SoD^2: Statically Optimizing Dynamic Deep Neural Network Execution

Wei Niu*
wniu@uga.edu
University of Georgia
Athens, GA, USA

Gagan Agrawal
gagrawal@uga.edu
University of Georgia
Athens, GA, USA

Bin Ren
bren@wm.edu
William & Mary
Williamsburg, VA, USA

ABSTRACT

Though many compilation and runtime systems have been developed for DNNs in recent years, the focus has largely been on *static DNNs*. Dynamic DNNs, where tensor shapes and sizes and even the set of operators used are dependent upon the input and/or execution, are becoming common. This paper presents SoD^2, a comprehensive framework for optimizing Dynamic DNNs. The basis of our approach is a classification of common operators that form DNNs, and the use of this classification towards a Rank and Dimension Propagation (RDP) method. This framework statically determines the shapes of operators as known constants, symbolic constants, or operations on these. Next, using RDP we enable a series of optimizations, like fused code generation, execution (order) planning, and even runtime memory allocation plan generation. By evaluating the framework on 10 emerging Dynamic DNNs and comparing it against several existing systems, we demonstrate both reductions in execution latency and memory requirements, with RDP-enabled key optimizations responsible for much of the gains. Our evaluation results show that SoD^2 runs up to 3.9× faster than these systems while saving up to 88% peak memory consumption.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

dynamic neural network, compiler optimization, mobile device

ACM Reference Format:
Wei Niu, Gagan Agrawal, and Bin Ren. 2024. SoD^2: Statically Optimizing Dynamic Deep Neural Network Execution. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3617232.3624869

*This work was primarily done while the author was at William & Mary.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0372-0/24/04…$15.00
https://doi.org/10.1145/3617232.3624869

1 INTRODUCTION

Deep Neural Networks are enabling several of the most exciting and innovative applications that are executed on a variety of computing devices, ranging from servers to edge and mobile devices. From a systems research viewpoint, this had led to a large set of ongoing projects on optimizing DNN inference (and training) tasks [1, 21, 23, 26, 32, 34, 61, 66, 69] as well as tensor compilers [31, 33, 54].

Most of the work on optimizing DNNs considers *static models* that are characterized by the following two properties: 1) input and output shapes and sizes for each layer are known a prior, and 2) the execution path is fixed, i.e., independent of the input. In *dynamic models*, in contrast, one or both of the above two properties are no longer true, and such models are now becoming prevalent. For example, Skipnet [63] decides, based on the input, whether to include or exclude certain operators (or layers). A different form of dynamism seen in transformers for NLP like BERT [13] or cutting-edge computer vision models [29, 55, 56] can take inputs with different shapes and/or apply variable portions of filter kernels during the execution. Consider a commonly used dataset like Wikipedia. The length of input sequences typically varies from 32 to 512 [71], creating significant dynamism in text processing. Similarly, neural networks for image/video processing often deal with images/videos of varying resolutions that dynamically change based on network conditions and player settings. At least three factors have contributed to the popularity of dynamic models and this trend is expected to continue: the need for adapting to computational capacities of different devices, the need for supporting different types of input (e.g. images of different resolutions), and the need for achieving high accuracy for different scenarios.

Dynamic shapes, sizes, and control flow in these models pose many challenges for the optimizations that have been key to obtaining high efficiency. For example, loop fusion [19, 42, 46, 59] cannot be applied [57, 73, 74] if we do not know that the index space of two loops (which likely is the same as the dimensions of respective input tensors) is identical. Planning the execution order [2] to reduce memory requirements or otherwise planning memory allocation [51] is, similarly, not possible if tensor sizes are not statically known.

While many of the existing systems for DNN execution can support dynamic models, they do with high overheads due to very conservative assumptions and/or expensive analyses at the runtime. For example, TFLite [1] and MNN [26] perform re-initialization (equivalent of recompilation) when the input shape to the model changes.

This paper presents the first nuanced approach for optimizing DNN inference in the presence of dynamic features. Our approach emphasizes reducing inference latency as well as memory requirements – the latter being quite important on the mobile devices we

Table 1: Inference overhead for shape dynamism w/ execution re-initialization. SL: shape propagation and layout selection. ST: schedule and tuning. Alloc: memory allocation. Infer: inference time. Experiments are conducted on a Samsung Galaxy S21 w/ MNN [26].

Model	CPU latency (ms)				GPU latency (ms)			
	SL	ST	Alloc	Infer	SL	ST	Alloc	Infer
YOLO-V6 [36]	6.9	1,155	2.2	476	0.8	1,678	30,605	102
Conformer [20]	3.8	127	7.8	926	3	1,021	73,170	1,193
CodeBERT [16]	2.3	253	2.8	370	1	856	4,568	498

target. The foundation of our approach is an in-depth study of operators that form the basis for modern DNNs. These operators are classified into several groups on the basis of how the output shapes relate to input shapes and values. Based on such a classification, we present a data-flow analysis framework, called Rank[1] and Dimension Propagation (RDP) that infers shapes and dimensions of intermediate tensors. RDP analysis considers known constants, symbolic constants, and expressions involving these. RDP analysis results are then used for enabling a number of optimizations, which includes operator fusion and fused code generation, static execution planning, runtime memory allocation, and multi-version code generation. This work integrates RDP and optimizations enabled by it together and builds a comprehensive framework for optimizing Dynamic DNNs, called SoD^2. SoD^2 is extensively evaluated on 10 cutting-edge DNN models with shape dynamism and/or control-flow dynamism. Specifically, these models include the ones for emerging Artificial General Intelligence (AGI) [18] such as StableDiffusion [56] and SegmentAnything [29]. Our evaluation results show that SoD^2 saves 27% to 88% memory consumption and results in 1.7× to 3.9× execution speedup compared with four state-of-the-art product-level DNN execution frameworks (such as ONNX Runtime [12], MNN [26], TVM [5] with Nimble extension [57], and TensorFlow Lite [1]) that support dynamic DNNs.

In all, this paper makes the following contributions. **DNN Operator Classification.** We classify the operators used for modern DNNs (specifically 150 operators used in ONNX (Open Neural Network Exchange)) into 4 categories, which are *Input Shape Determined Output*, *Input Shape Determined Output Shape*, *Input Shape & Value Determined Output Shape*, *Execution Determined Output*. We formally define these operators and explain their significance for inferring ranks and dimensions for the DNNs where the input can be of different sizes and the execution is data dependent.

Data-Flow Analysis for Rank and Dimension Propagation. Building on the operator classification, we have developed a static analysis framework for propagating shape and size information through a computational graph. This framework, called RDP, considers both known and symbolic constants as well as expressions involving these values. Though somewhat similar to the well-known constant propagation analysis [4], our work is different in having transfer functions specific to the operator (types), supporting both backward and forward analyses, and considering not only known and symbolic constants but also expressions involving them.

[1]Rank denotes the number of dimensions in a tensor.

Comprehensive Set of Static and Dynamic Optimizations. Using results from RDP analysis, we enable a series of optimizations. First, we enable code fusion, including generating multiple versions when sufficient static information is not available. Next, we perform execution planning, using the results of RDP to partition the original graph, and further using several heuristics based on RDP output. Finally, we enable runtime plan generation for memory allocation and also generate multiple versions of optimized implementations for individual operators.

2 EXISTING FRAMEWORKS AND LIMITATIONS

Existing DNN inference engines on mobile devices use two common approaches when handling dynamic DNNs.

Static Solutions. Many existing DNN inference engines for mobile platforms (specifically, TFLite [1] and MNN [26]) support dynamic features by extending their static model execution. For handling dynamic input shapes, this involves either execution re-initialization when the input shape changes or, alternatively, conservative (maximum) memory allocation when the input shapes are unknown. To handle dynamic control flow, it typically requires the execution of all possible paths, and stripping out invalid results. Not surprisingly, such simplistic handling of dynamic features incurs significant execution and/or memory overhead. To further illustrate, Table 1 shows a performance study of three models (YOLO-V6 [36], Conformer [20], and CodeBERT [16]) that can take input with dynamic shapes. MNN [26] runs these models on a Samsung Galaxy S21 with execution re-initialization to handle varied input shapes. These results show that the re-initialization usually takes even significantly longer time than the inference itself. This approach might be acceptable for cases where the overhead of re-initialization can be amortized over a number of inference tasks (e.g., certain video processing scenarios). However, many application scenarios (across the image, audio, and language processing) involve continuously changing inputs. An alternative way, as also indicated above, is to conservatively allocate large memory spaces. However, it incurs significant memory wastage, which can limit the ability to execute large models or to do so efficiently, especially on mobile (or edge) devices with limited memory.

Runtime Solutions. TVM (with Nimble extension) [5, 57] improves on the limitations of static solutions by providing a set of optimizations within a virtual machine. An example of this functionality is a *shape function* to infer the output tensor shape and use this information for dynamic memory allocation. However, such functions and the subsequent dynamic memory allocation introduces significant execution overhead.

3 OPERATOR CLASSIFICATION BASED ON DYNAMISM

Our observation is that DNN operators have different dynamism degrees, leading to distinct levels of challenges and opportunities in optimizing them. More specifically, this work categorizes DNN operators into four types: *Input Shape Determined Output*, *Input Shape Determined Output Shape*, *Input Shape & Value Determined Output Shape*, and *Execution Determined Output*. This section gives a formal definition.

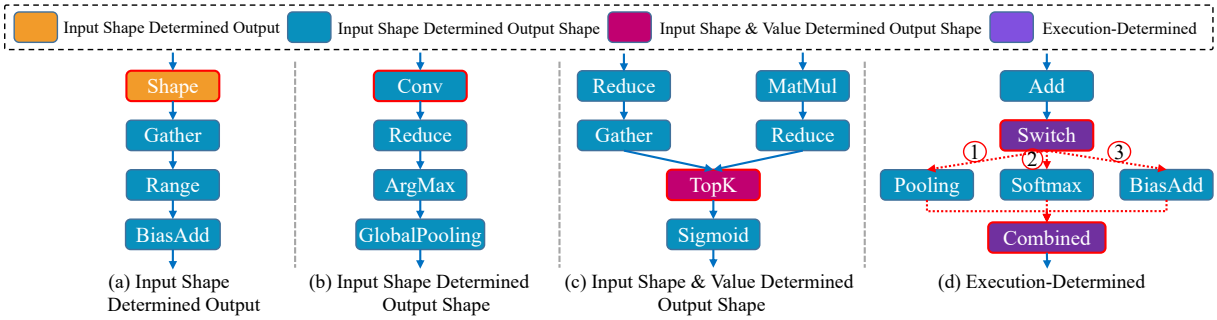


Figure 1: Different degrees of dynamism. Each node is a DNN operator. Yellow, blue, red, and purple mean *Input Shape Determined Output*, *Input Shape Determined Output Shape*, *Input Shape & Value Determined Output Shape*, and *Execution Determined Output*, respectively. In (d), `Switch`'s execution path is decided dynamically during runtime and red dot edges represent both the computation dependency and control flow.

Background and Notation. It is common to represent a DNN as a Computational Graph, which happens to be a Directed Acyclic Graph (DAG). Each tensor (which can be an input and/or output) can be categorized by a shape (including dimensions) and the contents or values. Each operator is denoted as L^l, where l is the operator index. Assume L^l has m input tensors (of which, $1, k$ are constant tensors while k, m are output tensors from previous operators) and n output tensors. The shape of the input tensor i for the l^{th} operator is denoted as IS_i^l and the corresponding tensor value can be denoted as IV_i^l, Similarly, each output tensor's shape and value are OS_i^l and OV_i^l, respectively. Now, intuitively, a class of functions relates the output shapes and values to the input shapes and values – F^{fs} for the shapes and F^v for the values.

- **Input Shape Determined Output:** The output (tensor), which is characterized by both its shape and value, has the following dependence on the input. The output tensor shapes are dependent on the input tensor shapes, whereas the output tensor values are determined by the input tensor shapes and possibly some of the constant tensors – input values do not impact the output. Examples include `Shape` and `EyeLike`. Formally, there is a pair of functions F^{fs}, F^v, such that:

$$OS_i^l \xleftarrow{F^{fs}} IS_1^l, \ldots, IS_m^l$$
$$OV_i^l \xleftarrow{F^v} IS_1^l, \ldots, IS_m^l, IV_1^l, \ldots, IV_{k-1}^l$$

  where $1 \leq k \leq m$.
- **Input Shape Determined Output Shape:** Similar to the previous category, the output shapes depend on the input shapes. However, what is different is that the output values rely on all the input values (including intermediate and constant input values). Examples include `Conv`, `Add`, and `Pooling`. The significance of this category, as compared to the next set of categories, is that if the input shape of this operator is known, compiler optimizations (e.g., operator fusion, execution/memory optimizations) are enabled. Formally, there is a pair of functions F^{fs}, F^v, such that:

$$OS_i^l \xleftarrow{F^{fs}} IS_1^l, \ldots, IS_m^l$$
$$OV_i^l \xleftarrow{F^v} IS_1^l, \ldots, IS_m^l, IV_1^l, \ldots, IV_m^l.$$

- **Input Shape & Value Determined Output Shape:** Similar to the previous category, the output values rely on the input shapes and all the input values. The difference is that the output shapes also rely on partial set of input values. Examples include `Extend` and `Range`). Formally, there is a pair of functions F^{fs}, F^v and a subset of input tensors $(p, \ldots, q)$ whose values specify the output shape, such that:

$$OS_i^l \xleftarrow{F^{fs}} IS_1^l, \ldots, IS_m^l, IV_p^l, \ldots, IV_q^l$$
$$OV_i^l \xleftarrow{F^v} IS_1^l, \ldots, IS_m^l, IV_1^l, \ldots, IV_m^l$$

  , where $1 \leq p \leq q \leq m$. If $p \leq q \leq k$, which is identical to *Input Shape Determined Output Shape*, and all the dependent input tensors are constant. In such cases, the input shapes can be calculated without knowing other intermediate input tensors. If only the input shape of this operator is known, only partial compiler optimizations with conservative analysis can be applied to it, and full optimizations need dynamic execution results.
- **Execution Determined Output:** Similar to the previous two categories, the output values rely on the input shapes and all the input values. Examples include `Nonzero` and `If`. Formally, there is a function F^v, such that:

$$OV_i^l \xleftarrow{F^v} IS_1^l, \ldots, IS_m^l, IV_1^l, \ldots, IV_m^l$$

  , and the shape of i-th output tensor can only be measured after materializing its value:

$$OS_i^l \leftarrow SHAPE\_OF\, OV_i^l$$

  , which means it is not able to know the output shapes until materializing the output tensors (i.e., after executing the layer). Only partial optimization with conservative analysis can be applied to this operator, and full optimizations need dynamic execution results.

Although these operator types are defined according to *forward transfer* , i.e. an output tensor shape and value are related to the input tensor shape and/or value. In practice, *Backward transfer* is also used, i.e., we can (and need to) backward propagate the known output shapes (either rank or dimension or both) to the unknown input shapes. For instance, if we know the output shape of `Add`, its input dimension might be 1 or identical to the corresponding output dimension due to broadcasting rules [11]. We define backward transfer functions as:

$$IS_i^l \xleftarrow{F^{bs}} OS_1^l, \ldots, OS_n^l.$$

Table 2 shows typical operators in ONNX [48] categorized by the above classification. As further illustration, Figure 1 shows four sub-graphs that represent operators with different dynamism degrees (marked with red boundary) and their connections. Figure 1 (a) shows an *Input Shape Determined Output* operator `Shape`. Once its input shape is known, its value result can be directly inferred (and in

Table 2: Classification of DNN operators based on dynamism degrees. Operators are from ONNX (Open Neural Network Exchange) [48].

Operator type	Operators	Representative
Input Shape Determined Output	`Shape, ConstantOfShape, Eyelike`	`Shape`
Input Shape Determined Output Shape	`Add, AveragePool, Cast, Concat, Conv, Elementwise w/ broadcast, Gather, MatMul, MaxPool, Reduce, Relu, Round, Sigmoid, Softmax`	`Conv, MatMul`
Input Shape & Value Determined Output Shape	`Expand, GroupNormalization, MaxUnpool, Onehot, Range, Reshape, Resize, Slice, TopK, Upsample`	`Reshape, Range`
Execution Determined Output	`If, Loop, NMS, Nonzero, <Switch, Combine>`†	`If, Loop`

† `<Switch, Combine>` is a pair of customized operators for dynamic control flow that is not defined in ONNX.

fact, this value can be propagated from `Shape` to `BiasAdd` because all following operators belong to the *Input Shape Determined Output Shape* group). Similarly, Figure 1 (b) implies that if the input shape to `Conv` is known, this shape information could be propagated to the entire sub-graph because all operators in this sub-graph belong to the *Input Shape Determined Output Shape* group. For the cases represented in both (a) and (b), even if the exact shape is unknown, it is still possible for us to perform compiler optimizations such as operator fusion and fused code generation, execution order optimization, and memory optimization, which will be elaborated in the next Section. In Figure 1 (c), the output shape of `TopK` depends on its input value (which is the left predecessor's branch in the example), i.e., the output shape of `TopK` (and its successors) is unknown until its left predecessor branch is executed. Figure 1 (d) represents a sub-graph involving a dynamic control flow. `Switch` results decide if path ①, ②, and/or ③ will be taken, and `Combine` merges the results from executed paths. Both (c) and (d) require dynamic execution, thus is more difficult to optimize statically.

Discussion. Although the examples in Table 2 and Figure 1 mentioned above simply classify each operator into one category, there are additional considerations. For example, an `Upsample` operator may belong to either *Input Shape Determined Output Shape* or *Input Shape & Value Determined Output Shape* depending on whether some of the input tensors are constant or not. Therefore, with constant propagation, an operator may transform from a more dynamic classification to a less dynamic one, offering us more aggressive optimization opportunities. This has motivated certain aspects of SoD^2. In addition, because our operator classification essentially models the dynamism degree of an operator by studying its computation logic and input/output tensor shapes and values, it is possible to create an automatic and generic tool based on existing intermediate representations like tensor expression (e.g., TVM expression) to categorize operators into different classifications.

4 DESIGN OF SOD^2

Based on the DNN operator classification introduced above, SoD^2 introduces a new static data-flow analysis framework to infer the intermediate result tensor shape. Such an analysis is the enabler of several optimizations, which are dynamic DNN operator fusion, execution path planning, memory planning, and multi-version code generation. All of these optimizations ensure a deterministic running sequence and a consistent output, given a particular input. At a high level, our approach does not require conservative static assumptions

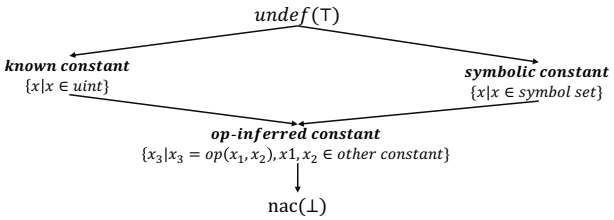


Figure 2: Domain of RDP dataflow analysis. It includes known, symbolic, and operation-inferred constants that form a lattice.

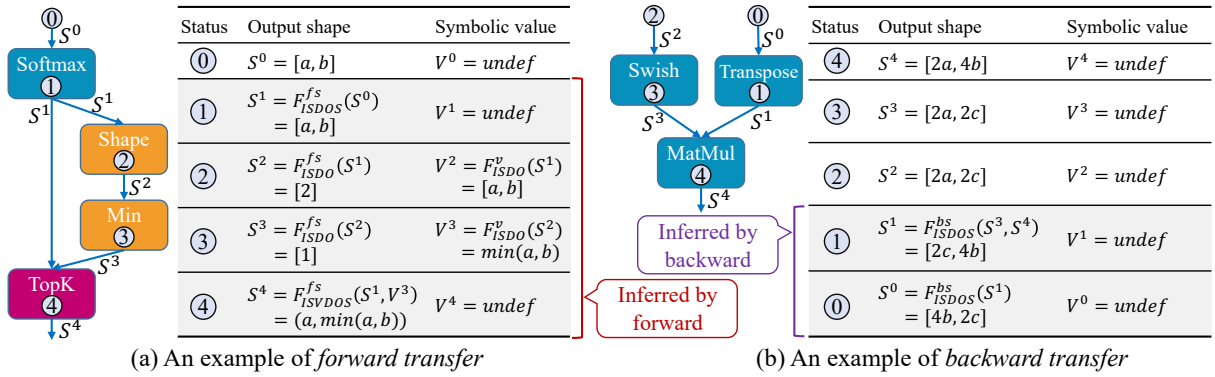
or runtime overheads, thus providing significant improvement over the existing state-of-the-art.

4.1 Pre-Deployment Data-Flow Analysis

To facilitate static optimizations for dynamic DNNs, a critical requirement is knowing (possibly symbolically) the intermediate result tensor shape (i.e., rank and dimension). Our key observation is that *for many operators and operator combinations (e.g., an Input Shape Determined Output operator and an Input Shape Determined Output Shape operator), even without knowing the input tensor shape, it is still possible to infer the shape of the intermediate result tensor to a certain degree.* Our framework is based on this observation and is called operator Rank and Dimension Propagation, or RDP. While RDP has certain similarities with the classical (symbolic) constant propagation frameworks [4], it needs to deal with nuances of the DNN operations and the computational graph. RDP also considers operations over multiple (symbolic) constants as a possibility in its lattice and requires iterative forward and backward analysis.

Formal Definition of Operator Rank and Dimension Propagation (RDP).. The entire RDP algorithm is expressed as a four-tuple $< G, D, L', F >$.

- G is an *extended computational graph* (a DAG), with control-flow operators <Switch, Combine>). If this extended computational graph involves multiple branches that all need to be executed, we assume the execution order is always from left to right. It is easy to prove that G is equivalent to a control-flow graph on operators, which serves as the foundation of this data-flow analysis.
- D is the direction of the data flow, which can be FORWARD and BACKWARD. Unlike most classic data-flow formulations, e.g., constant propagation or reaching definitions, RDP iteratively processes G in forward and backward directions



(a) An example of *forward transfer* (b) An example of *backward transfer*

Figure 3: Examples of forward and backward transfer. Each node is an operator. Yellow, blue, and red mean *Input Shape Determined Output*, *Input Shape Determined Output Shape*, and *Input Shape & Value Determined Output Shape*, respectively. Ids (e.g., ①) indicate the location where transfer functions apply and their applying orders for a forward transfer (a backward transfer reverses this order). S and V equations map values in the RDP domain to the shape and value of each tensor, in which, F denotes the transfer function. *fs* and *bs* of F denote forward and backward, and F's subscript is a short form of its type (e.g., *ISDOS* means *Input Shape Determined Output Shape*).

until the results converge. This is because the shape of a tensor could be inferred from its producing operator and/or consuming operator, and their inference results should be the same to guarantee the correctness of this DNN execution.

- L' itself is a three-tuple $< V', \wedge, m >$. V' is the domain of values (also shown in Figure 2) and includes known constants, symbolic constants, and operation-inferred constants that form a lattice. The lattice also includes undefined ($undef$) as the top ($\top$) of the lattice and Bottom ($\bot$) which is not-a-constant (*nac*). $\wedge$ is a meet operator, which follows the common definition for product lattice. m is a map function mapping values in lattice to two variables, Shape (S) and Value (V), representing these for the intermediate tensor. More specifically, RDP is a type of data-flow analysis, where L′ describes its analysis scope, i.e., how to map each shape and value property to a kind of constant that forms a lattice structure (in Figure 2). This lattice guarantees that the analyzed properties of RDP follow lattice theory, so RDP analysis will converge with a unique solution.
- $F : V' \rightarrow V'$ is the domain for transfer functions. F is designed for each operator (type) and transfers the Shape (S) and Value (V) from the input tensor to the output tensor based on the operator type. Similar to data-flow analysis for *constant propagation* RDP has two kinds of transfer functions, `Update`, and `Merge`. `Update` transfers from the input tensor to the output tensor for an individual operator; while `Merge` operates on branch control flow and merges (output) tensors from multiple possible execution paths. Because RDP has both FORWARD and BACKWARD directions, F also contains transfer functions for both directions.

Transfer Function Examples. SoD² contains 16 types of `Update` transfer functions. These functions are based on the classification of the operator's dynamism degree, as detailed in Table 3. The table includes four dynamic degrees, covering two directions (forward and backward) and two types of propagation (shape and value). During forward transfer, each operator employs its shape and value transfer functions in accordance with its associated dynamism degree to infer the shape (e.g., F^{fs}_{ISDO}) and value (e.g., F^{fv}_{ISDO}) of its output tensor, respectively. Similarly, F^{bs}_{ISDO} and F^{bv}_{ISDO} serve as two examples of a backward transfer function for shape and value during the backward transfer process. Figure 3 illustrates several common ones. The left-hand side (Figure 3 (a)) shows an example with four forward transfers that employ three types of `Update` transfer functions. Similarly, the right-hand side (Figure 3 (b)) shows an example with two backward transfer functions that belong to the same type. A point worth noting is that the appropriate transfer function to apply to an operator depends not only on the computational graph but also on the constants inferred during the RDP analysis process, which determines the dynamism classification of the operator. The `Merge` transfer function is straightforward – it merges the S-map and V-map from multiple control-flow branches based on the lattice in Figure 2. Table 4 summarizes the key components of RDP.

Table 3: Illustration of forward and backward transfer functions for different operator types.

Type	Forward		Backward	
	Shape	Value	Shape	Value
Input Shape Determined Output	F^{fs}_{ISDO}	F^{fv}_{ISDO}	F^{bs}_{ISDO}	F^{bv}_{ISDO}
Input Shape Determined Output Shape	F^{fs}_{ISDOS}	F^{fv}_{ISDOS}	F^{bs}_{ISDOS}	F^{bv}_{ISDOS}
Input Shape & Value Determined Output Shape	F^{fs}_{ISVDOS}	F^{fv}_{ISVDOS}	F^{bs}_{ISVDOS}	F^{bv}_{ISVDOS}
Execution Determined Output	F^{fs}_{EDO}	F^{fv}_{EDO}	F^{bs}_{EDO}	F^{bv}_{EDO}

Table 4: Definition of Rank and Dimensions Propagation (RDP).

Notation	Definition	Notation	Definition
Domain	Tensor Rank and Dimensions	Direction	*Forward, Backward*
Forward	$OUTL = F^{fs}_{P \in predL} P$	Backward	$INL := F^{bs} OUTL$
Initial	$OUTL = undef$	Terminate	No more changes

Algorithm 1: RDP's Optimized Chaos Algorithm

```
1  foreach node in ecg.sorted_node do
2      mark_as_undef(node) /* Initialize as undef */
3  set_model_input_shape(ecg)
4  do
5      changed ← false
       /* Traverse the Depth-first sorted nodes */
6      foreach node in ecg.sorted_node do
7          predecessors ← predecessor_of(node)
8          successors ← successor_of(node)
9          if node.type is Combine then
               /* Merge the rank and dims for Combine */
10             changed |= node.merge(predecessors)
11         if node.type is Switch or Combine then
12             continue /* Transit to all successors */
           /* 1 Forward transfer to current node */
13         changed |= forward_transfer(node, predecessors)
           /* 2 Backward transfer to predecessors */
14         foreach pred in predecessors do
15             changed |= backward_transfer(node, pred)
           /* 3 Update for Input Shape Determined Output */
16         if node.type ∈ Input Shape Determined Output then
17             if node.shape ∉ undef,nac then
18                 node.value ← get_symbolic_value(node.shape)
19 while changed

   Func forward_transfer: node, preds
20 if all(node.outputs.shape ∉ undef) then
21     return False /* Outputs are not in undef */
22 pred_shapes, pred_values ← shape_of(preds), value_of(preds) switch
     node.op_type do
23     case 'Input Shape Determined Output' do
           /* Only depends on the first input shape */
24         return FT_ISDO(node, pred_values[0])
25     case 'Input Shape Determined Output Shape' do
26         return FT_ISDOS(node, pred_shapes)
27     case 'Input Shape & Value Dependent Output Shape' do
28         return FT_ISVDOS(node, pred_shapes, pred_values)
29     case 'Execution-Determined' do
           /* Assign nac */
30 return False

   Func backward_transfer: node, pred
31 if all(pred.outputs.shape ∉ undef) then
32     return False /* Outputs are not in undef */
   /* Similar to forward_transfer */
```

RDP Solution. The method is shown as Alg. 1 and involves applying the transfer functions (F) to the extended computational graph (G) along the two directions iteratively. Elaborating on Alg. 1, it first sorts the nodes (i.e., operators) in the computational graph G with the dept-first order and initializes the output shape- and value-maps of each node as *undef* (Line 1 to Line 2). It next processes each node (n) by applying forward transfer functions to n's predecessors' output shape- and value-maps (i.e., n's input shape- and value-maps) (Line 13). Moreover, it propagates n's output shape- and value-maps to n's predecessors' output shape- and value-maps by backward transfer functions if any predecessors have *undef* analysis results (Line 14 to Line 15). These forward and backward transfer functions are defined based on the dynamism classification of DNN operators (as shown in Line 20 to Line 32). Alg. 1 needs to process two specific types of nodes (operators): i) control-flow nodes (like `Combine` or `Switch`), for which, it needs to call the `Merge` function to merge analysis results from multiple control-flow paths (Line 9 to Line 10), and ii) Input Shape Determined Output nodes, for which, it assigns a symbolic constant to the value map to facilitate subsequent analysis (Lines 16 to 18). Alg. 1 continues processing nodes in G until no updates happen on any node's shape-/value-maps. Similarly to other data-flow analysis, RDP follows *Lattice Theory* [28], so an optimized chaos implementation (based on worklist) is guaranteed to converge.

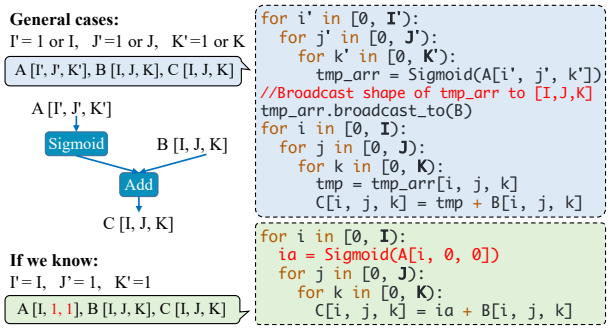


Figure 4: Operator fusion with dynamic shapes. The top code snippet shows that fusion is not feasible because of broadcasting [11]. Specifically, *Add* requires A's indices I', J', and K' to be either 1 or I, J, and K, resulting in 8 fusion scenarios. With RDP, such fusion is feasible (shown in the below code snippet). This fusion significantly reduces intermediate result materialization requirements.

4.2 Operator Fusion for Dynamic DNN based on RDP

Though fusion has been a successful optimization on DNNs [46], it is also known to be very hard to implement on dynamic DNNs [57]. A frequent issue is that without knowing the tensor shape of two operators, the DNN compiler either cannot fuse them at all or has to generate a large number of code versions, each for a possible combination of shapes for the two operators. In fact, as often more than two operators are merged, the possible combinations for which separate code should be generated increase rapidly. Our proposed RDP analysis can address this issue by using (possibly symbolic) shape information. Information such as the two operators having tensors of the same shape can enable and/or simplify fusion, even if the exact dimensions are not known till runtime.

Figure 4 shows a simplified example with two common DNN operators (`Sigmod` and `Add`) on tensors with shapes not known till runtime. `Sigmod` takes an input tensor A with a dynamic shape of [I', J', K']. `Add` performs an element-wise addition on `Sigmod`'s output and another input tensor B, whose shape happens to be [I, J, K]. Now, if A and B are of different shapes, a shape broadcast operation on the output tensor of `Sigmod` needs to be conducted immediately before the element-wise addition. Without our RDP analysis, the dynamic shape of A and B (and the possible shape broadcast operation) prevents the DNN compiler from fusing these two operators in an efficient way, i.e., the compiler either generates code without fusion (as shown in the blue box of Figure 4), or generates multiple code

versions (8 versions for this example) and selects a version during the runtime. Assuming our RDP analysis result is I' = I, J' = 1, and K' = 1, i.e., a mix of symbolic constant (I) and known constant, the DNN compiler can further generate a unique version of fused code (as shown in the green box of Figure 4). SoD^2 incorporates RDP and the above operator fusion based on RDP into a state-of-the-art operator fusion for static DNNs (DNNFusion [46]) to generate the fusion plan and optimized fused code.

4.3 Static Execution Planning based on RDP

A computational graph (DAG) typically allows for several different orderings for the execution of operators. The choice of ordering has an impact on the peak memory usage (for intermediate results), which further has consequences for cache performance and the execution latency. There has been previous work on this problem, which has in fact shown that generating an *optimal* execution plan (by a metric like memory consumption) is an NP-complete problem [2]. Thus, choosing an optimal plan can be difficult for modern large DNNs with hundreds or even thousands of operators.

The dynamic properties (e.g., dynamic shapes and control flow) further complicate this problem. In SoD^2, we develop a series of heuristics driven by the use of proposed RDP analysis. The overall idea is that since a globally optimal solution is almost infeasible, an approach based on *graph partitioning* is justified. It turns out that the results of RDP are able to guide both graph partitioning and choice of solution within each sub-graph. Particularly, we observe that known constants, symbolic constants, op-inferred constants, and $\perp$ or *nac* progressively increase the impediment on the generation of an optimal execution plan. More specifically, for a sub-graph *sg* with a limited number of operators:

First, if the shape of all tensors in *sg* are known constants, the optimal execution plan for *sg* can be obtained statically by an exhaustive search – a limited size of *sg* can further make such a search feasible. Second, if the shape of tensors in *sg* are mixed known constants, symbolic constants, and op-inferred constants, it is still possible to compare the memory requirements and thus generate a (close to) optimal execution plan. This is especially true if these shapes are derived from the same set of symbolic constants. Third, if an operator has an *nac* output tensor shape, it disables further analysis and execution planning. Such operators, it turns out, provide an opportunity to partition the original graph into sub-graphs that can be independently analyzed.

4.4 Other Optimizations

4.4.1 Memory Allocation Plan. Besides execution (order) planning, *memory planning* of DNNs is also a critical step [2, 51]. A *memory allocation plan*, which decides where in a linear memory space each intermediate tensor is allocated, and when it is deallocated, can restrict peak memory usage and improve execution performance – the latter by reducing memory fragmentation, avoiding memory movement, and limiting memory allocation/de-allocation. In contrast to execution planning that (even for dynamic DNNs) can be carried out at compilation time, memory planning for dynamic DNNs can only be performed at execution time when all tensor sizes are known. Memory planning of static DNN execution has also been proved NP-complete [2], while DNN model dynamism further complicates memory planning.

Existing memory planning methods for dynamic DNN execution (e.g., Nimble [57]) have addressed this. Without knowing the exact tensor shapes, the methods usually rely on a greedy strategy [51], (e.g., finding the minimal memory slot currently available that can hold the new tensor). In comparison, we use RDP results and the following two key insights. First, we base our approach on sub-graphs generated by our static execution planning method. It turns out that for sub-graphs with known constant shapes, as well as those with symbolic/op-inferred constant shapes that are defined solely by the input tensor of the sub-graph, the peak memory requirement can be inferred from static RDP analysis results and subsequent execution plan generation.

Second, we have observed that for most sub-graphs, the memory requirement decreases monotonically in both forward and backward directions from the location in the graph with peak memory usage. Therefore, initiating memory planning from the peak memory consumption location and traversing in the forward and backward directions, and picking the available memory slots for reuse works as a good strategy, and does not lead to the need for extra memory space.

Based on these insights, a lightweight greedy approach that starts from the peak memory requirement location can help to find optimal memory usage for many/most sub-graphs. Our evaluation (details omitted because of space limits) on ConvNet-AIG [62] shows that our RDP-based memory allocation plan requires 1.05× of optimal peak memory consumption (that results from an exhaustive search); while the one based on the greedy strategy mentioned above (MNN) requires 1.16× of optimal peak memory consumption.

4.4.2 RDP-based Multi-Version Code Generation. As we discussed in Section 4.2, RDP analysis enables and/or simplifies operator fusion by revealing (possibly symbolically) tensor shapes. In cases where a single (fused) version is not feasible, one of the advantages of the information obtained through RDP is that the number of different versions of the fused code generated can be reduced significantly.

SoD^2 further benefits from this property of RDP by generating *multi-version code* to optimize *hotspot operators* (e.g., CONV and GEMM) that dominate the DNN execution. Prior efforts [1, 26] have shown that the optimization opportunities for these operators depend on the shapes and sizes of the input/output tensors. Therefore, for static DNN executions, existing frameworks (such as TensorFlow Lite [1] and MNN [26]) usually employ multi-version codes that involve different optimizations (e.g., tiling, unrolling, choice of the number of thread blocks, etc.). However, this optimization is challenging for dynamic DNNs because an unknown tensor shape and/or tensor size implies that too many versions will be needed. The tensor shape (or shape relations) provided by RDP help to generate code for more specific tensor shapes only, thus resulting in fewer code versions.

More specifically, SoD^2 relies on an auto-tuner based on Genetic Algorithm to generate the exploration space (e.g., tiling shapes, loop permutation, and unrolling settings) for kernel code generation as DNNFusion [46]. One feature of this auto-tuner is the more effective exploitation of parallelism available in the hardware. To tackle

Table 5: Memory consumption (allocated for intermediate results) for ONNX Runtime, MNN, TVM with Nimble extension (TVM-N), and SoD^2 on a mobile CPU. "-" means this model is not supported by the framework yet. "S" stands for shape dynamism, and "C" represents for control-flow dynamism.

Model	#Layers	Model Size (MB)	Dynamism	Input Type	ORT (MB)		MNN (MB)		TVM-N(MB)		**SoD^2 (MB)**	
					Min	Max	Min	Max	Min	Max	Min	Max
StableDiffusion [56]	407	137	S	Text + Image	186	342	124	376	-	-	92	271
SegmentAnything [29]	857	17	S	Text + Image	-	-	-	-	-	-	16	22
Conformer [20]	1,703	303	S	Audio	-	-	61	78	-	-	49	58
CodeBERT [16]	985	502	S	Text	32	75	25	54	-	-	21	41
YOLO-V6 [36]	599	239	S	Image	288	430	148	404	964	1,103	89	206
SkipNet [63]	549	103	S + C	Image	168	597	27	124	522	700	18	86
DGNet [37]	847	91	C	Image	37	37	76	76	-	-	23	29
ConvNet-AIG [62]	282	104	S + C	Image	168	423	33	109	557	646	26	77
RaNet [68]	2,617	525	S + C	Image	675	1275	166	675	-	-	86	452
BlockDrop [65]	439	179	S + C	Image	242	460	35	105	523	723	24	69
Geo-mean memory consumption normalized by SoD^2 *					**3.64**×		**1.37**×		**8.62**×		1	

* This normalized geo-mean memory consumption is calculated by 1) averaging the memory usage of runs with all input samples for each model, 2) calculating the geo-mean of the average memory usage of all models, and 3) normalizing with SoD^2's geo-mean memory usage.

the challenge of dynamic shapes, SoD^2 employs a multi-version approach, where the versions are chosen based on empirical evidence relating to the impact of different shapes on performance. For instance, our auto-tuner considers fat, regular, and skinny matrices for both GEMM and CONV kernels.

5 EVALUATION

SoD^2 is implemented by extending an existing DNN execution framework (DNNFusion [46]) that supports static DNN execution only. This section evaluates the performance of SoD^2 by comparing it with four state-of-the-art frameworks. These frameworks are ONNX Runtime (ORT) [12] (V1.14.1), MNN [26] (Vdcb080c), TVM [5] w/ Nimble extension (TVM-N) [57] (V7831a79), and TFLite [1] (V2.11.1). ORT, MNN, and TVM-N support shape dynamism, while for DNNs with control flow, they execute all possible branches and strip out invalid ones. For fairness, this section also shows a performance comparison between SoD^2 and MNN by disabling SoD^2's <*Combine, Switch*> control-flow support and adopting the same "execute-all, strip-out-invalid" strategy. TFLite supports dynamic input shapes with memory re-initialization; however, it cannot run most of our dynamic models properly because it usually fails on some input shapes. It does not support dynamic control flow either as required by most of the models we target. Thus, we use TFLite as a baseline for comparing DNN executions with fixed inputs and paths only.

Our evaluation has four objectives: 1) demonstrating that SoD^2 outperforms other frameworks with respect to both memory requirements and execution latency (Section 5.2), 2) studying the performance effect of our key optimizations based on RDP (Section 5.3), 3) further confirming the performance advantage of SoD^2 by evaluating it under different situations (Section 5.4), and 4) showing that SoD^2 performs well on different mobile platforms (i.e., SoD^2 has good portability).

5.1 Evaluation Setup

Dynamic Models and Datasets. Our evaluation is conducted on three types of dynamic models: 1) models with shape dynamism, 2) models with control-flow dynamism, and 3) models with both shape and control-flow dynamism. The first category comprises five cutting-edge DNN models, which are StableDiffusion [56] (covering the Encoder part, referred to as SDE), SegmentAnything [29], Conformer [20], CodeBERT [16], and YOLO-V6 [36] (referred to as YL-V6). The second category includes DGNet [37]. The third category consists of four models, including SkipNet [63] (referred to as SNet), ConvNet-AIG [62] (referred to as CNet), RaNet [68], and BlockDrop [65] (referred to as BDrop).

Table 5 characterizes these models by showing the nature of dynamism, target input types, model size, and the total number of layers. Because the choice of training datasets has a negligible impact on the final inference latency or memory consumption (since the model size and structure are the same), this section reports results from one training dataset for each model. StableDiffusion-Encoder, SkipNet, DGNet, ConvNet-AIG, RaNet, and BlockDrop are trained on ImageNet dataset [9]; YOLO-V6 is trained on MS COCO dataset [40]; SegmentAnything is trained on SA-1B dataset [29]; CodeBERT is pre-trained on [10]; and finally, Conformer is trained on Librispeech dataset [49]. Since the model accuracy is the same across all frameworks, our evaluation focuses only on execution time and memory consumption.

Test Samples and Setup. Our inference performance evaluation randomly selects 50 input samples from the corresponding validation dataset for each model. Specifically, for models that take images as input, i.e., YOLO-V6, SkipNet, ConvNet-AIG, RaNet, and BlockDrop, our evaluation randomly selects 50 input images from the ImageNet dataset, with the size of dimensions ranging from 224 to 640. DGNet does not support dynamic input shapes, but it does support dynamic control flow. Therefore, we only tested images with a dimension of 224 for DGNet. As YOLO-V6 only accepts images with dimensions that are multiples of 32, only a subset of inputs could be used. For StableDiffusion-Encoder and SegmentAnything, the 50 randomly selected input images have dimensions ranging from 64 to 224. For CodeBERT and Conformer, our evaluation randomly selects 50 input samples with sequential lengths ranging from 32 to 384.

Table 6: End-to-end execution latency comparison among ONNX Runtime, MNN, TVM-N, and SoD^2 on mobile CPU and mobile GPU. "-" means this model is not supported by the framework yet.

Model	ORT (ms)				MNN (ms)				TVM-N (ms)				**SoD^2 (ms)**			
	CPU		GPU		CPU		GPU		CPU		GPU		CPU		GPU	
	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
StableDiffusion [56]	179	2,115	217	2,076	189	1,287	159	1,252	-	-	-	-	152	733	105	530
SegmentAnything [29]	-	-	-	-	-	-	-	-	-	-	-	-	66	108	42	71
Conformer [20]	-	-	-	-	51	300	265	498	-	-	-	-	40	225	35	150
CodeBERT [16]	141	752	-	-	125	1,265	-	-	-	-	-	-	102	452	72	366
YOLO-V6 [36]	174	1,386	155	733	168	925	47	287	251	2,108	-	-	118	546	33	178
SkipNet [63]	111	841	-	-	92	789	116	363	109	974	-	-	41	633	29	253
DGNet [37]	122	122	127	127	67	67	211	211	-	-	-	-	32	56	23	42
ConvNet-AIG [62]	90	693	-	-	88	731	96	305	98	947	-	-	46	526	22	203
RaNet [68]	102	641	-	-	139	663	114	208	-	-	-	-	63	403	31	150
BlockDrop [65]	153	1,199	-	-	145	1,421	139	468	185	1,622	-	-	79	668	42	295
Geo-mean latency*	**2.5**×		**3.9**×		**1.7**×		**2.3**×		**2.7**×		-		1		1	

* This normalized geo-mean execution latency is calculated by 1) averaging the execution latency of runs with all input samples for each model, 2) calculating the geo-mean of the average execution latency of all models, and 3) normalizing with SoD^2's geo-mean execution latency.

The experiments are performed on a Samsung Galaxy S21 smartphone powered by a Snapdragon 888 processor [53]. This processor features an octa-core Kryo 680 CPU, comprising one large core, three middle cores, and four small cores, and a Qualcomm Adreno 660 GPU with 1024 ALUs. Additionally, to demonstrate the portability of our approach, SoD^2 is also tested on an earlier generation of Snapdragon platform with more constrained resources, specifically the Snapdragon 835 [52] equipped with a Qualcomm Kryo 280 octa-core CPU, consisting of four middle cores and four small cores, and a Qualcomm Adreno 540 GPU with 384 ALUs. Our evaluation employs 8 threads on mobile CPUs and pipelined execution on mobile GPUs. The GPU execution uses a 16-bit floating-point representation, while the CPU execution uses a 32-bit floating-point representation. Each experiment is executed 50 times and only the average numbers are reported – as the variance was negligible, it is not reported for readability.

5.2 Overall Comparison

This section focuses on the end-to-end memory reduction and execution latency gains of SoD^2.

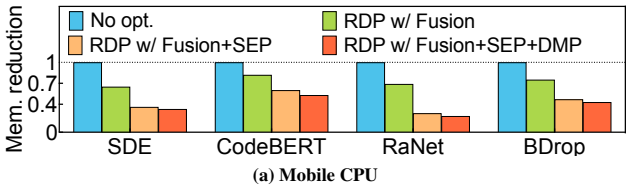
Overall Memory Consumption Comparison. Table 5 presents a comparison of end-to-end memory consumption on a mobile CPU using SoD^2, ONNX Runtime (ORT), MNN, and TVM with Nimble extension (TVM-N). As the results on mobile GPU show a similar trend, they are not included here. '-' implies that a model is not supported by a given framework. The 'Min' and 'Max' columns indicate the minimum and maximum memory consumption (excluding the memory for holding the model itself because this part is the same for all frameworks). The last row of the table shows the geometric mean memory consumption of each framework normalized by SoD^2. Its detailed calculation method is shown below the table and is over the cases where execution is possible. Among other frameworks, only MNN can support Conformer. SegmentAnything is not supported by other frameworks as either certain key operators are missing, and/or there are limitations in optimization, leading to large model execution footprints.

Table 7: Latency impact of input distribution on YOLO-V6. Each cell shows the latency speedup of SoD^2 over a corresponding baseline of ORT, MMN, or TVM-N.

Model	1th	25th	50th	75th	100th
ORT	1.43×	1.66×	1.95×	2.33×	2.52×
MNN	1.41×	1.44×	1.50×	1.58×	1.65×
TVM-N	2.13×	2.52×	3.03×	3.67×	3.90×

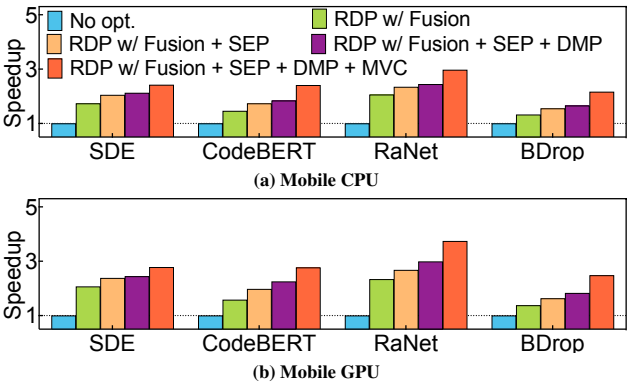
Compared with other frameworks, SoD^2 has significantly lower memory consumption. Specifically, ORT, MNN, and TVM-N need to use 3.64×, 1.37×, and 8.62× memory, respectively, over SoD^2. SoD^2 results in a greater reduction in memory consumption for image models (compared to other models) because image models generally have larger memory footprints, allowing for more significant optimization opportunities. It is worth noticing that TVM-N executes models as its own Android RPC application, which is one of the causes of higher memory requirements.

Overall Latency Comparison. Table 6 presents a comparison of end-to-end latency for SoD^2 against other frameworks on both mobile CPU and GPU. The table includes the minimum and maximum latency observed across different input samples for each model. On mobile CPU, SoD^2 achieves an average speedup of 2.5×, 1.7×, and 2.7× compared to ONNX Runtime, MNN, and TVM-N, respectively. TVM-N does not support dynamic models on a mobile GPU. Compared against the other two frameworks on mobile GPU, SoD^2 achieves a speedup of 3.9× and 2.3× over ORT and MNN, respectively. Notably, the minimum latency achieved by SoD^2 on mobile GPU is significantly lower than other frameworks for ConvNet-AIG, RaNet, and BlockDrop models. This is because our optimizations can handle different cases and mitigate the effect of execution path variations. It is worth pointing out that the distribution of inputs could impact results. However, it does not change our conclusion. To show this impact more explicitly, we conduct a set of experiments on YOLO-V6 by selecting 50 input samples from different percentiles ranging from 1st to 100th, and our results are as shown in Table 7.



(a) Mobile CPU

Figure 5: Memory reduction of different optimizations on CPU. Over the baseline w/o any RDP-enabled optimization (No opt.)



(a) Mobile CPU

(b) Mobile GPU

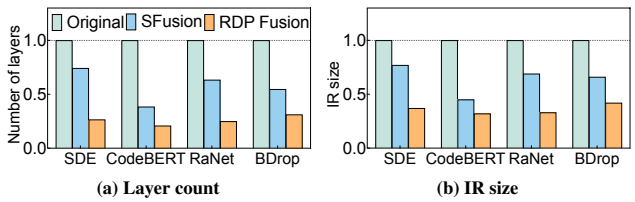
Figure 6: Execution speedup of different opt. on CPU and GPU. Over the baseline w/o any RDP-enabled optimization (No opt.)

5.3 Optimization Breakdown Analysis

This section studies the individual impact of the key optimizations in SoD^2 on both memory consumption and latency.

Memory Reduction w/ Different Optimizations. Figure 5 evaluates the memory reduction achieved through different optimizations for 4 models (StableDiffusion-Encoder, CodeBERT, RaNet, and BlockDrop), including RDP-enabled operator fusion (`Fusion`), static execution planning (`SEP`), and dynamic memory planning (`DMP`). The results for other models exhibit a similar trend and are excluded due to space limitations. The baseline version is referred to as `No opt` – despite the name, it includes general static optimizations, such as static operator fusion and constant folding. Building on this version, we study the benefits of optimizations enabled by RDP analysis. On mobile CPU, operator fusion, static execution planning, and dynamic memory planning bring 18% to 30%, an extra 22% to 37%, and another extra 3% to 7% memory reduction, respectively. Multi-version code generation (`MVC`) is primarily designed for latency improvement, its impact on memory reduction is negligible. The memory reduction on mobile GPU is omitted because our optimizations are general to both CPU and GPU, and the results are similar.

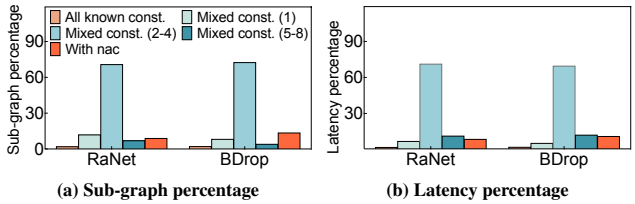
Latency Reduction w/ Different Optimizations. Figure 6 presents the speedup breakdown of our key optimizations on the same 4 models. On mobile CPU, our RDP-based operator fusion yields 1.3× to 1.9× speedup compared to `No opt`. Additionally, static execution planning provides 1.1× to 1.3× speedup, and dynamic memory planning gains 1.04× to 1.1× speedup, and Multi-version code generation brings an extra 1.3× to 1.6× speedup. On mobile GPU, these numbers are 1.4× to 2.3×, 1.2× to 1.3×, 1.06× to 1.2×, and 1.4× to 1.7×, respectively. Our optimizations provide more benefits for mobile GPU since GPU is more sensitive to memory and data movement and supports a higher degree of parallelism. We further study each optimization with more profiling results.



(a) Layer count

(b) IR size

Figure 7: Further break down effect of existing static fusion (SFusion) and RDP-based fusion (RDP Fusion). For both layer count and intermediate result size, normalized by no fusion opt.



(a) Sub-graph percentage

(b) Latency percentage

Figure 8: The percentage of different types of sub-graph.

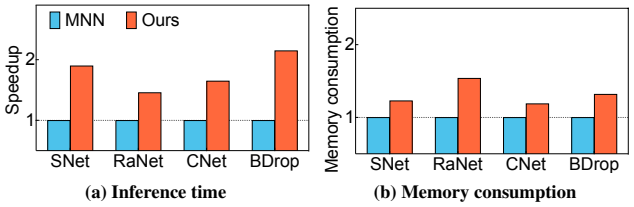
RDP-enabled Operator Fusion. Figure 7 further breaks down the effect of existing operator fusion for static DNNs only (`SFusion`) and our RDP-enabled operator fusion (`RDP Fusion`) on these four dynamic DNNs. These results are normalized by the original DNN without fusion (`Original`). `SFusion` reduces the layer counts by 26% to 61%; while `RDP Fusion` further reduces the layer counts by 16% to 46% additionally by leveraging RDP analysis results. In terms of intermediate result (IR) size, `RDP Fusion` saves an additional 13% to 40% on top of `SFusion`.

Subgraph Data. To better understand execution and memory planning, this part studies how many sub-graphs can benefit from RDP analysis results. Figure 8 (a) shows the percentage of different sub-graphs, i.e. those with all known constant shapes, with mixed constant shapes, and with statically unknown (*nac*) only for 2 representative models. The numbers (1, 2-4, and 5-8) after `Mixed const` denote the number of code versions that are required to optimize this sub-graph (the lower the better). This result shows that over 90% of the sub-graphs belong to all known constant or mixed constant categories whose execution plan and memory plan can be optimized by our framework. To further confirm this, Figure 8 (b) shows the latency percentage of each kind of sub-graphs.

5.4 Further Performance Analysis

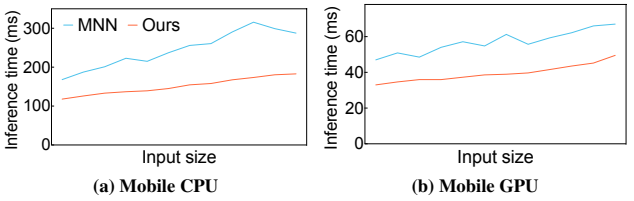
This section further studies SoD^2 under different cases.

Latency Comparison with the Same Execution Path. To provide an apple-to-apple comparison for control-flow dynamism, this test disables the control-flow logic in 4 models (SkipNet, RaNet, ConvNet-AIG, and BlockDrop) that have control-flow dynamism. Our execution included all paths, including all branches in the



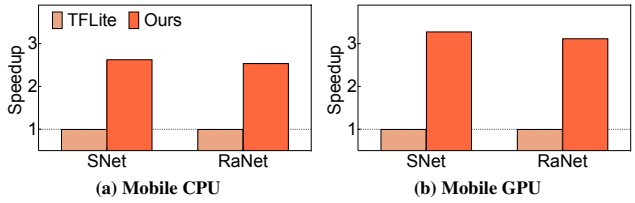
(a) Inference time (b) Memory consumption

Figure 9: Latency and memory consumption comparison between SoD^2 and MNN with the same execution path.



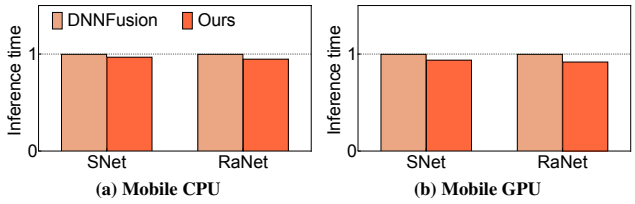
(a) Mobile CPU (b) Mobile GPU

Figure 10: Performance variation with different input sizes (shapes). The data is collected from YOLO-V6. A larger input size means more computations.



(a) Mobile CPU (b) Mobile GPU

Figure 11: Speedup with the same memory consumption.



(a) Mobile CPU (b) Mobile GPU

Figure 12: Inference time comparison with DNNFusion for static models (i.e., with both frozen shapes and control flows).

<*Switch, Combine*> pairs. Figure 9 illustrates the performance comparison with MNN because MNN performs the best among all baseline frameworks we compared. SoD^2 achieves 1.5× to 2.0× speedup and 1.2× to 1.5× memory reduction on the mobile CPU. This result further validates the effect of our RDP analysis and fusion, execution, and memory optimizations based on it even without the dynamic branch selection capability of SoD^2.

Latency Comparison with Different Input Sizes. To demonstrate the stability of SoD^2, this test randomly selects 15 input shapes for YOLO-V6, and Figure 10, shows their inference latency with MNN and SoD^2. These results demonstrate that SoD^2 outperforms MNN in terms of both latency and stability across increasing input sizes on mobile CPUs and GPUs. Specifically, SoD^2 exhibits lower and more consistent latency, while MNN exhibits significant variations.

Latency Comparison with Fixed Memory Budget. Figure 11 presents a latency comparison between SoD^2 and TFLite with the same memory budget. Specifically, TFLite fixes its memory consumption to match SoD^2's, and uses the XLA rematerialization policy [19] to handle the out-of-memory cases. SoD^2 outperforms TFLite by an even greater margin. Additionally, SoD^2 demonstrates a higher speedup on mobile GPU compared to mobile CPU due to the longer time required for mobile GPU to materialize intermediate tensors from its cache into main memory because of memory mapping.

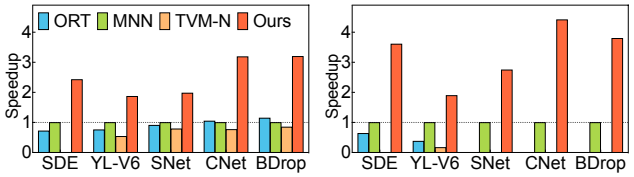
Latency Comparison with Static Models. Figure 12 examines the latency overhead of SoD^2 in contrast to our baseline, DNNFusion [46], for static models. Specifically, we evaluate the latency in SkipNet and RaNet where dynamic values were fixed statically and fully propagated, ensuring the absence of unknown values and dynamic control flows. As shown in Figure 12, SoD^2 incurs an average overhead of 3% and 7% performance slowdown when compared to the completely optimized static DNNFusion. This is attributed to the fact that DNNFusion, with full information available, results in a more comprehensive fusion optimization and does not include dynamic memory planning overhead.

5.5 Portability

To further investigate the effectiveness of portability, Figure 13 shows the execution speedup of SoD^2 over other frameworks on another mobile device – Snapdragon 835, and 5 models (StableDiffusion-Encoder, YOLO-V6, SkipNet, ConvNet-AIG, and BlockDrop). SoD^2 achieves similar speedup trends, and interestingly, it achieves higher speedups on this earlier generation of SoC because this SoC has more restricted resources (e.g., cache size and memory throughput). The RDP-based optimizations employed in SoD^2 significantly reduce memory requirements, leading to improved performance on these platforms.

6 RELATED WORK

Dynamic Neural Network Optimizations. Type analysis and type inference [8, 22, 33, 44, 58] are widely used to analyze tensor shapes, thus assisting in Dynamic Neural Network optimizations. Nimble [57], which has been integrated into TVM, is a compilation-based Dynamic DNN framework. This framework relies on expensive dynamic functions to interpret dynamic shapes at the runtime. This implementation, which we have extensively compared against, limits the opportunities for optimized code generation, such as performing operator fusion. DISC [74] extends MLIR-HLO [33] and propagates the shape information for operators that have certain constraints, e.g. same dimensions (the case of `Activation`) and same size (the case of `Transpose`). SoD^2 provides a more comprehensive operator classification based on dynamism degrees, bringing in significantly enhanced optimization opportunities. Axon [6] is a programming language that allows specification of symbolic shapes for input and output tensors for computational graphs. It uses a constraint solver



(a) Mobile CPU (b) Mobile GPU

Figure 13: Portability evaluation. The results are collected on Snapdragon 835. An empty bar means the model is not supported by the framework. Results are normalized by MNN for readability.

to find shapes whereas SoD^2 uses a forward and backward dataflow analysis (RDP), which also alleviates additional programmer involvement. In addition, SoD^2 includes a set of opts enabled by RDP.

Less closely related to SoD^2, DietCode [73] proposes an auto-scheduler framework based on TVM for dynamic shapes. The framework builds a cost model to predict runtime performance and reduces the search space to find optimal runtime parameters (e.g., loop tiling). Cortex [15], Cavs [67], and another effort [25] mainly aim to address recursive dynamism of neural networks, different from SoD^2's focus. Other efforts focus on dynamic batching for inference [14, 17, 41, 72] or are designed for dynamic DNN training [45].

DNN Execution and Memory Optimizations. Several studies exist for operator execution order scheduling, such as [2, 38, 39]. Among these efforts [38, 39] focus on minimizing peak memory consumption by reordering operators for resource-constrained devices (e.g., MCUs), and effort [2] proposes an optimized scheduling framework for complex models (irregularly wired neural networks). These approaches rely on static shapes only. There have aldo been recent efforts on optimizing memory allocation planning and memory management for DNNs. Works such as [35, 51] have designed various heuristic memory planning algorithms for static DNNs only. TelaMalloc [43] performs memory management on the fly for static control-flow graphs with known intermediate tensor shapes and sizes. It does not fully consider the DNN control-flow dynamism and dynamic shapes. A possible future work can be to integrate our RDP analysis and TelaMalloc's combination of heuristics with a solver-based approach to further improve our memory planning. When the available memory is limited, rematerialization [24, 30] and recomputation [3] methods achieve a trade-off between memory consumption and execution latency. These aspects can be considered for dynamic DNNs in the future.

DNN Inference Engines on Mobile. Support for DNN inference on mobile devices has become an area of active research in recent years. Efforts such as MCDNN [21], DeepX [32], DeepMon [23], DeepSense [69], and DeepCache [66] have primarily concentrated on optimizing the execution of static DNNs with static shapes and control flow. TensorFlow Lite (TFLite) [1], Pytorch-Mobile [50], TVM [5], and MNN [26] provide support for dynamic shapes relying on reinitialization or conservative (maximum) memory allocation. They either do not support dynamic control flow or require executions of all paths with a stripping of invalid results. As shown in our evaluation, these methods introduce high runtime overhead. One of the previous systems for static DNNs, DNNFusion [46], also involved a classification of DNN operators, however, the classification introduced here is orthogonal.

7 DISCUSSION AND FUTURE WORK

Generalizing to Other Platforms. The proposed techniques, such as RDP analysis, RDP-enabled fusion, and execution and memory planning, have broad applicability to various platforms, including data-center GPUs. This is particularly true for single-input inference scenarios. One potential nuance that may arise is the distinction between data-center GPUs and mobile GPUs in terms of their ability to perform batched inference. Unlike mobile GPUs, data-center GPUs have the capacity to process multiple inputs concurrently, thereby maximizing their computational power. However, it is possible that different input samples within a batch may necessitate the use of different execution paths. Therefore, the integration of dynamic batching with dynamic neural networks presents a potential direction for future research.

Scalability of Handling LLMs. The optimizations in SoD^2 can also be applied to massive large language models (LLMs). One of the primary procedures we employ is graph partitioning, as elaborated in Section 4.3. This procedure involves dividing the entire computational graph into a collection of sub-graphs, each of which encompasses a restricted number of layers. The optimal solution is determined offline for each sub-graph. However, Language Models (LLMs) have been characterized by an incredibly large number of parameters, numbering in the billions [7, 60, 70]. This poses a significant challenge for mobile devices in terms of computation and resource requirements. Our future work will enhance SoD^2 by combining it with the model pruning and quantization advances [27, 47, 64] to achieve an even better performance.

Extending beyond ONNX. Operator classification and associated optimization designs are also not limited to ONNX or other inference formats (e.g., TFLite, Caffe2). This is because our proposed analysis is based on the degree of dynamism defined by the computation logic of an operator and the relationship between its input and output, rather than relying on the specific representation or format of the operator. Some formats have yet to fully support dynamic computational graphs. For instance, PyTorch supports exporting models with dynamic shapes (such as Input Shape Determined Output, Input Shape Determined Output Shape, and Input Shape & Value Determined Output Shape) to ONNX. However, it is unable to convert models with dynamic control flow to ONNX. To address this limitation, we added a customized ONNX operator pair <Switch, Combine> (as shown in Figure 1d) and registered a customized export routine on PyTorch specifically for models with a dynamic control flow. SoD^2 does have limitations in handling very complicated (or user-defined) dynamic models (such as Graph Neural Networks or DNNs involving recursive executions) that can be represented well in PyTorch. We leave this further optimization as a future work.

8 CONCLUSIONS

This paper has presented a comprehensive framework, SoD^2, for optimizing DNNs. SoD^2 classifies common operators of Dynamic DNNs into four types, and comprises a novel static dataflow analysis

(RDP). This is followed by a set of optimizations enabled by RDP for Dynamic DNNs, including operator fusion, static execution (order) planning, dynamic memory allocation planning, and multi-version code generation. SoD^2 is extensively evaluated on a mobile system with 10 emerging dynamic DNNs and the evaluation results show that it saves up to 88% memory consumption and brings up to 3.9× execution speedup over four state-of-the-art DNN execution frameworks. As the underlying techniques are general and applicable to other devices as well, our future work will evaluate SoD^2's efficacy on other devices (e.g., edge GPUs and Raspberry Pi).

ACKNOWLEDGMENTS

The authors would like to express their gratitude to the anonymous reviewers and shepherd for their insightful and detailed comments. All of these have significantly contributed to the enhancement of this paper. This work was supported in part by the National Science Foundation (NSF) under the awards of CCF-2047516 (CAREER), CCF-2146873, CCF-2333895, CCF-2334273, CNS-2230944, CNS-2341378, IIS-2142681, III-2008557, and OAC-2333899. Any errors and opinions are not those of the NSF and are attributable solely to the author(s). The authors also acknowledge William & Mary Research Computing for providing computational resources.

REFERENCES

[1] Martin 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 G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *OSDI 2016*. USENIX Association, USA, 265–283.

[2] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. 2020. Ordering Chaos: Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/290.pdf

[3] Samuel Rota Bulò, Lorenzo Porzi, and Peter Kontschieder. 2018. In-Place Activated BatchNorm for Memory-Optimized Training of DNNs. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 5639–5647. https://doi.org/10.1109/CVPR.2018.00591

[4] David Callahan, Keith D Cooper, Ken Kennedy, and Linda Torczon. 1986. Interprocedural constant propagation. *ACM SIGPLAN Notices* 21, 7 (1986), 152–161.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI 2018*. 578–594.

[6] Alexander Collins and Vinod Grover. 2022. Axon: A Language for Dynamic Shapes in Deep Learning Graphs. *ArXiv preprint* abs/2210.02374 (2022). https://arxiv.org/abs/2210.02374

[7] Mike Conover, Matt Hayes, Ankit Mathur, Jianwei Xie, Jun Wan, Sam Shah, Ali Ghodsi, Patrick Wendell, Matei Zaharia, and Reynold Xin. 2023. Free Dolly: Introducing the World's First Truly Open Instruction-Tuned LLM. https://www.databricks.com/blog/2023/04/12/dolly-first-open-commercially-viable-instruction-tuned-llm

[8] Karl Crary and Stephanie Weirich. 1999. Flexible type analysis. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*. 233–248.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*. IEEE Computer Society, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[10] Microsoft Developer. 2023. CodeBERT. https://github.com/microsoft/CodeBERT.

[11] Numpy developers. 2023. Tensor Broadcasting. https://numpy.org/doc/stable/user/basics.broadcasting.html. Version: 1.24.

[12] ONNX Runtime developers. 2023. ONNX Runtime. https://onnxruntime.ai/. Version: 1.14.1.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[14] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: an efficient GPU serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 389–402.

[15] 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.

[16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[17] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency RNN inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[18] Ben Goertzel. 2014. Artificial general intelligence: concept, state of the art, and future prospects. *Journal of Artificial General Intelligence* 5, 1 (2014), 1.

[19] Google. 2023. Tensorflow XLA. https://www.tensorflow.org/xla.

[20] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, and Ruoming Pang. 2020. Conformer: Convolution-augmented Transformer for Speech Recognition. In *Interspeech 2020, 21st Annual Conference of the International Speech Communication Association, Virtual Event, Shanghai, China, 25-29 October 2020*, Helen Meng, Bo Xu, and Thomas Fang Zheng (Eds.). ISCA, 5036–5040. https://doi.org/10.21437/Interspeech.2020-3015

[21] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 123–136.

[22] Robert Harper and Greg Morrisett. 1995. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 130–141.

[23] Loc N Huynh, Youngki Lee, and Rajesh Krishna Balan. 2017. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 82–95. https://doi.org/10.1145/3081333.3081360

[24] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph Gonzalez. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/320.pdf

[25] Eunji Jeong, Joo Seong Jeong, Soojeong Kim, Gyeong-In Yu, and Byung-Gon Chun. 2018. Improving the expressiveness of deep learning frameworks with recursion. In *Proceedings of the Thirteenth EuroSys Conference*. 1–13.

[26] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lyu, and Zhihua Wu. 2020. MNN: A Universal and Efficient Inference Engine. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/287.pdf

[27] Qing Jin, Jian Ren, Richard Zhuang, Sumant Hanumante, Zhengang Li, Zhiyu Chen, Yanzhi Wang, Kaiyuan Yang, and Sergey Tulyakov. 2022. F8net: Fixed-point 8-bit only multiplication for network quantization. *arXiv preprint arXiv:2202.05239* (2022).

[28] Gary A Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 194–206.

[29] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. 2023. Segment Anything. *ArXiv preprint* abs/2304.02643 (2023). https://arxiv.org/abs/2304.02643

[30] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2021. Dynamic Tensor Rematerialization. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=Vfs_2RnOD0H

[31] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[32] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. 2016. DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. 1–12.

[33] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[34] Royson Lee, Stylianos I Venieris, Lukasz Dudziak, Sourav Bhattacharya, and Nicholas D Lane. 2019. Mobisr: Efficient on-device super-resolution through heterogeneous mobile processors. In *The 25th Annual International Conference on Mobile Computing and Networking*. 1–16.

[35] Maksim Levental. 2022. Memory Planning for Deep Neural Networks. *ArXiv preprint* abs/2203.00448 (2022). https://arxiv.org/abs/2203.00448

[36] Chuyi Li, Lulu Li, Hongliang Jiang, Kaiheng Weng, Yifei Geng, Liang Li, Zaidan Ke, Qingyuan Li, Meng Cheng, Weiqiang Nie, et al. 2022. YOLOv6: A single-stage object detection framework for industrial applications. *ArXiv preprint* abs/2209.02976 (2022). https://arxiv.org/abs/2209.02976

[37] Changlin Li, Guangrun Wang, Bing Wang, Xiaodan Liang, Zhihui Li, and Xiaojun Chang. 2021. Dynamic Slimmable Network. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*. Computer Vision Foundation / IEEE, 8607–8617. https://doi.org/10.1109/CVPR46437.2021.00850

[38] Edgar Liberis and Nicholas D Lane. 2019. Neural networks on microcontrollers: saving memory at inference via operator reordering. *ArXiv preprint* abs/1910.05110 (2019). https://arxiv.org/abs/1910.05110

[39] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. 2020. MCUNet: Tiny Deep Learning on IoT Devices. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/86c51678350f656dcc7f490a43946ee5-Abstract.html

[40] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Doll'a r, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. *CoRR* abs/1405.0312 (2014). arXiv:1405.0312 http://arxiv.org/abs/1405.0312

[41] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. 2017. Deep Learning with Dynamic Computation Graphs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=ryrGawqex

[42] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 881–897.

[43] Martin Maas, Ulysse Beaugnon, Arun Chauhan, and Berkin Ilbeyi. 2022. TelaMalloc: Efficient On-Chip Memory Allocation for Production Machine Learning Accelerators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 123–137. https://doi.org/10.1145/3567955.3567961

[44] Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375.

[45] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. 2017. Dynet: The dynamic neural network toolkit. *ArXiv preprint* abs/1701.03980 (2017). https://arxiv.org/abs/1701.03980

[46] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.

[47] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. 2020. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 907–922.

[48] ONNX. 2017. Open Neural Network Exchange. https://www.onnx.ai.

[49] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2015, South Brisbane, Queensland, Australia, April 19-24, 2015*. IEEE, 5206–5210. https://doi.org/10.1109/ICASSP.2015.7178964

[50] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[51] Yury Pisarchyk and Juhyun Lee. 2020. Efficient memory management for deep neural net inference. *ArXiv preprint* abs/2001.03288 (2020). https://arxiv.org/abs/2001.03288

[52] Qualcomm. 2016. Snapdragon 835. https://www.qualcomm.com/products/snapdragon-835-mobile-platform.

[53] Qualcomm. 2020. Snapdragon 888. https://www.qualcomm.com/products/snapdragon-888-5g-mobile-platform.

[54] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *PLDI 2013*. Association for Computing Machinery, New York, NY, USA, 519–530.

[55] Yongming Rao, Wenliang Zhao, Benlin Liu, Jiwen Lu, Jie Zhou, and Cho-Jui Hsieh. 2021. DynamicViT: Efficient Vision Transformers with Dynamic Token Sparsification. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 13937–13949. https://proceedings.neurips.cc/paper/2021/hash/747d3443e319a22747fbb873e8b2f9f2-Abstract.html

[56] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 10684–10695.

[57] 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.

[58] Jeremy Siek and Walid Taha. 2007. Gradual typing for objects. In *ECOOP 2007–Object-Oriented Programming: 21st European Conference, Berlin, Germany, July 30-August 3, 2007. Proceedings 21*. Springer, 2–27.

[59] TensorFlow. 2018. TensorFlow Grappler. https://www.tensorflow.org/guide/graph_optimization.

[60] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[61] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *ArXiv preprint* abs/1802.04730 (2018). https://arxiv.org/abs/1802.04730

[62] Andreas Veit and Serge Belongie. 2018. Convolutional Networks with Adaptive Inference Graphs. *European Conference on Computer Vision (ECCV)* (2018).

[63] 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 (ECCV)*. 409–424.

[64] Zifeng Wang, Zheng Zhan, Yifan Gong, Geng Yuan, Wei Niu, Tong Jian, Bin Ren, Stratis Ioannidis, Yanzhi Wang, and Jennifer Dy. 2022. SparCL: Sparse Continual Learning on the Edge. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 20366–20380. https://proceedings.neurips.cc/paper_files/paper/2022/file/80133d0f6eccaace15508f91e3c5a93c-Paper-Conference.pdf

[65] Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S. Davis, Kristen Grauman, and Rogério Schmidt Feris. 2018. BlockDrop: Dynamic Inference Paths in Residual Networks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 8817–8826. https://doi.org/10.1109/CVPR.2018.00919

[66] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. 2018. DeepCache: Principled Cache for Mobile Deep Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom '18)*. Association for Computing Machinery, New York, NY, USA, 129–144.

[67] Shizhen Xu, Hao Zhang, Graham Neubig, Wei Dai, Jin Kyu Kim, Zhijie Deng, Qirong Ho, Guangwen Yang, and Eric P Xing. 2018. Cavs: An efficient runtime system for dynamic neural networks. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 937–950.

[68] Le Yang, Yizeng Han, Xi Chen, Shiji Song, Jifeng Dai, and Gao Huang. 2020. Resolution Adaptive Networks for Efficient Inference. In *2020 IEEE/CVF Conference*

on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020. IEEE, 2366–2375. https://doi.org/10.1109/CVPR42600.2020.00244

[69] Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, and Tarek F. Abdelzaher. 2017. DeepSense: A Unified Deep Learning Framework for Time-Series Mobile Sensing Data Processing. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, Rick Barrett, Rick Cummings, Eugene Agichtein, and Evgeniy Gabrilovich (Eds.). ACM, 351–360. https://doi.org/10.1145/3038912.3052577

[70] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. 2022. Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414* (2022).

[71] Jinle Zeng, Min Li, Zhihua Wu, Jiaqi Liu, Yuang Liu, Dianhai Yu, and Yanjun Ma. 2022. Boosting Distributed Training Performance of the Unpadded BERT Model. *arXiv preprint arXiv:2208.08124* (2022).

[72] Yujia Zhai, Chengquan Jiang, Leyuan Wang, Xiaoying Jia, Shang Zhang, Zizhong Chen, Xin Liu, and Yibo Zhu. 2022. ByteTransformer: A High-Performance Transformer Boosted for Variable-Length Inputs. *ArXiv preprint* abs/2210.03052 (2022). https://arxiv.org/abs/2210.03052

[73] 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.

[74] Kai Zhu, WY Zhao, Zhen Zheng, TY Guo, PZ Zhao, JJ Bai, Jun Yang, XY Liu, LS Diao, and Wei Lin. 2021. DISC: A dynamic shape compiler for machine learning workloads. In *Proceedings of the 1st Workshop on Machine Learning and Systems*. 89–95.