DyCL: Dynamic Neural Network Compilation Via Program Rewriting and Graph Optimization

Simin Chen
simin.chen@UTDallas.edu
UT Dallas
Dallas, USA

Shiyi Wei
swei@utdallas.edu
UT Dallas
Dallas, USA

Cong Liu
congl@ucr.edu
UC Riverside
Riverside, USA

Wei Yang
wei.yang@utdallas.edu
UT Dallas
Dallas, USA

ABSTRACT

The deep learning (DL) compiler serves as a vital infrastructure component to enable the deployment of deep neural networks on diverse hardware platforms such as mobile devices and Raspberry Pi. DL compiler's primary function is to translate DNN programs written in high-level DL frameworks such as PyTorch and TensorFlow into portable executables. These executables can then be flexibly executed by the deployed host programs. However, existing DL compilers rely on a tracing mechanism, which involves feeding a runtime input to a neural network program and tracing the program execution paths to generate the computational graph necessary for compilation. Unfortunately, this mechanism falls short when dealing with modern dynamic neural networks (DyNNs) that possess varying computational graphs depending on the inputs. Consequently, conventional DL compilers struggle to accurately compile DyNNs into executable code. To address this limitation, we propose DyCL, a general approach that enables any existing DL compiler to successfully compile DyNNs. DyCL tackles the dynamic nature of DyNNs by introducing a compilation mechanism that redistributes the control and data flow of the original DNN programs during the compilation process. Specifically, DyCL develops program analysis and program transformation techniques to convert a dynamic neural network into multiple sub-neural networks. Each sub-neural network is devoid of conditional statements and is compiled independently. Furthermore, DyCL synthesizes a host module that models the control flow of the DyNNs and facilitates the invocation of the sub-neural networks. Our evaluation demonstrates the effectiveness of DyCL, achieving a 100% success rate in compiling all dynamic neural networks. Moreover, the compiled executables generated by DyCL exhibit significantly improved performance, running between 1.12× and 20.21× faster than the original DyNNs executed on general-purpose DL frameworks.

CCS CONCEPTS

• **Software and its engineering → Compilers**; • **Computing methodologies → Machine learning**.

KEYWORDS

Dynamic Neural Networks, Deep Learning Compilers

1 INTRODUCTION

With the growing popularity of deep learning(DL)-based applications, optimizing, executing, and deploying these applications becomes critical. DL compilers [3, 7, 11–13, 32, 33, 43, 53, 54] are fundamental infrastructures for achieving these goals, enabling DL application deployment on various hardware devices. DL compilers translate DL models written in high-level DL frameworks (*e.g.,* PyTorch [37] and TensorFlow [1]) into optimized and portable executables.

Over the past few years, significant advancements have been made in the development of DL compilers, aiming to streamline the deployment of neural networks on diverse hardware platforms [6, 8, 11, 44]. DL compilers offer two key advantages. First, they enable the compiled DNN model to function as a executable program, eliminating the need for model developers to install resource-intensive DL frameworks on target platforms to parse and execute the DNN models. Second, DL compilers optimize the inference time overhead of a given DNN model, making it suitable for real-time applications on resource-constrained platforms such as mobile devices.

However, existing DL compilers heavily rely on the *tracing mechanism* [6, 33], which requires providing a runtime input to a neural network program and tracing its execution path to generate the necessary computational graph for compilation. Unfortunately, these tracing mechanisms implicitly assume that a DNN model can be abstracted as a "static" computational graph, with a fixed execution path for computation. This assumption, however, does not hold for modern dynamic neural networks (DyNNs), where the execution

path is determined by individual inputs and varies with each invocation [9, 15, 19, 34, 46, 49]. For instance, an encoder-decoder model used in neural machine translation [4] may require invoking the underlying decoder multiple times to produce translation outputs, without specifying the exact number of invocations.

To understand the limitations of the tracing mechanism employed by existing DL compilers when it comes to compiling dynamic neural networks, we conducted an empirical study utilizing two widely-used DL compilers (*i.e.,* TVM [6] and OnnxRuntime [33]) to compile four types of DyNNs. The results revealed a discrepancy between the outputs produced by running the compiled executables and the original DNN models within DL frameworks. This discrepancy clearly indicates that the DL compilers fail to accurately compile DyNNs.

To overcome the limitations of existing DL compilers, we introduce an automatic tool, DyCL, which assists developers in accurately compiling DyNNs automatically. Our primary objective is to enable the flexible adaptation of optimizations found in current DL compilers while ensuring the correct handling of the inherent dynamism present in DyNNs. The design of DyCL is driven by two key observations. First, we identify that the main source of DL compilers' inability to compile DyNNs lies in the presence of conditional statements within DyNN programs (*e.g.,* conditional statements). When a source program does not contain such conditional statements, a DyNN program effectively transforms into a regular DNN program, with a computational graph that can be determined statically. Second, we recognize that DL applications typically involve pre- and post-processing stages, such as image normalization and token mapping. Consequently, the compiled DNN executable cannot function as a standalone program. Instead, a host program (*i.e.,* Listing 2) is responsible for running both the pre- and post-processing stages, as well as invoking the DNN executable for inference. It is worth noting that the host program is often developed using a high-level programming language (*e.g.,* Java, C/C++, and Python), whose compiler is equipped to adequately handle dynamism.

Based on these observations, we design DyCL to move the dynamism of DyNN models to the host programs and reuse existing DL compilers to compile the sub-DNN models without dynamism. Specifically, DyCL coverts a DyNN into multiple standard neural networks (we refer to the separated neural networks as sub-DNNs) with conditional statements determining which sub-DNNs are invoked, and then apply the existing DL compilers to compile the sub-DNNs and incorporate the conditional statements into the host program.

We identify two challenges to correctly moving the dynamism from DyNN programs to the host programs. The first challenge is maintaining the essential contexts (*i.e.,* concrete model instance and model input shape) for compiling each sub-DNN. To address this challenge, our insight is that each sub-DNN is a "fixed" part and has no dependency on the DyNN inputs. In other words, we can obtain the concrete DNN instance for each sub-DNN by performing constant propagation. Motivated by such intuition, we developed a program rewriting engine (Section 5.2) that first conducts loop unrolling and then constant propagation to make each variable that has no dependency on the DyNN's input constant. As for maintaining the input tensor shape for each sub-DNN, our insight

is that each sub-DNN input is the output of its predecessor sub-DNNs. Based on this insight, we introduced a novel concept called the Heterogeneous Control Flow Graph (HCFG) (Section 5.3), a special type of control flow graph, to model the dynamic behavior and the data flow of DyNNs. After that, we propose a novel traverse algorithm (Section 5.5) to traverse the HCFG, trace each sub-DNNs output shape, and use them as the input shapes for compiling the subsequent sub-DNNs. By doing so, we can successfully collect the necessary context to compile each sub-DNN.

The second challenge we encountered involves co-optimizing the host program and the compiled sub-DNNs to achieve enhanced optimization. Our insight for tackling this challenge stems from the observation that the host program and the compiled sub-DNNs are typically executed on different devices, such as CPU and GPU. Consequently, unnecessary overhead may arise due to data transfers between these devices. Thus, we can further optimize each sub-DNN and put the computation-free operations (*e.g.,* memory manipulation) on the host program to reduce the data transfer overheads. We then propose two strategies to identify the computation-free operations (*e.g.,* constant assignment and tensor copy) in each sub-DNN and move the operations from the computational graph of the sub-DNN to the corresponding host program to ensure semantic equivalence.

We conducted extensive experiments to evaluate DyCL. Specifically, we evaluate two open-source DL compilers, TVM [7] and OnnxRuntime [33], on nine DyNN models, and we select two popular hardware platforms (*i.e.,* Nvidia TX2 and Nvidia AGX) as the backends. The selected DyNN models are diverse in terms of model architecture, model size, and application. The selected hardware backends are popular embedded platforms for deploying neural networks to assist system decision-making. We evaluate DyCL in terms of compilation correctness and acceleration. Moreover, we conduct an ablation study to understand the contribution of our proposed graph optimization module. The results show that DyCL can 100% correctly compile all dynamic neural networks (*i.e.,* the final decision after the post-processing of the compiled DyNN has a 100% consistent rate with the decision of the original DyNN model), and the maximum numeric error between the compiled DyNN and the original DyNN is around $10^{-4.72}$, significantly less than directly applying DL compiler to compile the DyNN (range from 10^0 to 10^4). Moreover, the compiled executables run 1.12× to 20.21× faster than the original DyNNs running on the general-purpose DL frameworks, indicating the benefits of applying DyCL to deploy DyNN models. Finally, our ablation study shows that the proposed graph optimization module can further benefit the compilation process.

This paper made the following contributions.

- We conduct an empirical study to use two popular DL compilers (*i.e.,* TVM and OnnxRuntime) to compile four types of DyNNs. The study results illustrate the limitations of the existing DL compilers when compiling DyNNs.
- We present a program rewriting approach that allows adapting many existing DL compilers to compile DyNNs correctly. The key novelty of our approach is to identify and represent the dynamism of DyNN programs in heterogeneous control
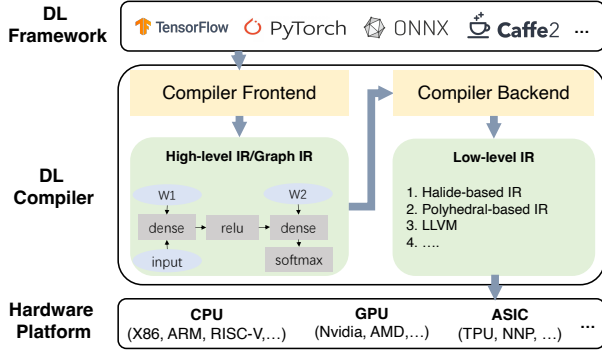
Figure 1: An overview of design architecture of DL compilers.

flow graphs. The sub-DNNs in HCFGs are compiled individually, and our approach generates a host API to call the compiled sub-DNNs.

- Based on the novel ideas, we implement DyCL; our evaluation results show DyCL can correctly compile nine DyNN models and accelerate the DyNN's inference time overheads range from 1.12× to 20.21×.

2 BACKGROUND

2.1 Deep Learning Compiler

DL compilers are designed to optimize deep neural networks from high-level deep learning frameworks (*e.g.,* Pytorch [37], Caffe [22] and TensorFlow [27]) and produce executables for AI-programs running on different hardware platforms [25, 48]. As shown in Fig. 1, a DL compiler primarily contains two parts [26]: the compiler frontend and the compiler backend. To compile a DNN, the compiler frontend first translates the DL model into high-level intermediate representations (IR) for hardware-independent optimizations. After that, the compiler backend converts the high-level IR into low-level IR for hardware-specific optimizations and code generation. The high-level IR in a DL compiler is typically represented by a graph, called *computational graph*. In a computational graph, a node represents an operation on a tensor or a program input, and an edge represents the data dependence between operations. The low-level IRs are language and machine-dependant, capturing the hardware characteristics (*e.g.,* memory management).

```
# Load a pretrained model                                      1
model = torchvision.models.resnet18(pretrained=True)           2
model = model.eval()                                           3
                                                               4
# Grab the TorchScripted model via tracing                     5
input_shape = [1, 3, 224, 224]                                 6
example_data = torch.randn(input_shape)                        7
scripted_model = torch.jit.trace(model, example_data)          8
                                                               9
# Transfer the PyTorch model to Relay                          10
input_name = "input0"                                          11
shape_list = [(input_name, img.shape)]                         12
mod, params = relay.frontend.from_pytorch(scripted_model,      13
    shape_list)
                                                               14
# Compile the model for the target platform                    15
target = tvm.target.Target("llvm", host="llvm")                16
```

```
dev = tvm.cpu(0)                                               17
with tvm.transform.PassContext(opt_level=3):                   18
    lib = relay.build(mod, target=target, params=params)      19
```

Listing 1: Example of compiling DNNs.

```
# Preprocess the image and convert to tensor                   1
img = Image.open(img_path).resize((224, 224))                  2
my_preprocess = transforms.Compose([                           3
        transforms.Resize(256),                                4
        transforms.CenterCrop(224),                            5
        transforms.ToTensor(),                                 6
        transforms.Normalize(                                  7
        mean=[0.485, 0.456, 0.406],                            8
        std=[0.229, 0.224, 0.225]),                            9
    ])                                                         10
img = my_preprocess(img)                                       11
img = np.expand_dims(img, 0)                                   12
                                                               13
# Load the compiled DNN and inference                          14
m = graph_executor.GraphModule(lib["default"](dev))            15
img = img.astype("float32")                                    16
m.set_input(input_name, tvm.nd.array(img))                     17
m.run()                                                        18
tvm_output = m.get_output(0)                                   19
                                                               20
# Postprocess the inference results                            21
top1_tvm = np.argmax(tvm_output.numpy()[0])                    22
tvm_class_key = class_id_to_key[top1_tvm]                      23
```

Listing 2: Example host program deploying compiled DNNs.

Listing 1[1] shows an example using TVM [6] to compile a DNN for mobile programs. As shown in line 8, to compile a DNN, the first step is to trace the DNN. The trace step requires two inputs: a DNN instance (*i.e.,* model) and an input example (*i.e.,* example_data). It outputs one scripted module. Listing 2 shows an example of how the compiled DNNs are deployed in a mobile platform. Lines 1-12 show the pre-processing step to normalize the input image; lines 14-17 show how to load the compiled DNN and use it for inference. Lines 21-23 show the post-processing step. In this paper, we refer to the program that loads the compiled DNN executable as the *host program*.
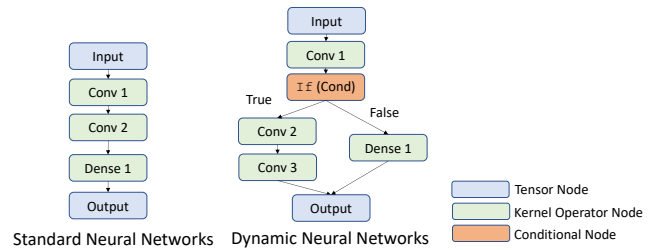

Figure 2: An overview of the standard neural networks vs. dynamic neural networks.

2.2 Dynamic Neural Network Model

Before we introduce Dynamic Neural Networks (DyNNs), we first introduce the basic concepts of standard neural networks. As shown in the left subfigure of Fig. 2, a neural network can be abstracted as

---
[1]https://tvm.apache.org/docs/how_to/compile_models/from_pytorch.html

a directed acyclic graph (DAG). The node in the graph represents either the tensor or the kernel operators (*e.g.,* `convolutional` operator and dense operator), and the edge represents the data-flow dependency between the nodes.

However, standard neural networks cannot satisfy the modern needs of many real-world applications. Take natural language processing as an example, where the neural networks require different input and output dimensions, making dynamic neural networks inherently necessary. In response to these demands, researchers have proposed DyNNs [9, 15, 19, 28, 30, 35, 39, 46, 49]. DyNNs merge neural networks with conditional logic, allowing their execution paths to be modified based on varying inputs. In the right subfigure of Fig. 2, DyNNs include additional components such as the `If` conditional nodes alongside the tensor and kernel operator nodes. When presented with specific input, the `If` conditional node utilizes the computed intermediate tensor (*e.g.,* the output tensor of the `Conv1` node in our example) to determine which sections of the overall graph should be activated for computation. For instance, in early-exit DyNNs, the `If` conditional node employs the computed confidence score from its inter-classifier to decide whether to continue the computation. Similarly, in neural machine translation DyNNs, the `If` conditional node leverages the output token value to determine whether the translation should be completed.

In this study, our focus lies on dynamic neural networks that exclusively consist of input-independent loops, as the inclusion of an input-dependent loop could potentially lead to an infinite loop scenario. DyNNs are unique tensor programs in which the parameters are learned from data rather than manually defined. However, this reliance on trained parameters renders them susceptible to adversarial examples [4], where the adversarial agent can manipulate the inputs to deceive the DNN and create an infinite loop. Consequently, to mitigate this vulnerability, current implementations of DyNNs often incorporate a flag that enforces a constant maximum number of iterations, independent of the inputs, thus safeguarding against potential adversarial attacks. This implementation style has been seen in existing work [4] which studied 1,455 DyNN implementations and all of the studied DyNNs apply such an implementation style.

3 EMPIRICAL STUDY

In this section, we perform an empirical study to understand if existing DL compilers can correctly compile DyNN models.

3.1 Study Setup

Target DL Compilers. We target two DL compilers in this study: Apache `TVM` [7] and Microsoft `OnnxRuntime` [33], both of which are popular in academic research and industry work. `TVM` is a DL compiler for deploying DNNs on various platforms. It first converts a deep neural network into a static computational graph for high-level optimization and then generates hardware-specific code for each node in the transformed graph. On the other hand, `OnnxRuntime` is a runtime-based framework. It also converts a deep neural network into a graph representation for optimization. After that, `OnnxRuntime` maps the graph node to pre-compiled kernel operator functions.

Target DyNN Models. We selected DyNN models for our study using the following four criteria. The selected DyNN models *(i)* are the state-of-the-art models, which are published at top-tier conferences and outperform others according to their publication; *(ii)* are popularly used in work conducted by both academia and industry; *(iii)* differ from each other in terms of input domains and dynamic mechanisms; *(vi)* are publicly available, and their code can be successfully executed.

Based on the above criteria, we selected four DyNNs from the survey by Huang et al. [17], listed in Table 1. Two of them are *energy-saving DyNNs* [2]: Shallow-Deep and SkipNet [23, 47]. Shallow-Deep is an early-termination DyNN that has multiple exits in a deep neural network. If one of the exits is confident about the prediction, the execution is stopped early. SkipNet is a conditional-skipping network that decides to skip or execute a DNN block based on the intermediate gate values. AttentionNet [45] and En-Decoder [50] are *generative DyNNs*. AttentionNet is a Neural Machine Translation (NMT) model that uses an attention mechanism to draw dependencies between input and output. En-Decoder uses two different types of attention mechanisms to generate captions for images.

3.2 Study Process

To check whether existing DL Compilers can correctly compile the DyNNs, our intuition is that a correct compilation process should not change the semantics of the DyNNs. With this intuition, we compare the semantics of the original DyNNs and the compiled DyNNs. Specifically, we generate random inputs and feed the generated inputs to both the original DyNN and the compiled DyNN for inference and collect the outputs. Given the same inputs, the compiled and original DyNNs should produce identical outputs. Otherwise, the compiled DyNN is not semantically equivalent to the original DyNN, implying the compilation process fails.

Recall the original DyNN model first runs on a general-purpose DL framework (*e.g.,* `PyTorch` and `TensorFlow`) to train its parameters. Thus, we can use kernel functions from the general-purpose DL framework to launch the original DyNN, and we denote this program as the vendor program $\mathcal{V}(\cdot)$. We then follow each target DL compiler's documentation to compile the original DyNN and generate the compiled executable $C(\cdot)$. We randomly sample 1,000 inputs from a DyNN model's hold-out testing dataset as the test suite. We feed the test suite to both $\mathcal{V}(\cdot)$ and $C(\cdot)$ and collect the outputs before and after the post-prepossess (*e.g.,* the variable `tvm_output` in line 19 and `tvm_class_key` in line 23 in Listing 2). We use the subscripts *before* and *post* to distinguish these two outputs. We then define two metrics to evaluate whether the target DL compiler can produce correct programs.

$$\delta = log \left\{ \max_{i=1}^{N} ||C_{before}(x_i) - \mathcal{V}_{before}(x_i)|| + \epsilon \right\}$$
$$\eta = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(\mathcal{V}_{post}(x_i) \neq C_{post}(x_i)) \tag{1}$$

Metrics. As shown in Eq.(1), our first metric is *maximum numeric error* δ, which computes the maximum numeric error between the vendor and compiled DyNN programs outputs before the post-prepossess (*i.e.,* the maximum error between the outputs of line

Table 1: The Subject DyNNs in Our Preliminary Study

DyNNs	Domain	Dynamic Mechanism	# of Branch	Github Star	Citation
Shallow-Deep	Image Classification	layer wise early stopping DNN	13	24	89
SkipNet	Image classification	block wise selective execution	53	208	355
En-Decoder	Image caption	token wise caption generation	50	1.2k	8863
AttentionNet	machine translation	token wise caption generation	200	1.2k	40711

Table 2: The rate of inconsistency outputs predictions.

DL Compiler	Shallow-Deep	SkipNet	En-Decoder	AttentionNet	ResNet
OnnxRuntime	0.83	1.00	0.87	0.83	0.00
TVM	0.83	1.00	0.87	0.83	0.00

19 in Listing 2), where ϵ is set as 10^{-10} to avoid the division-by-zero issue. The DNN compilation process needs to perform some numeric matrix operations, and errors naturally exist in the matrix operations. Thus, numeric inconsistencies are inevitable in the DNN compilation process.

However, some numeric inconsistencies may not affect DyNN programs' final decision after post-processing. Thus, besides *maximum numeric error*, we propose *final inconsistent rate η* to measure the inconsistent rate of the post-prepossessed outputs between the vendor DyNN programs and the optimized DyNN programs, where $\mathbb{I}$ is the identity function, and it outputs 1 if the expression inside this function is evaluated to be true; otherwise 0. If the *final inconsistent rate* does not equal to 0, then it means that C will produce different outputs with $\mathcal{V}$ given the same inputs, implying the DL compiler changes the DyNN's semantics. For classification DyNNs, the post-processing step involves computing the predicted label. This is achieved by searching for the category with the highest confidence scores [18]. For generation DyNNs, we set the post-prepossess as computing the generated sequences, which is done by searching the token that has the maximum likelihood among each output position [45].

Comparison Baselines. To show the compilation process is correct, we also compile a standard DNN program with no branches, ResNet50 [18], as the baseline.

3.3 Study Results

The number of inconsistent predictions from the compiled DL executable and the DL framework is shown in Table 2. The results demonstrate that the majority of randomly selected inputs (more than 80%) produce inconsistent outputs between vendor programs and compiled executables, implying that the produced DNN executable is semantically inequivalent to the original DyNN programs.
Failure Analysis. We attempted to comprehend the enormous amount of inconsistent predictions in Table 2. In particular, we would like to show that the inconsistency is caused by the design limitation of the DL compiler rather than the numerical errors in the compiler optimization process. We visualize the error distribution of the outputs from $\mathcal{V}(\cdot)$ and $C(\cdot)$. The error distribution is shown in Fig. 3 (for better presentation, we only show the results of SkipNet and En-Decoder, more results could be found on our website), where
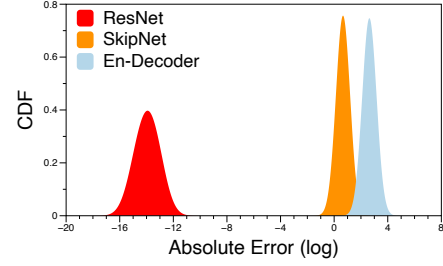
Figure 3: The error distribution of standard DNN and DyNNs.

the x-axis represents the absolute error and the y-axis represents the cumulative distribution function (CDF) [21]. We observe that the absolute errors of ResNet are mostly located in the range of $[10^{-15}, 10^{-13}]$. However, the errors of DyNNs are located in the range of $[10^{-1}, 10^4]$, depending on the DyNN models. Such a significant error gap (*i.e.,* more than $10^{14}\times$) demonstrates that the discrepancy in DyNN is not due to numerical errors.

To gain further insight into the reasons for compilation failure, we conducted an analysis focusing on consistent outputs (it is worth noting that some DyNNs, as shown in Table 2, are capable of producing consistent outputs). In our analysis, we specifically tracked the execution traces of these inputs and compared them to the traces used during the compilation process of the DyNN. As indicated in line 8 of Listing 1, the DL compiler relies on tracing an example data to compile the model. During our investigation, we made a crucial observation: these traces were found to be identical. This observation can be attributed to a fundamental characteristic of existing DL compilers - their "static" nature of the tracing mechanism in the compilation process. Consequently, during the compilation process, the DL compiler disregards conditional branches (such as if statements) and generates a fixed computational graph based on the execution trace of the example input. As a result, the compiled DyNN executable utilizes the same execution path for all inputs, as dictated by the fixed computational graph. While it is possible for inputs to follow the same execution path as the example data, resulting in the compiled executable producing identical results to the original DyNN program, the likelihood of this occurring is exceptionally low. This is particularly evident in the case of SkipNet, which encompasses an astounding 2^{53} distinct execution paths.

Summary. Our experimental results confirm the limitations of existing DL compilers. Specifically, while some of these compilers do offer support for control flow, they struggle to correctly compile DyNNs due to their reliance on tracing mechanisms for obtaining computational graphs.

```
1
def adaptive_forward_compile(self, x):                    2
    # some code for tensor computation                    3
                                                          4
    for g in range(3):                                    5
        for i in range(self.num_layers[g]):               6
                                                          7
            if g == 0 and i < self.num_layers[g] - 1:     8
                i = i + 1                                 9
                                                         10
            name = 'group{}_ds{}'.format(g + 1, i)       11
            if name in self.attr_layers:                 12
                model = self.attr_layers[name]           13
                prev = model(prev)                       14
                                                         15
            if mask == 0:                                16
                x = (1 - mask).expand_as(prev) * prev    17
            else:                                        18
                ly_name = 'group{}_layer{}'.format(g+1,i)19
                layer = self.attr_layers[ly_name]        20
                x = layer(x)                             21
                x = mask.expand_as(x) * x                22
                cnt = cnt + 1                            23
            prev = x                                     24
#### Some codes                                          25
```

Listing 3: An example to demonstrate the challenge of identifying sub-DNNs from the source code.

4 CHALLENGES AND INTUITIONS

DyNN model compilation imposes several unique challenges compared to standard "static" neural network model compilation. We enumerate two major challenges and provide a high-level idea of how DyCL addresses them.

Goal: Designing a general approach for various DL Compilers using different types of IRs. To enable existing DL compilers to effectively compile DyNNs, the initial step is to represent the dynamic behavior of DyNNs within these compilers. Furthermore, to fully leverage the capabilities of various optimization strategies present in these DL compilers, a comprehensive approach needs to be designed for each compiler, taking into account their unique designs and characteristics.

Solution: Our solution is based on the following two observations. First, the dynamic behavior of DyNNs comes only from conditional statements (as shown in Fig. 2). DyNNs can be separated into multiple standard neural networks (we refer to the separated neural networks as sub-DNNs) with conditional statements determining which sub-DNN to be invoked, and each sub-DNN can be correctly compiled by existing DL compilers. Second, a compiled DL program consists of an external library function and a host program (as shown in Listing 2). The host program is a program often implemented by a modern high-level programming language (*e.g.,* C/C++ and Python). These languages already have programming constructs to represent the dynamic behaviors (*i.e.,* conditional statements). Combing these two observations, our idea is to split the conditional statements from tensor computations in DyNNs, and then apply the existing DL compilers to compile the tensor computation part and incorporate the conditional statements into the host program.

Challenge 1: Maintaining the contexts for compiling sub-DNN. The first challenge is compiling each sub-DNN in the correct contexts. Recall that in Listing 1, compiling a neural network needs two contexts: the DNN model instance and a DNN model input. Unfortunately, both contexts are not explicitly exhibited in the DyNN implementation. Listing 3 shows an example of a simplified implementation of SkipNet [47]. From the code snippet, it is not easy to identify the necessary context for compilation because some contexts are represented in an implicit way. For example, an existing DL compile is unable to compile a basic block (e.g., lines 13-14) because the compiler is unable to obtain the concrete model instance model (*i.e.,* model is determined by the variable name as the statement in line 13 assigns self.attr_layers[name] to variable model) nor the input tensor shape of the model instance.

Intuition 1: To address this challenge, our observation is that the dynamic behavior of the DyNN model comes from its conditional If node. Removing the conditional If node, the rest sub-DNNs are "fixed"; thus, sub-DNNs have fixed context and have no dependency on the DyNN inputs. In other words, we can obtain the concrete DNN instance for each sub-DNN by performing constant propagation and loop unrolling. As for the shape of each sub-DNN's input, our intuition is that we can start from the entry of the DyNN models' computational graph, iteratively compute the output shape of each node in the graph, and set the output shape as the input shape for the successor node.

Challenge 2: Co-optimization between the host program and tensor computation. Recall that the compiled DyNNs will be invoked as an API function in the host program. To ensure the semantic equivalence between the generated API and the original DyNNs, for each sub-DNNs, we need to track (i) the required input variable of the corresponding code snippet and (ii) the live variable after the corresponding code snippet, and then set these variables as the input/output of the sub-DNNs. However, the output variable of a sub-DNN may be a constant value or identical to one of the sub-DNN's input variables. Putting these variables to DL compilers to get acceleration on the accelerator (*e.g.,* GPU) may introduce unnecessary data transfer time. Thus, it calls for a co-optimization between the host program and the tensor computation.

Intuition 2: Our intuition is that DL compilers are designed to accelerate the "computing" for modern hardware platforms. We seek to reduce as many computation-free data transfer overheads as possible to accelerate the inference process. Based on this intuition, we propose two graph optimization strategies to further optimize the computational graph of each sub-DNN. Our optimization strategy will put computation-free data transfer left in the host program to reduce the data transfer time from the host program to the accelerator.

5 OUR APPROACH: DYCL

Given a DyNN model P_{DyNN}, our goal is to compile it and generate an optimized host API function P_{Host} that is semantically equivalent to the P_{DyNN}. Formally denoted as

$$P_{DyNN}(x) = P_{Host}(x) \qquad \forall x \in X \qquad (2)$$
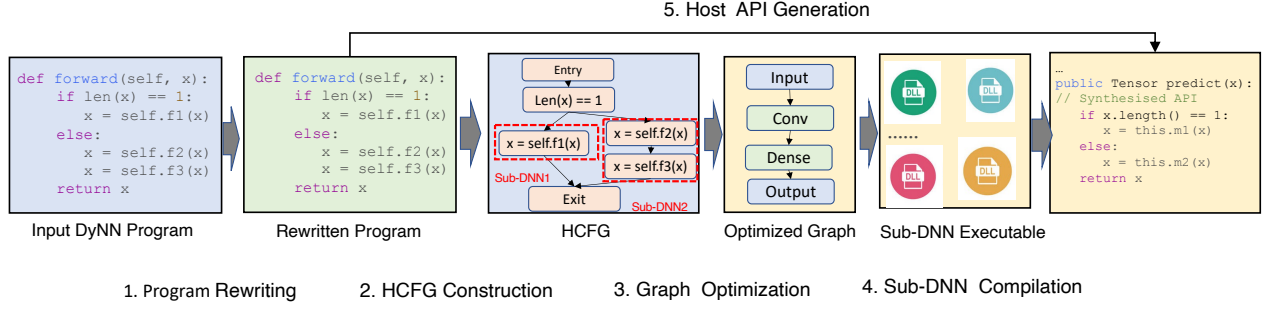
Figure 4: Design overview of DyCL.

For any inputs that belong to the program input domain, the host API will produce the same output as the original DyNN. Moreover, the host program can call the API using a similar way in Listing 2.

5.1 Design Overview

Figure 4 shows the design overview of our approach. The key insight of DyCL is to partition a DyNN into several sub-DNNs without dynamism for compilation and leave the dynamism part to the host program. Based on this insight, DyCL carries out the following five steps. ① *DyNN source program rewriting.* Given a DyNN program from the high-level DL framework (*e.g.,* PyTorch), DyCL first rewrites it and makes the context for each sub-DNN explicit. ② *HCFG Construction.* After rewriting the DyNN program, the next step is constructing a heterogeneous control flow graph (HCFG) to represent the logic conditional nodes and sub-DNN nodes. ③ *Sub-DNN Optimization.* In this step, we build a computational graph for each sub-DNN and propose two strategies to optimize the computational graph of each sub-DNN. ④ *Sub-DNN compilation.* After optimizing the computational graph of each sub-DNN, we start from HCFG's entry node and iteratively compile the node in the HCFG to obtain a set of compiled DNN executables. ⑤ *Host API generation.* Finally, we modify the rewritten DyNN's abstract syntax tree (AST) and covert the modified AST back to a host API function.

Algorithm 1 HCFG Construction Algorithm.

Input: Rewritten DyNN program P_{DyNN}.
Output: HCFG of the input DyNN program.

1: CFG = ConstructCFG(P_{DyNN}) {get the CFG of P_{DyNN}}
2: HCFG = Dict() {Initlize HCFG as an dictionary}
3: **for** each N in CFG.Nodes **do**
4:    s = N.statements[-1] {get the last statement of N}
5:    **if** s is logic conditional statement **then**
6:      s, N2 = BlockPartation(N) {partition node N}
7:      HCFG.update(s) {Add statement s to HCFG}
8:      N = N2. {Update N}
9:    **end if**
10:    HCFG.update(N) {Add node N to HCFG}
11: **end for**

5.2 DyNN Program Rewriting

Recall that the purpose of this step is to make the context for each sub-DNN explicit so that we can automatically compile each sub-DNN. To achieve this goal, we first conduct a loop unrolling process to get a program that contains no cycle in its CFG. As we introduced in Section 2, there is no cycle in the computational graph of the DyNN model. Thus, we need to unroll all loop statements in the original DyNN program. Recall that our tool focuses on DyNNs that do not contain input-dependent loops. This restriction is in place to avoid the potential introduction of infinite loops, as discussed in Section 2. Consequently, our loop unrolling technique is always feasible within this scope. After that, we perform constant propagation to make sure that all variables have no dependency on the DyNN's input constant. Consider the example provided in Listing 3. Excerpts of the rewritten code are presented in Listing 4, showcasing the unrolling of the for loop statements from lines 4-5 and lines 9-10 of Listing 4. These correspond to the for loop statements found in lines 5-6 of Listing 3. Additionally, the if statement present in lines 8-9 of Listing 3 is eliminated following constant propagation.

```
def adaptive_forward_compile(self, x):          1
    # some code for tensor computation          2
    g = 0                                        3
    i = 1                                        4
    name = 'group1_ds1'                          5
    #### Some repeated code                      6
                                                 7
    g = 0                                        8
    i = 2                                        9
    name = 'group2_ds2'                          10
    #### Some repeated code                      11
                                                 12
```

Listing 4: The code snippet of the DyNN after rewriting

5.3 HCFG Construction

After rewriting the original DyNN program, the CFG of the program will be a DAG containing no cycle, and each variable that has no data dependency on DyNN's input will be presented as a constant.

Recall that our goal is to put the dynamic logic of DyNNs on the host program. To achieve this goal, we propose a heterogeneous control flow graph (HCFG), which is a special control flow graph (CFG). HCFG includes two types of nodes, the first comprises only one logic conditional statement, and the other contains multiple
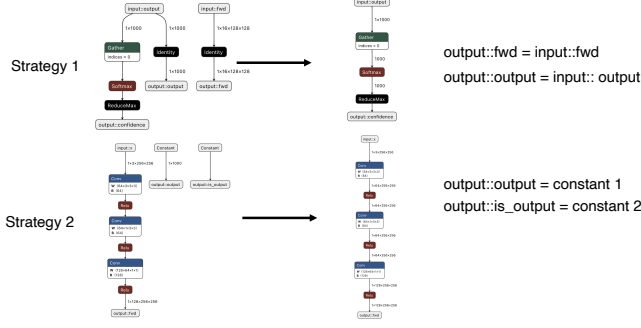
Figure 5: The proposed graph optimization strategy.

sequential statements. The edges in HCFG represent the control flow paths, similar to the edges in CFG. The HCFG construction algorithm is shown in Algorithm 1. On line 1, we first construct the CFG of the rewritten DyNN program. Then we traverse the basic blocks in the CFG (line 3), and if the last statement of the block is an if statement (line 5), we split the block into two blocks and update the HCFG (lines 6-8). Each node in the HCFG is either an if statement or a sequence of statements that contain only the tensor computation statements. We treat each node that contains only the tensor computation statements as a sub-DNN for compilation.

5.4 Sub-DNN Optimization

For each sub-DNN in the constructed HCFG, we perform further optimization on its computational graph to avoid unnecessary data transfer overheads between the host program and the accelerator (*e.g.,* GPU). As discussed in Section 4, we seek to identify the computation-free operations in each sub-DNNs' computational graph and put these operations on the host program.

The first strategy is to eliminate the identity tensor copy operation. For this strategy, we start from each output node and enumerate all paths from the input node to this output node. If a path contains only the identity operator, we will remove this path and add a corresponding assignment statement in the host program. The second strategy is similar to constant propagation in compiler optimization. Firstly, we identify the output node for which all of its sink nodes (nodes without any incoming edges) are constant; we then remove these paths in the computational graph and add the corresponding assignment statement in the host program. Fig. 5 shows an example of our proposed graph optimization strategy. For each strategy, the first column shows the original computational graph, and the second column is the optimized computational graph. The third column shows the corresponding statements we add to the host program.

5.5 Sub-DNN Compilation

Recall that using the DL compiler to compile a DNN model requires feeding an example input to the compiler (*i.e.,* line 8 in Listing 1). To create such an example input, the input tensor shape must be manually defined, which is a time-consuming process for DyNNs with hundreds of sub-DNNs. To address this challenge, we propose an automatic algorithm to compile each sub-DNNs.

Algorithm 2 Sub-DNN Compilation Algorithm.

Input: Heterogeneous Control Flow Graph (HCFG).
Input: Start Node (N_0).
Input: An example input of the DyNN model (x_0).
Output: A mapping from node to compiled sub-DNN ($\mathcal{M}$).

1: $L = [\ N_0\ ]$ {Maintain a search list L}
2: S = Dict() {Maintain S to store tensor shape}
3: **while** L is nor empty **do**
4:    N = Select(L) {select N based on N's precursor}
5:    N_{visit} = True {set N as visited}
6:    $M = N.successor$ {collect N's successor}
7:    $L.update(M)$ {update the search list}
8:    **if** N is N_0 **then**
9:      $out =$ =Compute(N, x_0) {compute node N}
10:     $S[N.id] = out$ {record the output shape for N}
11:     $exe =$ =Compile(N, x) {compile node N}
12:     $\mathcal{M}[N.id] = exe$ {store the executable N}
13:   **else**
14:     **if** N is Logic Node **then**
15:       $S[N.id] = S[N.precursor]$ {set output shape of logic statement the same as its precursor}
16:       Continue
17:     **else**
18:       $x = S[N.precursor]$ {get input shape from its precursor}

19:       $out =$ =Compute(N, x) {compute node N}
20:       $S[N.id] = out$ {record the output shape}
21:       $exe =$ =Compile(N, x) {compile node N}
22:       $\mathcal{M}[N.id] = exe$ {store the executable}
23:     **end if**
24:   **end if**
25: **end while**

Our automatic sub-DNN compilation algorithm is shown in Algorithm 2, which takes an HCFG, a start node N_0, and a DyNN input x_0 as inputs and outputs a mapping $\mathcal{M}$ from node id to the compiled sub-DNNs. In general, the algorithm maintains a search list L and a dictionary S that stores the output shape for each node. While the search list L is not empty, we select a node N from L iteratively, depending on whether all predecessors of N have been computed (line 4). We then set N as visited and collect all non-visited successors of N to update our search list (lines 5-7). For each node N, there are three conditions: (1) N is the start node N_0; (2) N is the logic conditional node; (3) N is a regular node other than N_0. For the first condition, we compute N's output shape and compile node N using input x_0 (lines 8-12). For the second condition, we set N's output shape the same as its predecessor's output shape because the conditional statement will not modify the variable in the program (lines 14-16). As for the third condition, we first obtain an input x from node N's predecessors, then use x to compile N, and finally compute N's output shape (lines 18-23). Our algorithm iteratively fetches N from the search list and compiles N until L is empty (lines 3-4). Finally, our algorithm outputs the mapping $\mathcal{M}$, whose key is the node id and the value is the compiled sub-DNN.

5.6 Host API Generation

Finally, we use a template-based approach to generate the code that invokes each compiled sub-DNN. We modify the rewritten DyNN program's AST by replacing the original tensor computation statements with the statements to invoke our compiled sub-DNNs. We generate our host API function from the modified AST. Our AST modification step only replaces the original tensor computation statements in the original DyNN program with a compiled library function call. Considering the fact that existing DL compilers have proven correct and effective in compiling standard neural networks without conditional statements, the generated API call will be semantically equivalent to the original DyNN model.

6 EVALUATION

6.1 Experimental Setup

In this section, we evaluate DyCL with empirical experiments and answer the following research questions. Our code and data are available on our website [5].[2]

- **RQ1 (Correctness):** Can DyCL correctly compile DyNNs and produce semantic equivalent API for host programs?
- **RQ2 (Acceleration):** Can DyCL optimize DyNNs over multiple platforms in terms of execution time?
- **RQ3 (Ablation Study):** How much does the proposed graph optimization module in DyCL enhance the execution time?
- **RQ4 (Overhead):** What is the overhead of DyCL in compiling DyNNs?

DyNN models. In addition to the four DyNN models in Table 1, we apply DyCL on other five DyNN models. The IDs and names of our subject DyNNs are shown in columns 1 and 2 in Table 3 and we will use IDs to represent these DyNNs in all future tables. The selected DyNN models cover different model architectures (*e.g.*, MobileNet and ResNet), different applications (*e.g.*, image classification and text generation), and different scales (*e.g.*, model sizes range from 2.3MB to 430MB). More detailed information about the evaluated DyNN models can be found on our website.

DL Compilers. We consider the DL compilers used in our empirical study as the target compilers: TVM and OnnxRuntime.

Hardware Platforms. We choose two different NVIDIA platforms imposing different architectural features as our hardware platforms to show DyCL can benefit the process of deploying DyNN models on various hardware platforms. The first hardware platform is NVIDIA Jetson TX2, which has 6 ARM-based cores and a 256-core Pascal-based GPU. The second platform is NVIDIA Jetson AGX Xavier [36], a powerful platform for robotics and autonomous driving with an 8-core NVIDIA Carmel CPU and a 512-core Volta-based GPU.

Comparison Baselines. Recall that DyNNs compiled by existing DL compilers can only correctly infer the inputs that have the same execution paths as the example input used in the compilation process. Thus, comparing DyCL with existing DL compilers is meaningless in terms of correctness. Therefore, for the correctness (RQ1) and the acceleration (RQ2) evaluation, we compare DyCL with original DyNNs without compilation (*i.e.,* using the original DL framework). We refer to the baseline as vendor.

[2]https://github.com/DyCL

Experimental Process and Metrics. For RQ1, we use DyCL to compile each vendor version DyNN $\mathcal{V}(\cdot)$ and get the compiled version $C(\cdot)$. Similar to our study process in Section 3, we feed 100 randomly generated inputs to these two versions and collect the outputs before and after the post-prepossess (*i.e.,* as shown in the last two lines in Listing 2), and use the metrics defined in Eq.(1) to evaluate the correctness of compilation. If DyCL can correctly compile the DyNN model, then the results for the metric *final inconsistent rate* should be zero, and the results for the metric *numeric maximum error* should be significantly different from the numeric errors in Fig. 3.

For RQ2, we first deploy the vendor version DyNN $\mathcal{V}(\cdot)$ and the compiled version $C(\cdot)$ on two different hardware platforms. After that, we feed the same inputs to these two versions for inference and record the inference overheads. For each input, we infer five times and report the average inference overheads.

For RQ3, we remove the graph module in DyCL and apply DyCL to compile each DyNN and get the compiled DyNN $C_{no}(\cdot)$. After that, we deploy $C_{no}(\cdot)$ and $C(\cdot)$ on five different hardware platforms and evaluate their inference overheads, similar to the process in RQ2. Intuitively, if our proposed co-optimization method can benefit the compilation process, then the compiled DyNN $C(\cdot)$ will have lower runtime overhead.

For RQ4, we report the time overheads of each module in DyCL when compiling each DyNN.

6.2 RQ1: Correctness

Table 3: Correctness of DyCL.

ID	Base DNN	Final Inconsistent Rate				Max Numeric Error			
		Nvidia TX2		Nvidia AGX		Nvidia TX2		Nvidia AGX	
		TVM	OnnxR	TVM	OnnxR	TVM	OnnxR	TVM	OnnxR
1	MobileNet	0	0	0	0	-10.00	-10.00	-10.00	-10.00
2	VGG19	0	0	0	0	-9.17	-9.35	-9.17	-9.35
3	ResNet50	0	0	0	0	-4.84	-4.94	-4.89	-5.53
4	WideResNet	0	0	0	0	-4.72	-5.60	-4.75	-5.48
5	ResNet38 + RNN	0	0	0	0	-5.62	-5.81	-5.62	-5.81
6	ResNet38 + Dense	0	0	0	0	-6.45	-6.45	-6.45	-6.45
7	ResxNext + LSTM	0	0	0	0	-10.00	-10.00	-10.00	-10.00
8	GoogLeNet+ LSTM	0	0	0	0	-10.00	-10.00	-10.00	-10.00
9	FlatResNet32	0	0	0	0	-10.00	-10.00	-10.00	-10.00

The compilation correctness results are shown in Table 3. Column 2 shows the names of the DyNN models under evaluation; columns 3 to 6 show the final inconsistent rate (η in Eq.(1)) between the outputs $\mathcal{V}_{post}(\cdot)$ and $C_{post}(\cdot)$, and the last four columns show the maximum numeric error (δ in Eq.(1)) between the outputs $\mathcal{V}_{before}(\cdot)$ and $C_{before}(\cdot)$.

We made the following observations. First, for all the evaluation subjects, DyCL can succesfully compile the original DyNN models for deployment, and the final outputs of the deployed host programs (*i.e.,* the results after post-process) are the same as the final outputs of the original DyNNs. This shows that the DyCL compiled DyNN is semantic-equivalent to the original DyNN, indicating that DyCL can correctly compile the DyNN model. Second, the differences of *maximum numeric error* between the outputs of the compiled DyNN and original DyNN are small, ranging from 0 to $10^{-4.72}$. The differences of numeric errors are within an allowed error range (*i.e.,* PyTorch sets a maximum default error as 10^{-5} for the compilation process). Recall that in Fig. 3, directly applying existing DL

compilers to compile the DyNN models results in the maximum error range $[10^0, 10^4]$. The small differences of maximum numeric error also confirm the correctness of DyCL; otherwise, if there is a difference between the execution path between the original DyNN and the DyCL compiled DyNN, the maximum numeric error will be much larger.

> Answers to **RQ1**: With DyCL, existing "static" DL compilers can successful compile the DyNN models. The numeric error between the original DyNN and the compiled DyNN is minimal and does not affects the final prediction.
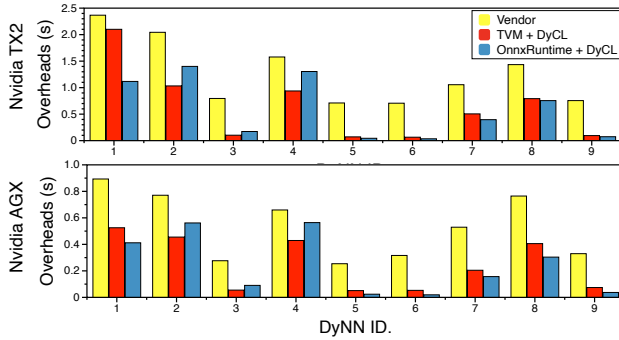
Figure 6: The inference overhead of the original DyNNs and the DyCL compiled DyNNs.

6.3 RQ2: Accelaration

The overheads of the original DyNN model and the compiled DyNN model are shown in Fig. 6, where the x-axis represents the DyNN model ID, and the y-axis shows the average inference overheads in seconds. The different color bars represent different running mechanisms. We observe that for all our experimental settings, the compilation process can accelerate the DyNN models' inference, and the acceleration rate range from 1.12× to 20.21×. Such significant acceleration shows the advantage of compiling the DyNN model for deployment, especially for deploying the DyNN model on resource-constrained platforms. Another interesting observation is that no DL compiler can consistently outperform the others on all experimental subjects, indicating that the optimization strategies in existing DL compilers are complimentary. This result demonstrates the benefit of DyCL's capability of adapting different DL compilers to compile DyNNs.

> Answers to **RQ2**: DyCL compiled DyNN models are consistently faster than the original DyNN models in terms of inference time, across different platforms and DL compilers.

Table 4: The inference overheads of compiled DyNN that are compiled with and without the graph optimization module.

ID.	TVM			ONNX		
	C_{no}	C	Accelerate	C_{no}	C	Accelerate
1	0.530	0.525	0.939	0.417	0.412	1.222
2	0.589	0.455	29.449	0.621	0.562	10.576
3	0.056	0.055	1.325	0.088	0.090	-2.524
4	0.525	0.429	22.297	0.585	0.564	3.685
5	0.058	0.051	13.030	0.026	0.024	10.241
6	0.054	0.053	1.873	0.020	0.019	6.796
7	0.236	0.205	15.482	0.162	0.157	3.593
8	0.407	0.406	0.329	0.311	0.304	2.252
9	0.074	0.074	0.219	0.037	0.037	0.223
Avg	0.281	0.250	9.438	0.252	0.241	4.007

6.4 RQ3: Ablation Study

The results of our ablation study are shown in Table 4. We show the results on Nvidia AGX platform due to the limit of space, more results could be found on our website. Column 1 shows the same subject DyNN IDs as in Table 3. The data in columns C and C_{no} represent the inference time overheads of the programs that are compiled by DyCL with and without the graph optimization module, respectively. We observe that for most settings, the graph optimization module can accelerate the compiled DyNN with an average acceleration rate ranging from 4.007% to 9.438%. Considering that existing DL compilers have almost done extreme optimization on the tensor computation on modern hardware platforms, the acceleration results in Table 4 are significant. This result confirms that our graph optimization strategy can benefit DyCL in compiling dynamic neural networks.

> Answers to **RQ3**: The graph optimization module of DyCL further accelerates the compiled DyNNs, on average 4% to 9% faster than those compiled without the module.

6.5 RQ4: Overheads

Table 5: The overheads of DyCL.

ID.	DyCL (s)		Original Overheads (s)		Extra Percentage (%)	
	Rewriting	Graph Opt	OnnxR	TVM	OnnxR	TVM
1	0.06	9.19	37.18	191.53	24.87	4.83
2	0.04	14.24	56.17	155.84	25.42	9.16
3	0.05	0.28	68.23	374.62	0.48	0.09
4	0.08	1.95	94.35	283.78	2.15	0.71
5	0.14	0.85	59.67	607.27	1.65	0.16
6	0.16	0.76	202.80	594.12	0.45	0.16
7	0.09	8.96	80.85	323.92	11.20	2.80
8	0.06	49.09	219.30	445.60	22.41	11.03
9	0.14	0.85	210.20	746.98	0.47	0.13
Avg	0.091	9.573	114.307	413.740	9.900	3.230

Table 5 shows the overheads of DyCL. The second and third columns show the overheads of the rewriting and the graph optimization module (overheads of other modules are ignorable), the

fourth and fifth columns show the overheads of applying the existing DL compiler to compile all sub-DNNs, and the last two columns show the overhead percentage of DyCL in terms of the total overheads of the compilation.

From the results, we observe that the main overheads of DyCL come from the graph optimization module. This is because each sub-DNN may be a computational graph with hundreds of nodes, thus emulating all paths from the input node to the output node is time-consuming. Moreover, the overheads of DyCL are ignorable when compared with the overheads of applying the DL compiler to compile the sub-DNNs. DyCL occupies only 3.23% to 9.90% percentage of the total overheads of compiling all sub-DNNs.

> Answers to **RQ4**: DyCL is a lightweight approach and does not significantly increase compilation overheads.

7 THREATS TO VALIDITY

Our selection of the DyNN systems, namely, ShallowDeep, SkipNet, AttentionNet, and En-Decoder, *etc.*, might be a threat to the *external validity* of our experimental conclusions. We have tried to alleviate this threat through the following efforts: (1) The DyNNs are very popular, which can be seen through the number of citations of the works (Table 1); (2) the underlying DNN models are state-of-the-art models, which are used significantly; (3) these systems differ from each other in terms of model architecture and functionality. Therefore, our experimental conclusions should generally hold because of the diverse model subjects. Moreover, it is important to address the issue of noisy latency measurements, as it can potentially impact the validity of our experimental conclusions. To tackle this challenge, we conducted multiple latency measurements and recorded both the average and variance values. Our results demonstrate that the variances are significantly smaller than the average value, indicating that the impact of system noise on our experiments is minimal and does not compromise the validity of our findings.

8 RELATED WORK

Dynamic Neural Networks. As discussed in Section 2, Dynamic Neural Networks can be separated into two categories: Energy-saving DyNNs and Generative DyNNs. The Energy-saving DyNNs can be divided into two categories: Conditional-skipping DyNNs and Early-termination DyNN. Among conditional-skipping models, Hua *et al.* [20] and Gao *et al.* [16] explored channel gating to determine computational blind spots for channel-specific regions unessential to classification. Liu *et al.* [31] presented a new type of DyNN that utilizes reinforcement learning to achieve selective execution of neurons. SkipNet [47] used gating techniques to skip residual blocks. On the other hand, Figurnov *et al.* [14] and Surat *et al.* [42] presented SACT and BranchyNet respectively, which are Early-termination DyNNs. SACT terminates the computation within a residual block early based on intermediate outputs, while BranchyNet uses separate exits within network for early termination.

Deep Learning Compilers. Deep learning Compilers have been one of the main focuses of the research community due to the requirement of flexibly deploying ML models on modern hardware platforms [10, 24, 29, 41, 51, 52, 55, 56]. Compilers like Apache TVM [6], Facebook's Glow [11], Intel's nGraph [8], Nvidia's TensorRT [44], Google's XLA [38] and Tensorflow Lite [27] are noteworthy compilers that are widely used to compile deep learning models. These compilers are fed with a Deep Learning model and generate highly optimized code as output. However, these compilers are not able to generate the correctly optimized code that is needed to represent DyNNs, as shown in Section 3. Recently, Nimble [40] proposed a virtual machine (VM)-based compiler that can handle the control flow execution logic and the DNN kernels accordingly. However, such VM-based solution increases the DyNN's inference time overheads. The contributions of Nimble and DyCL are orthogonal, because VM-based compiler can not generate model-persistence DNNs, and it is not feasible to re-design all existing non-VM based DL compilers run on VMs.

9 CONCLUSION

In this work, first, we study the limitation of existing DL compilers to compile Dynamic Neural Networks. The significant inconsistent rate in our study results validates that existing DL compilers cannot handle dynamic neural networks. Then, we propose a program rewriting approach to split the tensor computation and the conditional statements, apply the DL compiler to compile the tensor computation parts and leave the conditional statements to the host program. Based on this idea, we propose DyCL, the first tool that can reuse the existing "static" DL compiler in the context of dynamic neural networks. Our evaluation of nine publicly available DyNN models shows that DyCL can correctly compile DyNN models. Moreover, evaluation results show that DyCL can achieve up to 20× inference time acceleration.

ACKNOWLEDGMENTS

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Simin Chen, Mirazul Haque, Cong Liu, and Wei Yang. 2022. DeepPerform: An Efficient Approach for Performance Testing of Resource-Constrained Neural Networks. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 31:1–31:13. https://doi.org/10.1145/3551349.3561158

[3] Simin Chen, Hamed Khanpour, Cong Liu, and Wei Yang. 2022. Learn to Reverse DNNs from AI Programs Automatically. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, Luc De Raedt (Ed.). ijcai.org, 666–672. https://doi.org/10.24963/ijcai.2022/94

[4] Simin Chen, Cong Liu, Mirazul Haque, Zihe Song, and Wei Yang. 2022. NMTSloth: understanding and testing efficiency degradation of neural machine translation systems. In *Proceedings of the 30th ACM Joint European Software Engineering*

Conference and Symposium on the Foundations of Software Engineering. 1148–1160.

[5] Simin Chen, Shiyi Wei, Cong, and Wei Yang. 2023. Reproduction package for "DyCL: Dynamic Neural Network Compilation Via Program Rewriting and Graph Optimization". (2023). https://doi.org/10.5281/zenodo.7978245

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799* 11 (2018), 20.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[8] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar Vijay, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. *CoRR* abs/1801.08058 (2018). arXiv:1801.08058 http://arxiv.org/abs/1801.08058

[9] Andrew S. Davis and Itamar Arel. 2014. Low-Rank Approximations for Conditional Feedforward Computation in Deep Neural Networks. (2014). http://arxiv.org/abs/1312.4461

[10] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. 2021. IOS: Inter-Operator Scheduler for CNN Acceleration. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org. https://proceedings.mlsys.org/paper/2021/hash/38b3eff8baf56627478ec76a704e9b52-Abstract.html

[11] Facebook. 2018. Facebook Glow. https://ai.facebook.com/tools/glow/

[12] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2021. ETO: accelerating optimization of DNN operators by high-performance tensor program reuse. *Proceedings of the VLDB Endowment* 15, 2 (2021), 183–195.

[13] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. 2021. Cortex: A compiler for recursive deep learning models. *Proceedings of Machine Learning and Systems* 3 (2021), 38–54.

[14] Michael Figurnov, Maxwell D Collins, Yukun Zhu, Li Zhang, Jonathan Huang, Dmitry Vetrov, and Ruslan Salakhutdinov. 2017. Spatially Adaptive Computation Time for Residual Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1039–1048.

[15] Mingfei Gao, Ruichi Yu, Ang Li, Vlad I. Morariu, and Larry S. Davis. 2018. Dynamic Zoom-In Network for Fast Object Detection in Large Images. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 6926–6935. https://doi.org/10.1109/CVPR.2018.00724

[16] Xitong Gao, Yiren Zhao, Lukasz Dudziak, Robert D. Mullins, and Cheng-Zhong Xu. 2019. Dynamic Channel Pruning: Feature Boosting and Suppression. (2019). https://openreview.net/forum?id=BJxh2j0qYm

[17] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. 2021. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[19] Weizhe Hua, Yuan Zhou, Christopher De Sa, Zhiru Zhang, and G Edward Suh. 2018. Channel gating neural networks. *arXiv preprint arXiv:1805.12549* (2018).

[20] Weizhe Hua, Yuan Zhou, Christopher M De Sa, Zhiru Zhang, and G Edward Suh. 2019. Channel gating neural networks. In *Advances in Neural Information Processing Systems*. 1884–1894.

[21] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An introduction to statistical learning*. Vol. 112. Springer.

[22] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. 675–678.

[23] Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. 2019. Shallow-deep networks: Understanding and mitigating network overthinking. In *International Conference on Machine Learning*. PMLR, 3301–3310.

[24] Dimitrios Koutsoukos, Supun Nakandala, Konstantinos Karanasos, Karla Saur, Gustavo Alonso, and Matteo Interlandi. 2021. Tensors: An abstraction for general data processing. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1797–1804.

[25] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727.

[26] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727.

[27] Shuangfeng Li. 2020. Tensorflow lite: On-device machine learning framework. *Journal of Computer Research and Development* 57, 9 (2020), 1839.

[28] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. 2017. Runtime Neural Pruning. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 2181–2191. https://proceedings.neurips.cc/paper/2017/hash/a51fb975227d6640e4fe47854476d133-Abstract.html

[29] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 530–543. https://doi.org/10.1145/3575693.3575707

[30] Lanlan Liu and Jia Deng. 2018. Dynamic Deep Neural Networks: Optimizing Accuracy-Efficiency Trade-Offs by Selective Execution. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 3675–3682. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16291

[31] Lanlan Liu and Jia Deng. 2018. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[32] Steven Solomon Lyubomirsky. 2022. *Compiler and Runtime Techniques for Optimizing Deep Learning Applications*. Ph. D. Dissertation. University of Washington.

[33] Microsoft. [n. d.]. ONNX Runtime: a cross-platform inference and training machine-learning accelerator. https://github.com/microsoft/onnxruntime.

[34] Mahyar Najibi, Bharat Singh, and Larry S Davis. 2019. Autofocus: Efficient multi-scale inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 9745–9755.

[35] Feng Nan and Venkatesh Saligrama. 2017. Adaptive classification for prediction under a budget. *arXiv preprint arXiv:1705.10194* (2017).

[36] Nvidia. [n. d.]. Jetson AGX Xavier Series. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[38] Amit Sabne. 2020. XLA: Compiling Machine Learning for Peak Performance. (2020).

[39] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

[40] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. 2021. Nimble: Efficiently compiling dynamic neural networks for model inference. *Proceedings of Machine Learning and Systems* 3 (2021), 208–222.

[41] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael B. Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure tensor program rewriting via access patterns (representation pearl). In *MAPS@PLDI 2021: Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming, Virtual Event, Canada, 21 June, 2021*, Roopsha Samanta and Isil Dillig (Eds.). ACM, 21–31. https://doi.org/10.1145/3460945.3464953

[42] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. 2016. Branchynet: Fast Inference via Early Exiting from Deep Neural Networks. In *Proceedings of the International Conference on Pattern Recognition*. 2464–2469.

[43] TensorFlow. [n. d.]. TensorFlow Lite. https://www.tensorflow.org/lite.

[44] Han Vanholder. 2016. Efficient inference with tensorrt. In *GPU Technology Conference*, Vol. 1. 2.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[46] Andreas Veit and Serge Belongie. 2018. Convolutional networks with adaptive inference graphs. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 3–18.

[47] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E Gonzalez. 2018. Skipnet: Learning Dynamic Routing in Convolutional Networks. In *Proceedings of the European Conference on Computer Vision*. 409–424.

[48] Ruoyu Wu, Taegyu Kim, Dave Jing Tian, Antonio Bianchi, and Dongyan Xu. 2022. {DnD}: A {Cross-Architecture} Deep Neural Network Decompiler. In *31st USENIX Security Symposium (USENIX Security 22)*. 2135–2152.

[49] Zuxuan Wu, Caiming Xiong, Chih-Yao Ma, Richard Socher, and Larry S Davis. 2019. Adaframe: Adaptive frame selection for fast video recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

1278–1287.

[50] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. 2015. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015 (JMLR Workshop and Conference Proceedings, Vol. 37)*. JMLR.org, 2048–2057. http://proceedings.mlr.press/v37/xuc15.html

[51] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. Sparse-TIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 660–678. https://doi.org/10.1145/3582016.3582047

[52] Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Xulong Tang, Chenchen Liu, and Xiang Chen. 2021. A Survey of Large-Scale Deep Learning Serving System Optimization: Challenges and Opportunities. *CoRR* abs/2111.14247 (2021). arXiv:2111.14247 https://arxiv.org/abs/2111.14247

[53] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. 2022. DietCode: Automatic optimization for dynamic tensor programs. *Proceedings of*

Machine Learning and Systems 4 (2022), 848–863.

[54] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 863–879.

[55] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 874–887. https://doi.org/10.1145/3470496.3527440

[56] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. 2022. {ROLLER}: Fast and Efficient Tensor Compilation for Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 233–248.