) graph representing the self-timed execution. (d) A transaction partial order (TPO) graph formed by eliminating the computing actors. This graph is used to create a transaction order of these communication actors. (e) The IPC graph embedded with the transaction order.

Definition 3: (DIGRAPH) *The digraph* $\Gamma(T)$ *of a* $n \times n$ *matrix* T *with entries defined in* $\mathbb{R}_{max}$ *is the tuple* $\langle A, E \rangle$*, where* A *is the set of vertices, i.e.,* $A = \{1, 2, \cdots n\}$ *and* E *is the set of connected ordered arcs between vertices i.e.,* $E = \{(i,j) \mid T_{i,j} \neq -\infty\}$.

Definition 4: (WALK) *A walk* w *in digraph* $\Gamma(T)$ *is the sequence of arcs* $(x_1, x_2)(x_2, x_3) \cdots (x_{k-1}, x_k)$*; head of an arc in the sequence is either the start vertex of the walk or tail vertex of a preceding arc; and the tail vertex of an arc in the sequence is either the end vertex of the walk or head vertex of a succeeding arc. Weight of the walk is given by*

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

Definition 5: (CYCLE) *A cycle* c *in digraph* $\Gamma(T)$ *is the walk* $(x_1, x_2)(x_2, x_3) \cdots (x_{k-1}, x_k)$*, such that* $x_k = x_1$.

Definition 6: (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.,*

$$\rho_{max}(T) = \max_{\forall c \text{ in } \Gamma(T)} \frac{|c|_T}{|c|} = \max_{x_1, \cdots, x_{k-1}} \frac{T_{x_1 x_2} + \cdots T_{x_{k-1} x_k}}{k-1} \tag{2}$$

Throughput of an SDFG is measured as the inverse of its *maximum cycle mean* (Equation 2), i.e.,

$$\text{Performance (throughput)} = \frac{1}{\rho_{\max}(T)} \tag{3}$$

With this background, we now introduce PRISM.

III. PRISM: A PERFORMANCE ORIENTED REAL-TIME SCHEDULER FOR MACHINE LEARNING OPERATIONS

PRISM operates in three steps. First, it creates an interprocessor communication (IPC) graph from a machine learning application using the mapping of its operations on an NSoC's resources and a self-timed schedule. Next, it estimates the throughput by embedding a transaction order for the inter-processor communication into the IPC graph and applying the maximum cycle mean formulation (Definition 6). Finally, it explores the design space of scheduling operations to resources using a Hill Climbing heuristic to maximize the throughput. We now discuss these steps in details.

A. Creating IPC Graph

PRISM uses SDFGs to model an IPC graph. We illustrate this using the example of an inception block shown in Figure 4a. This is the building block of our evaluated InceptionV3 machine learning model [40]. There are 9 operations (called actors) – input, conv 1-6, pool, and concat. PRISM uses the self-timed execution, where each computing resource executes the actors assigned to it in a fixed order that is specified at compile time. Before firing an actor, it must wait for all of the actor's tokens to be available. We illustrate a self-timed schedule in Figure 4b for the specific actor allocation where input, conv 1, 3, & 4 are mapped on GPP 1, conv 2 & 3 on NPU 1, and pool, conv 6, and concat on NPU 2. To use self-timed strategy [16], PRISM enforces the actor execution order for each resource. For instance, GPP 1 executes input, conv 3, conv 1, and conv 4 in this same order every time the inception block (i.e., its SDFG) is executed.

Figure 4c shows the IPC graph that PRISM constructs for the inception block. Observe that the graph consists of computing actors, which represent operations of the inception block and additionally, a few extra actors which are shown with letters `S` and `R`. They represent send and receive communication actors, respectively. PRISM categorizes each channel of an SDFG as intra- or inter-processor channel, where a channel is called *intra-processor channel* if its source and destination actors are mapped to the same resource, and *inter-processor channel* otherwise. In Figure 4b, there are 6 intra-processor and 5 inter-processor channels. For each inter-processor channel, PRISM adds two communications actors – a send actor at its source and a receive actor at its destination. In Figure 4b, there are 5 send actors and 5 receive actors.

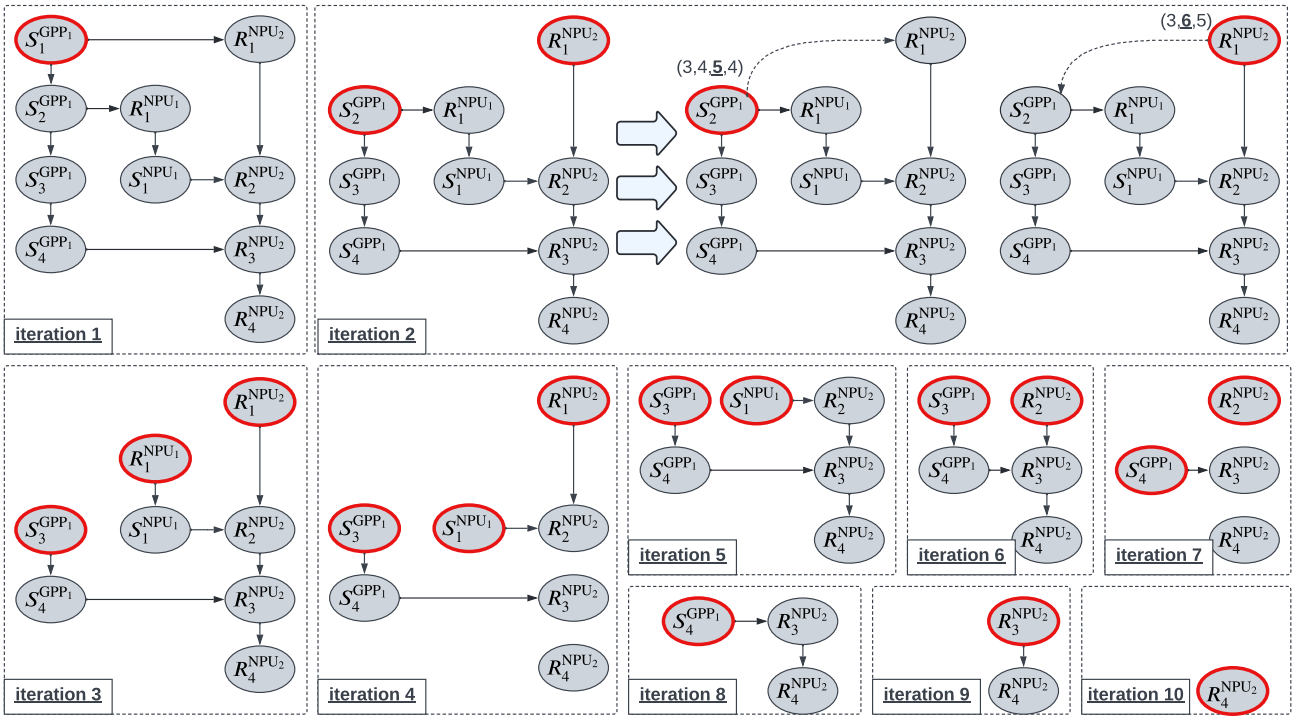


Fig. 5. Iterations of the transaction partial order algorithm to generate the transaction order of communication actors from the TPO graph of Figure 4d. Operations in each iteration is demonstrated using iteration 2 as an examples. First, ready actors are identified. Next, MCMs are computed for each ready actor by creating channels to other ready actors. The ready actor with the smallest MCM is selected as the candidate and deleted from the graph.

B. Embedding A Transaction Order in an IPC Graph

For an NSoC, inter-processor communications take place via the platform's shared resources. So, a transaction order must be created for the communication actors. PRISM uses the transaction partial order (TPO) graph, which it generates from an IPC graph by eliminating its computing actors [53]. Figure 4d is the TPO graph of the IPC graph of Figure 4c.

Figure 5 illustrates the iterations of a transaction partial order algorithm to generate the transaction order of communication actors . Algorithm 1 shows its pseudo-code. For each iteration, the algorithm performs the following steps. First, it prepares a list of ready actors (i.e., those that do not have any incoming channel). In Figure 5, ready actors are S_1^{GPP1} for iteration 1, S_2^{GPP1} & R_1^{NPU2} for iteration 2, S_3^{GPP1}, R_1^{NPU1}, & R_1^{NPU2} for iteration 3, and so on. Next, for each ready actor, it creates new outgoing channels to other ready actors. We illustrate these channels using dashed lines for iteration 2. Next, it evaluates the MCM for these ready actors using Equation 2. For iteration 2, we underline the MCMs out of other cycle means. It calls a ready actor with the smallest MCM as candidate and deletes it from the TPO graph along with all its outgoing channels. In iteration 2, actor S_2^{GPP1} is the candidate as it has the smallest MCM. The current iteration completes once the candidate is deleted from the TPO graph. Subsequently, the algorithm repeats these steps for the next iteration, eventually terminating when all actors are deleted from the TPO graph. Finally, the algorithm generates a transaction order by connecting all candidates in the sequence in which they are deleted. Figure 4e shows the IPC graph with the transaction order of its communication actors shown using red dashed lines. PRISM uses this IPC graph with embedded transaction order to compute the throughput using Equation 3.

C. Scheduling using a Hill Climbing Heuristic

Scheduling SDCNN operations on an NSoC consists of determining the mapping, ordering, and timing [16]. Since PRISM uses self-timed execution, it is not necessary to obtain the precise timing information. Here, we discuss how PRISM explores the design space of mapping and ordering.

We introduce the following notations.

$$
\begin{aligned}
G_{\text{IPC}}(A, C) &= \text{An IPC graph with embedded transaction order} \\
A &= \text{Set of computing and communication actors and } |A| = N_A \\
C &= \text{Set of channels between actors} \\
G_{\text{NSoC}}(R, E) &= \text{An NSoC platform graph} \\
R &= \text{Set of resources of the NSoC and } |R| = N_R \\
E &= \text{Set of edges/links between the resources} \\
\mathcal{M} &= \{x_{i,j}\} = \text{ Mapping of } G_{\text{IPC}} \text{ on } G_{\text{NSoC}} \\
x_{i,j} &= \begin{cases} 1 & \text{if actor } a_i \in A \text{ is mapped to resource } r_j \in R \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

The problem of finding a mapping of actors to resources draws parallel to mapping tasks on parallel computers [54]. This problem has been shown to be NP-complete for non-trivial optimization objectives [55]. Therefore, heuristic solu-

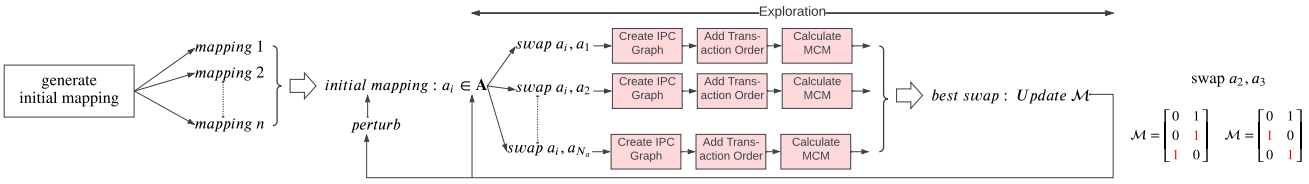


Fig. 6. Overview of the Hill Climbing heuristic to generate mapping of operations to computing units of an NSoC platform. The swapping procedure is demonstrated to the right of this figure.

tions such as Hill Climbing, Simulated Annealing, and Genetic Algorithms are effective in finding a solution [56]. We use a Hill Climbing heuristic because of its simplicity [57].

Algorithm 1: Transaction partial order algorithm.

Input: TPO graph $G_{TPO}(\mathbf{V}, \mathbf{E})$
Output: Transaction order $\mathbf{O}$

1 $\mathbf{O} = \emptyset$ /* Initialize the transaction order. */
2 **while** $\mathbf{V} \neq \emptyset$ **do** /* Start of an iteration */
3 $\mathbf{R} = \{v \in \mathbf{V} | inC(v) = \emptyset\}$ /* Set of ready actors. */
4 **for** $r \in \mathbf{R}$ **do**
5 Create $Ch(r, u)$ $\forall u \in \mathbf{R}$ and $u \neq r$ /* Create a new channel from each ready actor to other ready actors. */
6 L_a.append(r) and L_m.append(ComputeMCM(G_{TPO})) /* Store the ready actor and the corresponding MCM in local variables L_a and L_m, respectively. */
7 $x = \text{argmin}(L_m)$ /* Find the index to the minimum MCM. */
8 candidate $= L_a[x]$ /* Select an actor with the minimum MCM as candidate. */
9 $\mathbf{O}$.append(candidate) /* Insert the candidate in the transaction order $\mathbf{O}$. */
10 $\mathbf{V} \setminus \{\text{candidate}\}$ and $\mathbf{E} \setminus \{OutC(\text{candidate})\}$ /* Delete the candidate and all its outgoing channels. */
11 **return** $\mathbf{O}$

Figure 6 illustrates the steps for the proposed Hill Climbing heuristic. It starts with the *generate initial mapping* block, which generates a set of initial mappings. Next, it selects one of these starting mappings and proceeds to the exploration stage. Here, it performs a trial swap for each actor $a_i \in \mathbf{A}$ with another actor that is mapped to a different resource. A trial swap operation involves changing the mapping matrix temporarily, as we illustrate to the right of the figure. For each of these trial swaps, our heuristic evaluates the optimization objective – the MCM. For this, we follow the procedure outlined in Sections III-A & III-B, which involves (1) creating an IPC graph from the mapping obtained after performing a swap, (2) embedding it with a transaction order for communication actors using Alg. 1, and (3) computing the MCM using Eq. 2.

Next, the heuristic selects a swap with the minimum MCM because reducing the MCM increases its throughput. In case of a tie, it selects a trial swap randomly. Once a swap is finalized, it makes the swap permanent by updating the mapping matrix. It then iterates through these steps for the next actor. A *pass* in this heuristic involves completing trial swaps for every pair of actors mapped to different resources. If the objective function improves during a pass, we initiate another pass of our heuristic. Otherwise, we call the heuristic to be stuck at a local minimum. To come out of this local minimum, we perturb the current mapping and restart the exploration, where perturbing involves performing a fixed number of random swaps.

During the Hill Climbing heuristic, we record all mappings, i.e., the initial mapping and the mappings obtained after performing each swap operation. For all these mappings, we estimate the energy consumption of their corresponding schedule. A set of Pareto-optimal mappings are only retained for use-case mapping, which is described next.

IV. Mapping Multi-Application Use-Cases on NSoC

Figure 7 illustrates how PRISM creates schedules for more than one application at the same time. Imagine a user initiating application B at time t_0, when the platform is already executing application A. Therefore, operations of B must be scheduled on the same resources that are currently executing A. This is a huge undertaking and it involves guaranteeing performance for both these applications to deliver a quality-of-service. One solution could be to analyze every combination of applications at design time and store the corresponding schedule – with n applications, there are n^2 use-cases to analyze. However, storing all use-cases from design-time can increase the storage overhead of an embedded platform and provides limited flexibility for run-time adaptation.

To address this, PRISM initiates a local and global exploration. The objective of PRISM's local exploration is to create a quick schedule for B keeping the current schedule of A unchanged. This is to admit B in the shortest possible time while providing a certain performance guarantee and ensuring that the performance of A does not degrade excessively. In the figure, PRISM admits B to the platform at time t_1.

The objective of PRISM's global exploration is to find optimized schedules for both A and B, which provide the best system performance while sharing the resources of an NSoC. In the figure, PRISM implements the optimized schedule for A and B from time t_2 onwards.

A. Local Exploration of PRISM

Algorithm 2 shows the pseudo-code of PRISM's local exploration. Here, $SchA$ is the schedule of application A, i.e., the schedule currently running on the NSoC. First, PRISM retrieves all Pareto-optimal schedules of the newly enabled application B and store it in the set S_B (line 2). Next, for each schedule $Sch_B \in S_B$, it computes the resource contention related overhead using a probabilistic framework (line 4). Finally, it selects a schedule for B that minimizes this overhead. The key idea is to reduce the expected wait time before actors from the two applications can be scheduled on

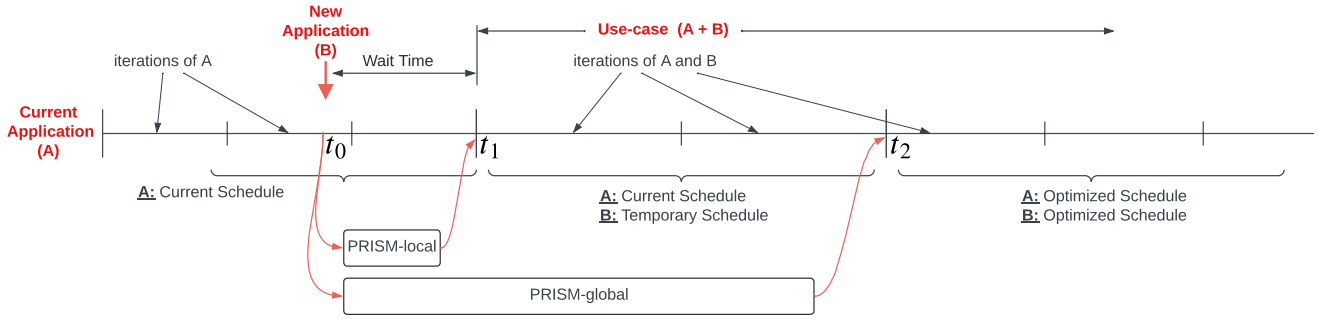


Fig. 7. Scheduling multi-application use-cases on an NSoC via PRISM's local and global explorations.

contending resources. PRISM uses this new schedule for B after it finishes exploring all alternatives. In the mean time, the platform continues to execute the current schedule of A. In this way, PRISM ensures a non-preemptive system where the actors that are already executing on the platform are not preempted to provide resources for the new application.

Algorithm 2: Local exploration of PRISM.

Input: Schedule database sDB, schedule $SchA$ of A, and application B
Output: Schedule $SchB$ of B.

1 $\rho_A = MCM(SchA)$ /* MCM of A. */
2 $S_B = sDB(B)$ /* Pareto-optimal schedules of B. */
3 **for** $SchB \in S_B$ **do** /* For each schedule in S_B */
4 $SchA, SchB = \texttt{Contention}(SchA, SchB)$ /* Compute resource sharing related contention. */
5 $\hat{\rho}_A = MCM(SchA)$ and $\rho_B = MCM(SchB)$ /* Compute MCM of A and B considering resource contention. */
6 $\Delta\rho_A = 100 * \frac{\hat{\rho}_A - \rho_A}{\rho_A}$ /* Compute the performance degradation of A due to resource contention. */
7 **if** $\Delta\rho_A < \Delta\rho_{max}$ *and* $\rho_B < \rho_{constraint}$ **then**
8 **return** $SchB$

To formulate the resource contention related overhead, we consider that each actor must be in one of the three states at any given time – (1) not-ready state, when it waits for all of its tokens to be available, (2) wait state, when it waits for its resource to be available, and (3) execution state, when it is currently executing a machine learning operation. Now assume that actor b ∈ B is mapped on the same resource as actor a ∈ A. We are to model the expected execution time of a and b when contending for the resource. When b is enabled, it can find a in one of the three states. Let the random variable $S(t)$ denote the state of actor a and Y denote the wait time of b.

We define the following probabilities.

$$\begin{aligned} P(S(t)=S_w) &= \text{probability of a in wait state} \\ &= t_{wait}\cdot\rho_{max} \\ P(S(t)=S_e) &= \text{probability of a in execution state} \\ &= t_a\cdot\rho_{max} \\ P(S(t)=S_n) &= \text{probability of a in not-ready state} \\ &= 1-P(S(t)=S_w)-P(S(t)=S_e) \end{aligned} \tag{4}$$

where t_{wait} is the average wait time and ρ_{max} is the MCM of the IPC of A. The above steady-state probabilities are derived assuming stationarity of an actor in each state.

The expected wait time is obtained as

$$E(Y) = \int_{-\infty}^{\infty} y\ P(y)\ dy, \tag{5}$$

where $P(y)$ is the probability density function of Y. We make the following consideration in solving Equation 5. First, if b arrives when a is in not-ready state, then b can be scheduled immediately. Second, if b arrives when a is in wait state (due to resource contention from other actors), then b must wait for the entire duration of the a's execution time. In this case, b is pushed to the back of the ready queue for the resource. Here, we uses first-come-first-serve (FCFS) scheduling for each resource. Finally, if b arrives when a has started executing, then b must wait for the remaining time until a finishes execution. Therefore, Equation 5 can be rewritten as

$$\begin{aligned} E(Y) = \quad & y_{|_{S(t)=S_n}}\cdot P(S(t)=S_n) \\ & + y_{|_{S(t)=S_w}}\cdot P(S(t)=S_w) \\ & + y_{|_{S(t)=S_e}}\cdot P(S(t)=S_e) \end{aligned} \tag{6}$$

Assuming the remaining execution time of a is uniformly distributed within the time duration of the execution time of a, the above equation reduces to

$$E(Y) = 0\cdot P(S(t)=S_n) + t_a\cdot t_{wait}\cdot\rho_{max} + \frac{t_a}{2}\cdot t_a\cdot\rho_{max} \tag{7}$$

Since $E(Y)$ is the average wait time t_{wait}, Equation 7 can be solved to obtain

$$t_{wait} = \frac{t_a^2\cdot\rho_{max}/2}{1-t_a\cdot\rho_{max}} \tag{8}$$

Finally, the modified execution time of b is

$$t_b = t_b + t_{wait} \tag{9}$$

This formulation needs to be extended for every actor of the two applications A and B. Once completed, Algorithm 2 computes the modified MCM for the two applications (line 5). It estimates the performance degradation of A (line 6). If they both are acceptable, i.e., they are within the user defined limits, Algorithm 2 terminates, returning the new schedule of B (lines 7-8). Here, $\Delta\rho_{max}$ is the maximum acceptable throughput degradation of A and $\rho_{constraint}$ is the throughput constraint of B. These are user-defined parameters.

B. Global Exploration of PRISM

For global exploration, PRISM first merges the IPC graphs of the two application. Let $G_A(A_1, C_1)$ be the IPC graph of application A with the set A_1 of actors and set C_1 of channels.

Let $G_B(A_2, C_2)$ be the IPC graph of application B with the set A_2 of actors and set C_2 of channels. The merged graph is

$$G_{AB}(A, C) \mid A = A_1 \cup A_2 \text{ and } C = C_1 \cup C_2 \tag{10}$$

Next, the merged graph is analyzed using the formulation that we presented in Section II-B. New schedules for A and B are implemented after completing their ongoing iterations.

V. Evaluation Methodology

A. Simulation Framework

We implement PRISM inside NeuroXplorer [58], an architectural simulator of neuromorphic system-on-chip platforms. We perform all simulations on a Lambda workstation, which has AMD Threadripper 3960X with 24 cores, 128 MB cache, 128 GB RAM, and 2 RTX3090 GPUs. Table III shows our simulation parameters. The code is available online at https://github.com/drexel-DISCO/PRISM.

TABLE III
Major simulation parameters.

Number of GPPs	2
Number of NPUs	128
NSoC Platform	μBrain [59], SPECK [29],& GrAI [28]
Design Parameters	Energy [60], Throughput [19], Reliability [61], [62], and Technology [63]
NPU supported operations – μBrain	Activation, Add, Conv2D, Concatenate, Dense, InputLayer, Normalization
NPU supported operations – SPECK	Activation, Add, AveragePooling2D, Concatenate, Conv2D, Dense, Dropout, Flatten, GlobalAveragePooling2D, GlobalMaxPooling2D, InputLayer, MaxPooling2D, Normalization, ReLU
NPU supported operations – GrAI	BatchNormalization, ZeroPadding2D, DepthwiseConv2D, Reshape, Rescaling, Multiply, and all supported operations of SPECK

TABLE IV
Evaluated models and ImageNet Top-1 accuracy.

Models	Accuracy	Models	Accuracy	Models	Accuracy	Models	Accuracy
LeNet	49.0%	ResNet50	77.1%	DenseNet121	74.4%	MobeleNet	74.2%
AlexNet	60.5%	ResNet101	78.0%	DenseNet169	76.1%	ShuffleNet	70.8%
ZFNet	64.0%	InceptionV3	78.1%	SqueezeNet	57.5%	NASNetMobile	77.9%
VGG16	73.4%	ResNext	83.5%	Xception	78.6%	NoisyStudent	82.0%

B. Evaluated Applications

We evaluate PRISM using 20 models – 16 models trained on the ImageNet dataset and four additional state-of-the-art models. The ImageNet models are summarized in Table IV. Although improving accuracy is not the focus here, we provide accuracy numbers for completeness. The four additional models are (1) PilotNet [64] for self-driving cars, (2) the transformer-based BERT [42] for natural language processing, (3) U-Net [41] for medical image segmentation, and (4) ConvLSTM [43] for time-series data processing.

We use the following use-cases for evaluating the local and global explorations of PRISM for scheduling use-cases.

1) **Usecase-1:** A combination of BERT and MobileNet
2) **Usecase-2:** A combination of ResNet50 and U-Net
3) **Usecase-3:** A combination of SqueezeNet and AlexNet
4) **Usecase-4:** A combination of VGG16 and ShuffleNet
5) **Usecase-5:** A combination of Xception and PilotNet

C. Models Supported on Evaluated NSoCs

As we discuss in Section I, an NPU may not support all operations of a machine learning model. To give further insight, Figure 8 shows the fraction of unsupported operations for three state-of-the-art NSoCs – μBrain [32], SPECK [29], and GrAI [28]. We report results for 5 ImageNet models and average across 16 such models of Table IV. We also report results for the four additional models and the average across all 20 evaluated models. We make the following observations.

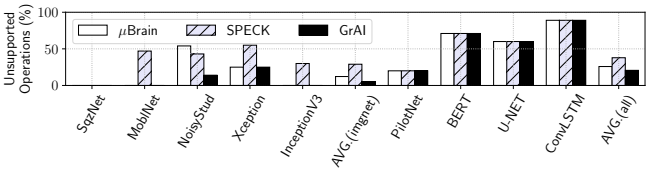


Fig. 8. Unsupported operations on three state-of-the-art NSoC platforms.

Some models such as SqueezeNet (abbreviated as SqzNet) is supported on all three platforms (unsupported operations is close to 0%). This is because this model consists of standard CNN operations such as convolution, pooling, and dense, which are all supported on these three platforms. On the other hand, most operations of BERT, U-Net and ConvLSTM are not supported on any of these platforms due to the use of specialized operations. For ImageNet models, μBrain, SPECK, and GrAI do not support on average 12%, 30%, and 5% of the total operations, respectively. Considering all models, these fractions are 26%, 38%, and 21%, respectively. PRISM uses the GPPs to schedule all unsupported operations.

D. Evaluated Schedulers

We evaluate the following schedulers.

1) **Baseline [8]:** This is an NPU-only policy. It uses a GPP to schedule only the operations not supported on the NPU. It exploits only batch parallelism.
2) **SentryOS [32]:** This is the baseline that exploits both pipeline and batch parallelism.
3) **PRISM**: The is the proposed scheduler which exploits platform heterogeneity to schedule operations. Here, an operation can be scheduled on a GPP even if it is supported on an NPU. Using a Hill Climbing heuristic, PRISM creates opportunities for all three forms of parallelism – batch, pipeline, and operation.

VI. Results

A. Throughput Performance

Figure 9 reports throughput of the evaluated schedulers We make the following observations.

Although both Baseline and SentryOS exploit batch parallelism, SentryOS additionally exploits pipeline parallelism within each batch. So, the throughput of SentryOS is higher (on average, 1.9x higher for ImageNet models and 1.6x higher

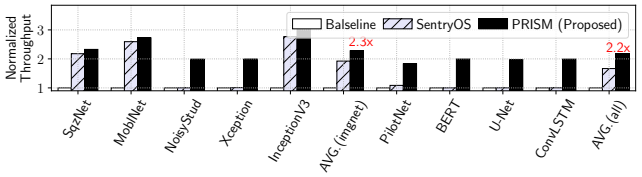


Fig. 9. Throughput normalized to Baseline.

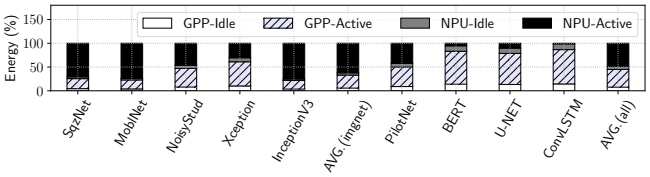


Fig. 11. Total energy distributed into GPP and NPU energy (idle and active).

for all 20 models). Between SentryOS and the proposed PRISM, SentryOS uses a GPP only for those operations that are not supported on an NPU. Therefore, the degree of pipeline and operation parallelism that can be exploited is limited. On the other hand, PRISM has a higher degree of freedom in mapping operations to GPPs and NPUs. So, PRISM can better exploit these forms of parallelism using the Hill Climbing heuristic. For ImageNet models, PRISM has 2.3x and 1.2x higher throughput than Baseline and SentryOS, respectively. Considering all 20 models, the throughput of PRISM is 2.2x and 1.3x higher, respectively.

B. Performance Per Watt

Figure 10 reports the throughput per watt of the evaluated schedulers. We make the following observations.

Between Baseline and SentryOS, SentryOS has 18% higher throughput per watt on average than Baseline. Although SentryOS has 1.9x higher speedup than Baseline (see Section VI-A), it also has a higher energy consumption due to an increased utilization of resources in exploiting pipeline parallelism. For the 4 non-ImageNet models where the throughput improvement is relatively small, the throughput per watt is lower than Baseline. Considering all 20 models, SentryOS has only 3% higher throughput per watt compared to Baseline.

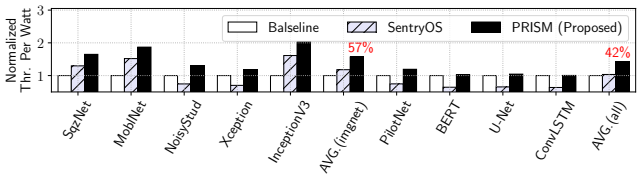


Fig. 10. Throughput per Watt normalized to Baseline (higher is better).

Between SentryOS and PRISM, PRISM has a higher throughput (average 1.3x higher) and lower energy (average 5% lower). So, the throughput per watt is 33% higher than SentryOS for ImageNet models and 38% higher for all 20 models. Compared to Baseline, PRISM has 57% and 42% higher throughput per watt, respectively.

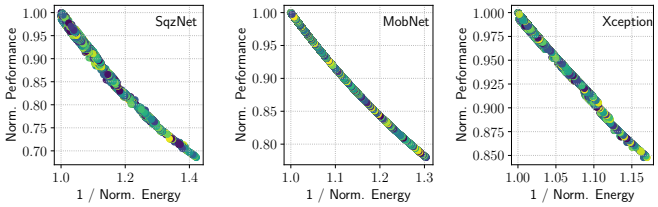
C. Energy Breakdown

Figure 11 reports the total energy of each model, distributed into GPP and NPU energy. The figure reports both active and idle energy, where active energy is the energy consumed when a resource is performing an operation, and idle energy is the energy consumed while it is not performing any operation. We make the following observations.

When idle, resources still draw a portion of the peak power. Therefore, the idle GPP and NPU energy constitute on average 8% and 6% of the total energy, respectively. Even though all operations of SqueezeNet are supported on an NPU (see Figure 8), GPP active energy constitutes about 21% of the total energy. This is because PRISM schedules its operations using both GPPs and NPUs to improve the throughput. For BERT, U-Net, and ConvLSTM, PRISM uses the GPP to schedule most operations. So, the GPP active energy is the dominant component of the total energy (on average 69%).

D. Pareto Exploration during Hill Climbing Optimization

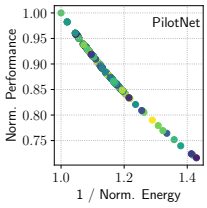
Figure 12 shows the energy-throughput Pareto points retained during the proposed Hill Climbing based mapping exploration for six models. PRISM selects the highest throughput point for every model for a given energy constraint.



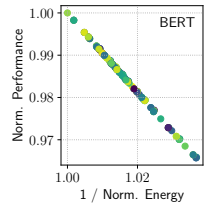
(a) SqueezeNet.

(b) MobileNetV3.

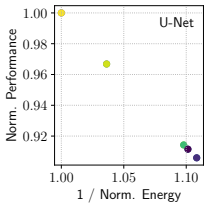
(c) Xception.



(d) PilotNet.



(e) BERT.



(f) U-Net.

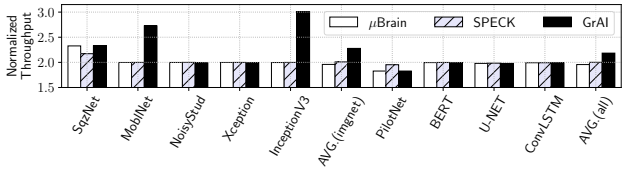
Fig. 12. Pareto points retained during the Hill Climbing exploration.

E. Throughput Scalability

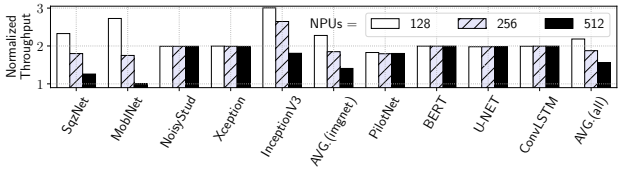
Figure 13a reports the throughput of PRISM normalized to Baseline for the three evaluated NSoCs. For BERT, U-Net, and ConvLSTM, the throughput is comparable. This is because most operations of these three models are not supported on an NPU. Therefore, PRISM uses GPPs for these operations, which results in similar performance across the three NSoC platforms. For other applications, throughput is higher for the platform where the heterogeneity can be exploited better.

Figure 13b reports the throughput of PRISM normalized to Baseline as we increase the number of NPUs from 128 (base config) to 512. We observe that the relative throughput reduces due to this change. This is because with more NPUs, batch parallelism is the dominant factor that contributes to performance. Since both PRISM and Baseline exploits this parallelism, the relative performance between these two schedulers reduces.

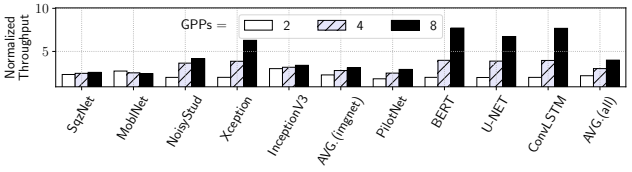
PRISM is still better because of the pipeline and operation parallelism that it additionally exploits from the hardware.



(a) Throughput normalized to Baseline across evaluated platforms.



(b) Throughput normalized to Baseline with increasing number of NPUs.



(c) Throughput normalized to Baseline with increasing number of GPPs.

Fig. 13. Speedup of PRISM for (a) different evaluated platforms, (b) increasing number of NPUs, and (c) increasing number of GPPs.

Figure 13c reports the throughput of PRISM normalized to Baseline as we increase the number of GPPs from 2 (base config) to 8. We observe that the relative throughput improves due to this change. This is because with more GPPs, PRISM can better exploit operation parallelism in an NSoC, which improves the throughput.

F. Use-case Performance

Since no existing schedulers can map use-cases, we created a baseline where the second application of a use-case is mapped by randomly allocating its actors. Figure 14 reports the throughput using PRISM's local and global explorations normalized to this baseline for five use-cases.

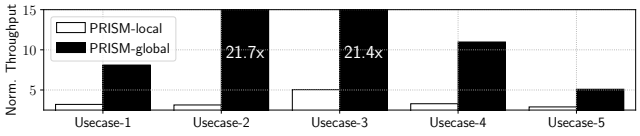


Fig. 14. Throughput normalized to baseline. The first application of a use-case is the current application that is executing on an NSoC. The second application is the one that is invoked by a user at run-time.

We observe that PRISM's local exploration results in an average 3.5x higher throughput than baseline. This is because during local exploration, PRISM analyzes all pre-computed schedules of the new application using a probabilistic formulation and selects one that minimizes the resource contention related slowdown for both applications (ongoing and new). On the other hand, baseline does not incorporate performance when mapping the new application of a use-case. PRISM's global exploration results in 3.8x and 13.4x higher throughput than the local exploration and baseline, respectively. The improvement is because during global exploration, PRISM performs the Hill Climbing heuristic on the merged graph of the two applications to find optimized schedules for both. This results in generating schedules that give higher throughput.

G. Wait Times

Table V reports wait times for the second application of each use-case when an NSoC is currently executing the first application. Here, the wait time is measured as the time from when a user invokes the second application to the time when the application starts executing on the hardware. This wait time depends on when a schedule is created for the second application. We observe that the local exploration is faster than global exploration (on average, 12x lower wait time). This improves the user experience.

TABLE V
WAIT TIMES OF PRISM'S LOCAL AND GLOBAL EXPLORATIONS.

PRISM	Usecase-1	Usecase-2	Usecase-3	Usecase-4	Usecase-5
local	12s	21s	6s	9s	14s
global	223s	116s	28s	138s	226s

VII. CONCLUSION

We propose PRISM, a real-time performance-oriented scheduler for neuromorphic system-on-chips (NSoCs). PRISM operates in four steps. First, it creates an interprocessor communication (IPC) graph of a machine learning model by considering the mapping of its operations and a self-timed schedule. Next, it embeds a transaction order for the inter-processor communications into the IPC graph. Next, it schedules the graph by overlapping communication with the computation. Finally, it uses a Hill Climbing heuristic to explore the design space of mapping and scheduling IPC graphs to an NSoC, exploiting the platform heterogeneity in improving opportunities for batch, pipeline, and operation parallelism. For multi-application uses-cases, PRISM uses a probabilistic framework to model resource contention related slowdown. It creates a schedule to reduce the expected wait time before concurrent operations are scheduled on contending resources. Our extensive evaluations with 20 machine learning workloads and five use-cases show that PRISM significantly improves the performance per watt for both individual applications and multi-application use-cases when compared to state-of-the-art schedulers.

ACKNOWLEDGMENTS

This work is supported by U.S. Department of Energy under Award Number DE-SC0022014 and the National Science Foundation under Awards CCF-1937419 & CCF-1942697.

REFERENCES

[1] B. Han, A. Sengupta, and K. Roy, "On the energy benefits of spiking deep neural networks: A case study," in *International Joint Conference on Neural Networks (IJCNN)*, 2016.

[2] A. Sengupta, Y. Ye, R. Wang, C. Liu, and K. Roy, "Going deeper in spiking neural networks: VGG and residual architectures," *Frontiers in Neuroscience*, vol. 13, p. 95, 2019.

[3] Y. Cao, Y. Chen, and D. Khosla, "Spiking deep convolutional neural networks for energy-efficient object recognition," *International Journal of Computer Vision*, vol. 113, pp. 54–66, 2015.

[4] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, vol. 10, pp. 1659–1671, 1997.

[5] G. Datta and P. A. Beerel, "Can deep neural networks be converted to ultra low-latency spiking neural networks?" in *Design, Automation, and Test in Europe (DATE) Conference and Exhibition*, 2022.

[6] F. Xing, Y. Yuan, H. Huo, and T. Fang, "Homeostasis-based cnn-to-snn conversion of inception and residual architectures," in *International Conference on Neural Information Processing (ICONIP)*, 2019.

[7] W. Fang, Z. Yu, Y. Chen, T. Huang, T. Masquelier, and Y. Tian, "Deep residual learning in spiking neural networks," *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[8] S. Song, H. Chong, A. Balaji, A. Das, J. Shackleford, and N. Kandasamy, "DFSynthesizer: Dataflow-based synthesis of spiking neural networks to neuromorphic hardware," *ACM Transactions on Embedded Computing Systems*, vol. 2, pp. 1–4, 2021.

[9] A. Paul, M. A. S. Tajin, A. Das, W. Mongan, and K. Dandekar, "Energy-efficient respiratory anomaly detection in premature newborn infants," *Electronics*, pp. 689–694, 2022.

[10] A. Balaji, F. Corradi, A. Das, S. Pande, S. Schaafsma, and F. Catthoor, "Power-accuracy trade-offs for heartbeat classification on neural networks hardware," *Journal of Low Power Electronics*, vol. 14, pp. 508–519, 2018.

[11] C. Mead, "Neuromorphic electronic systems," *Proceedings of the IEEE*, vol. 78, pp. 1629–1636, 1990.

[12] S. Li, E. Hanson, X. Qian, H. H. Li, and Y. Chen, "ESCALATE: Boosting the efficiency of sparse CNN accelerator with kernel decomposition," in *International Symposium on Microarchitecture (MICRO)*, 2021.

[13] Z. Shao, X. Chen, L. Du, L. Chen, Y. Du, W. Zhuang, H. Wei, C. Xie, and Z. Wang, "Memory-efficient CNN accelerator based on interlayer feature map compression," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, pp. 668–681, 2021.

[14] V. Mrazek, Z. Vasícek, L. Sekanina, M. A. Hanif, and M. Shafique, "ALWANN: Automatic layer-wise approximation of deep neural network accelerators without retraining," in *International Conference on Computer-Aided Design (ICCAD)*, 2019.

[15] F. Catthoor, S. Mitra, A. Das, and S. Schaafsma, "Very large-scale neuromorphic systems for biological signal processing," in *CMOS Circuits for Biological Sensing and Processing*, 2018.

[16] S. Sriram and S. Bhattacharyya, *Embedded Multiprocessors; Scheduling and Synchronization*, 2000.

[17] A. Das and A. Kumar, "Dataflow-based mapping of spiking neural networks on neuromorphic hardware," in *Great Lakes Symposium on VLSI (GLSVLSI)*, 2018.

[18] A. Balaji and A. Das, "A framework for the analysis of throughput-constraints of SNNs on neuromorphic hardware," in *IEEE Annual Symposium on VLSI (ISVLSI)*, 2019.

[19] S. Song, A. Balaji, A. Das, N. Kandasamy, and J. Shackleford, "Compiling spiking neural networks to neuromorphic hardware," in *International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2020.

[20] S. Song, L. V. Mirtinti, A. Das, and N. Kandasamy, "A design flow for mapping spiking neural networks to many-core neuromorphic hardware," in *International Conference on Computer-Aided Design (ICCAD)*, 2021.

[21] Qualcomm. (2022) Snapdragon Neural Processing Engine (SNPE).

[22] Nvidia. (2022) Jetson AGX Xavier.

[23] A. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, T. Hartley, and L. Van Gool, "AI benchmark: Running deep neural networks on android smartphones," in *ECCV Workshops*, 2018.

[24] M. Davies, N. Srinivasa, T. H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, pp. 82–99, 2018.

[25] C.-K. Lin, A. Wild, G. N. Chinya, T.-H. Lin, M. Davies, and H. Wang, "Mapping spiking neural networks onto a manycore neuromorphic architecture," in *Programming Language Design and Implementation (PLDI)*, 2018.

[26] BrainChip. (2022) Akida Neuromorphic System- on-Chip.

[27] BrainChip. (2022) MetaTF development environment.

[28] "GrAI Chip and GrAIFlow Software," https://www.graimatterlabs.ai/product, accessed: 2022-05-10.

[29] Q. Liu, O. Richter, C. Nielsen, S. Sheik, G. Indiveri, and N. Qiao, "Live demonstration: face recognition on an ultra-low power event-driven convolutional neural network ASIC," in *CVPR Workshops*, 2019.

[30] S. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The SpiNNaker project," *Proceedings of the IEEE*, vol. 102, pp. 652–665, 2014.

[31] F. Galluppi, S. Davies, A. Rast, T. Sharp, L. A. Plana, and S. Furber, "A hierachical configuration system for a massively parallel neural hardware platform," in *International Conference on Computing Frontiers (CF)*, 2012.

[32] M. L. Varshika, A. Balaji, F. Corradi, A. Das, J. Stuijt, and F. Catthoor, "Design of many-core big little μBrains for energy-efficient embedded neuromorphic computing," in *Design, Automation, and Test in Europe (DATE) Conference and Exhibition*, 2022.

[33] L. Shi, J. Pei, N. Deng, D. Wang, L. Deng, Y. Wang, Y. Zhang, F. Chen, M. Zhao, S. Song *et al.*, "Development of a neuromorphic computing system," in *International Electron Devices Meeting (IEDM)*, 2015.

[34] Y. Ji, Y. Zhang, S. Li, P. Chi, C. Jiang, P. Qu, Y. Xie, and W. Chen, "NEUTRAMS: Neural network transformation and co-design under neuromorphic hardware constraints," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[35] S. Moradi, N. Qiao, F. Stefanini, and G. Indiveri, "A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs)," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 12, pp. 106–122, 2017.

[36] A. Das, Y. Wu, K. Huynh, F. Dell'Anna, F. Catthoor, and S. Schaafsma, "Mapping of local and global synapses on spiking neuromorphic hardware," in *Design, Automation, and Test in Europe (DATE) Conference and Exhibition*, 2018.

[37] A. Balaji, A. Das, Y. Wu, K. Huynh, F. G. Dell'Anna, G. Indiveri, J. L. Krichmar, N. D. Dutt, S. Schaafsma, and F. Catthoor, "Mapping spiking neural networks to neuromorphic hardware," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, pp. 76–86, 2019.

[38] A. Balaji, S. Song, A. Das, J. Krichmar, N. Dutt, J. Shackleford, N. Kandasamy, and F. Catthoor, "Enabling resource-aware mapping of spiking neural networks via spatial decomposition," *Embedded Systems Letters*, vol. 13, pp. 142–145, 2020.

[39] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size," *arXiv*, 2016.

[40] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Computer Vision and Pattern Recognition Conference (CVPR)*, 2016.

[41] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2015.

[42] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[43] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W.-K. Wong, and W.-c. Woo, "Convolutional LSTM network: A machine learning approach for precipitation nowcasting," *Conference on Neural Information Processing Systems (NeurIPS)*, 2015.

[44] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, 1987.

[45] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor system-on-chip (MPSoC) technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, pp. 1701–1713, 2008.

[46] K. Rosvall and I. Sander, "A constraint-based design space exploration framework for real-time applications on MPSoCs," in *Design, Automation, and Test in Europe (DATE) Conference and Exhibition*. IEEE, 2014.

[47] A. H. Ghamarian, M. C. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. Moonen, and M. J. Bekooij, "Throughput analysis of synchronous data flow graphs," in *International Conference on Application of Concurrency to System Design (ACSD)*, 2006.

[48] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *Design Automation Conference (DAC)*, 2006.

[49] A. Das, A. Kumar, and B. Veeravalli, "Energy-aware task mapping and scheduling for reliable embedded computing systems," *ACM Transactions on Embedded Computing Systems*, vol. 13, pp. 1–27, 2014.

[50] A. K. Singh, A. Das, and A. Kumar, "Energy optimization by exploiting execution slacks in streaming applications on multiprocessor systems," 2013.

[51] A. Das, A. Kumar, and B. Veeravalli, "Temperature aware energy-reliability trade-offs for mapping of throughput-constrained applications on multimedia MPSoCs," in *Design, Automation, and Test in Europe (DATE) Conference and Exhibition*, 2014.

[52] A. Das, A. Kumar, and B. Veeravalli, "Reliability and energy-aware mapping and scheduling of multimedia applications on multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, pp. 869–884, 2015.

[53] N. Bambha, V. Kianzad, M. Khandelia, and S. S. Bhattacharyya, "Intermediate representations for design automation of multiprocessor DSP systems," *Springer Design Automation for Embedded Systems*, vol. 7, pp. 307–323, 2002.

[54] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: survey of current and emerging trends," in *Design Automation Conference (DAC)*, 2013.

[55] A. L. Rosenberg, "Data encodings and their costs," *Acta Informatica*, 1978.

[56] E.-G. Talbi and T. Muntean, "Hill-climbing, simulated annealing and genetic algorithms: a comparative study and application to the mapping problem," in *Hawaii International Conference on System Sciences*, 1993.

[57] D. L. Smitley and I. Lee, "Comparative analysis of hill climbing mapping algorithms," *Technical Reports (CIS)*, 1988.

[58] A. Balaji, S. Song, T. Titirsha, A. Das, J. Krichmar, N. Dutt, J. Shackleford, N. Kandasamy, and F. Catthoor, "NeuroXplorer 1.0: An extensible framework for architectural exploration with spiking neural networks," in *International Conference on Neuromorphic Systems (ICONS)*, 2021.

[59] J. Chen, W.-B. Jone, J.-S. Wang, H.-I. Lu, and T.-F. Chen, "Segmented bus design for low-power systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, pp. 25–29, 1999.

[60] T. Titirsha, S. Song, A. Balaji, and A. Das, "On the role of system software in energy management of neuromorphic computing," in *International Conference on Computing Frontiers (CF)*, 2021.

[61] S. Song and A. Das, "A case for lifetime reliability-aware neuromorphic computing," in *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2020.

[62] T. Titirsha, S. Song, A. Das, J. Krichmar, N. Dutt, N. Kandasamy, and F. Catthoor, "Endurance-aware mapping of spiking neural networks to neuromorphic hardware," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, pp. 288–301, 2021.

[63] A. Paul, S. Song, T. Titirsha, and A. Das, "On the mitigation of read disturbances in neuromorphic inference hardware," *IEEE Design & Test*, 2022.

[64] M. Bojarski, C. Chen, J. Daw, A. Değirmenci, J. Deri, B. Firner, B. Flepp, S. Gogri, J. Hong, L. Jackel *et al.*, "The NVIDIA pilotnet experiments," *arXiv preprint arXiv:2010.08776*, 2020.