DNA: A General Dynamic Neural Network Accelerator

Lian Liu, Jinxin Yu, Mengdi Wang, *Member, IEEE*, Xiaowei Li, *Member, IEEE*, Yinhe Han, *Senior Member, IEEE* and Ying Wang, *Member, IEEE*

***Abstract*—Due to the demonstrated superiority, dynamic neural networks (NNs), which adapt their network structures to different inputs, have been recognized as an optimized alternative to conventional static NNs. However, researchers have not explored the implications of dynamic NN on neural processing unit (NPU) architecture design. Consequently, we analyze the characteristics and inefficient sources of executing dynamic NNs on existing hardware. From our analysis, existing NPUs, designed for static NNs, cannot effectively handle the execution of dynamic operator and agent-dependent data loading in dynamic NNs.**

To this end, we present DNA, an efficient accelerator optimized to deal with the challenges of running general dynamic NNs. Firstly, to improve the execution efficiency of dynamic operators, we propose a transverter-based online scheduling strategy to rapidly generate efficient scheduling for each dynamic operator. Secondly, to mitigate hardware idleness caused by the non-deterministic and agent-dependent data access patterns in dynamic NNs, we propose a novel predictor-based prefetching strategy that achieves effective data preloading with negligible cost. We implemented our accelerator, DNA, by integrating an additional online scheduler into a typical many-core baseline accelerator. According to our evaluation of various dynamic NNs, DNA achieves $3.48\times$ speedup and $3.03\times$ energy savings over the baseline accelerator.

***Index Terms*—Dynamic NN, NPU Design, Accelerator.**

I. Introduction

CURRENT dynamic neural networks (NNs), which adapt their structure, parameters, or connections at runtime based on input features [1], are increasingly replacing traditional static networks [2]. Their input-specific adaptability enhances representation capacity, leading to improved accuracy [3], [4], computational efficiency [5], robustness [6], and generality [7]. LeCun et al. [8] have emphasized that dynamic architectures now dominate mainstream applications. As a result, dynamic NNs have become a central focus in deep learning and are widely adopted across domains. For instance, large models like GPT-4 and Switch Transformer [9], [10] dynamically activate only a subset of experts based on input tokens (Figure 1a). Despite their growing prominence, the architectural implications of dynamic NNs for neural processing units (NPUs) remain underexplored.

This work is motivated by the need to understand the execution characteristics of dynamic NNs and their impact on NPU design. While traditional static NNs are structured as operator graphs with fixed execution order [11], [12], dynamic NNs have architectures that adapt based on input data [4], [6]. As shown in Figure 1, each dynamic layer introduces an *Agent* that modifies a *backbone operator*—the original operator with fixed shape and parameters—into a *dynamic operator* tailored to the input. Execution proceeds in two stages: (1) the *Agent* processes the input to customize the backbone operator, and (2) the resulting dynamic operator computes the output. This process repeats across layers until the inference is complete. Thus, executing dynamic NNs requires runtime adaptation of operator structures, introducing new challenges for optimizing both operator computation and data loading efficiency.

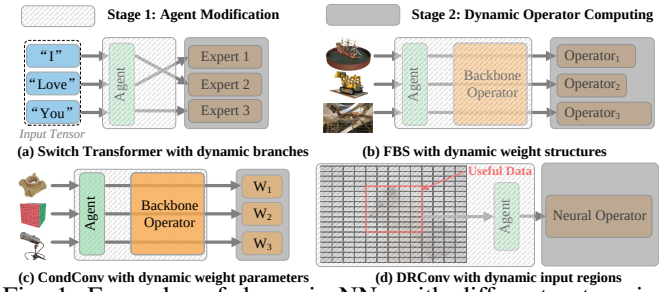


Fig. 1: Examples of dynamic NNs with different categories. The execution flow in dynamic NN contains two stages: (1) agent modification and (2) dynamic operator computing.

Unfortunately, existing NPUs [13], designed primarily for static neural networks, struggle to address these challenges. As shown in Figure 2a, executing dynamic NNs requires online scheduling of operators and data loading based on the *Agent*'s outputs prior to computation. Current compilation frameworks [14], [15] employ dynamic graph scheduling, which uses runtime information to select a kernel from a vendor-provided template library to optimize dynamic operator execution. However, these libraries, typically offering only a limited set of pre-implemented kernels, cannot cover the vast scheduling space required by dynamic operators (Section II-A). As illustrated in Figure 2a, dynamic operators are ultimately processed similarly to static ones, leading to low PE utilization (20–50%) during execution. Furthermore, data dependencies in dynamic NNs are agent-driven; for example, in Figure 1a, selecting expert weights and input tokens depends on agent decisions. These dependencies introduce execution stalls, further degrading on-chip buffer and PE utilization in existing NPUs.

To this end, we present DNA, a novel accelerator design for the unique characteristics of dynamic NNs. As indicated in Figure 2b, DNA controls the scheduling of dynamic operators and optimizes data loading. On the one hand, to generate efficient scheduling for the dynamic operator, we propose a novel transverter-based online scheduling technique, that

Manuscript received 21 October 2024; revised 25 April 2025; accepted 26 June 2025. This work was partially supported by the National Natural Science Foundation of China (Grant No. 62025404 and No. 62222411) and the National Key R&D Program of China (Grant No. 2023YFB4404400). This article was recommended by Associate Editor Malloy, Heather. Ying Wang is the corresponding author (wangying2009@ict.ac.cn).

Lian Liu, Jinxin Yu, Mengdi Wang, Xiaowei Li, Yinhe Han and Ying Wang are with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with the Department of Computer Science, University of Chinese Academy of Sciences, Beijing 100190, China.

Lian Liu and Xiaowei Li are also with the Zhongguancun Laboratory.

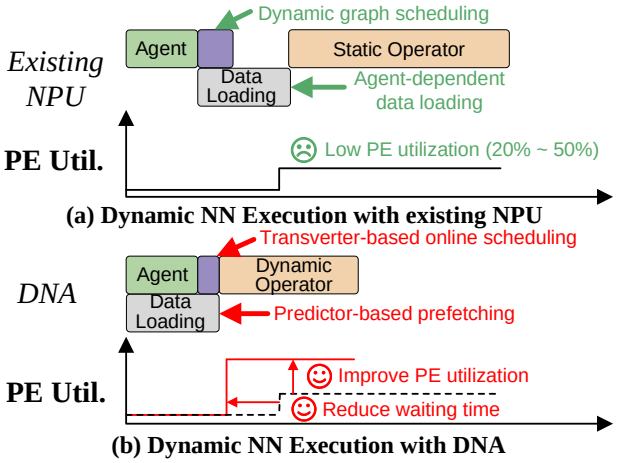


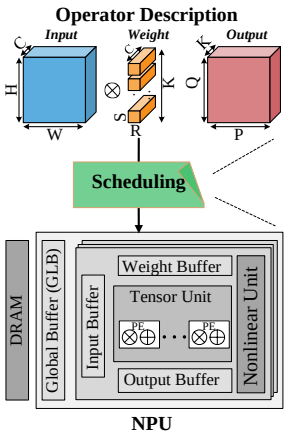
Fig. 2: DNA utilizes two novel optimizations to achieve high-performance execution with negligible extra cost.

utilizes the offline scheduling of the backbone operator and online transforms it to accommodate the dynamic operator. Specifically, we heuristically devise a greedy algorithm to achieve the transformation above and implement a hardware-based transverter[1] to reduce the cost of online scheduling (Section V). On the other hand, to break the dependency of data loading on agent computation results, we propose a predictor-based prefetching strategy that preloads data in parallel with agent computation. Specifically, we observe an imbalance and correlation in data access patterns across samples in dynamic NNs (Section III-B). This motivates the use of prior data access traces to predict accesses in the current input, thereby enabling predictor-based prefetching for data preloading (Section VI). As presented in Figure 2b, the proposed transverter-based online scheduling and predictor-based prefetching optimizations effectively improve PE utilization and reduce waiting time for operator execution, respectively.

In general, this work makes the following contributions:

- To the best of our knowledge, this paper is the first to propose an accelerator for general dynamic NNs. By examining the source of inefficiency in existing NPUs for dynamic NN support, we design a versatile and input-adaptive architecture to accelerate dynamic NN as well as conventional static NN models.
- We propose a novel transverter-based online scheduling that generates efficient scheduling for the dynamic operators modified by the *Agent*. The hardware-assisted online scheduling mechanism dramatically improves hardware utilization with negligible runtime cost.
- We leverage predictor-based prefetching to break the *Agent* dependencies in data loading for dynamic operators. This improves on-chip buffer utilization in the NPU and reduces stalling incurred by dependencies.
- We implement and demonstrate the effectiveness of the proposed DNA architecture using several representative dynamic NNs. Evaluation on DNA achieves 3.48× speedup and 3.03× energy saving on average over the baseline accelerator architecture.

[1]The term "transverter" is originally from the field of electrical engineering. Here, it is used to refer to a scheduling converter.




```
//DRAM level
for n0 = [0 : 1):
  for q2 = [0 : 2) :
// Global Buffer level
for p2 = [0 : 7) :
  for q1 = [0 : 7) :
    spatial_for r0 = [0 : 3) :
      spatial_for k1 = [0 : 4) :
// Input Buffer level
for k0 = [0 : 2) :
// Weight Buffer level
for c1 = [0 : 2) :
  for p1 = [0 : 2) :
// Output Buffer level
for s0 = [0 : 3) :
  spatial_for p0 = [0 : 2) :
    spatial_for c0 = [0 : 8) :
// Register level
for q0 = [0 : 2) :
    O[k][p][q] +=
        I[:c][p:p+r][q:q+s] *
        W[k][:c][:r][:s]
```

Fig. 3: DNN operator scheduling framework. The scheduling framework considers operator dimensions and accelerator parameters to generate scheduling for the operator ($R = S = 3, P = Q = 28, C = K = 8, N = 1$) in the rightmost.

II. Background & Related Work

A. Operator Description and Scheduling

Operator Description. An operator contains both weight parameters and operator dimensions, as shown in the upper left corner of Figure 3. The operator dimension can be expressed by a nested loop with 7 variables: R, S, P, Q, C, K, N. R and S refer to the weight width and height, P and Q refer to the input width and height, C and K refer to the weight channels, which are also identical to the input and output channel respectively, and N refers to the batch size. A unique operator can be obtained once the operator dimensions and corresponding weight parameters are determined.

Operator Scheduling Framework. Operator execution efficiency on NPUs is highly dependent on scheduling; for instance, optimized schedules can deliver up to 3.2× performance over random ones [16]. As shown in Figure 3, a typical scheduling framework takes operator dimensions and accelerator configurations to generate high-performance schedules. NPUs [17] typically consist of PE arrays connected via on-chip networks and a multi-level memory hierarchy (e.g., DRAM, global buffers). Operator scheduling comprises three key components: (1) tiling strategy, which defines tile sizes across dimensions (e.g., H/W of inputs); (2) parallelism strategy, specifying loop parallelization over PE arrays via `spatial_for`; and (3) computation order, which determines loop nesting at each memory level. These factors significantly affect performance [18], but optimizing them is challenging due to the vast search space—up to 10^{15} options for the second VGG16 layer [11]—necessitating advanced compilers [19]. In consequence, existing scheduling works take seconds, minutes, or even hours to generate a solution [16], [19]. However, since executing one operator on conventional NPUs typically takes milliseconds [20], regenerating the scheduling for every dynamic operator is impractical.

B. Dynamic NN

Compared to static models, dynamic NNs can adapt their operator structures, parameters, or connections to different inputs during runtime. To better illustrate the characteristics of dynamic NNs, we first describe the execution flow, and then provide diverse categories and examples of dynamic NNs.

Dynamic NN Execution. In dynamic NNs, each layer begins by passing input data to an *Agent*, which modifies a backbone operator to produce a corresponding dynamic operator. This operator processes the input to generate output, which feeds into the next network layer, repeating until inference completes. Notably, dynamic operators are not created from scratch. Instead, agents adjust the structure, parameters, or connections of backbone operators. Agent computations typically involve lightweight nonlinear operations (e.g., softmax-based expert selection), whereas dynamic operators perform compute-intensive tasks such as matrix multiplications or convolutions. Consequently, as shown in Figure 3, we assign agents to the NPU's nonlinear unit and dynamic operators to the tensor unit. Since dynamic operators dominate execution time, this paper focuses on optimizing their performance.

Categories and Examples of Dynamic NN. Based on the different strategies of operator modification, we can effectively classify dynamic NNs into several categories. As depicted in Figure 1, we divide dynamic NNs into four categories and provide specific examples. Our goal in establishing these categories is to find underlying optimization opportunities and enable broad coverage of diverse dynamic NNs. Figure 1a presents a typical dynamic branches network, Switch Transformer [10], that modifies the operator connection for each token. For example, for the token "Love", the *Agent* selects Expert 1 to execute, avoiding sending the token to Expert 2 and 3. It reduces computation and data access by minimizing the number of experts required for each token. Figure 1b presents FBS [21], a network that dynamically prunes the weight channels (C/K for weight) based on different images. Specifically, for a backbone operator with operator dimensions $C = K = 256$, the possible dynamic operators can occupy from 1 to 256 channel sizes (both C and K). Figure 1b uses different operator lengths to denote different channel sizes. Figure 1c presents a dynamic weight parameters network, Condconv [22], that only changes the weight parameters for different inputs. Figure 1d presents a network, DRConv [23], that utilizes *Agent* to select the useful region (H/W for input data) of input to reduce the computing cost. Therefore, the input dimensions for different input data are quite different. Overall, dynamic NNs adjust operator connectivity (Figure 1a), dimensions (Figure 1b,d), and parameters (Figure 1c) based on runtime inputs. In this paper, we primarily exemplify our proposed optimizations using FBS (Section V-D) and Switch Transformer (Section VI-C), but the proposed techniques are general and can also apply to other dynamic NNs. For instance, the early-exit based BranchyNet [24] can also utilize the proposed prediction-based prefetching to alleviate unnecessary data loading (detailed in Section VI-C).

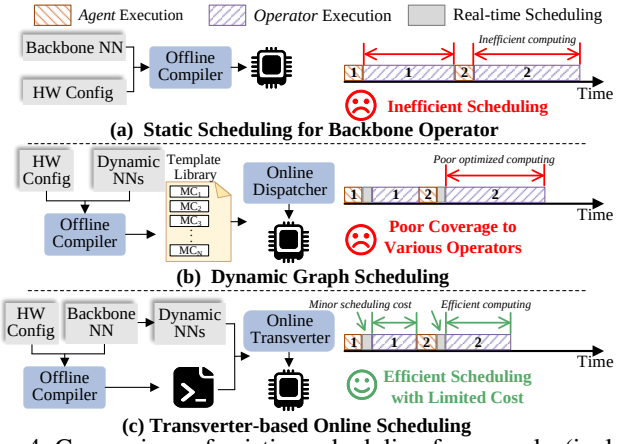


Fig. 4: Comparison of existing scheduling frameworks (including our design) for dynamic operators.

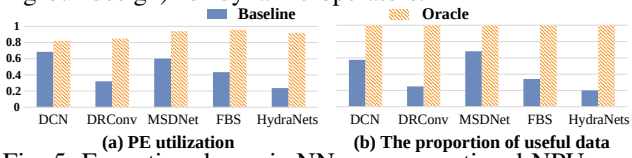


Fig. 5: Executing dynamic NNs on conventional NPUs results in (a) low PE utilization and (b) unnecessary data loading.

C. Specific Dynamic NN and Accelerator Co-design

With the rise of dynamic NNs, some specialized accelerators are proposed to optimize certain dynamic NN implementations, as summarized in Table I. DRQ [25] devised a sensitive region prediction mechanism to optimize dynamic precision networks. DCNA and CoDeNet [26], [27] proposed the table tracking method and supports several fixed data access patterns to minimize the data movement cost in deformable convolution networks. DPACS [28] is an algorithm-hardware co-design that accelerates its self-optimized dynamic pruning networks. All these hardware solutions in prior works are closely associated with their customized dynamic NN architectures, and can hardly be applied to the wide area of common dynamic NNs. In consequence, a general high-performance accelerator that efficiently deals with both the diverse dynamic NN family and the conventional static NNs is desired.

III. Analysis & Motivation

Compared to conventional NPUs, dynamic NN accelerators need to efficiently optimize the dynamism of the model introduced by dynamic operators. We analyze the sources of inefficiency for existing NPUs and discuss corresponding motivations for efficient accelerator design as follows.

A. Inefficient Dynamic Operator Scheduling

Inefficiency with Existing Scheduling. Existing DL optimization frameworks [19], [29] treat JNN models as static data flows and generate static scheduling for backbone operators (Figure 4a), overlooking runtime variations in dynamic operators. To address dynamic input shapes, recent compilers [14], [15], [29] adopt dynamic graph scheduling on GPUs (Figure 4b). These frameworks decompose operators into sub-units that are invariant to input-dependent changes, optimize them offline, and store the results in a fixed template

TABLE I: Comparison of DNA and Specialized Accelerators on Dynamic NNs.

Works	Dynamic NNs supported	Category	*Dynamic NN Challenges and Solutions*		
			Low PE Utilization	Repetitive Data Access	Irregular Memory Access
DRQ [25]	dynamic precision networks	dynamic parameter	✗	✗	sensitive region prediction
DCNA [26]	DCNs	dynamic input region	✗	parallel BLI processing	dependency table tracking
CoDeNet [27]	limited DCNs	dynamic input region	✗	✗	certain deformable operators
DPACS [28]	dynamic pruning networks	dynamic architecture	algorithm-optimized pruning	✗	✗
DNA (this work)	**general dynamic NNs**	**all categories**	transverter-based dynamic mapping	fused-graph scheduling	prediction-based prefetching

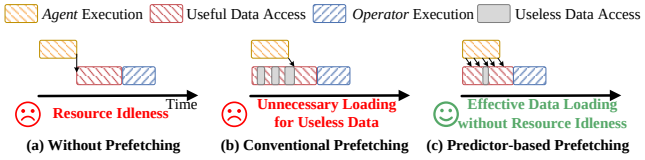
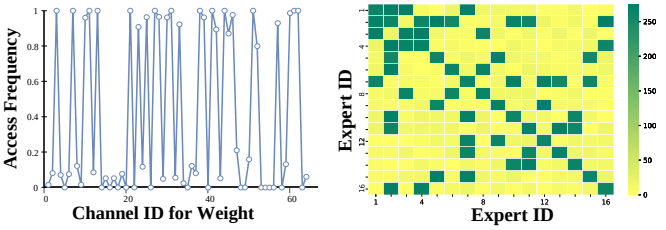


Fig. 6: The predictor-based prefetching can avoid resource idleness with minor useless data loading.



(a) FBS [21] (b) Switch Transformer [10]

Fig. 7: The imbalanced and correlated data access distribution pattern in two typical dynamic NNs.

library. At runtime, the system selects an appropriate schedule from this library based on the actual dynamic operator. For example, Nimble [15] applies tiling-based kernel splitting and sampling-based compilation, while DietCode [14] uses decision trees to map dynamic operators to pre-compiled micro-kernels. Despite these efforts, the limited size of template libraries—typically only dozens of entries, which fails to cover the vast scheduling space (Section II-A). For instance, a single dynamic operator in FBS [21] can have up to 50,000 variants. Consequently, the baseline (using Nimble-style scheduling) achieves only 41.8% average PE utilization across dynamic NNs (Figure 5a).

Motivation for Hardware Assistant Online Scheduling. To balance execution efficiency and runtime overhead, a novel approach is desired that enables high-performance scheduling for each dynamic operator with minimal generation cost. Achieving this requires an efficient scheduling strategy and a reduced scheduling space. Recall the execution flow of dynamic NN, we notice that the dynamic operator is ***modified*** from the backbone operator with the corresponding *Agent*. Our insight is that: ***The preliminary scheduling of the backbone operator contains potential information for subsequent online scheduling of the dynamic operator.*** Motivated by this, we introduce a transverter-based online scheduling strategy (Figure 4c), which reuses backbone scheduling to guide dynamic operator scheduling. This approach simplifies the problem by ignoring changes to computation order and focusing solely on adjusting dynamically varying operator dimensions (Section V-B). Unlike prior software-based frameworks, our method avoids reliance on fixed template libraries and enables on-the-fly tuning to produce optimized schedules even for dynamic operators that templates cannot cover.

B. Agent-dependent Dynamic Data Access

Inefficiency with Agent-dependent Data Loading. There is a close dependency between *Agent* and *dynamic operator* during the execution of dynamic NNs: only after finishing the execution of *Agent* can we determine the data needed by the dynamic operator. Therefore, the agent-dependent data loading will cause long periods of resource idleness (Figure 6a). For example, during the execution of DCN [3], PEs in the NPU will be idle over 21% of the whole execution time. One approach to mitigate the performance degradation incurred by such data dependencies is prefetching. However, existing prefetching strategies in NPUs target static NNs, determining required data loading based on model architecture extracted during offline compilation [30]. Unfortunately, only a small fraction of data needs to be loaded in dynamic NNs. As shown in Figure 5b, only 36.7% (on average) of data is useful during execution among various dynamic NNs. Consequently, adopting conventional prefetching strategies would introduce unnecessary data loading (Figure 6b).

Motivation for Predictor-based Prefetching. Consequently, we need a method to predict required data loading before agent computation, in order to control hardware prefetching behavior. To achieve this, we analyze the data access characteristics of two typical dynamic neural networks, FBS and Switch Transformer. FBS [21] performs dynamic pruning across weight channel dimensions. As shown in Figure 7a, some channels (indexed by Channel ID) are more prone to pruning than others. Switch Transformer [10] chooses different experts for each input token. As indicated in Figure 7b, the selection between different experts exhibits a high correlation. For instance, when a token selects the first expert, it is highly likely also to select the second, third, and seventh experts. Therefore, the imbalanced and correlated data access distribution patterns motivate ut to utilize prior data access traces to predict data access characteristics for the current input, enabling predictor-based prefetching.

IV. DNA: Overview

Based on the analysis and motivations in Section III, we present a novel dynamic NN accelerator, DNA. As illustrated in Figure 8, DNA integrates an execution scheduler and a predictor on top of typical NPU design [13] to optimize the execution efficiency for dynamic NN. A workflow different from the classical framework is also used to support the scheduling optimization for dynamic NNs.

A. Architecture Overview

Core Architecture. DNA employs a many-core compute engine consisting of multiple PE arrays, following the design

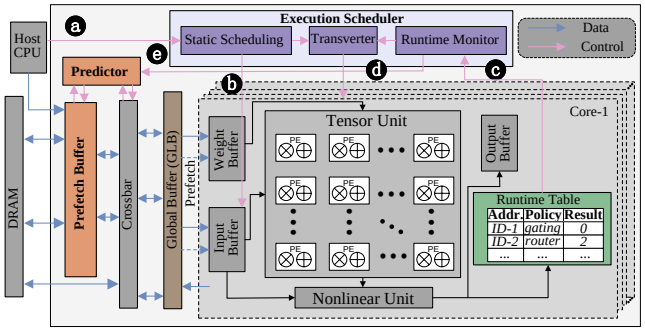


Fig. 8: The overall architecture and workflow of DNA.

of existing NPUs [31], [32]. Each core includes a set of PEs and a dedicated nonlinear unit to execute *operators* and *Agents*, respectively. To capture agent decisions at runtime, each core maintains a runtime table that stores operator modification data, including agent-controlled addresses (e.g., *channel ID*, *expert ID* in Figure 7), policies (e.g., gating [33], routing [10]), and results. These elements are specific to each dynamic NN. For instance, in the FBS pruning network, the agent uses the gating policy on the *channel ID* to determine whether a channel is pruned (e.g., result 0 means the channel is retained). All agent outcomes are forwarded to the runtime monitor. Furthermore, each core contains private buffers for weight, input, and output, respectively.

Inter-core Connection and Memory Hierarchy. A global buffer (GLB) is closely connected with each core and an additional prefetch buffer is designed to bridge the computing engine to the off-chip memory. Unlike traditional NPUs, the GLB in DNA not only distributes required data to each core for computation, but also can transfer the computation results from one core to others. This enables dynamic NNs with dynamic parameters to benefit from fast on-chip data forwarding. For example, for dynamic weight parameters network Condconv [22] (Figure 1c), the inter-core connection can transfer the computing results of the agent to other cores to finish subsequent dynamic operator computing.

Online Scheduler. One of the major differences between DNA and conventional NPUs is the online scheduler, including the execution scheduler and predictor. The execution scheduler is critical for promoting scheduling efficiency for dynamic operators during runtime. Based on the preliminary scheduling for backbone operator, the execution scheduler collects the runtime information and conducts a novel scheduling transformation algorithm within the transverter unit, which generates the high-performance scheduling for every dynamic operator modified from the backbone operator. This scheduling transformation procedure is named transverter-based online scheduling. On the other hand, the predictor controls the prefetch buffer to achieve accurate data prefetching and eviction, avoiding hardware resource idleness and reducing useless data movement with negligible misprediction cost.

B. Workflow Overview

The overall DNA workflow includes *Offline Compilation* and *Online Optimization*. *Offline Compilation* generates appropriate static schedulings for agents and backbone operators (Figure 8ⓐ). Compilation for agents maps agent computation onto nonlinear units and configures the runtime table to record the agent-controlled address, policy and result (ⓑ). The generated schedulings for backbone operators are stored in the execution scheduler for subsequent online scheduling.

Online Optimization includes two main components, transverter-based online scheduling and predictor-based prefetching, to achieve efficient scheduling and effective data access for dynamic operators, respectively. We utilize the preliminary scheduling for the backbone operator and the corresponding runtime information acquired by the runtime monitor (ⓒ) to guide the scheduling transformation of dynamic operators. Specifically, the transverter integrates our proposed scheduling transformation algorithm (Section V-B) to improve the execution efficiency (ⓓ). Meanwhile, the predictor will direct the behavior of the prefetch buffer to preload useful data in operator computing (ⓔ). Details of the prediction strategy will be presented in Section VI-A.

V. DNA: Transverter-based Online Scheduling

A. Scheduling Problem Formalization

In this paper, we use the formulation described in CoSA [34] to interpret the operator scheduling procedure, and all the parameters used for scheduling description are summarized in Table II. Note that other scheduling descriptions [16] can also be adopted in our framework so the DNA toolchain can be easily bridged to popular NN compiling stacks [19], [29].

Scheduling can be formulated as a prime factor allocation problem [34]. Generally, the size of each operator dimension j can be factorized into multiple values and recorded into a two-dimensional table $F_{(j,n)}$. Take the workload in Figure 3 as an example, the third operator dimension $P = 28$ can be factorized into three prime factors $2, 2, 7$ so that we can record them into the table as $F_{(3,1:3)} = 2, 2, 7$. By allocating the prime factors to the corresponding memory levels i and determining the parallelism strategy k, we can obtain a scheduling represented by a binary matrix $X_{i,(j,n),k}$ (Simplified as the Scheduling Table in Figure 9b). The three scheduling aspects described in Section II-A are all contained in the binary matrix X. Specifically, *tiling strategy* is located by assigning prime factors $F_{(j,n)}$ to the corresponding memory level i, *parallelism strategy* is denoted by k, where $k = 0$ indicates `spatial_for` and *computation order* is chronicled by extending rank indices $O_0, O_1, ..., O_z$ to the memory level of interest. For example, different computation orders in the global buffer cause different memory access patterns, so we extend rank indices, from one to the number of all prime factors, for the global buffer to cover all possible computation order representations.

Furthermore, to satisfy the requirements in hardware resource η, two basic constraint rules are considered in the scheduling configuration m, denoted as R_1 and R_2:

$$\begin{aligned} R_1(m,\eta) &: m.U_i \le \eta.MS_i, \quad \forall i \\ R_2(m,\eta) &: m.C_i \le \eta.P_i, \quad \forall i \end{aligned} \tag{1}$$

Where $m.U_i$ and $m.C_i$ represent the occupied memory and computing resources at each memory level i (e.g. GLB, Weight

TABLE II: Notations of Scheduling Space.

Notation	Description	Details
Indices		
i	memory level for accelerator	Register, GlobalBuffer, etc.
j	operator dimension	R, S, P, Q, C, K, N
k	parallelism strategy	spatial / temporal
v	data tensor	weight, input, output
n	prime factor	-
z	computation order	$O_1, O_2, ..., O_z$
Scheduling Strategy		
$X_{i,(j,n),k}$	Scheduling configuration	A binary matrix

Buffer), respectively. $\eta.MS_i$ and $\eta.P_i$ denote the total buffer and computing resources at each memory level (Figure 9b).

B. Online Scheduling Transformation

Our online scheduling transforms the preliminary backbone operator scheduling into online schedulings for the input-decided dynamic operators. Following the formulation above, the scheduling transformation procedure can be viewed as a prime factor reallocation problem. The reallocation operation deemed in our framework is to assign the prime factor $F_{(j,n)}$ indexed by (j, n) from memory level i_1 and parallelism strategy k_1 to another one (i_2 and k_2). In the absence of any prior knowledge, the reallocation problem is as complex as the original allocation problem. In consequence, a problem that should be figured out is: *What information is preserved in the preliminary backbone operator scheduling?* From our observation, about 97% of sampled dynamic operators, under the oracle scheduling circumstance, occupy the same computation order as preliminary scheduling. So in this procedure, only *tiling strategy* and *parallelism strategy* are considered.

Algorithm 1: Scheduling Transformation

Input : input-dependent dynamic and its corresponding backbone operator of dynamic NN model τ, α; hardware constraint table η; preliminary backbone operator scheduling ϖ;
Output: transformed efficient scheduling m;

1 $m \leftarrow$ `Initialization`(α, ϖ, τ);
2 **for** $i = I-1; i \geq 1; i--$ **do**
3 $m \leftarrow$ `MaxBuff`(m, η, i);
4 $m \leftarrow$ `MaxComp`(m, η, i);
5 **Function** `Initialization`(α, ϖ, τ)**:**
6 $m \leftarrow \varpi$;
7 Compare α and τ to record modified dimensions;
8 **forall** *operator dimension j has been changed* **do**
9 factorize new obtained loop bound into $F_{(j,n)}$;
10 $m.X_{i,(j,n),k} = 0, \forall i \neq I$ and $m.X_{I,(j,n),1} = 1$;
11 **Function** `MaxBuff`(m, η, i)**:**
12 **forall** *operator dimension* $j \in m$ & $A_{j,v}B_{i,v}$ **do**
13 **if** $\frac{\eta.MS_i}{m.U_i} \geq \min F_{(j,n)}$ & $m.X_{i+1,(j,n),1}$ **then**
14 $m.X_{i,(j,n),1} = 1$, $m.X_{i+1,(j,n),1} = 0$;
15 **Function** `MaxComp`(m, η, i)**:**
16 **while** $r = \frac{\eta.P_i}{m.C_i} \geq 2$ **do**
17 **for** $i' = i; i' \geq 0; i'--$ **do**
18 **forall** *operator dimension* $j \in m$ **do**
19 **if** $F_{i',(j,n)} \leq r$ & $m.X_{i',(j,n),1}$ **then**
20 $m.X_{i,(j,n),0} = 1, m.X_{i',(j,n),1} = 0$;
21 $r /= F_{(j,n)}$;

However, the search space is still larger than 10^9 for the given example in Figure 3, which defies real-time online scheduling. Fortunately, the dynamic operators do not change in every operator dimension compared to its backbone operator, so we simplify the reallocation procedure by reserving allocation choices for unchanged operator dimensions.

We propose a greedy-based algorithm, as shown in Algorithm 1, to achieve rapid scheduling transformation instead of finding the optimal scheduling. The transformation consists of the following three stages. Given an input-dependent dynamic operator τ, the first stage INITIALIZATION generates an executable scheduling with maybe-poor efficiency. Then, MAXBUFF maximizes the on-chip memory utilization by exchanging the tiling configuration of each memory level i. Lastly, MAXCOMP maximizes computing resources utilization by adjusting parallelism strategy k. The second and third stages of scheduling transformation are iteratively executed for each memory level from outside-in (from GLB to Register). The overview of this procedure is presented in *line* 1-5. We explain the details as follows.

Initialization. This algorithm begins by assigning preliminary backbone operator scheduling ϖ to the desired scheduling m for a dynamic operator and identifying the difference operator dimensions (e.g. C/K for dynamic pruning) between the backbone operator α and the modified dynamic operator τ (*line* 7-8). The detailed modification information is accessed from the runtime table in Figure 8 when executing *Agent*. Then, this function traverses all the changed operator dimensions j and factorizes new dimension size into the table $F_{(j,n)}$. Additionally, all data of changed operator dimension is mapped to the outermost memory level I in a temporal way by adjusting assignments in X (*line* 9-11). After that, we can obtain an executable scheduling as an initial seed, which is not a premier solution since all newly generated prime factors are mapped into the outermost memory level.

Maximizing On-chip Memory Utilization. This stage first traverses all possible operator dimensions stored in memory level i. For instance, R/S dimension of weights can be stored in the Global Buffer and Weight Buffer. A and B in Algorithm 1 are two binary matrices encoding the relation between operator dimension and data tensor, data tensor and memory level, respectively [34]. As a result, we calculate the occupied memory resources $m.U_i$ for memory level i as:

$$U_I = \sum_{v=0}^{2} \prod_{i=0}^{I-1} \prod_{j=0,n=0}^{6,N_j} \prod_{k=0}^{1} \begin{cases} F_{(j,n)}, & X_{(j,n),i,k} A_{j,v} B_{I,v} = 1 \\ 1, & \text{otherwise} \end{cases} \tag{2}$$

Where $A_{j,v}B_{i,v}$ locates the given data tensor v. In detail, we use $A_{j,v}B_{I,v}$ to denote which operator dimension belongs to the current memory level, and accumulate all tensors that can be allocated within the current memory level (e.g., weight tensor can only be mapped on global memory/buffer and weight buffer), to compute the occupied memory resources. Both spatial and temporal factors should be included in the memory utilization.

This procedure obtains the total and occupied buffer size ratio $\frac{\eta.MS_i}{m.U_i}$, and compares it with the minimal prime factor $F_{(j,n)}$ in outer memory $i+1$. If constraint R_1 (Equation 1) is met, the data in the outer memory is moved to the current memory level i to improve memory utilization. This process

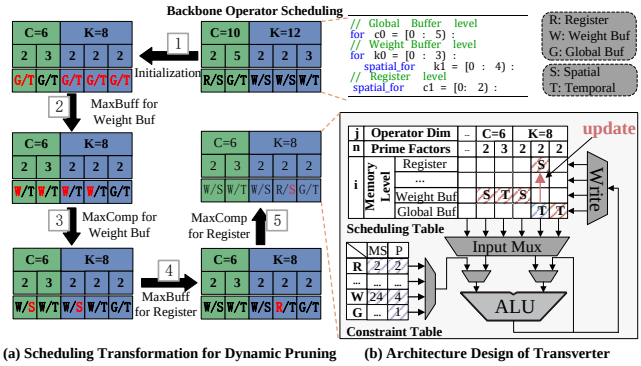


Fig. 9: The architecture and example of transverter unit.

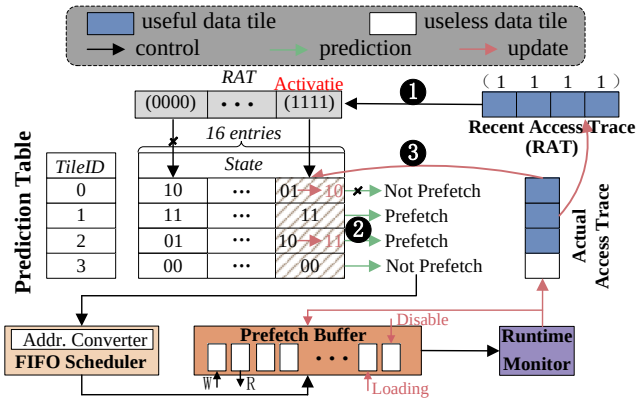


Fig. 10: The mechanism and architecture of predictor.

is described as the function `MaxBuff` in Algorithm 1.

Maximizing PE Utilization. Similar to the second stage, this stage (`MaxComp` in Algorithm 1) begins with calculating the ratio of total and utilized computing resources $r = \frac{\eta.P_i}{m.C_i}$ for each memory level i. The utilized computing resources for given scheduling m are expressed as follows.

$$C_I = \prod_{i=0}^{I} \prod_{j=0,n=0}^{6,N_j} \begin{cases} F_{(j,n)}, & X_{(j,n),i,0} = 1 \\ 1, & \text{otherwise} \end{cases} \tag{3}$$

The utilized computing resources can be expressed as how many PEs are used at one time. In consequence, we compute the utilized computing resources as the number of spatial mapped PEs ($k = 0$) in each memory level i.

If the ratio r is greater than 2 (the smallest prime factor), this stage traverses all inner memory levels ($i' \leq i$) for all operator dimensions j. When prime factor $F_{(j,n)}$ located in current memory level i' is less than the original ratio r, as well as mapped in a temporal execution manner ($k = 1$), the prime factor is then remapped to computing units spatially. With this operation, PE utilization is appropriately improved.

C. Online Scheduler Implementation

As revealed in Section III, efficient and fast online scheduling is critical to the performance of dynamic NN. We need to generate the corresponding scheduling for every dynamic operator. However, implementing the proposed scheduling transformation algorithm on the host CPU would incur extra synchronization latency comparable to operator execution, which is unacceptable. Consequently, we propose a specialized hardware-implemented transverter in our on-chip execution scheduler, as illustrated in Figure 9b. Two look-up tables are created to record the scheduling configuration m and hardware resource constraints η. The mission of the transverter is to calculate the occupied buffer and computing resources for the scheduling configuration, and the corresponding comparison results $\frac{\eta.MS_i}{m.U_i}$ and $\frac{\eta.P_i}{m.C_i}$. As a result, we integrate an ALU to the table to cumulatively derive these results, following Equation (2) and (3). With the help of multiplexer (Mux), transverter selects inputs from the scheduling table and constraint table, or inherits data from the output of ALU. Once the transverter receives all the results, it enables the write signal to modify the values in the scheduling table. For example, by rewriting the values in the scheduling table corresponding to the second prime factor of dimension K's memory level and parallelism strategy, we can achieve one update operation, as Figure 9b shows. Through multiple updates, we complete the scheduling transformation and obtain an efficient scheduling for the dynamic operator.

D. Discussion: Case-study and Coverage

We present an example of scheduling transformation for a dynamic pruning network (prunes C from 10 to 6 and K from 12 to 8) in Figure 9a. For simplicity, we omit a large number of operator dimensions and memory levels, only considering scheduling operator dimensions C, K into registers, weight and global buffer (R, W, G). The preliminary scheduling ϖ for the backbone operator is shown in the top right corner of Figure 9. As both C and K are modified, the initialization procedure (Figure 9 1) assigns all prime factors into GLB in a temporal manner (denoted as G/T). Under the constraints reported in the constraint table, transverter adopts the greedy optimization strategy to maximize buffer and PE utilization for memory levels W and R (Figure 9 2-5). For example, the total memory and computing resources at memory level W are 24 and 4, respectively, as the constraint table in Figure 9b shows. Therefore, we assign all the prime factors of C and the first two prime factors of K from global buffer to weight buffer (G $\rightarrow$ W) in procedure 2, and also assign the first prime factor of C and K to spatial parallelism choice (T $\rightarrow$ S) in procedure 3. Compared to preliminary scheduling, the final acquired online scheduling improves both the PE and buffer utilization by 2.5×, demonstrating its effectiveness.

The online scheduling can optimize not only the dynamic weight structure operators in Figure 1b, but also support other categories of dynamic NNs. For example, networks with dynamic input regions (Figure 1d) can achieve high-performance execution by adjusting the input dimension P, Q during the scheduling procedure. In addition, networks with dynamic branches (Figure 1a) can also treat the input dimension of each expert as a variable to achieve online scheduling.

VI. DNA: Predictor-based Prefetching

In addition to inefficient dynamic operator scheduling, agent-dependent data access also hampers the execution efficiency of dynamic NNs. As demonstrated in Figure 6, sequentially computing agent and dynamic operator without prefetching introduces computing resources idleness, while

adopting the conventional prefetching strategy causes unnecessary access to useless data. Fortunately, we found that it is possible to predict the memory access pattern of dynamic operators before *Agent* execution. Specifically, based on the intrinsic similarity among data loading behaviors, prior data access traces can be used to anticipate future data requirements. Therefore, DNA incorporates an access predictor in its DMA controller to control prefetching behavior and mitigate memory access overhead. The predictor actively prefetches useful data and discards non-reusable data. We illustrate the details of prediction methods in Section VI-A and introduce the corresponding hardware implementation in Section VI-B. To demonstrate the efficacy of our proposed method, we also conduct a case study in Section VI-C.

A. Memory Access Pattern Prediction

Since the data is computed in the granularity of sub-tensors (e.g. expert-level and channel-level in Figure 1a,b), the prefetching and eviction of data will also be performed according to the granularity of the corresponding model. In this section, we refer to the granularity of data access as *tile*. We discuss the prediction methods for access patterns in the granularity of data *tile*. The *TileID* in Figure 10 denotes the index for one specific data *tile*.

As discussed in Section III-B, the imbalanced and correlated patterns of agent-dependent data access motivate the use of prior access traces to anticipate data requirements for incoming samples. Prior works [10], [25], [35] have shown that semantically similar inputs often exhibit analogous data usage. For example, in Switch Transformer, tokens like “Love” and “Admire” frequently activate the same expert (e.g., Expert 1), leading to repeated access of similar weight tiles. Based on the observation, we divide the overall prediction process into two stages. First, we need to identify similar samples that are analogous to the current input. Then we leverage the historical data access traces of those samples to predict subsequent data access patterns for the current input sample. Therefore, we propose a two-level prediction table that combines the recent and historical access trace for data *tile*, as shown in Figure 10. The data access trace represents whether each agent-controlled data *tile* is used in subsequent operator computing. For example, in Switch Transformer, a data *tile* represents the weights for each expert, and the corresponding data access trace records whether the weights for a particular expert are loaded. All the access traces are stored in the runtime table and passed to the predictor via the runtime monitor, as shown in Figure 8. Our prediction strategy leverages the recent access trace (*RAT*) of the current sample to identify analogous samples, since samples with similar access traces tend to have representational similarities. Additionally, we maintain the historical access traces of samples in the prediction table. As presented in Figure 10, each entry in the prediction table is denoted by a 2-bit $State$, which indicates the prediction result for the corresponding data *tile*, e.g. 10 and 11 represent useful data *tile* and expect a prefetch tag, while 00 and 01 are the opposite connotations. The table entries are located by two parameters: data *tile* address $TileID$ and recent access

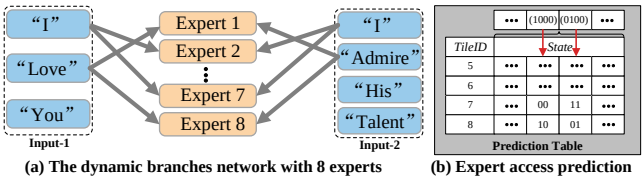


Fig. 11: The example of predictor-based prefetching.

trace RAT. The predicting procedure is as follows. At first, the prefetch buffer addresses the table entries according to RAT (Figure 10a❶). For example, the recent access trace is (1111), so the last column of the prediction table is matched, which means the matched history-sample entries are found and can be used for prediction. The prediction result is decided according to the $State$ value of each entry (❷). After finishing computing the corresponding agent and getting the actual access trace for current data *tile*, the corresponding $State$ value of each entry will be updated (❸). Specifically, once the data *tile* is really used by the current sample, the $State$ value is increased by 1, otherwise decreased by 1. Concurrently, the actual access trace is viewed as *RAT* for the next iteration.

B. Implementation of Predictor

According to the prediction approach above, a lookup table is required in the predictor, as depicted in Figure 10. All the prefetch operations are pushed into the FIFO scheduler. Once the prefetch buffer is available, the FIFO scheduler dispatches the corresponding operation at the head of the queue. Concurrently, the address converter in the FIFO scheduler converts the corresponding *TileIDs* into actual addresses, to control the data access behavior in the prefetch buffer. The predictor also receives data from the runtime monitor to update the entries in the prediction table and handle misprediction errors. When a misprediction occurs, the predictor will directly control the prefetch buffer without using the FIFO scheduler. As shown in Figure 10, when the required data is not prefetched, `loading` instructions will be immediately issued to the prefetch buffer to obtain the corresponding data; when the accessed data is useless, the predictor will directly use the `disable` instruction to set that data to null. Since there is no need to roll back for misprediction, the overhead of misprediction is negligible (details in Section VII-D).

C. Case-study for Predictor-based Prefetching

We present an example of weight access pattern prediction for MoE model with 8 optional experts, as shown in Figure 11. The MoE selects experts for each input token, as illustrated in Figure 11a. Assume Input-1 has already finished executing on the dynamic branches network. Now we need to execute Input-2. First, for each token in Input-2, we determine its execution status for the first four experts, and obtain the corresponding recent access trace (*RAT*) based on the results. As we can notice, the *RAT* for the token ”I” is 0100 as only Expert 2 was activated in the previous execution. According to the acquired *RAT*, we query the prediction table and find we can prefetch the weight data with *TileID* 7, corresponding to the weights for Expert 7. Similarly, we will prefetch the weights for Expert

8 ahead of time for the token "Admire" to reduce the idleness of the computing resources. Compared to the conventional prefetching strategy, our method effectively reduces the unnecessary loading for weights in Expert 5 and 6. Similarly, the proposed prediction-based prefetching method can also be applied to other dynamic NNs such as BranchyNet [24] by stopping prefetching the weight parameters in the next layer. In detail, DNA can utilize the prediction strategy to decide whether to early-exit to prefetch weight parameters from the next layer or from the first layer.

VII. Evaluation

A. Experimental Setup

DNA Implementation. DNA is equipped with 16 cores, each contains an 8×8 PE array and an 8 KB buffer, which is divided into input, weight and output buffers. The L2 global buffer is set as 128 KB. Another 128 KB L3 buffer is adopted as the prefetch buffer. The transverter contains 16 items for each layer to buffer prime factors, and the predictor occupies an additional 1 KB buffer as the prediction table.

We implemented DNA design in RTL and synthesized it under the TSMC 45nm technology with Synopsys Design Compiler [36]. For system-level evaluation, we developed a cycle-accurate simulator based on STONNE [37], cross-verified with the RTL implementation. We extended the simulator to satisfy the many-core design and added our own NoC simulator. We also implemented the transverter and predictor for our DNA design. The DRAM access latency and power estimation are obtained from DRAMSim3 [38]. DNA runs dynamic NN models at a 500 MHz clock frequency. According to the synthesis report, DNA occupies $8.82\ mm^2$ area.

Baselines. For a fair comparison, we take the baseline hardware implementation with resources identical to the DNA architecture, but without any of our proposed optimizations. The baseline follows the setting in Figure 2a that employs the dynamic graph scheduling method with an agent-dependent data loading strategy. Besides, we separately optimize online scheduling (Section V) and predictor-based prefetching (Section VI) on baseline architecture to evaluate their corresponding improvement. These two cases are denoted as baseline with online scheduling and prefetching, respectively. An oracle design that has prophetic knowledge is regarded as a theoretical upper bound of hardware performance.

Specialized Accelerator. We also compare DNA with several specialized dynamic NN accelerators introduced in Section II-C. DCNA [26] is adopted to evaluate the performance of DCN and DPACS [28] is used to evaluate dynamic pruning networks. To make fair comparisons, we set all designs to have the same number of PEs (1024) and buffer size. All these accelerators are operated at 500 MHz. An additional 32 KB index buffer is allocated to support the optimization proposed by DCNA. The PE block in DPACS, previously implemented with an FPGA, was revised to an ASIC implementation for fair comparison. Additionally, the corresponding weight management unit is augmented to support the optimization.

GPU System with SOTA Compilation Frameworks. We compare DNA with state-of-the-art compilation frameworks, including Nimble [15] and DietCode [14], using the Jetson Xavier NX as our evaluation platform. This embedded system features a 6-core ARMv8.2 CPU and a 384-core Volta GPU with 48 Tensor Cores and a 6MB L2 cache. As a baseline, we adopt TensorRT's static compilation strategy. We also evaluate Nimble and DietCode as high-performance dynamic NN compilation frameworks. To assess our hardware-assisted online scheduler, we extend GPGPU-Sim 4.0 [39] to model the Jetson Xavier NX, denoted as the DNA-extended GPU. This system incurs only a 0.89% area overhead compared to the baseline.

TABLE III: Dynamic Neural Networks for Evaluation.

Network	Category	Backbone
DCN [3], [4] Token Merging [23] (TM)	Dynamic input regions	Mask R-CNN ViT
FBS [21] MoEfication [41] (MOEF)	Dynamic weight structures	VGG16 BERT
SwitchTransformer [10] (ST) Mixtral MoE [40]	Dynamic branches	Transformer LLM
CondConv [22]	Dynamic weight parameters	EfficientNet-B0

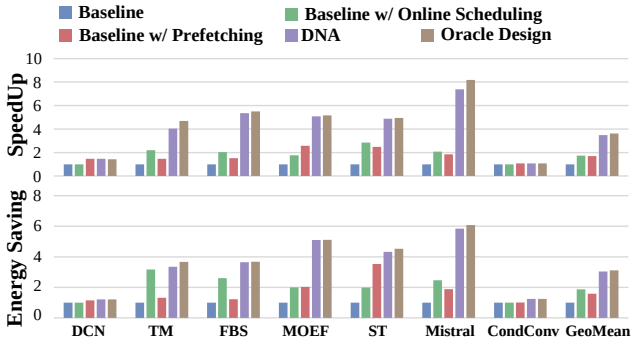


Fig. 12: Performance comparison of dynamic NNs.

Benchmarks. Table III describes the workload, including all categories of dynamic NNs in Figure 1. All these models employ classical neural networks as the backbone. Since the parameter size of SwitchTransformer [10] and Mixtral MoE [40] exceeds the capacity of DNA (edge device), we cannot directly evaluate them on our accelerator. Therefore, we scale down the parameter size of these models to 125M.

B. Performance

Comparison to Baselines. Figure 12 shows the speedup and energy saving delivered by DNA compared to the baseline designs. As one can notice, utilizing online scheduling or predictor-based prefetching alone can also achieve $1.73\times$ and $1.70\times$ speedup, $1.87\times$ and $1.58\times$ energy savings, respectively. The benefits of online scheduling are mainly from improved PE utilization. However, it can only accelerate networks with dynamic operator structures rather than models with dynamic parameters (e.g. CondConv [22]). On the other hand, although predictor-based prefetching introduces additional data movement from mispredictions, it reduces hardware static energy by decreasing idleness. On average, DNA achieves $3.48\times$ speedup and $3.03\times$ energy saving over baseline and reaches 96% improvement of the oracle design, revealing its efficiency.

Comparison with Specialized Accelerator. Figure 13 presents the performance between various specialized accelerators and DNA. DCNA adopts dependency table tracking (with the help of index buffer) to reduce useless data access

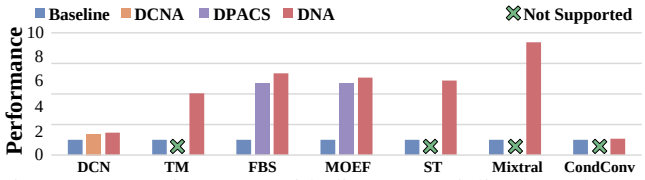


Fig. 13: Comparing DNA with diverse specialized accelerators.

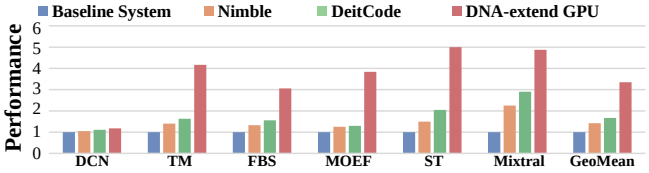


Fig. 14: Comparison with SOTA compilation frameworks.

for DCNs. However, it uses a monolithic systolic array design, which hinders the benefits of inter-core communication in Section IV-A. As a result, DNA achieves $1.05\times$ speedup over DCNA on DCN. In contrast to the precise memory access of DCNA, the mispredictions in DNA caused a slight increase in off-chip data movement ($1.01\times$). DPACS is designed to accelerate their self-optimized dynamic pruning networks with limited structures but is not suited for all cases of dynamic pruning networks, so it can only reach the 87.9% performance of DNA. Since DPACS and DNA adopt identical compute and memory configurations, the power consumption during execution is essentially equivalent between the two architectures. Consequently, DNA achieves $1.1\times$ energy saving compared to DPACS. Overall, DNA can even achieve better performance than specialized accelerators without sacrificing generality.

C. Comparison with SOTA Compilation Framework

Figure 14 shows the performance improvement achieved by adopting advanced compilation frameworks. As one can notice, when optimizing dynamic NNs with SOTA compilation frameworks, there is a notable improvement in the overall execution efficiency. Compared to the baseline system, utilizing Nimble and DeitCode yields up to $3.34\times$ and $2.96\times$ ($2.35\times$ and $2.01\times$ on average) performance speedup, respectively. However, as analyzed, software-based compilation framework optimizations cannot explore scheduling optimizations for each dynamic operator but can only explore feasible optimization spaces based on sampling strategies. In contrast to this, the hardware extension of the GPU utilizing the scheduling optimization strategies proposed in this paper (DNA-extended GPU) achieves up to $4.99\times$ ($3.35\times$ on average) performance speedup, compared to the baseline system.

D. Breakdown Analysis

Energy Breakdown. Figure 15a shows the energy breakdown when DNA runs various dynamic NNs. It can be noticed that both the transverter and predictor induce negligible energy cost. Furthermore, how the energy is reduced also depends on the characteristics of dynamic NNs. For example, FBS with high prediction accuracy (95.2%) consumes more energy on the prefetch buffer than GLB. We can also notice that the memory-bounded transformer models (MOEF, ST and Mixtral) spends more energy on data loading. Despite memory

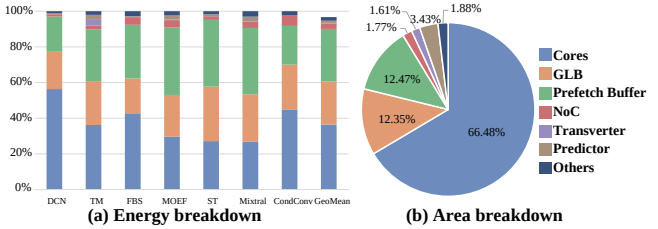


Fig. 15: The energy and area breakdown of DNA.

data movement accounting for 53.6% of total energy consumption, the overall cost of misprediction is limited to 2.02% on average. This is because DNA can achieve high prediction accuracy and not all mispredictions incur data movement overheads (only when useless data are prefetched).

Area Breakdown. The area overhead of DNA is increased by 5.0% compared to baseline architecture. The area breakdown of DNA is presented in Figure 15b. From our observation, the computing engine and on-chip memory, including prefetch buffer and GLB occupy most of the silicon area. Our proposed transverter and predictor only take 1.61% and 3.43% of total area, respectively. However, since the transverter and predictor are independent of the number of cores in DNA, the additional area overhead compared to the baseline decreases as the number of cores increases.

E. Evaluation on Transverter-based Online Scheduling

To validate the efficiency of transverter-based online scheduling framework (Section V), we evaluate the performance on two typical backbone networks VGG16 and ResNet50 [11], [12], combining the mechanisms of dynamic pruning and dynamic scaling [42], [43] methods on every operator. Furthermore, we adopt a fabricated genuine computing architecture, simba [44], which is quite different from DNA, to prove the universality of our online scheduling framework.

Figure 16 highlights the efficiency of our transverter-based online scheduling. Static scheduling (Figure 4a) targets only the backbone operator, while dynamic graph scheduling (Figure 4b), following the approach in [29], selects dynamic operator schedules using a predefined template library. Oracle scheduling, in contrast, assumes perfect foresight for each dynamic operator. Our method achieves 2.83× speedup and 3.03× energy savings (GeoMean), reaching 93% of oracle efficiency. While dynamic graph scheduling occasionally outperforms our approach in specific cases (e.g., Conv_4_2 in VGG), its limited kernel coverage restricts its overall effectiveness—achieving only 59.6% of our method's performance gain.

F. Evaluation on Predictor-based Prefetching

Compared to the conventional prefetching method [32], [45], our predictor-based prefetching can effectively reduce the unnecessary memory access overhead. Figure 17 presents the percentage reduction in off-chip memory accesses compared to the baseline and the corresponding prediction accuracy of various dynamic NNs. In general, the DNA accelerator with the predictor achieves $2.09\times$ main memory access reduction over the baseline on average. For dynamic input region networks (DCN, TM), the useless access for input features is greatly

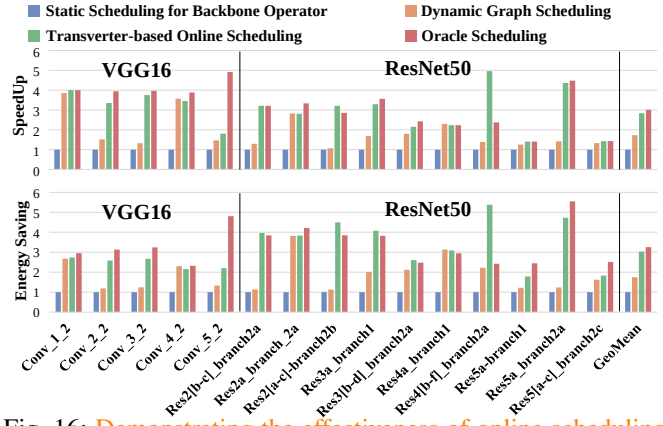


Fig. 16: Demonstrating the effectiveness of online scheduling.

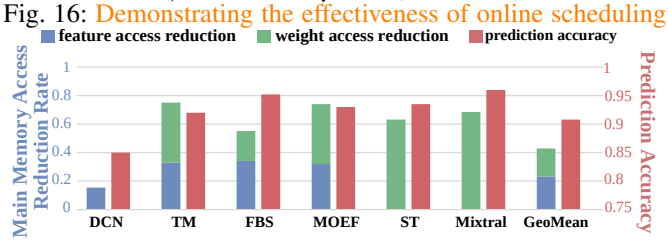


Fig. 17: The reduced main memory access over conventional prefetching and prediction accuracy for various dynamic NNs.

reduced. In this situation, DNA effectively reduces redundant off-chip access for the same data by predicting the reusability of each data tile. For dynamic weight branch networks (ST, Mixtral), the memory access prediction method contributes to the reduction of unnecessary data access for weight data. For dynamic weight structure networks (FBS, MOEF), some weight data and their corresponding input feature data can be reduced together, as they are highly related. For one of the dynamic parameter networks, CondConv, the memory access patterns are 100% predictable due to their fixed data flow and thus ignored in the figure. Overall, our memory access predictor achieves 92.5% accuracy on average for various dynamic networks and effectively reduces energy consumption and execution latency.

VIII. Conclusion

Existing NPUs cannot efficiently execute dynamic NNs. The main challenges come from the dynamism of operators and the data loading dependencies on agent computing. Consequently, we propose a versatile architectural design, DNA, to support the efficient execution of dynamic NNs. DNA integrates a transverter-based online scheduling mechanism to optimize the execution efficiency of dynamic operators, and adopts predictor-based prefetching strategies to optimize the overhead of memory access during model execution. Extensive evaluation shows that the proposed DNA accelerator achieves significant performance and energy efficiency boost.

References

[1] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang, "Dynamic neural networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 11, pp. 7436–7456, 2021.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[3] J. Dai, H. Qi, Y. Xiong, Y. Li, G. Zhang, H. Hu, and Y. Wei, "Deformable convolutional networks," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 764–773.

[4] X. Zhu, H. Hu, S. Lin, and J. Dai, "Deformable convnets v2: More deformable, better results," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 9308–9316.

[5] A. Bhattacharjee, Y. Venkatesha, A. Moitra, and P. Panda, "Mime: Adapting a single neural network for multi-task inference with memory-efficient dynamic pruning," *arXiv preprint arXiv:2204.05274*, 2022.

[6] S. Cai, Y. Shu, and W. Wang, "Dynamic routing networks," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2021, pp. 3588–3597.

[7] M. Elbayad, J. Gu, E. Grave, and M. Auli, "Depth-adaptive transformer," in *ICLR 2020-Eighth International Conference on Learning Representations*, 2020, pp. 1–14.

[8] Y. LeCun, "1.1 deep learning hardware: Past, present, and future," in *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2019, pp. 12–19.

[9] OpenAI, "Gpt-4 technical report," 2023.

[10] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *The Journal of Machine Learning Research*, vol. 23, no. 1, pp. 5232–5270, 2022.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[13] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[14] B. Zheng, Z. Jiang, C. H. Yu, H. Shen, J. Fromm, Y. Liu, Y. Wang, L. Ceze, T. Chen, and G. Pekhimenko, "Dietcode: Automatic optimization for dynamic tensor programs," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 848–863, 2022.

[15] H. Shen, J. Roesch, Z. Chen, W. Chen, Y. Wu, M. Li, V. Sharma, Z. Tatlock, and Y. Wang, "Nimble: Efficiently compiling dynamic neural networks for model inference," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 208–222, 2021.

[16] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, "Mind mappings: enabling efficient algorithm-accelerator mapping space search," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 943–958.

[17] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367–379, 2016.

[18] S.-C. Kao and T. Krishna, "Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.

[19] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.

[20] Coral.ai, "Edge tpu performance benchmarks," 2021, https://coral.ai/docs/edgetpu/benchmarks/,.

[21] X. Gao, Y. Zhao, Ł. Dudziak, R. Mullins, and C.-z. Xu, "Dynamic channel pruning: Feature boosting and suppression," in *International Conference on Learning Representations*.

[22] B. Yang, G. Bender, Q. V. Le, and J. Ngiam, "Condconv: conditionally parameterized convolutions for efficient inference," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 2019, pp. 1307–1318.

[23] J. Chen, X. Wang, Z. Guo, X. Zhang, and J. Sun, "Dynamic region-aware convolution," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 8064–8073.

[24] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *2016 23rd international conference on pattern recognition (ICPR)*. IEEE, 2016, pp. 2464–2469.

[25] Z. Song, B. Fu, F. Wu, Z. Jiang, L. Jiang, N. Jing, and X. Liang, "Drq: dynamic region-based quantization for deep neural network ac-

celeration," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 1010–1021.
[26] D. Xu, C. Chu, C. Liu, Y. Wang, H. Li, X. Li, and K.-T. Cheng, "Energy-efficient accelerator design for deformable convolution networks," *arXiv preprint arXiv:2107.02547*, 2021.
[27] Q. Huang, D. Wang, Z. Dong, Y. Gao, Y. Cai, T. Li, B. Wu, K. Keutzer, and J. Wawrzynek, "Codenet: Efficient deployment of input-adaptive object detection on embedded fpgas," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 206–216.
[28] Y. Gao, B. Zhang, X. Qi, and H. K.-H. So, "Dpacs: Hardware accelerated dynamic neural network pruning through algorithm-architecture co-design," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 237–251.
[29] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.
[30] A. Azizimazreah and L. Chen, "Shortcut mining: Exploiting cross-layer shortcut reuse in dcnn accelerators," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 94–105.
[31] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.
[32] X. Wang, B. Zhao, R. Hou, A. Awad, Z. Tian, and D. Meng, "Nasguard: a novel accelerator architecture for robust neural architecture search (nas) networks," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 776–789.
[33] X. Wang, F. Yu, L. Dunlap, Y.-A. Ma, R. Wang, A. Mirhoseini, T. Darrell, and J. E. Gonzalez, "Deep mixture of experts via shallow embedding," in *Uncertainty in Artificial Intelligence*. PMLR, 2020, pp. 552–562.
[34] Q. Huang, A. Kalaiah, M. Kang, J. Demmel, G. Dinh, J. Wawrzynek, T. Norell, and Y. S. Shao, "Cosa: Scheduling by constrained optimization for spatial accelerators," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 554–566.
[35] F. Li, G. Li, X. He, and J. Cheng, "Dynamic dual gating neural networks," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 5330–5339.
[36] Synopsys, "Design compiler," http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages.
[37] F. Muñoz-Martínez, J. L. Abellán, M. E. Acacio, and T. Krishna, "Stonne: Enabling cycle-level microarchitectural simulation for dnn inference accelerators," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2021, pp. 201–213.
[38] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "Dramsim3: a cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.
[39] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE international symposium on performance analysis of systems and software*. IEEE, 2009, pp. 163–174.
[40] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.
[41] Z. Zhang, Y. Lin, Z. Liu, P. Li, M. Sun, and J. Zhou, "Moefication: Transformer feed-forward layers are mixtures of experts," in *Findings of the Association for Computational Linguistics: ACL 2022*, 2022, pp. 877–890.
[42] W. Hua, Y. Zhou, C. De Sa, Z. Zhang, and G. E. Suh, "Channel gating neural networks," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 2019, pp. 1886–1896.
[43] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Weinberger, "Multi-scale dense networks for resource efficient image classification," in *International Conference on Learning Representations*, 2018.
[44] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina *et al.*, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 14–27.
[45] X. Wei, Y. Liang, and J. Cong, "Overcoming data transfer bottlenecks in fpga-based dnn accelerators via layer conscious memory management," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

Lian Liu received the BS degree in computer science and technology from Nankai University, Tianjin, China, in 2021. He is working toward a PhD in computer science with the State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include computer architecture, hardware-software codesign and processing in memory.

Jinxin Yu received the BS degree in Measurement and Control Technology and Instruments from University of Science and Technology Beijing, China, in 2021. He is currently pursuing his PhD degree in computer science with the State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include computer architecture, hardware-software codesign and LLM system optimisation.

Mengdi Wang (Member, IEEE) received the BE degree in computer science from Huazhong University, Wuhan, China, in 2017, and the Ph.D. degree in computer science from the Institute of Computing Technolog (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2023. She is currently an assistant researcher with ICT, CAS. Her research interests include computer architecture and VLSI.

Xiaowei Li (Senior Member, IEEE) (Senior Member, IEEE) received the B.Eng. and M.Eng. degrees in computer science from the Hefei University of Technology, Hefei, China, in 1985 and 1988, respectively, and the Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 1991. In 2000, he joined ICT, CAS, as a Professor. He is also a Professor with the Department of Computer Science, University of Chinese Academy of Sciences, Beijing. His current research interests include VLSI testing, design verification, and dependable computing.

Yinhe Han (Member, IEEE) received the MS and PhD degrees in computer science from ICT, CAS, in 2003 and 2006, respectively. He is currently a professor with ICT, CAS, His research interests include VLSI/SOC interconnection, testing and fault-tolerance. He is a recipient of the Best Paper Award at Asian Test Symposium 2003. He is a member of CCF, ACM, IEEE society. He was the program co-chair of Workshop of RTL and High Level Testing (WRTLT) in 2009, and serves on the Technical Program Committees of several IEEE and ACM conferences, including ATS, GVLSI, etc.

Ying Wang (Member, IEEE) is currently a Professor with the State Key Laboratory of Computer Architecture, Institute of Computing Technology (ICT), Chinese Academy of Sciences, Beijing, China. From 2011 to 2013, he worked with SAFARI Group, Carnegie Mellon University (CMU), Pittsburgh, PA, USA, as a Visiting Researcher. At ICT and CMU, his research interests primarily focus on reliable computer architecture and VLSI design, with an emphasis on memory systems, energy-efficient accelerators, and approximate/error-tolerant computing.