

DynaStitch: Dynamic ViT Inference via Neural Stitching for Resource-Constrained Edge Devices

Jiwon Kim, Luca Mottola, Jeho Lee, Seonghoon Park, Hojung Cha, and Thiemo Voigt

Abstract—Neural stitching enables Vision Transformers (ViTs) to operate at dynamic computational costs, providing adaptive inference that is highly promising for deployment on resource-constrained edge devices. However, neural stitching-based ViTs inherently exhibit high memory usage and lack resource-efficient inference capabilities, limiting practical deployment. We present DynaStitch, an inference system that enables flexible deployment and resource-efficient operation of neural stitching-based ViTs on varying edge device resources. To reduce memory usage, we propose dynamically constructing a partial model within memory constraints by selectively removing blocks. To find an optimal partial model among numerous candidates at runtime, we formulate an optimization problem and significantly reduce the search space using two pruning conditions. We then solve the problem using a dynamic programming algorithm. For resource-efficient inference, DynaStitch controls computational cost of the partial model to meet the target latency and updates the partial model and processor frequency to maximize inference accuracy and energy efficiency. In our evaluation, DynaStitch maximizes inference performance under memory constraints and target latencies, delivering up to 44.7% memory reduction and 19.5% average power reduction while maintaining comparable accuracy against state-of-the-art neural stitching-based ViTs.

Index Terms—Vision transformer, on-device inference, neural stitching, resource-aware systems.

I. INTRODUCTION

VISION Transformers (ViTs) have emerged as state-of-the-art vision backbones, demonstrating superior generalization capabilities [1]. Deploying ViTs on edge devices is essential to support high-quality vision processing [2], [3]. Modern edge applications, such as autonomous robots, are increasingly required to execute vision processing concurrently with other tasks, such as user interaction [4]. Consequently, the computational resources available for vision tasks are not only limited but also fluctuating. In such dynamic environments, a model offering a diverse range of computational cost options is essential to adaptively maximize inference accuracy within the varying resource budget.

Flexible ViT architectures that offer diverse computational cost options have recently been proposed, enabling adaptive inference at runtime. One approach is early-exit ViTs [5], [6], which incorporate early exit mechanisms into ViTs. These

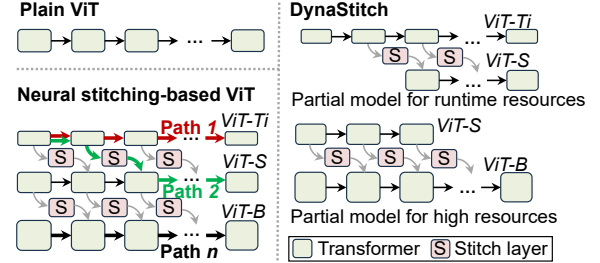


Fig. 1. Existing ViTs and DynaStitch. DynaStitch configures partial models adapted to the available runtime resources.

models add early exits between transformer blocks and operate with flexible computational costs. However, this approach suffers from accuracy degradation as more exits are added [7], limiting its flexibility. On the other hand, slimmable ViTs [8] adjust the embedding dimensions of transformer blocks to scale computational cost dynamically. However, increasing the scaling granularity leads to accuracy loss and heavy training efforts, imposing constraints on flexibility. Neural stitching-based ViTs [9], [10] provide an alternative approach that enables highly flexible computational costs with minimal accuracy loss. By stitching multiple well-pretrained ViTs of different scales within a model family, they support tens of execution paths with different costs, achieving high flexibility without requiring extensive retraining (see Fig. 1). This architecture aligns well with the critical need for diverse computational cost options in dynamic edge environments.

While neural stitching-based ViTs enable fine-grained computational cost control over dynamic resources, their practical on-device deployment requires an inference system that addresses the following key challenges:

- 1) *Meeting memory constraints*: neural stitching-based ViTs comprise multiple ViTs, significantly increasing the number of model parameters and leading to high memory overhead. On edge devices, limited memory is shared among multiple applications. For instance, as shown in Fig. 2, an autonomous robot typically relies on a vision backbone for navigation. However, it may eventually need to support user interaction using memory-intensive language models [4]. Even lightweight language workloads consume GB-level memory (see Table I), which may reduce the memory available for vision inference. The vision model must adapt to the reduced memory budget. Unfortunately, existing neural stitching-based ViTs do not offer this crucial adaptation.
- 2) *Lack of an effective inference strategy*: balancing the trade-offs between accuracy and energy consumption is essential to meet diverse application needs. While a higher

Jiwon Kim is with Ewha Womans University, Seoul, Republic of Korea (e-mail: jiwon_kim@ewha.ac.kr; corresponding author).

Luca Mottola is with Politecnico di Milano, Milan, Italy, RISE Research Institutes of Sweden, Sweden, and Uppsala University, Uppsala, Sweden (e-mail: luca.mottola@polimi.it).

Jeho Lee, Seonghoon Park, and Hojung Cha are with Yonsei University, Seoul, Republic of Korea (e-mail: jeholee@yonsei.ac.kr; seonghoon.park@yonsei.ac.kr; hjcha@yonsei.ac.kr).

Thiemo Voigt is with the Department of Electrical Engineering, Uppsala University, Uppsala, Sweden, and RISE Research Institutes of Sweden, Sweden (e-mail: thiemo.voigt@angstrom.uu.se).

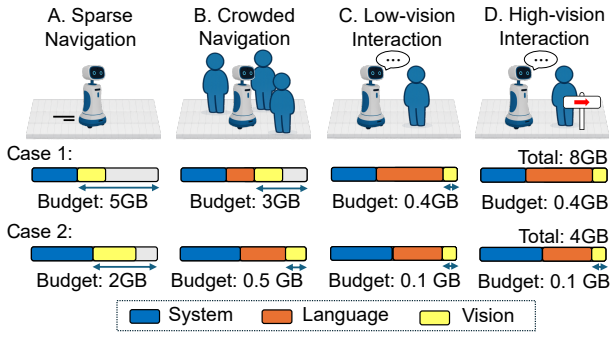


Fig. 2. Robot scenario with phase-dependent vision demand and memory budget under co-running language workloads.

computational cost improves accuracy, it also increases energy consumption. The desirable balance depends on the situation. For example, as shown in Fig. 2, a robot navigating a sparse area can prioritize energy efficiency to extend operation time, whereas navigation in a crowded area requires higher accuracy for pedestrian safety. Existing neural stitching-based ViTs, however, are not able to balance between accuracy and energy efficiency.

We propose DynaStitch, an inference system based on neural stitching-based ViTs that addresses these challenges and adapts its inference strategy dynamically to changing runtime resources. DynaStitch is rooted in the following key techniques. (1) On-Demand Model Assembly: We construct a partial model by selecting specific blocks of a neural stitching-based ViT to meet memory constraints, as shown in Fig. 1. However, identifying the optimal configuration presents a significant challenge due to the massive search space where the number of potential partial models scales exponentially. To address this, we propose an optimization method that quickly finds and configures the optimal partial model, aligned with the available computational resources.

(2) Computation Control and Configuration Update: We propose an inference strategy that adapts to runtime resources and optimizes resource efficiency. We control the computational cost (i.e., execution path) of the partial model to meet the target latency. Upon significant shifts in resource or target latency, we adaptively update the partial model and processor frequency. This optimizes inference accuracy and energy efficiency based on two operational modes: *accuracy-first* and *energy-first*, which prioritize either accuracy or energy efficiency according to the application's needs.

We extensively evaluate DynaStitch against existing works and across three edge devices—Jetson AGX Orin 64GB, AGX Xavier, and Orin Nano [11] (hereafter referred to as Orin, Xavier, and Nano, respectively)—that differ in computational resources. The results show that DynaStitch maximizes performance while meeting various runtime constraints; our partial model and inference strategy reduce memory usage by 44.7% and average power consumption by 19.5%, respectively, while maintaining comparable accuracy under memory budgets.

TABLE I
STANDALONE MEMORY USAGE OF ON-DEVICE LLM WORKLOADS
MEASURED ON AGX XAVIER WITH OLLAMA [14].

Model	Alive Mem.	Engine Mem.
Qwen3-0.6B	1.9 GB	1.4 GB
Qwen3-1.7B	2.8 GB	1.9 GB
Qwen3-4B	4.6 GB	1.9 GB
Qwen3-8B	7.0 GB	1.9 GB

II. BACKGROUND AND RELATED WORK

Since the advent of ViTs in 2020 [1], variants, such as DeiT [12] and Swin Transformer [13], have successfully established Transformer architectures as state-of-the-art for vision tasks by addressing the core problems of the original ViT, including data inefficiency and lack of inductive bias. However, ViTs are inherently parameter-heavy and computationally intensive. Due to these high resource demands, achieving peak performance of ViTs on edge devices necessitates utilizing the runtime computational resources to their fullest potential. Ideally, this requires executing a ViT with the maximum accuracy while meeting the target latency, despite resource constraints and fluctuating available resources at runtime.

ViTs customized for resource constraints. Several techniques exist to generate ViTs customized for specific resource constraints. One prominent direction is the supernet-based approach, which enables flexible model extraction by embedding multiple architectures within a single network. AutoFormer [15] constructs a supernet, from which various subnets of different sizes are derived. Similarly, MatFormer [16] builds a block-level supernet, generating and combining transformer blocks of varying sizes to create a ViT. However, training the supernets requires significant computational resources. More recently, learngene-based approaches have emerged [17], [18], [19]. These methods extract structured components (i.e., learngenes) from a well-trained model and use them to initialize the target model, alleviating the heavy training overhead of supernets. Overall, while these approaches successfully produce ViTs suited for edge devices, they yield a ViT with a fixed computational cost, resulting in an inability to adapt to fluctuating resources or target latencies at runtime.

ViTs with dynamic computational costs. ViT architectures capable of operating with dynamic computational costs have been proposed [5], [6], [8], [9], [10], [20]. One approach applies an early exit mechanism [21] to ViTs, where exit heads are inserted between transformer blocks to enable early inference termination [5], [6]. However, increasing the number of exits for high control granularity leads to significant training efforts due to complex loss functions that balance multiple exit losses, resulting in overall model performance degradation. Another method, slimmable ViTs [8], adjusts embedding dimensions within transformer blocks, but also suffers from accuracy losses as the scaling granularity increases. Similarly, Vision Mixture of Experts (V-MoE) [20] dynamically adjusts the number of feedforward layers. However, the use of a large number of experts to achieve high flexibility often results in unstable training, and the model size increases linearly with the number of experts.

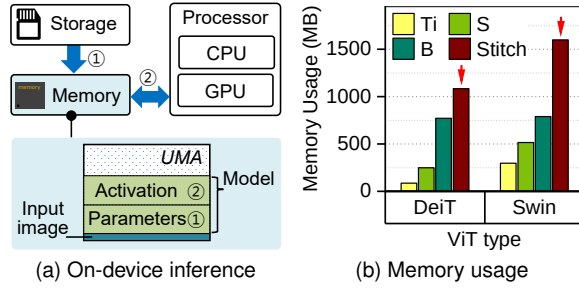


Fig. 3. Neural stitching-based ViTs exhibit increased memory usage compared to single ViTs.

Neural stitching-based ViTs. Neural stitching approaches [9], [10] enable dynamic inference paths by interconnecting different pre-trained architectures. Recently, this paradigm has been extended to various domains, including sampling acceleration in diffusion models [22] and efficient reasoning in LLMs [23]. ViTs based on neural stitching [9], [10] effectively address the limitations observed in prior flexible designs, such as limited flexibility, high training complexity, and a linearly increasing model size.

Neural stitching-based ViTs offer superior flexibility with tens of execution paths by inserting stitch layers between blocks within pretrained models of different scales, as shown in Fig. 1. Based on the observed similarity between feature maps [24], this approach connects blocks from ViTs of different scales within a model family using stitch layers, which are implemented using 1×1 Conv, creating multiple paths for execution. The computational cost varies depending on the selected path. For example, in Fig. 1, the computational cost of Path 1 is lower than that of Path 2 and significantly lower than that of Path n , as the blocks in ViT-Ti are lighter than those in ViT-S, which in turn are lighter than those in ViT-B. By leveraging well-pretrained ViTs, this method achieves high computational flexibility with minimal training effort and accuracy loss. By enabling fine-grained computational control, this approach can adaptively operate across a wide range of dynamic computational resources. This superior flexibility positions neural stitching-based ViTs as a promising candidate for adapting inference to the varying computational resources of edge devices.

III. MOTIVATION

We analyze the resource usage of neural stitching-based ViTs and identify optimization opportunities for on-device deployment. Unless otherwise noted, results are obtained on Xavier using the ONNX implementation in Section VII.

A. Resource Usage Analysis

Memory resource usage. Fig. 3(a) illustrates the memory usage of a deep learning model during on-device inference on edge devices. The model is first loaded from storage into memory to initiate inference (①). Then, processors access the memory to perform inference tasks (②). In ①, the model parameters, e.g., weights and bias, are loaded into memory. In

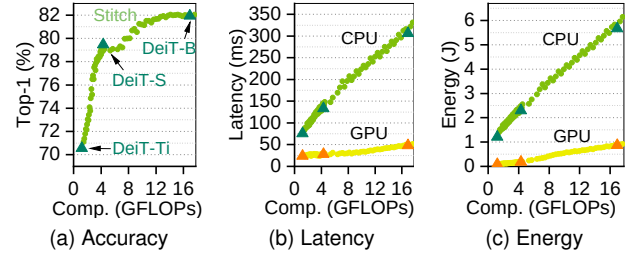


Fig. 4. Impact of computational cost on accuracy, latency, and energy, illustrating the trade-off between inference accuracy and energy consumption.

②, activations, such as intermediate values required for calculations during inference, are allocated. Unlike during training, where backpropagation requires storing intermediate activations for gradient computation, inference generally requires less activation memory. Hence, reducing model parameters is crucial for lowering memory usage, particularly in memory-constrained environments.

However, neural stitching-based ViTs contain significantly more parameters than a single ViT because they are constructed by stitching multiple ViTs (see Fig. 1). Fig. 3(b) shows the memory usage of three different scales of models (Ti, S, and B) within two types of ViTs, DeiT and Swin Transformers, and the neural stitching-based model (Stitch) constructed from these (the experimental setup is described in Section 7.1). Ti, S, and B refer to DeiT-Ti, DeiT-S, and DeiT-B for DeiT [12], and Swin-Ti, Swin-S, and Swin-B, for Swin [13], respectively. The results show that the neural stitching-based models consume significantly more memory compared to single models. At runtime, only the parameters of the selected path in the stitched model are activated, while many other parameters remain unused but still occupy memory. On edge devices, memory resources are limited. Failing to accommodate the model within the available memory budget renders it unusable. The inefficient memory usage hinders the deployment of neural stitching-based ViTs.

Computation and energy usage. The computational costs of neural stitching-based ViTs, that can be controlled at runtime, are closely related to accuracy, latency, and energy consumption. Fig. 4(a) shows that increasing the computational cost leads to higher accuracy, as the model can extract more complex features. However, in Fig. 4(b), higher computational costs also lead to higher latency, consequently resulting in increased energy consumption (Fig. 4(c)). These figures illustrate the inherent trade-off: higher computational costs are essential for improving accuracy, whereas lower costs are necessary for energy savings. This emphasizes the need for an inference strategy that manages computational costs considering the trade-off based on application-specific priorities.

B. Opportunities

Memory usage. Neural stitching-based ViTs contain significantly more transformer blocks than single ViTs. Here, we identify opportunities for memory reduction by selectively removing blocks from the stitched model. Fig. 5(a) shows the original stitched model and an example where blocks are removed. We assume that ViT-Ti, ViT-S, and ViT-B each

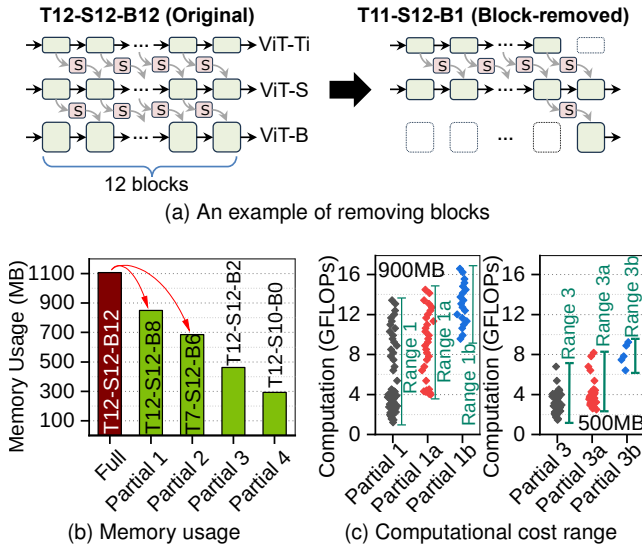


Fig. 5. Memory usage reduction and computation control opportunities. Removing blocks reduces memory usage, and the computational costs depend on which blocks are removed.

consist of twelve transformer blocks, denoted as T12-S12-B12 for the original model. In the example, T11-S12-B1 represents a configuration where one block is removed from ViT-Ti and eleven blocks from ViT-B. The block-removed model has fewer blocks and reduced execution paths compared to the original model, while still offering flexible computational costs with multiple paths. Fig. 5(b) presents the memory usage of the original model (Full) and four different block-removed models (Partial 1, Partial 2, Partial 3, and Partial 4), constructed using DeiT-Ti, DeiT-S, and DeiT-B. As shown in the graph, the block-removed models consume less memory than the original model, with variations in memory usage depending on their configuration.

Computational costs. While block-removed models exhibit reduced memory usage compared to the original model, they have fewer paths. Interestingly, even when different block-removed models have similar memory usage, the computational costs of their available paths are distributed across distinct ranges depending on the block configuration. Fig. 5(c) shows the computational costs of six different partial models, where each dot represents the computational cost of a path. For example, the Partial 1 offers multiple cost options shown as black dots within Range 1. The left figure shows three models configured under a 900 MB memory constraint. The memory usages of Partial 1 (T12-S12-B8), Partial 1a (T4-S12-B9), and Partial 1b (T0-S9-B11) are 849.33 MB, 856.42 MB, and 874.58 MB, respectively. Although these three models consume similar memory resources, they have different configurations and computational cost ranges, as shown in the graph (Range 1, Range 1a, and Range 1b). By selectively utilizing Partial 1 and Partial 1b, we can cover the full range of computational costs supported by the full model while adhering to the memory budget. Similarly, Partial 3 (T12-S12-B2), Partial 3a (T8-S12-B3), and Partial 3b (T0-S11-B4) are configured under a 500 MB memory constraint, consuming 462.83 MB, 498.58 MB, and 485 MB, respectively. They are

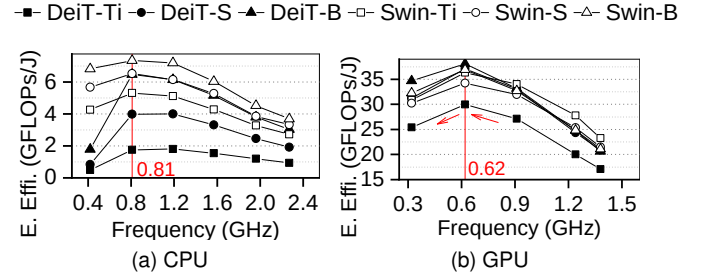


Fig. 6. Energy efficiency across different processor frequencies. Efficiency drops beyond a threshold (red line) due to increased latency.

configured with the same memory constraint but show varying ranges of computational costs.

Energy consumption. To identify opportunities for energy consumption reduction, we measure the energy efficiency of ViTs during inference at various processor frequencies, as shown in Fig. 6. The results indicate that energy efficiency is highly sensitive to processor frequency. Notably, energy efficiency generally improves as the frequency decreases. This is because edge devices operate based on DVFS (Dynamic Voltage and Frequency Scaling); lowering the frequency allows for a reduction in operating voltage, thereby significantly reducing power consumption. However, the results reveal a critical threshold (red line) below which efficiency degrades, as the increased inference latency outweighs the power savings. This experiment indicates that strategically controlling the processor frequency while meeting the target latency is effective for improving energy efficiency.

C. Take-aways

First, our analysis shows that neural stitching-based ViTs exhibit substantial memory overhead, while selectively removing blocks enables adaptation to memory constraints. Crucially, even among partial models with similar memory usage, the supported computational cost ranges vary significantly, indicating an *optimal configuration* that satisfies the memory constraint while aligning its computational cost range with the available runtime resources.

Second, our experiments show that lower computational cost reduces energy consumption, while processor frequency control further improves energy efficiency. This highlights a key energy reduction opportunity: adjusting the processor frequency within the target latency while remaining above a critical efficiency threshold, rather than increasing computational cost, minimizes energy consumption.

IV. DYNASTITCH

Motivated by our insights, we design DynaStitch, a system that enables the effective deployment of neural stitching-based ViTs on resource-constrained edge devices. The goal of the system is to maximize inference accuracy and energy efficiency, while adaptively meeting runtime memory constraints and the target latency. To this end, the system (I) generates a model tailored to runtime memory constraints, (II) executes it within the target latency, and (III) updates the model configuration and processor frequency to maximize accuracy

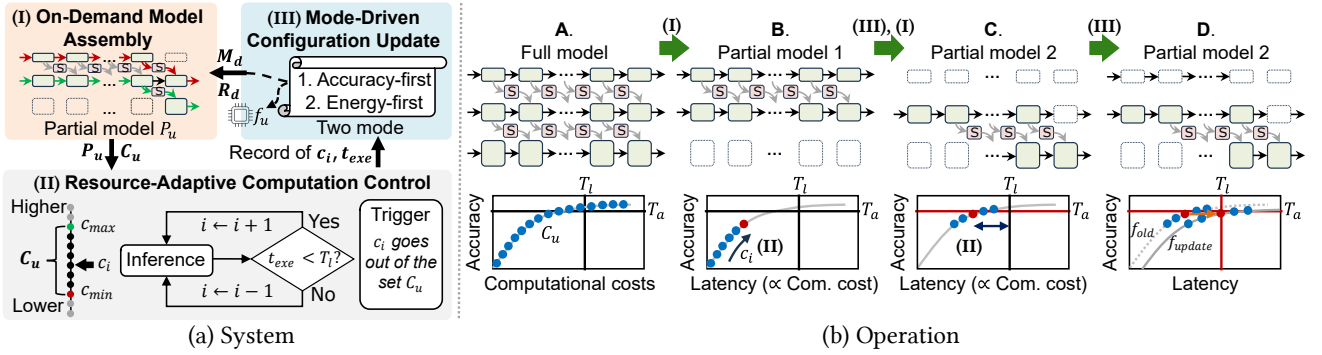


Fig. 7. DynaStitch system and operation. (a) The system comprises three components. (b) A: Initial full model. B: Partial model 1 loaded under M_d ; its cost options (dots) fail to reach target accuracy (T_a). C: Updating to Partial model 2 provides higher cost options, meeting T_a while leaving latency headroom for target latency (T_l). D: Lowering frequency ($f_{old} \rightarrow f_{update}$) trades latency slack for energy efficiency, satisfying both T_a and T_l .

or energy efficiency according to application needs. As shown in Fig. 7(a), our system consists of three components, each corresponding to a specific goal:

(I) On-demand model assembly. In this component, the system provides an optimal neural stitching-based ViT tailored to the runtime memory constraints and computational resources. We configure a model by selectively removing the blocks of neural stitching-based ViTs at runtime. We define the configured model as *partial model* and the original model as *full model*. As shown in Fig. 7(a), the partial model (P_u) is configured based on the given runtime memory constraints (M_d) and required computational cost range (R_d). Here, we propose a technique to quickly find the optimal partial model that maximizes the number of execution paths (computational cost options) at runtime. To this end, we reduce the state space by imposing two conditions and solve a combinatorial optimization problem. The configured partial model P_u can operate with different computational cost and we denote the set of these costs as C_u . Note that since we assume the partial model has at least one path, the smallest partial model corresponds to the smallest ViT in the full model (e.g., ViT-Ti in Fig. 7).

(II) Resource-adaptive computation control. The computational cost of the partial model P_u is controlled by selecting one element c_i from the cost set C_u to meet the target latency. A higher index i corresponds to a higher computational cost. If the inference latency (t_{exe}) is lower than the target latency (T_c), the system increases the computational cost to improve inference accuracy until the target accuracy T_a ; otherwise, we decrease it to ensure the latency remains within the target. The cases where the selected cost c_i reaches the minimum (c_{min}) or the maximum (c_{max}) in the cost set C_u mean that the partial model is not suitable for current computational resources. In these cases, the system requests a configuration update—such as a partial model update and processor frequency control.

(III) Mode-driven configuration update. To optimize accuracy and energy efficiency under runtime computational resources, this component updates the computational-cost range R_d for partial model reconfiguration or controls the processor frequency (f_u). In Section III, we observed the trade-off between accuracy and energy consumption, and how processor frequency affects energy efficiency. Based on these observations, we define two modes, *accuracy-first* and *energy-*

first, which determine the priority order for updating R_d and f_u . The On-Demand Model Assembly component reconfigures the partial model only when R_d is updated.

Operation. Fig. 7(b) shows model and resource optimization of DynaStitch in accuracy-first mode. Stage A: Initially, the full model containing all computational costs (blue dots) is stored in storage. Stage B: Under the runtime memory constraint, the On-demand Model Assembly constructs and loads Partial Model 1 into the memory. However, as depicted in the accuracy-latency graph, the maximum cost of this partial model (red dot) falls short of the target accuracy (T_a), necessitating a model update. Stage C: The system triggers the assembly of Partial Model 2 by replacing blocks with those supporting higher computational costs. While this model successfully meets T_a , the inference latency is still significantly lower than the target latency (T_l). This large margin indicates that the processor is running at an unnecessarily high frequency, wasting energy. Stage D: To eliminate this inefficiency, Mode-Driven Configuration Update lowers the frequency f_u from f_{old} to f_{update} , shifting the latency curve to the right. Consequently, the operational point aligns with T_l while maintaining T_a , thereby achieving the optimal state that maximizes energy efficiency without violating Service Level Objectives (SLOs). The operation of the energy-first mode is similar, but it conducts frequency control to maximize energy efficiency first.

V. ON-DEMAND MODEL ASSEMBLY

As described in Section IV, we construct a partial model that maximizes the number of computational cost options under the constraints (M_d , R_d). However, naive block selection severely degrades the available options, while finding the optimal configuration involves a vast search space, making it infeasible for on-demand execution.

To overcome this, we employ search space pruning and dynamic programming, enabling the system to quickly find the optimal configuration. We first formulate the problem of finding a partial model as a combinatorial optimization problem. Then, we describe dynamic programming (DP) to solve this problem.

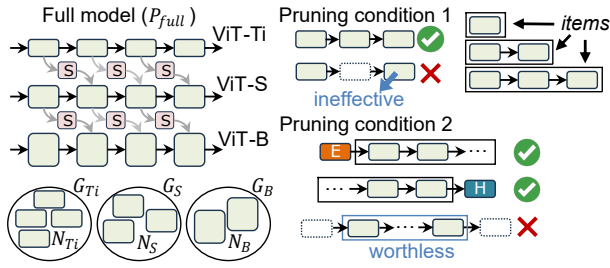


Fig. 8. On-demand model assembly. *Pruning condition 1*: Consecutive blocks only; isolated fragments are ineffective. *Pruning condition 2*: Items must attach to the embedding or head; intermediate floating is worthless.

A. Problem Formulation

As shown in Fig. 8, we assume a full model P_{full} is constructed with ViT-Ti, ViT-S, and ViT-B, with the sets of transformer blocks in each model denoted as G_{Ti} , G_S , G_B , respectively. The number of elements (i.e., transformer blocks) in each set is N_{Ti} , N_S , N_B , respectively. A partial model P_u is configured with parts of P_{full} . As all blocks of P_{full} are selected or not, the number of possible configurations for P_u is $2^{N_{Ti}} \cdot 2^{N_S} \cdot 2^{N_B}$. Thus, finding the optimal partial model requires solving a combinatorial optimization problem with a complexity of $O(2^n)$.

To address this high complexity, we propose reducing the state space using two pruning conditions, relaxing the problem complexity to $O(n^3)$. Note that a partial model necessarily and always contains at least one feasible path. The conditions we impose do not affect the feasibility of the partial model itself but rather serve as pruning conditions that reduce the state space by eliminating suboptimal configuration within it. Specifically, fragmented block selections either do not contribute to valid paths or require expensive larger-model suffixes, resulting in low path-count efficiency relative to their memory cost. Embedding layers, head layers, and stitch layers are excluded from the candidates for combination, as their impact on complexity is negligible compared to transformer blocks, and their inclusion would not affect the state space pruning.

Pruning the solution space. As shown in Fig. 8, the first condition (Pruning condition 1) is that only consecutive blocks are considered. In a neural stitching-based ViT, inference is performed as the input traverses the selected path. If non-consecutive blocks are selected, isolated fragments that are not connected to a valid prefix or suffix do not contribute to feasible paths while still occupying memory. Even when fragmented ranges in an intermediate ViT can form stitched paths, they typically require keeping heavier suffix blocks from the larger ViT, making them inefficient for maximizing path-count flexibility under a memory budget. Therefore, selectable items are restricted to consecutive blocks.

The second condition (Pruning condition 2) is that the selected items (consecutive blocks) are positioned immediately after the initial embedding layer or right before the final output head. If the adjacent blocks on either side of the selected blocks are removed, path formation becomes impossible. Additionally, selectable items for the smallest and largest ViTs (in

this case, ViT-Ti and ViT-B) are restricted to being positioned right after the initial embedding layer and right before the final output head, respectively. This is because placing a selected item from the smallest ViT at the output head or a selected item from the largest ViT at the embedding layer provides no benefit for path formation. Based on these two pruning conditions, the sets of computational and memory costs of selectable items are defined as follows.

$$C_{Ti} = \{c_0^{Ti}, c_1^{Ti,F}, c_2^{Ti,F}, \dots, c_{N_{Ti}}^{Ti}\}, \quad (1)$$

$$M_{Ti} = \{m_0^{Ti}, m_1^{Ti,F}, m_2^{Ti,F}, \dots, m_{N_{Ti}}^{Ti}\}.$$

$$C_S = \{c_0^S, c_1^{S,F}, \dots, c_{N_S-1}^{S,F}, c_{N_S}^{S,B}, \dots, c_{N_S}^{S,B}\}, \quad (2)$$

$$M_S = \{m_0^S, m_1^{S,F}, \dots, m_{N_S-1}^{S,F}, m_{N_S}^{S,B}, \dots, m_{N_S}^{S,B}\}.$$

$$C_B = \{c_0^B, c_1^{B,B}, c_2^{B,B}, \dots, c_{N_B}^{B,B}\}, \quad (3)$$

$$M_B = \{m_0^B, m_1^{B,B}, m_2^{B,B}, \dots, m_{N_B}^{B,B}\}.$$

C_{Ti} and M_{Ti} represent the sets of computational cost and memory cost of selectable items from ViT-Ti, respectively. C_S and M_S correspond to ViT-S, and C_B and M_B to ViT-B. In Eq. (2), $c_j^{S,F}$ and $m_j^{S,F}$ denote the costs of j consecutive blocks placed right after the initial embedding layer, and $c_j^{S,B}$ and $m_j^{S,B}$ placed right before the final output head. c_0^S and m_0^S correspond to no blocks selected, while $c_{N_S}^S$ and $m_{N_S}^S$ denote selecting all blocks. With these definitions, a partial model is constructed by selecting elements from the sets. Assuming k_{Ti} , k_S , and k_B are the indices of the selected items for ViT-Ti, ViT-S, and ViT-B, respectively, the following condition represents the constraints for constructing a partial model within the given memory constraint (M_d).

$$M_{Ti}(k_{Ti}) + M_S(k_S) + M_B(k_B) + \alpha \leq M_d, \quad (4)$$

$$0 \leq k_{Ti} \leq N_{Ti}, \quad 0 \leq k_S \leq 2N_S - 1, \quad 0 \leq k_B \leq N_B.$$

α is a constant representing the memory occupied by the embedding, head, and stitch layers of each ViT, which we omit hereafter to simplify the equations.

Combinatorial optimization problem. Our problem is finding a partial model that achieves the highest flexibility (number of paths) within the given constraints, M_d and R_d . This is equivalent to selecting three items (indices are k_{Ti} , k_S , k_B) with weights $M_{Ti}(k_{Ti})$, $M_S(k_S)$, and $M_B(k_B)$ to fit into a knapsack with a total capacity of M_d , while maximizing their combined value (flexibility).

To define this problem, we quantify the item values as flexibility, that is the number of created paths by selecting the items. Each independent ViT generates a path only when all its blocks are selected. For instance, for the ViT-Ti, a path is formed only when all blocks are present ($M_{Ti}(k_{Ti}) = m_{N_{Ti}}^{Ti}$). Thus, the number of paths created by each item is quantified as follows:

$$V_{Ti} = \{v_0^{Ti}, \dots, v_{N_{Ti}-1}^{Ti}, v_{N_{Ti}}^{Ti}\} = \{0, \dots, 0, 1\}. \quad (5)$$

$$V_S = \{v_0^S, \dots, v_{2N_S-2}^S, v_{2N_S-1}^S\} = \{0, \dots, 0, 1\}. \quad (6)$$

$$V_B = \{v_0^B, \dots, v_{N_B-1}^B, v_{N_B}^B\} = \{0, \dots, 0, 1\}. \quad (7)$$

Meanwhile, due to the nature of neural stitching-based ViTs,

where paths are formed through stitch layers, selecting two different items creates additional paths through their connecting stitch layers. For example, when selecting one item from ViT-Ti and one item from ViT-S, the number of paths generated corresponds to the number of stitch layers between them. Therefore, we define the synergy value function between selected items as follows. Note that f_{stitch} is a function that returns the number of stitch layers between two selected items, indicated by k_{Ti} , k_S , and k_B .

$$S_{Ti,S}(k_{Ti}, k_S) = f_{stitch}^{Ti,S}(k_{Ti}, k_S). \quad (8)$$

$$S_{S,B}(k_S, k_B) = f_{stitch}^{S,B}(k_S, k_B). \quad (9)$$

The total value for k_{Ti} , k_S , and k_B -th items is:

$$V(k_{Ti}, k_S, k_B) = v_{Ti}(k_{Ti}) + v_S(k_S) + v_B(k_B) + S_{Ti,S}(k_{Ti}, k_S) + S_{S,B}(k_S, k_B). \quad (10)$$

We predefine the value function V . Then, we apply the given computational cost (R_d) at runtime to create V_{R_d} , which is used to find the partial model, as follows.

$$c(k_{Ti}, k_S, k_B) = c_{Ti}(k_{Ti}) + c_S(k_S) + c_B(k_B). \quad (11)$$

$$V_{R_d}(k_{Ti}, k_S, k_B) = \begin{cases} V(k_{Ti}, k_S, k_B), & \text{if } c(k_{Ti}, k_S, k_B) \in R_d, \\ 0, & \text{else.} \end{cases} \quad (12)$$

$C(k_{Ti}, k_S, k_B)$ represents the total computational cost of the selected items, and its value is preserved only if it lies within R_d . Ultimately, our objective is to maximize V_{R_d} within the memory constraint, as expressed in the following equation.

$$\begin{aligned} & \text{maximize } V_{R_d}(k_{Ti}, k_S, k_B) \\ & \text{s.t. } M_{Ti}(k_{Ti}) + M_S(k_S) + M_B(k_B) + \alpha \leq M_d \end{aligned} \quad (13)$$

This objective counts only paths whose computational costs fall within the required range R_d . Since R_d reflects the target latency and runtime resources, maximizing V_{R_d} increases the number of operating points for accuracy-energy control.

B. Dynamic Programming

To find k_{Ti} , k_S , and k_B that maximize V_{R_d} , the required computations are $N_{Ti} \cdot (2N_S - 1) \cdot N_B$, resulting in a computational complexity of a cubic polynomial. We adopt a DP approach based on a two-dimensional DP table to reduce redundant computations, thereby relaxing the complexity to $O(n \cdot m)$. In DP table D , the row index represents the item's index, and the column index represents the memory capacity. Thus, the number of rows is the total number of selectable items, $n_{Ti} + n_S + n_B$, and the number of columns corresponds to the memory capacity m_d . Here, $n_{Ti} = N_{Ti} + 1$, $n_S = 2N_S$, $n_B = N_B + 1$, and m_d represents M_d divided by the unit (e.g., for an 800 MB with a 2 MB unit, m_d is 400). The DP table values with the row index x and the column index y consist of two values: $D(x, y) = \{v_D(x, y), k_D(x, y)\}$. $v_D(x, y)$ is the

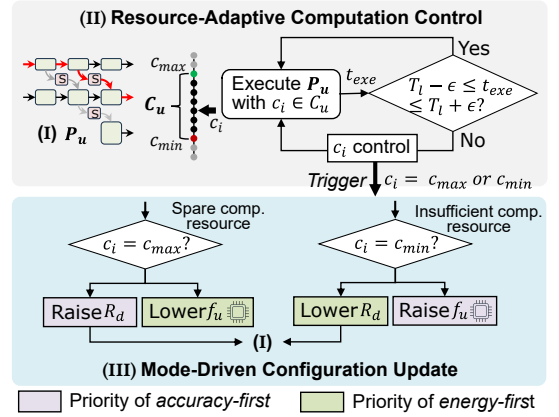


Fig. 9. Resource-Adaptive Computation Control and Mode-Driven Configuration Update.

value and $k_D(x, y)$ stores the index of the selected item. The DP table is populated using the following equation.

$$n_{pre} = \begin{cases} 0, & 1 \leq x \leq n_{Ti} \\ n_{Ti}, & n_{Ti} + 1 \leq x \leq n_{Ti} + n_S \\ n_{Ti} + n_S, & n_{Ti} + n_S + 1 \leq x \leq n_{Ti} + n_S + n_B \end{cases} \quad (14)$$

$$v_{R_d}(k_i, k_j) = \begin{cases} V_{R_d}(k_i, k_j, 0), & k_i = k_{Ti}, k_j = k_S \\ V_{R_d}(0, k_i, k_j), & k_i = k_S, k_j = k_B \end{cases} \quad (15)$$

$$v_{select}(x, y) = v(n_{pre}, y - M(x)) + v_{R_d}(k_D(n_{pre}), x - n_{pre}) \quad (16)$$

$$v_D(x, y) = \begin{cases} v_D(x - 1, y), & v_{select}(x, y) \leq v_D(x - 1, y) \\ v_{select}(x, y), & v_{select}(x, y) > v_D(x - 1, y) \end{cases} \quad (17)$$

$$k_D(x, y) = \begin{cases} k_D(x - 1, y), & v_{select}(x, y) \leq v_D(x - 1, y) \\ x - n_{pre}, & v_{select}(x, y) > v_D(x - 1, y) \end{cases} \quad (18)$$

n_{pre} denotes the last index of the ViT model preceding the one that contains the current item index x . For example, if the item indexed by x belongs to ViT-S, n_{pre} is the last index of the items in ViT-Ti. v_{R_d} represents the value of two items, derived using V_{R_d} , and M is the sum of M_{Ti} , M_S , and M_B maintaining the index order. v_{select} is the total value when the item x is selected. Based on these, we complete the table: $D(x, y) = \{v_D(x, y), k_D(x, y)\}$. Finally, P_u is obtained by tracking the values in the table from the last index, requiring $(n_{Ti} + n_S + n_B) \cdot m_d$ computations.

VI. COMPUTATION CONTROL AND CONFIGURATION UPDATE

Resource-adaptive computation control. This component controls the execution of the partial model configured in the (I) On-Demand Model Assembly component. As shown in Fig. 9, the partial model P_u is executed with a computational cost c_i selected from the available cost set C_u . This cost is dynamically adjusted based on the comparison between

the runtime inference latency t_{exe} and the target latency T_l . To prevent unnecessarily frequent adjustments and improve reliability in meeting the target latency, we introduce a margin ϵ to T_l .

During inference, if the required computational cost exceeds the range of the cost set C_u (Trigger), a configuration update is requested along with the recorded history of selected costs and their corresponding execution times t_{exe} . This process is described at the end of this section.

Note that, since runtime computational resources are unknown prior to execution, the initial partial model P_u is configured with an unconstrained computational cost range to facilitate the search for a configuration with the broadest options. The initial computational cost c_i is set to the median value of the cost set C_u , while c_{min} and c_{max} are derived from the boundaries of C_u .

Mode-driven configuration update. When a configuration update is requested, this component updates either the computational cost range for partial model reconfiguration or the processor frequency, based on the operation mode, i.e., accuracy-first or energy-first. Fig. 9 shows the detailed operation. The request is triggered in two cases: the selected cost c_i reaches c_{max} or c_{min} , indicating that the current computational resources are insufficient or spare, respectively. We address these two cases separately.

In the first case, indicating spare computational resources, we have two options: raising the computational cost range R_d or lowering the frequency f_u . Raising R_d improves model accuracy, whereas lowering f_u results in enhanced energy efficiency. Therefore, in accuracy-first mode, the system first raises R_d , and if it is not feasible (due to memory constraints or met target accuracy T_a), f_u is then lowered. On the other hand, in energy-first mode, the system first lowers f_u and then raises R_d . Throughout the control process, we ensure that f_u does not fall below the threshold f_{th} (red line in Fig. 5) to prevent simultaneous degradation of energy efficiency and processor performance. In our implementation, f_{th} is empirically determined offline for each device and processor type. We profile representative ViT executions across available processor frequencies and set f_{th} to the lowest frequency before energy efficiency starts to degrade.

The second case, indicating insufficient computational resources, also presents two options: lowering R_d or raising f_u . Lowering R_d compromises the overall model accuracy, whereas raising f_u reduces energy efficiency. Thus, in accuracy-first mode the system prioritizes raising f_u . In energy-first mode, the system prioritizes lowering R_d .

Computational cost range update. To lower or increase the computational cost range R_d , we utilize the past record of the selected costs and their latencies. Since the selected cost c_i captures the state of computational resources when the latency t_{exe} is close to the target latency T_l , we determine a new R_d in three steps: 1) Create a frequency histogram of valid c_i : Only values of c_i where $|t_{exe} - T_l| < \epsilon$ are considered, ensuring they reflect runtime computational resource conditions. 2) Calculate scores based on weighted frequency: More recent records are more relevant, so scores are assigned to each computational cost with higher weights given to recent entries. 3) Update the

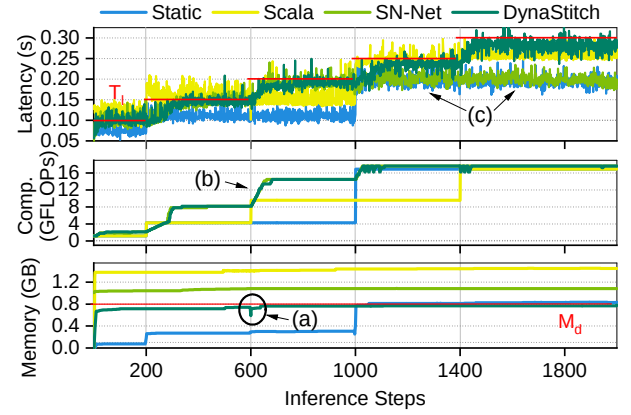


Fig. 10. Comparison with the baselines. DynaStitch enables (a) partial model configuration within the memory constraint, (b) fine-grained computational cost control, and (c) energy consumption reduction.

cost range R_d : R_d is updated upward or downward to include the top γ computational costs with the highest scores. The size of R_d and γ are empirically determined according to the ViT type and application.

VII. EVALUATION

A. Settings

Implementation. We implement DynaStitch in three steps: (1) We train a full neural stitching-based ViT using the SN-Net framework [9]. (2) The model parameters are split at the block level and stored separately using the ONNX library. (3) At runtime, the system dynamically loads or unloads blocks based on the given configuration, enabling partial model execution. CPU experiments use the ONNX Runtime CPU Execution Provider, while GPU experiments use the ONNX Runtime CUDA Execution Provider. All experiments use the ONNX Runtime for inference. We implement our approach on three types of backbone models—DeiT [12], DeiT3 [25], representative plain ViTs, and Swin Transformer [13] (denoted as Swin), a widely adopted hierarchical ViT. DeiT and Swin are applied for the image classification and DeiT3 for the semantic segmentation. We use the ImageNet-1K dataset for classification and the ADE20K for segmentation. We use accuracy-first mode by default for consistent comparison; energy-first is analyzed in Section VII-C.

Baselines. We compare DynaStitch against multiple baselines: (1) Static: selectively utilizing an appropriate model from a fixed pool of pre-trained models (e.g., DeiT-{Ti, S, B}). (2) State-of-the-art flexible ViTs: SN-Net [9], a neural stitching-based ViT, and Scala [8], a slimmable ViT.

Metrics. We evaluate the system using four metrics: memory usage (MB) and inference latency (s) to assess whether the system operates within the memory constraints and target latency; and accuracy—Top-1 (%) for classification and mIoU (%) for segmentation—and power consumption (W) to measure improvements in inference accuracy and energy efficiency. Power consumption and memory usage are logged using `tegrastats`, a monitoring tool for NVIDIA Jetson devices.

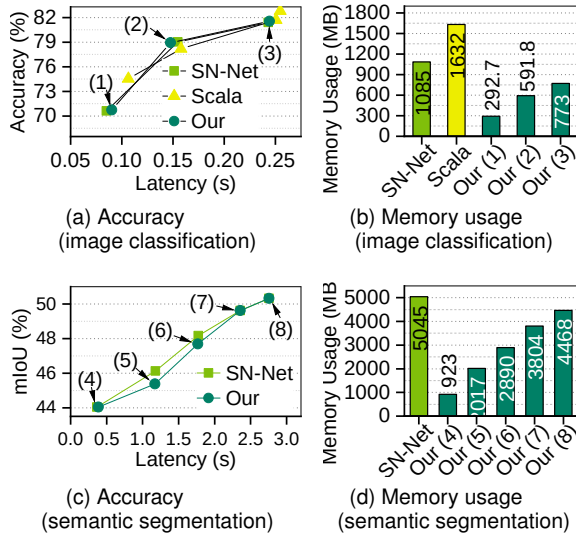


Fig. 11. Comparison with the state-of-the-art flexible ViTs. DynaStitch achieves comparable accuracy with lower memory usage.

B. State-of-the-Art Comparison

We compare DynaStitch with the baselines as the target latency (T_l) increases from 0.1 s to 0.3 s in 0.05 s steps, with $M_d = 0.8$ GB. Fig. 10 shows the DeiT results on Orin, where DynaStitch consistently outperforms the baselines.

(a) Meeting Memory Constraints: The memory trace highlights a key difference: unlike DynaStitch, other flexible ViTs (i.e., Scala and SN-Net) do not consider runtime memory constraints and incur significantly higher memory usage. DynaStitch stays within M_d by partially reconfiguring the model at the outset, from T12-S12-B7 to T0-S3-B12 at (a). Crucially, this reconfiguration allows DynaStitch to cover the full range of computational costs supported by the original full model while meeting the memory constraint.

(b) Adaptive Computational Cost Control: In the computational cost trace, Static offers only three fixed options, providing coarse-grained adaptation to meet the target latency. This results in unnecessarily low computational costs and lower inference accuracy than other methods. Scala provides slightly finer cost adjustment than Static. However, due to the structural complexity of adjusting embedding dimensions, Scala exhibits higher latency than other methods at the same computational cost. Its implementation supports only four distinct costs (13 reported in the paper), yielding much coarser granularity than the 70+ costs provided by neural stitching-based ViTs. In contrast, as marked in (b), SN-Net and DynaStitch show the superior flexibility of neural stitching by adjusting cost at fine granularity within the target latency.

(c) Maximizing Energy Efficiency: SN-Net and DynaStitch exhibit different latencies at the mark (c) even when operating at the same computational cost. This disparity is due to the frequency control of DynaStitch. When DynaStitch operates at maximum computational cost but still achieves a latency lower than the target T_l , it automatically reduces the processor frequency while meeting T_l , thus operating more energy-efficiently. In the $T_l = 0.25$ interval, DynaStitch autonomously drops the frequency from 2.2 GHz (18.4 W, 3.67 J per sample)

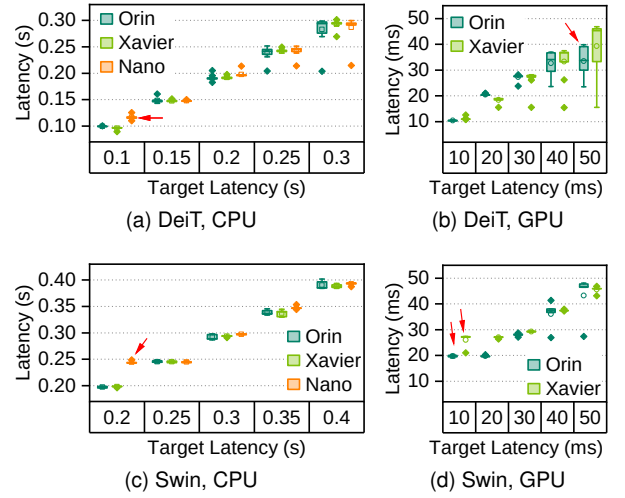


Fig. 12. Latency. DynaStitch meets the target latencies under diverse settings. Red arrows indicate the cases that are physically unachievable due to inherent hardware limitations.

to 1.96 GHz (13.2 W, 3.03 J per sample). In the $T_l = 0.3$ interval, it further reduces the frequency to 1.73 GHz (9.6 W, 2.69 J per sample). This maximizes energy efficiency while meeting the constraints.

We further compare DynaStitch with state-of-the-art flexible ViTs under various memory constraints. For image classification, Fig. 11(a)-(b) shows that DynaStitch achieves comparable accuracy to SN-Net and Scala with much lower memory usage. For example, DynaStitch achieves comparable accuracy under 300, 600, and 800 MB constraints at $T_l = 0.1, 0.15$, and 0.25 , respectively. This is mainly because runtime partial-model assembly eliminates the memory overhead seen in the baselines. SN-Net preloads all candidate ViT blocks and additional stitch layers, while Scala retains all model weights to support different sub-network configurations, causing high memory usage. Fig. 11(c)-(d) shows the semantic segmentation results. For a fair comparison, we evaluate only SN-Net, the only flexible ViT with a segmentation-compatible design, because segmentation requires performance-influencing custom heads. DynaStitch shows a similar trend, achieving comparable mIoU across diverse settings (T_l : 0.5–3.0 s, M_d : 1000–4500 MB) with consistently lower memory usage than SN-Net.

C. System Performance

The goal of DynaStitch is to maximize inference accuracy and energy efficiency while meeting runtime memory constraints and target latencies across edge devices with diverse resources. We evaluate it over 340 settings on three devices (i.e., Orin, Xavier, and Nano) under diverse memory constraints and target latencies. We measure memory usage, inference latency, accuracy, and energy efficiency to evaluate both meeting constraints and performance maximization. Each setting includes 50,000 sequential inference steps. The memory constraint (M_d) ranges from 100 MB to 1600 MB at 100 MB intervals, while the target latency (T_l) is set between 0.1–0.4 s for CPU delegation and 10–50 ms for GPU delegation. These settings are determined within feasible operational limits, as the minimum required partial model

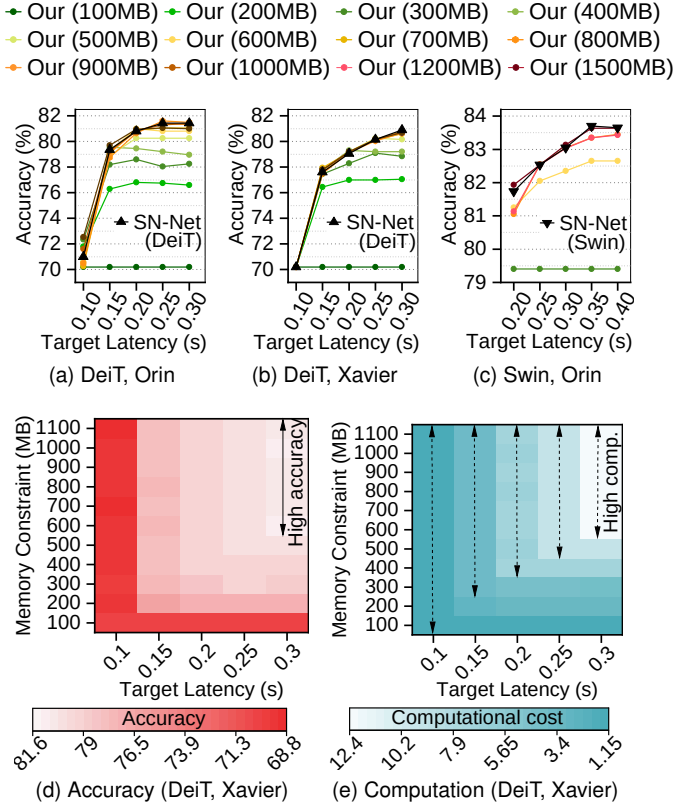


Fig. 13. Accuracy. DynaStitch maximizes inference accuracy under memory constraints and target latency.

size and achievable latency depend on the backbone type and processor performance.

1) *Meeting Constraints*: While Section VII-B showed that DynaStitch meets memory constraints, here we focus on evaluating meeting target latencies on different devices. Fig. 12 presents the distribution of inference latencies corresponding to the target latencies across all scenarios on the three devices. These distributions are derived from the average inference latency per setting, grouped by target latency. The results demonstrate that DynaStitch effectively meets the target latency across different devices and backbone models.

However, as indicated by the arrows in Fig. 12(a) and Fig. 12(c), Nano fails to meet the 0.2 s target latency. This is because, even when executing the model with the smallest computational cost, Nano's CPU performance is inherently insufficient to reach the desired latency. A similar issue is observed in Fig. 12(d) for GPU delegation, where both Orin and Xavier fail to meet the 10 ms target latency (red arrows). This is due to the high minimum computational cost of the Swin Transformer. On the other hand, in Fig. 12(b), the latencies in the arrow-indicated case are considerably lower than the target. This means that even when executing the most computationally intensive model at the lowest processor frequency (f_{th}), inference is still faster than the target latency. Overall, the results show that DynaStitch effectively meets the target latencies in most cases. However, in certain scenarios, its ability to meet the target latency is constrained by the processor's performance and model type.

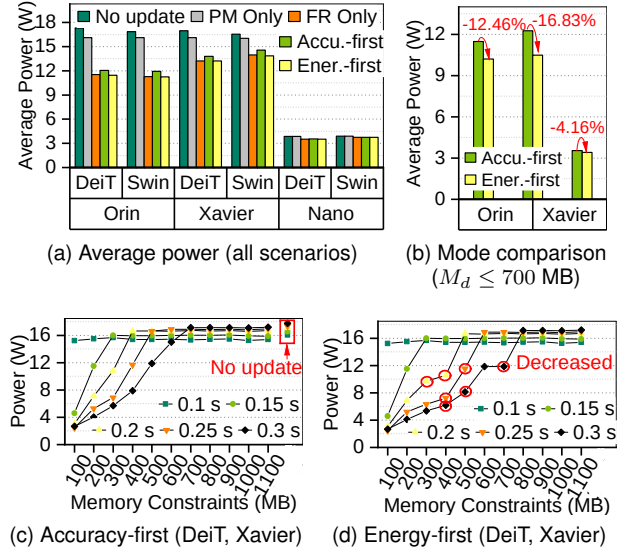


Fig. 14. Power consumption of the two operation modes. DynaStitch significantly reduces power consumption in both modes.

2) *Performance Maximization: Accuracy*. We compare the inference accuracy of DynaStitch using partial models against the full model (i.e., SN-Net), which operates without memory constraints. Fig. 13 presents their Top-1 accuracy across target latencies, devices, and backbone models. DynaStitch maintains high accuracy under memory constraints by constructing partial models. In Figs. 13(a) and 13(b), it achieves accuracy comparable to SN-Net (DeiT) under tight memory constraints. Figs. 13(d) and 13(e) further analyze the DeiT/Xavier case in Fig. 13(b), showing accuracy and supported computational cost for each memory constraint and target latency. DynaStitch maintains comparable accuracy down to 600 MB because its partial model retains high-cost execution paths comparable to full SN-Net, which uses 1085 MB (up to 44.7% memory reduction). Under stricter constraints, the supported computational cost and accuracy naturally decrease, showing that DynaStitch removes unused blocks while retaining paths needed for high-accuracy inference.

Similarly, in Fig. 13(c), DynaStitch achieves comparable accuracy to SN-Net (Swin) under a 900 MB constraint, compared to SN-Net's 1600 MB. This demonstrates that DynaStitch configures partial models tailored to runtime computational resources, maximizing inference accuracy. While accuracy naturally declines under stricter constraints due to reduced model capacity, DynaStitch ensures the optimal configuration within the limit, as validated in Section VII-D.

Energy. DynaStitch improves energy efficiency by updating partial models and processor frequency according to the operation mode. We compare five policies: (1) No update: The initial partial model remains unchanged without any updates, (2) PM Only: Only partial model updates are performed, without frequency control, (3) FR Only: Only frequency control is performed, without partial model updates, (4) Accuracy-first: DynaStitch with accuracy-first mode, and (5) Energy-first: DynaStitch with energy-first mode.

Fig. 14(a) presents the average power consumption across

TABLE II
TOP-1 ACCURACY (%) (DeiT, XAVIER, $M_d = 500$ MB).

Target Latency	0.1 s	0.15 s	0.2 s	0.25 s	0.3 s
No update	70.2	77.8	78.35	78.35	78.35
PM Only	70.2	77.8	79.2	80.15	80.15
FR Only	70.2	77.8	78.35	78.35	78.35
Accuracy-first	70.2	77.8	79.2	80.15	80.15
Energy-first	70.2	77.8	79.05	79.05	79.1

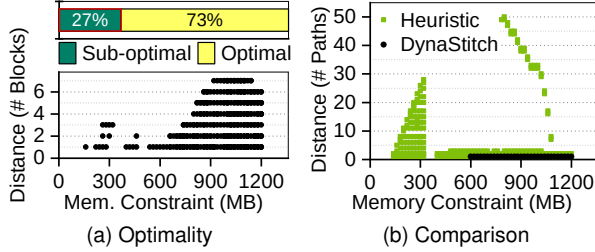


Fig. 15. Algorithm validation.

all scenarios for each policy. Both No update and PM Only policies exhibit high power consumption, demonstrating that frequency control plays a significant role in improving energy efficiency. Specifically, frequency control reduces energy consumption by 30.45% on Orin, 18.7% on Xavier, and 9.25% on Nano. FR Only, without partial model updates, also significantly reduces power consumption. However, as shown in Table II, the FR Only yields lower accuracy because it limits the model to the initial configuration. Accuracy-first achieves the highest accuracy at slightly higher power than FR Only and Energy-first, while Energy-first minimizes power while still improving accuracy through partial-model updates.

Although the power difference between Accuracy-first and Energy-first appears marginal on average, Fig. 14(b) highlights a pronounced disparity in specific scenarios (i.e., constraints below 700 MB). This suggests that the impact of each mode varies by scenario. Figs. 14(c) and 14(d) provide a detailed comparison of the two modes: As memory constraints increase, larger models are accommodated, and as target latency decreases, the system has less flexibility to adjust frequency, limiting power savings. Thus, energy-saving potential depends on device type, memory constraints, and target latency. Overall, both modes effectively balance energy efficiency and accuracy, significantly reducing power consumption while maintaining high performance.

D. On-demand Model Assembly

Optimality and effectiveness. We validate the optimality and effectiveness of our DP solution described in Section V against optimal and heuristic baselines across 4,430 settings with varying memory constraints and computational cost ranges. Optimal solutions are obtained using an optimization solver, Google OR-Tools [26]. Heuristic solutions are generated as follows: at least one valid path must be ensured within the memory constraint. The heuristic first selects all blocks from the largest possible model in the model family that fits within the memory constraint. It then greedily adds blocks from smaller and larger anchors to maximize path diversity within

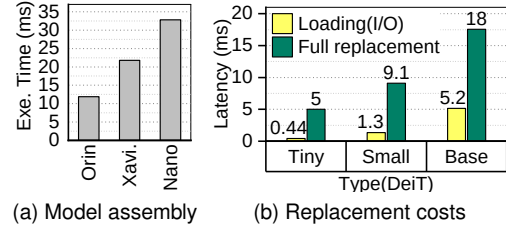


Fig. 16. Overhead evaluation.

the remaining memory budget. For example, if a DeiT-S model can fit within the constraint, it is selected first, and additional blocks from DeiT-T and DeiT-B are added in that order.

Compared with optimal solutions, Fig. 15(a) shows that our partial-model configurations exactly match 73%, while 27% differ in block composition. This discrepancy, shown in the graph as the number of differing blocks, tends to increase under looser memory constraints, where the number of feasible partial models grows.

With many valid options, our DP algorithm may differ from OR-Tools because it resolves ties by selecting the first valid candidate encountered. Importantly, even among these 27% configurations, Fig. 15(b) shows that the number of supported paths—a distance metric for flexibility—differs by at most one from the OR-Tools optimal solutions (black dots), demonstrating consistently near-optimal results.

Fig. 15(b) also compares the heuristic baseline (green dots), which shows a much greater distance from the optimal solutions in the number of supported paths. This indicates that the heuristic configuration supports fewer execution paths than our method, reducing flexibility in adapting computational cost. This degradation is particularly evident under very low or high memory constraints. In such cases, the initially selected model is either too small (e.g., DeiT-T) or too large (e.g., DeiT-B), and the greedy strategy fails to diversify execution paths.

Overhead. We evaluate partial-model update overhead by measuring the On-Demand Assembly execution time on three devices. Fig. 16(a) shows average execution times of tens of milliseconds. In comparison, naive exhaustive search over block subsets before applying the two pruning conditions took 53.8 hours on a desktop server (Intel Xeon E5-2620 v3 CPU). The pruning conditions reduce the search space to selectable anchored block ranges, enabling the DP solver to find a configuration within tens of milliseconds on Jetson devices.

Next, we decompose full block replacement time into pure block loading (I/O) and remaining runtime overhead (Fig. 16(b)). The total replacement overhead is 5.02, 9.08, and 17.57 ms for Tiny, Small, and Base blocks, respectively. Pure loading accounts for only a small fraction, indicating that most overhead comes from runtime processing, such as session construction, graph optimization, and memory cleanup, rather than file access. Under synthetic memory pressure generated by *stress-ng* with 8 VM workers and 2 GB per worker, the full replacement latency of a Base block increases to 43.85 ms in the worst case, which is 2.5 $\times$ higher than the no-contention case. DynaStitch handles resource dynamics primarily by regulating computational cost, triggering partial model updates

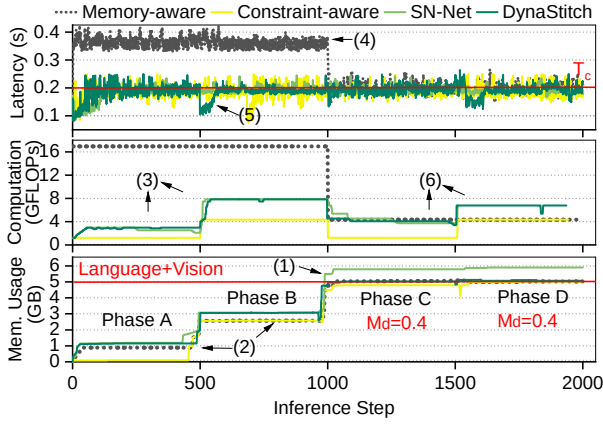


Fig. 17. Baseline comparison in Case 1.

only in response to significant shifts in resources or target latency. According to our experiments, these updates occur infrequently, observed at most three times throughout an entire dynamic session, even in the most extreme scenarios (e.g., only one update in the case shown in Fig. 10). Given this frequency, the impact on the overall latency is expected to be negligible.

VIII. CASE STUDY

We conduct a case study motivated by recent service robots [27] combining visual perception with natural-language interaction [4] (Fig. 2). Such language workloads support user interaction and contextual understanding but incur substantial memory usage depending on model size and runtime state (Table I).

We consider varying memory pressure from a co-running language workload and vision demand across four operating phases (Fig. 2). In Phase A, the robot navigates without a language workload, leaving a large vision memory budget (energy-first). In Phase B, it navigates a crowded area requiring higher-accuracy vision (accuracy-first), with the language engine on standby for prompt interaction. In Phase C, user interaction activates the language model, sharply reducing the vision budget while requiring only lightweight visual processing (energy-first). In Phase D, the interaction additionally requires detailed visual understanding, such as recognizing a directional sign, demanding a higher-cost vision path under the same tight memory budget (accuracy-first).

In this scenario, the language workload runs on the GPU and vision on the CPU. We compare DynaStitch with three baselines. *SN-Net* uses the same path-level Resource-Adaptive Computation Control and frequency control as DynaStitch but does not perform partial model assembly. *Memory-aware Static* selects the largest single model among Tiny, Small, and Base (DeiT-Ti, DeiT-S, and DeiT-B) that fits the current memory budget. *Constraint-aware Static* selects the largest of these models that satisfies both the current memory budget and target latency. The static baselines cannot adjust computational cost after model selection but use the same frequency control as DynaStitch.

Case 1. Case 1 considers an 8-GB Jetson-class robot platform [11]. We conservatively reserve about 3 GB for

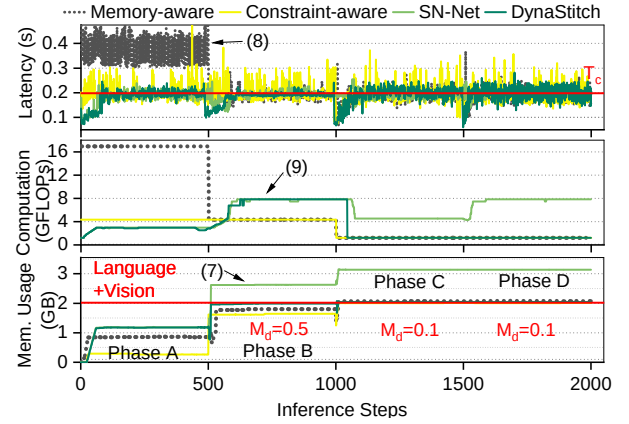


Fig. 18. Baseline comparison in Case 2.

the operating system, robot middleware, navigation/perception stack, and runtime headroom [28], leaving 5 GB for vision in Phase A. Qwen3-4B in standby uses 1.9 GB of measured engine memory, reducing the vision budget to about 3 GB in Phase B. When active, its measured alive memory footprint is 4.6 GB, leaving only 0.4 GB for vision in Phases C and D. Fig. 17 compares DynaStitch with the baselines under this trace.

SN-Net provides many execution paths but keeps the full stitched model in memory. Thus, when the language workload becomes active, it exceeds the vision budget in Phases C and D, as marked by (1). DynaStitch instead constructs partial models and remains within the budget throughout the trace. The static baselines exhibit different limitations. *Memory-aware Static* selects the largest model that fits the memory budget but ignores target latency, causing it to select large models in Phases A and B that violate the latency target even at high frequency, as marked by (4). *Constraint-aware Static* also considers target latency but can choose only among Tiny, Small, and Base. Its coarse operating points therefore leave usable computation headroom unused, as marked by (2).

DynaStitch adapts to both memory budget and phase-dependent vision demand. In Phase A, it uses a low computational cost in energy-first mode while meeting the target latency. In Phase B, it increases the cost in accuracy-first mode to support stronger perception in the crowded area (3). Phases C and D share the same tight memory budget but require different vision demands. DynaStitch reconfigures the partial model and expands the supported computational-cost range in Phase D (6), showing that it can provide different cost ranges under the same memory constraint. Table III shows the resulting accuracy gap between *Constraint-aware Static* and DynaStitch, both of which satisfy the memory budget and target latency.

Case 2. We consider a more constrained 4-GB platform, reserving about 2 GB for system/runtime headroom and using Qwen3-0.6B as a compact language workload. This leaves 2 GB for vision in Phase A (Fig. 2). The vision budget drops to about 0.5 GB in Phase B with Qwen3-0.6B in standby using 1.4 GB of engine memory, and to 0.1 GB in Phases C and D when the active model uses 1.9 GB. Fig. 18 compares

DynaStitch with the baselines under this constrained trace.

SN-Net exceeds the vision budget from Phase B onward (7) because it keeps the full stitched model resident in memory, whereas DynaStitch remains within the tight budget using partial models. The static baselines also have clear limitations. As in Case 1, Memory-aware Static cannot meet the target latency (8). Constraint-aware Static satisfies both memory and latency by selecting a smaller model, but its fixed operating point cannot adapt computation to phase-dependent vision demands. DynaStitch instead adjusts computational cost within the memory budget, increasing it when higher vision performance is required (9), demonstrating its effectiveness under severe memory pressure.

IX. DISCUSSION

A. Scope

Deployment. DynaStitch enables flexible vision inference with dynamically adjustable computational costs to accommodate changing latency, accuracy, or energy requirements while operating within a given memory budget. It is particularly beneficial when the required operating point or available memory changes at runtime. In such settings, a static model may either underutilize available computational resources or fail to meet the current inference requirements, whereas retaining the full stitched model may exceed the memory budget. Conversely, when resource availability and application requirements remain stable and a single static model can satisfy them, the additional adaptability of DynaStitch may provide limited benefit.

Energy and hardware. In our evaluation, frequency control provides most of the power reduction, while partial-model assembly preserves a useful computational-cost range for accuracy under memory constraints. Fixed-frequency accelerators can disable frequency control while retaining model assembly and path selection. On memory-bound platforms, the energy model should also account for path-dependent memory traffic.

Multitasking and runtime integration. DynaStitch neither schedules nor preempts co-running applications; it assumes an external resource manager provides the available vision-memory budget M_d . It handles compute contention through measured inference latency and adjusts the execution path accordingly. General-purpose scheduling and preemption remain outside the current scope.

B. Generalizability

Neural stitching primarily targets general-purpose ViT families, including plain and hierarchical backbones widely used across downstream tasks. Its applicability to task-specific architectures (e.g., those for object detection and human pose estimation) remains less explored because deviations from standard ViT structures and fewer modular stitch points can limit flexibility. Stitching also requires multiple pretrained models of different capacities within the same family, limiting applicability when only one model is available; however, the growing availability of multi-scale ViT families makes this constraint increasingly less significant.

TABLE III
ACCURACY OF FEASIBLE METHODS IN CASE 1

Metric	Constraint-aware		DynaStitch	
	A & B	C & D	A & B	C & D
Top-1 Accuracy (%)	74.4	74.4	76.77	78.02

DynaStitch is a system-level approach to inference-time and memory adaptability, complementary to model-level optimizations such as pruning, quantization, and knowledge distillation, and to compact edge-oriented ViTs such as EfficientFormer [29] and TinyViT [30]. These approaches optimize or redesign individual models before deployment, whereas DynaStitch adapts resident blocks and execution paths at runtime; they can be combined when stitch compatibility is preserved. In practice, quantization requires efficient hardware and runtime support for selected precision-operator combinations; pruning must preserve block interfaces and may require sparse-kernel support; and distilled or compact ViT families require compatible model variants and stitch points, potentially with architecture-specific retraining.

X. CONCLUSION

This paper presents DynaStitch, a system for deploying neural stitching-based ViTs on resource-constrained edge devices under varying runtime resources. We propose: (1) fast optimal partial model configuration under memory constraints; (2) runtime computational cost control within target latencies; and (3) a mode-driven strategy that optimizes inference accuracy and energy efficiency via adaptive model reconfiguration and processor frequency scaling. Evaluation shows that DynaStitch adapts to diverse edge resources, reducing memory usage and power consumption while maintaining comparable accuracy.

ACKNOWLEDGMENTS

This work was supported by Sweden’s Innovation Agency (Vinnova), by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2024-00344323), and by the Ewha Womans University Research Grant of 2026.

OpenAI ChatGPT was used for language editing of author-written text in portions of Sections II and VII–IX.

REFERENCES

- [1] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, “An image is worth 16x16 words: Transformers for image recognition at scale,” *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2021.
- [2] C. Jung, J. Lee, G. Kim, J. Kim, S. Park, and H. Cha, “Aria: Optimizing vision foundation model inference on heterogeneous mobile processors for augmented reality,” in *Proc. ACM Int. Conf. Mobile Syst., Appl., Services (MobiSys)*, 2025, pp. 249–262.
- [3] J. Lee, C. Jung, J. Kim, and H. Cha, “Panopticus: Omnidirectional 3d object detection on resource-constrained edge devices,” in *Proc. ACM Int. Conf. Mobile Comput. Netw. (MobiCom)*, 2024, pp. 1207–1221.
- [4] T. S. Gurunathan, M. S. Raza, A. K. Janakiraman, M. A. Khan, B. Pal, and A. Gangopadhyay, “Edge llms for real-time contextual understanding with ground robots,” in *Proc. AAAI Symp. Ser.*, vol. 5, no. 1, 2025, pp. 159–166.

- [5] A. Bakhtiarnia, Q. Zhang, and A. Iosifidis, “Multi-exit vision transformer for dynamic inference,” *arXiv preprint arXiv:2106.15183*, 2021.
- [6] G. Xu, J. Hao, L. Shen, H. Hu, Y. Luo, H. Lin, and J. Shen, “Lgvit: Dynamic early exiting for accelerating vision transformer,” in *Proc. ACM Int. Conf. Multimedia (MM)*, 2023, pp. 9103–9114.
- [7] S. Laskaridis, A. Kouris, and N. D. Lane, “Adaptive inference through early-exit networks: Design, challenges and directions,” in *Proc. Int. Workshop Embedded Mobile Deep Learn.*, 2021, pp. 1–6.
- [8] Y. Zhang, X. Ma, H. Wang, K. Ma, S. Chen, D. Hu, Y. Fu *et al.*, “Slicing vision transformer for flexible inference,” *Adv. Neural Inf. Process. Syst.*, vol. 37, pp. 42 649–42 671, 2024.
- [9] Z. Pan, J. Cai, and B. Zhuang, “Stitchable neural networks,” in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2023, pp. 16 102–16 112.
- [10] Z. Pan, J. Liu, H. He, J. Cai, and B. Zhuang, “Stitched vits are flexible vision backbones,” in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2024, pp. 258–274.
- [11] NVIDIA, “Jetson orin nano,” <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/>, Mar. 2025.
- [12] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. J’egou, “Training data-efficient image transformers & distillation through attention,” in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2021, pp. 10 347–10 357.
- [13] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, “Swin transformer: Hierarchical vision transformer using shifted windows,” in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, 2021, pp. 10 012–10 022.
- [14] Ollama, “Ollama on jetson,” <https://www.jetson-ai-lab.com/tutorials/ollama/>, Jun. 2026.
- [15] M. Chen, H. Peng, J. Fu, and H. Ling, “Autoformer: Searching transformers for visual recognition,” in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, 2021, pp. 12 270–12 280.
- [16] F. Devvrit, S. Kudugunta, A. Kusupati, T. Dettmers, K. Chen, I. Dhillon, Y. Tsvetkov, H. Hajishirzi, S. Kakade, A. Farhadi *et al.*, “MatFormer: Nested transformer for elastic inference,” *Adv. Neural Inf. Process. Syst.*, vol. 37, pp. 140 535–140 564, 2024.
- [17] B. Shi, S. Xia, X. Yang, H. Chen, Z. Kou, and X. Geng, “Building variable-sized models via learngene pool,” in *Proc. AAAI Conf. Artif. Intell.*, vol. 38, no. 13, 2024, pp. 14 946–14 954.
- [18] S. Xia, M. Zhang, X. Yang, R. Chen, H. Chen, and X. Geng, “Transformer as linear expansion of learngene,” in *Proc. AAAI Conf. Artif. Intell.*, vol. 38, no. 14, 2024, pp. 16 014–16 022.
- [19] S. Xia, Y. Zu, X. Yang, and X. Geng, “Initializing variable-sized vision transformers from learngene with learnable transformation,” *Adv. Neural Inf. Process. Syst.*, vol. 37, pp. 43 341–43 366, 2024.
- [20] C. Riquelme, J. Puigcerver, B. Mustafa, M. Neumann, R. Jenatton, A. Susano Pinto, D. Keysers, and N. Houlsby, “Scaling vision with sparse mixture of experts,” *Adv. Neural Inf. Process. Syst.*, vol. 34, pp. 8583–8595, 2021.
- [21] S. Teerapittayanon, B. McDanel, and H.-T. Kung, “Branchynet: Fast inference via early exiting from deep neural networks,” in *Proc. Int. Conf. Pattern Recognit. (ICPR)*, 2016, pp. 2464–2469.
- [22] Z. Pan, B. Zhuang, D.-A. Huang, W. Nie, Z. Yu, C. Xiao, J. Cai, and A. Anandkumar, “T-stitch: Accelerating sampling in pre-trained diffusion models with trajectory stitching,” *arXiv preprint arXiv:2402.14167*, 2024.
- [23] Z. Chen, Z. Chen, J. He, L. Sheng, M. Tan, J. Cai, and B. Zhuang, “R-stitch: Dynamic trajectory stitching for efficient reasoning,” *arXiv preprint arXiv:2507.17307*, 2025.
- [24] S. Kornblith, M. Norouzi, H. Lee, and G. Hinton, “Similarity of neural network representations revisited,” in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2019, pp. 3519–3529.
- [25] H. Touvron, M. Cord, and H. J’egou, “Deit iii: Revenge of the vit,” in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2022, pp. 516–533.
- [26] Google, “Google or-tools,” <https://developers.google.com/optimization>, Mar. 2025.
- [27] Passenger Self Service, “Robots guide passengers,” 2025, accessed: 2026-08-14. [Online]. Available: <https://www.passengerselfservice.com/2025/08/robots-guide-passengers-at-halifax-stanfield/>
- [28] NVIDIA Isaac ROS, “Getting started,” https://nvidia-isaac-ros.github.io/v/release-3.2/getting_started/index.html, accessed: Jun. 19, 2026.
- [29] Y. Li, G. Yuan, Y. Wen, J. Hu, G. Evangelidis, S. Tulyakov, Y. Wang, and J. Ren, “EfficientFormer: Vision transformers at MobileNet speed,” *Adv. Neural Inf. Process. Syst.*, vol. 35, pp. 12 934–12 949, 2022.
- [30] K. Wu, J. Zhang, H. Peng, M. Liu, B. Xiao, J. Fu, and L. Yuan, “TinyViT: Fast pretraining distillation for small vision transformers,” in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2022, pp. 68–85.



Jiwon Kim received the B.S. degree in Electronic Engineering from Ewha Womans University in 2016 and the Ph.D. degree in Computer Science from Yonsei University, Seoul, Republic of Korea in 2024. She is currently an Assistant Professor in the Department of Artificial Intelligence at Ewha Womans University. Her research interests include on-device AI, energy-aware mobile systems, and intelligent energy storage systems.

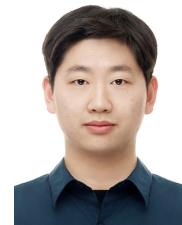


Luca Mottola is a Full Professor at Politecnico di Milano (Italy), a Senior Researcher at RISE Sweden, and a Visiting Faculty at Uppsala University (Sweden). He was granted multiple times with the ACM SigMobile Research Highlight. He received the ACM SENSYS Test of Time Award in 2022, is a Google Faculty Award winner, and received eight Best Paper Awards at flagship conferences. He is past General Chair for CPS-IoT Week 2022, past PC Chair for ACM MobiSys, ACM SenSys, and ACM/IEEE IPSN, as well as an AE of IEEE

Transactions on Networking and ACM Transactions on Sensor Networks.



Jeho Lee received the B.S. degree in Computer Engineering from Ajou University in 2020 and is currently pursuing the Ph.D. degree in Computer Science and Engineering at Yonsei University, Seoul, Republic of Korea. His research interests include on-device AI and mobile systems, with a focus on efficient neural network inference on heterogeneous mobile SoCs and mobile NPUs.



Seonghoon Park received the B.S. and Ph.D. degrees in Computer Science and Engineering from Yonsei University, Seoul, Republic of Korea, in 2018 and 2025, respectively. He is currently an Assistant Professor in the Department of Computer Science and Engineering at Yonsei University. His research interests include mobile immersive computing, on-device AI, and energy-efficient mobile systems.



Hojung Cha received the B.S. and M.S. degrees in computer engineering from Seoul National University, Seoul, South Korea, in 1985 and 1987, respectively, and the Ph.D. degree in computer science from the University of Manchester, Manchester, U.K., in 1991. He is currently a Professor with the Department of Computer Science and Engineering, Yonsei University, Seoul. His research interests include on-device AI system optimization and energy-aware mobile systems.



Thiemo Voigt received the Ph.D. degree from Uppsala University, Sweden, in 2002. He is currently a Full Professor at the Department of Electrical Engineering, Uppsala University, and a Senior Researcher at RISE Computer Science. His current research interests include low-power networking, system software, and machine learning for embedded networked devices and the Internet of Things, as well as in-body communication and 6G Ambient IoT. His work has been cited more than 22 900 times and has an H-index of 64. He is an Editorial Board

Member of ACM Transactions on Sensor Networks (TOSN).