Edge-PRUNE: A Dataflow-Based Framework for Distributed Signal Processing and Machine Learning

Jani Boutellier, *Senior Member, IEEE*, Bo Tan, *Member, IEEE*, Jari Nurmi, *Senior Member, IEEE*, and Shuvra S. Bhattacharyya, *Fellow, IEEE*

***Abstract*—Distributed sensing through video, audio, radar and other sensors is strongly growing with application areas such as smart homes and Internet of Things. The concept of edge computing proposes shifting signal and data analysis from centralized servers close to the sensors, providing reduction in data communication bandwidth requirements and centralized server computation load as well as improving data privacy. Previous works in the domain of edge computing have paid little attention to formal modeling of computing across devices. This work proposes the VR-PRUNE-E model of computation that is based on the well-known dataflow abstraction. Within VR-PRUNE-E, a specific type of resilient network graph is introduced, which allows the distributed system to continue its operation after the failure of any single node or connection. Besides the formal model, the manuscript introduces the Edge-PRUNE software framework that supports the proposed dataflow abstraction, as well as concrete experimental results on real edge computing scenarios. The explored setups cover networks with up to 128 endpoint nodes and two servers. Application examples cover popular machine learning applications of image classification, object detection and radar signal processing, built on CNN and transformer architectures, extended with redundant system configurations that provide fault tolerance. The proposed work is also benchmarked in terms of processing time and shown to outperform previous work by 34% in computation efficiency.**

***Index Terms*—Dataflow computing, signal processing, machine learning, edge computing, fault tolerance.**

Received 11 October 2024; revised 18 March 2025 and 23 May 2025; accepted 10 August 2025. Date of publication 13 August 2025; date of current version 4 September 2025. This work was supported in part by the Research Council of Finland under Grant 345681 and Grant 345683, in part by the Business Finland EVET-ZEM Project DAZE 11019/31/2022, in part by Finnish Scientific Advisory Board for Defence Project VN/17548/2023-SAAP-25, and in part by the US Army Research Office and US Army Research Laboratory accomplished under Grant W911NF-21-1-0258. The associate editor coordinating the review of this article and approving it for publication was Pin-Yu Chen. *(Corresponding author: Jani Boutellier.)*

Jani Boutellier is with the School of Technology and Innovation, University of Vaasa, 65200 Vaasa, Finland (e-mail: jani.boutellier@uwasa.fi).

Bo Tan and Jari Nurmi are with Tampere Wireless Research Centre, The Faculty of Information Technology and Communication Sciences, Tampere University, 33720 Tampere, Finland (e-mail: bo.tan@tuni.fi; jari.nurmi@tuni.fi).

Shuvra S. Bhattacharyya is with the Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 USA (e-mail: ssb@umd.edu).

Digital Object Identifier 10.1109/TSP.2025.3598453

I. Introduction

EDGE computing is becoming increasingly important with applications such as smart homes [1], surveillance [2] and machinery condition monitoring [3]. The analytics algorithms involved in these applications are commonly at the core of signal processing, or from closely related fields such as machine learning.

Edge computing provides a variety of benefits to the deployment of algorithms and applications: the possibility to flexibly distribute the computing load between *endpoint devices* and edge servers, reducing communication bandwidth needs, and – while less obvious – increasing the degree of privacy [4] for the signal measured by the sensing endpoint devices, which is especially important in smart home contexts and with video type signals.

Although a variety of works have been published on edge computing and collaborative edge-cloud signal/data analytics [5], [6], [7], [8], existing frameworks for edge computing have either been developed without considering computation models at all, or resorting to less formally specified DAG (directed acyclic graph) based models of computation that do not capture details of computation and data, which are important to signal processing: definitions of samples, sampling rates and feedback loops. In contrast, a well-defined, formal model of computation allows analyzing application correctness at design time, preventing the introduction of subtle application design mistakes that can lead to costly error correction after system deployment.

Dataflow [9], [10] has provided a well-defined model of computation (MoC) across a variety of signal processing applications [11], and more recently also for machine learning [12], [13]. In the context of signal processing and machine learning (SPML) systems, the general form of dataflow can be viewed as encompassing a large class of more specialized models of computation [10], which are streamlined for specific application domains, platform types or other usage or analysis scenarios (e.g., see [14]). This paper contributes a framework for dataflow-based modeling and design that is geared towards efficient implementation of SPML systems on edge computing platforms.

Consider Fig. 1, which depicts a toy example of a dataflow graph for camera image-based face detection and recognition (`facedet` and `facerec` blocks). Here, the `camera` block emits images as data packages called *tokens*. Each image token

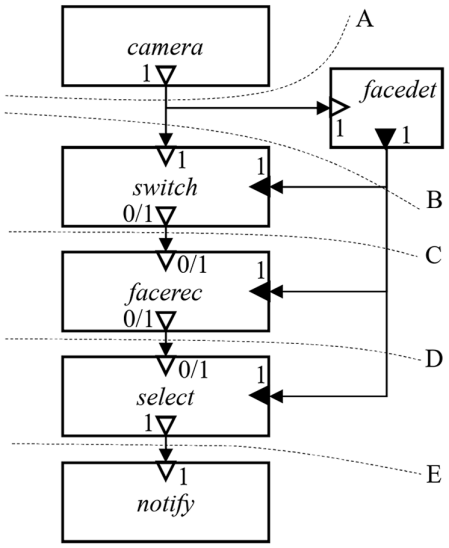


Fig. 1. An example of a dynamic dataflow graph. Rectangles represent actors, whereas solid arrows denote dataflow edges. Capital letters associated with dashed lines depict partition points.

is received by the `facedet` block and by the `switch` block. If the `facedet` block detects a face in the input image, it transmits an "enable" token to the blocks `switch`, `facerec` and `select` — otherwise it sends a "disable" token to these blocks. Upon an "enable" token, the `switch` block relays the camera image to the `facerec` block, which performs face detection and sends the output to the `select` block that finally triggers the `notify` block. Otherwise, if `switch` receives a "disable" token from `facedet`, the camera image is not forwarded to `facerec`, and the `select` block outputs a null value that does not cause a notification event in the `notify` block.

In an edge computing scenario, we can consider distributing the computations associated with the blocks between two systems. Assuming a smart camera with some processing capacity and an edge server with abundant processing resources, Fig. 1 depicts five different *partition points* denoted with letters A through E. For example, applying partition point A would mean that the smart camera executes only the `camera` block, transmits the `camera` output over a remote communication channel to the edge server, and lets the edge server process all the remaining blocks.

The most suitable partition point among A through E can be selected based on a variety of criteria that can include communication bandwidth minimization, or balancing of the computation load. However, the question of partitioning becomes radically changed if we consider that the remote connection between the smart camera and the edge server is unreliable, with a chance of data loss. It turns out that with unreliability, partition points B, C and D have severe disadvantages: loss or unavailability of a token emitted by the `facedet` block can cause synchronization problems around the `facerec` block such that recognition is triggered for images that do not contain a face and vice-versa. More severely, if over a long period of time, more and more tokens emitted by the `facedet` block are lost, unprocessed image tokens start accumulating at the input of the `switch` block causing eventually a buffer overflow with possibly catastrophic consequences.

In contrast, if partition point A or partition point E is selected, token loss has only mild consequences: individual images (point A) or individual notifications (point E) can be lost, but internal system synchrony is maintained, and buffers do not accumulate tokens. In dataflow terminology, uncontrolled accumulation of tokens is related to a concept called "consistency"; in particular, one possible consequence of a dataflow graph being inconsistent is that there may be no way to execute the graph iteratively, over an indefinitely large input stream, without having an unbounded number of tokens accumulate on one or more buffers within the graph. In contrast, consistent dataflow graphs permit the handling of arbitrarily large input streams while providing guarantees of bounded buffer memory requirements. This provision for bounded memory execution on indefinitely large input streams is important in embedded signal and information processing systems, where there is often no pre-specified bound on the duration of the input signals, and even when a bound is available, there is typically insufficient storage space available to hold an entire input stream all at once. Reliable execution under such conditions requires that the system operate under a *bounded memory guarantee*, where the memory bound is less than the storage capacity of the target platform.

Consistency analysis is one of the fundamental tasks in dataflow-based application modeling. For most [10], [15], [16] dataflow models of computation that are expressive enough to describe a dynamic system such as in Fig. 1, *decidable* consistency analysis is not guaranteed — that is — the consistency analysis cannot be performed in finite time.

This paper proposes VR-PRUNE-E, an edge computing elaboration of the recent VR-PRUNE dataflow MoC [17]. The VR-PRUNE-E MoC is flexible, allowing the modeling of dynamic computations (similar to Fig. 1) ranging from individual image processing filters to neural network layers, and enabling application execution mapping from individual processor cores across networked distributed edge scenarios. VR-PRUNE-E has been developed with error-resilience in mind: following formal design rules ensures that the dataflow graph consistency analysis is a decidable problem. Furthermore, VR-PRUNE-E specifies graph topologies that provide fault tolerance such that the overall system remains operational even in the case when a single component (node, connection) fails. Details on how VR-PRUNE-E relates to and goes beyond VR-PRUNE will be given in Section IV-D.

Besides the theoretical VR-PRUNE-E MoC, this paper also presents the Edge-PRUNE Framework that implements the VR-PRUNE-E MoC and allows deploying applications to real-life distributed systems. A preliminary version of Edge-PRUNE has previously been presented in [18] and its fault-tolerance aspects have been discussed in [19]. This paper provides:

1) A complete coverage of the VR-PRUNE-E model of computation,

2) A description of the Edge-PRUNE software framework[1],
3) Experimental results on real edge computing devices and applications, and
4) Evaluation against previous work.

Compared to [18], [19], items #1, #3 and #4 are fully novel content in this manuscript. An early description of the Edge-PRUNE framework was presented in [18], however it did not include, e.g., description of the Open Neural Network Exchange (ONNX) support that is of key importance for Edge-PRUNE.

The rest of the paper is organized as follows: Section II presents relevant related works, Section III presents the VR-PRUNE MoC, Section IV elaborates the MoC to distributed computing, Section V presents the Edge-PRUNE framework, Section VI shows experimental evaluation of the proposed work, Section VII discusses the results and concludes the paper.

II. Background

A. Dataflow

Dataflow is a MoC that has gained significant popularity as a formal abstraction of signal processing systems [9], [20] and more recently in the field of machine learning [12]. The dataflow abstraction captures the flow of data samples between computations, enabling formal analyses [21], as well as practical tools for software deployment [22]. Several significant dataflow models of computation have been introduced in the past, and in general the MoCs provide different trade-offs between expressiveness and analyzability. On one extreme there are fully design-time analyzable static MoCs such as Synchronous Dataflow [9], and on the other side fully dynamic dataflow MoCs such as Dataflow Process Networks [10]. Whereas fully dynamic dataflow MoCs allow modeling almost any type of application behavior, means for application correctness analysis remain limited. Between the two extremes, there are a number of dataflow MoCs which allow capturing restricted types of dynamic application behavior, such as parameterized SDF [20], PiSDF [23], and VR-PRUNE [17]. VR-PRUNE allows describing signal processing applications that are allowed to change their run-time behavior under formally defined restrictions that preserve application design time analyzability. VR-PRUNE can be considered as a predecessor of VR-PRUNE-E; whereas VR-PRUNE addressed applications that are executed on a single system, the proposed VR-PRUNE-E MoC focuses on distributed dataflow computing and fault tolerance.

B. Collaborative Inference and Distributed Computing

The developments of machine learning (ML) have recently become a driving force in technology. Many of the central mathematical operations (e.g., convolution) and algorithm patterns of ML stem from signal processing, making the fields closely related. In the context of Edge-PRUNE, collaborative machine learning inference is an especially relevant application area, where the computations of a deep neural network (DNN) are partitioned between a low-resource *endpoint* device (such as the smart camera of Fig. 1) and a more powerful *edge server*.

One of the pioneering works in collaborative inference, Neurosurgeon [24], proposed a prediction-based scheduler on top of Caffe, for automatic partitioning of DNNs between mobile devices and data centers. Autodidactic Neurosurgeon (ANS) [25] elaborated on [24] by introducing adaptive learning-based partition point optimization. Simultaneously with [24], the DDNN framework [30] proposed homogeneously distributed machine learning with *early exits* across cloud, edge and multiple endpoint devices. Recently, the DDNN framework has been implemented within the Adaptive Computing Framework (ACF) [31]. In contrast to endpoint-server collaborative inference, several works [6], [29], [32] have proposed the distribution of DNN inference across multiple endpoint devices.

Edgent [5] (and its successor Boomerang [26]) continued in the vein of Neurosurgeon, proposing DNN *right-sizing* using early exits (similar to DDNN [30]), with an implementation based on the Chainer deep learning framework. In contrast, IONN [33] proposed an incremental offloading scheme, where a client device uploads partitions of a DNN model to a server for distributed inference; like Neurosurgeon, IONN is Caffe-based. JointDNN [34] formulated the partitioning of computations as a graph shortest path problem, also paying attention to autoencoder and generative model type DNNs, where output size grows towards the last layers. JALAD [35] proposed an optimization framework for distributing DNN computations between edge and cloud, considering the possibility of feature compression, accuracy and latency optimization.

DADS [27] proposed the DAG based edge-cloud DNN inference model (ECDI) that can be used to optimize the edge-cloud partitioning of more complex than chain-like DNN structures. For the practical implementation, DADS relies on a modified version of Caffe. Similar to DADS, the industrial effort Auto-Split [28], and D^3 [36] are DAG-based. D^3 follows the three-layer (device, edge, cloud) concept of DDNN [30], as well as parallel distribution across edge nodes like, e.g., MoDNN [32]. SPINN [7] is a PyTorch-based framework for distributed inference that leverages concepts of early exit and dynamic splitting, which can accommodate to run-time changes of the environment. The successor of SPINN, DynO [8], introduced several optimization targets and bit-width optimizations.

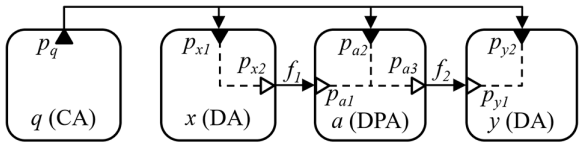
On the distributed computing front, TensorFlow (TF) [12] represents a major commercial tool for distributed machine learning; compared to Edge-PRUNE, the underlying dataflow model of TF does not provide similar design time analyzability [17]. DASK [37] is a Python-based library for distributed computing with mentions on fault-tolerance features. Finally, DAL [38] is a Kahn Process Networks (KPN) [39] based model for distributed computing, with fault tolerance support. KPN, unfortunately, does not provide design time analyzability of application correctness.

Recently, the concept of semantic communication systems [40] has been proposed, where instead of transmitting the whole source bit sequence, only the key information is transmitted, eliminating irrelevant information without any performance degradation [41]. Sun et al. [41] have formulated a semantic

[1] Available at https://gitlab.com/jboutell/vprf/-/tree/edge-prune

TABLE I
COMPARISON OF REPRESENTATIVE EDGE COMPUTING WORKS

Work	ML support	Fault toler.	Open source	Formal model
Edge-PRUNE	ONNX	Yes	Yes	VR-PRUNE-E
Neurosurg. [24]	Caffe	No	No	No
ANS [25]	TF+Pytorch	No	Yes	No
Boomer. [26]	Chainer	No	No	No
DADS [27]	Caffe	No	No	DAG
Auto-Split [28]	TF/C++	No	No	DAG
EdgeSP [29]	Unknown	No	No	No
DynO [8]	PyTorch	No	No	DAG



T	p_{x2}	p_{a1}	p_{a3}	p_{y1}	[lrl, url]
p_q	1	1	1	1	[0, 1]

Fig. 2. An example of an application graph G that forms a minimal DPG. Control relationship between an input control port (black triangle) and a DRP (white triangle) is depicted by a dashed line.

computation rate maximization problem, considering computation offloading decisions, among others, in the context of aerial-ground networks [42].

Table I shows a summary of previous works and compares them to Edge-PRUNE. Compared to the related works, Edge-PRUNE offers a unique combination of a formally defined dataflow MoC with design time analyzability, as well as support for distributed computing with fault tolerance. The table shows that besides the general lack of support for fault tolerance among previous works, very few works have released a software infrastructure that could be used for further works. In the following two sections, the MoC of Edge-PRUNE is presented.

III. MODEL OF COMPUTATION

In this section we compactly describe the key concepts that are common between the VR-PRUNE [17] MoC and its proposed elaboration VR-PRUNE-E to make the MoC presentation self-sustained and unambiguous.

In VR-PRUNE(-E), the top-level entity for describing an application (for instance, a DNN) is the application graph $G = (A, F)$, where the set of nodes A represent computations (e.g., DNN layers or operations), and the set of directed edges F represent data buffers between nodes. The edges carry data in first-in-first-out (FIFO) order. The connection point between an edge $f \in F$ and a node $a \in A$ is called a *port* p_a such that $f = fifo(p_a)$ and $parent(p_a) = a$. Fig. 2 shows a minimal example of a graph G, where $A = \{q, x, a, y\}$ such that x and y are a pair of dynamic actors, and q is an associated control actor (See Section III-A for an explanation of actor types). Here, for example, $f_1 = fifo(p_{x2}) = fifo(p_{a1})$ and $parent(p_{a1}) = a$.

Within edges, data flows in the form of *tokens* that are data packets of pre-defined size. In the machine learning context, tokens equal to tensors – matrices of intermediate features between DNN layers. In classical signal processing tokens can be interpreted as samples.

In dataflow MoCs, nodes are called *actors*. Computation in an actor is triggered based on data availability: an actor can start to compute (it *fires*) when all the input edges of that actor have a sufficient number of tokens available – for each input port of each actor, the *input token rate* indicates the number of tokens required for one firing. Upon firing, the actor consumes a number of tokens (indicated by the token rate) from each input edge and produces a number of tokens to each departing edge of that actor – specified by the *output token rate* that is defined for each port with output direction.

Each edge $f \in F$ of the graph G is connected to exactly one output port p^+ of an actor, and to exactly one input port p^- of an actor. The port $p^+ = source(f)$ is referred to as the *source port* of edge f and $p^- = sink(f)$ is referred to as the *sink port* of f. If $p^+ = source(f)$ and $p^- = sink(f)$, p^+ and p^- are *connected ports*. In the context of Fig. 2 we could write $p^+_{x2} = source(f_1)$ and $p^-_{a1} = sink(f_1)$ if the port directions need to be emphasized.

In the application graph G, each edge $f \in F$ represents the classical bounded-size FIFO channel that is assumed to carry all data tokens from the source port p^+ of f to the sink port p^- of f without data loss; this is emphasized in VR-PRUNE-E by denoting edges within the graph G as *regular edges*. Two actors $a, b \in G$ are *adjacent*, if $p^+_a = source(f)$ and $p^-_b = sink(f)$. The maximum token capacity of edge $f \in G$ is indicated as $capacity(f)$. The current token count in edge f is expressed by $tokencount(f)$ such that $0 \leq tokencount(f) \leq capacity(f)$. Initial tokens (delays) on edges $f \in F$ are denoted as $delay(f)$.

Ports of graph G actors A exist in three different types:

Static regular port (SRP). An SRP has a token rate that is fixed to a positive integer value.

Control port (CP). A CP is similar to an SRP except that its token rate is required to be fixed to 1.

Dynamic regular port (DRP). As the name suggests, the DRP allows some flexibility in its token rate. At design time, for each DRP p, a lower rate limit (*lrl*) and an upper rate limit (*url*) are fixed such that $0 \leq lrl(p) \leq url(p)$. At runtime the active token rate, $atr(p)$, of a DRP can vary such that $lrl(p) \leq atr(p) \leq url(p)$.

The VR-PRUNE(-E) MoC has two features that set it apart from other dataflow models of computation: a) support for variable token rates, and b) the symmetric token rate requirement. Whereas the variable token rate feature allows the token rate of port p to vary at runtime within $lrl(p)$ and $url(p)$, the symmetric token rate requirement, on the other hand, necessitates that for actors $a, b \in A$ always holds $atr(p_a) = atr(p_b)$ if p_a and p_b are connected ports.

A. Actor Types

Each actor $a \in A$ belongs to one of the four actor types: *static processing actor* (SPA), *dynamic actor* (DA), *configuration actor* (CA), or *dynamic processing actor* (DPA).

Static processing actor. The SPA type of actor can have zero or more input SRPs, and zero or more output SRPs. In terms of behavior (dataflow *firing rules*) the SPA resembles the classical synchronous dataflow [9] actor: in order to fire, the SPA needs to have $tokencount(fifo(p^-)) \geq atr(p^-)$ tokens available on each input port p^-, and $capacity(fifo(p^+)) - tokencount(fifo(p^+)) \geq atr(p^+)$ of free token space for the edge associated with each output port p^+ such that $parent(p^+) = parent(p^-) = a$.

Configuration actor. The CA is required to have one or more output CPs and may additionally have zero or more SRPs with input or output direction. In terms of firing rules, configuration actors behave identically to SPAs.

Dynamic actor. A DA has one or more DRPs of input direction, or one or more DRPs of output direction — a DA cannot have both input and output DRPs. In addition to the DRPs, a DA must have one input CP and may have any number of SRPs (input or output direction). Due to the presence of DRPs, a DA has more than one firing rule, which are however set deterministically by the value of the token consumed from the input CP.

Dynamic processing actor. A DPA must have one or more DRPs of input direction, and one or more DRPs of output direction, as well as exactly one input CP. A DPA is not allowed to have any SRPs. Similar to DAs, DPA firing rules are deterministically set by token values read from the DPA's control port.

B. Control Table

In the application graph G, actors with DRPs (DPAs and DAs) are only allowed to exist within (sub)graphs called Dynamic Processing Graphs (DPGs) explained in Section III-D. Restricting the appearance of dynamic computation and communication behavior to clearly defined graph structures enables decidable graph analyzability. The DPG concept is flexible in the sense that the application graph G may consist of several interconnected DPGs, however the VR-PRUNE(-E) Design Rules that are described is Section III-C disallow individual actors from being associated to more than one DPG.

Each DPG has an associated *control table T*, which defines (1) the control relationships between CAs, DAs and DPAs, and (2) variable token rates between DRPs. The control table has h rows and $w + 1$ columns, where h is the number of output control ports in the DPG, and w is the number of DRPs, respectively. The first w columns of the control table contain a binary value of 0 or 1, indicating whether there is (value 1) a control relationship between the output control port (row) and DRP (column) or not (value 0). The last column of the control table indicates the *lrl* and *url* token rates that apply to all DRPs *controlled* by the same control output port. The graph of Fig. 2 shows a minimal example of a DPG and the associated control table.

C. Design Rules

VR-PRUNE(-E) defines a set of Design Rules that apply to the application graph G, but not to the VR-PRUNE-E network graph introduced in Section IV. The Design Rules mainly concern interactions and control of DAs and DPAs, providing design space limits preventing application graph design faults related to variable token rates.

The VR-PRUNE(-E) approach for consistent deployment of variable token rates in application graphs is defined around a pair of dynamic actors labeled x and y, and an associated control actor q. For an example, consider Fig. 2. To properly define the relationships between x, y, and q, some formal definitions are required:

Definition 1: Chain. A chain is a non-empty sequence of actors $S = \{a_1, a_2, ..., a_N\}$ such that a_i and a_{i+1} are adjacent $\forall i = 1, 2, ..., N$.

Thus, S *connects* a_1 and a_N. In the case that all a_i are distinct, S is a *simple chain*.

Definition 2: Linked ports. Ports p_x and p_y are called linked ports, denoted as $\{p_x, p_y\}$, if (1) x and y are adjacent, or (2) x and y are connected by a simple chain $S = \{x, a_1, a_2, ..., a_N, y\}$.

As a special case of linked ports, if both p_x and p_y are DRPs, $\{p_x, p_y\}$ are called *linked DRPs*.

For all Design Rules, we assume that x and y are a pair of dynamic actors with linked DRPs $\{p_x, p_y\}$, and S_1 is a *connecting subchain* $S_1 = a_1, a_2, ..., a_N$ between them. In Fig. 2, $\{p_{x2}, p_{y1}\}$ are linked DRPs, and S_1 consists of a single actor a. Notice that in general there can be more than one connecting subchain between x and y. The Design Rules are:

1) The *Linked Port Control Rule* requires each pair of linked DRPs $\{p_x, p_y\}$, as well as every DRP within the actors $a_i \in S_1$ to be controlled by the same output control port p_q.
2) *Balanced Delay Rule*: The input control ports associated with the linked DRPs $\{p_x, p_y\}$, and each DRP within actors of S_1 must be connected to p_q with the same delay.
3) The *Connecting Subchain Rule* requires that for each actor $a_i \in S$ (a) a_i must be of type SPA or DPA, and (b) a_i may not belong to any connecting subchain S_2 that is associated with a dynamic actor $z \notin \{x, y\}$.
4) The *Single-sided Dynamism Rule* requires that all DRPs of the DA x have output direction, and all DRPs of DA y have input direction.
5) The *Encapsulation Rule*. Let $k \notin \{S_1\}$ be an actor connected to S_1. The encapsulation rule requires that (a) if $k \notin \{x, y\}$, k must belong to another connecting subchain $S_2 = b_1, b_2, ..., b_N$ controlled by a set of linked DRPs $\{p_{x2}, p_{y2}\}$ such that $parent(p_{x2}) = x$ and $parent(p_{y2}) = y$. If (b) $k \in \{x, y\}$, then k is allowed to be connect to $\{x, y\}$ only through a DRP p_k such that $parent(p_k) = k$.

For illustrations and examples on Design Rules, the reader is advised to refer to [17].

D. Dynamic Processing Graphs

VR-PRUNE(-E) actor types and Design Rules provide generic constraints that apply to all application graphs and their subgraphs and set a necessary basis for decidable graph consistency analysis. Besides the formal actor and design rule

definitions, VR-PRUNE(-E) uses the *Switch DPG* (sDPG) as a graph structure for deploying actors with variable token rates in a consistent manner. The sDPG was originally introduced with the VR-PRUNE MoC that is in terms of variable token rate behavior compatible with the proposed VR-PRUNE-E MoC. The proof for sDPG consistency analysis decidability is not repeated here, but the interested reader can refer to [17].

A valid sDPG consists of exactly two dynamic actors x and y, and exactly one control actor q. x and y are connected by any positive number of linked DRPs. As stated in Section III-C, a pair of linked DRPs may be connected by a chain of actors S, which in the case of sDPG can contain SPAs or DPAs. For each S, validness of an sDPG requires that no $fifo(p)$, where p is a DRP, may have $delay(fifo(p)) > 0$.

IV. The VR-PRUNE-E Network Graph

The previous section compactly presented the essential model concepts shared between VR-PRUNE and VR-PRUNE-E. In this section the MoC is elaborated to encompass distributed system concepts that are not part of VR-PRUNE [17].

The application graph G defines the computations of an application and assumes that 1) all computations are carried out on a single computing platform that can have one or more execution units (CPU cores or GPUs), and 2) data transfer between the actors of G is error-free. It is clear that these assumptions of the graph G are not well suited for distributed edge computing scenarios. For this reason, VR-PRUNE-E defines the network graph N that is formally, stepwise constructed from the graph G. N consists of a set of actors α and a set of edges Φ, $N = (\alpha, \Phi)$, which are copied instances (actors, edges) from the graph G, interconnected by additional graph primitives (ports and edges) for unreliable remote connections:

Interface edge (IE). The IE type of edge radically differs from the regular edge presented in Section III in that a) there is no assumption on the boundedness of the interface edge token capacity, and b) data loss can occur in an IE f — a token written to the source port of f may never appear at the sink port of f.

Interface port (IP). An interface port p is required to have $lrl(p) = 0$ and $url(p) = 1$, and hence the $atr(p)$ can vary at runtime within the range $[0, 1]$. An IP type of port may only be connected to an IE type of edge. The output IP p implements *fault-tolerant writes*, i.e. if the $fifo(p)$ cannot accept tokens, the actor $parent(p)$ needs to omit writing to $fifo(p)$ and continue its operation otherwise. Each input IP p must similarly implement *fault-tolerant reads*: if the $fifo(p)$ is unreachable, e.g., due to communication failure, $parent(p)$ must continue its operation otherwise.

The VR-PRUNE-E network graph N can be defined in several different topologies that offer various features. In the following, we define a particular type of network graph, the *bipartite network graph* (BNG), show how to construct it from the application graph G, and analyze its fault resilience.

A. The Bipartite Network Graph

The BNG is a network graph that consists of $m + n$ nodes and mn edges, forming a K_{mn} complete bipartite graph such that m nodes form the *server* set of nodes and n nodes form the *device* set of nodes. Each node of the server set is directly connected with every node of the device set, but nodes within the same set are not connected with each other. In terms of fault tolerance, nodes belonging to the same set can be considered as redundant instances of the same functionality, edge server or endpoint node. The BNG structure is very generic and can cover a variety of computational scenarios from the MLSP field including, but not limited to collaborative neural inference ($m = n = 1$), sensor network data acquisition and processing ($m = 1$, $n > 1$), and sensor networks with additional fault tolerance ($m > 1, n > 1$).

It is important to notice that in the BNG structure, individual nodes of the sets m and n are not single actors, but sets of actors, defined as described in the following section.

B. Constructing the Bipartite Network Graph

The BNG graph N is constructed from the application graph G in a nine-step procedure. For brevity, we adopt the notation $nport(n, p)$ which is the short form for port p of node n.

1) Identify the set of bridges $B \in F$ from graph G. Here, a bridge is an edge f such that (a) $p^+ = source(f)$ and $p^- = sink(f)$ are both SRPs with $atr(p^+) = atr(p^-) = 1$, and (b) removal of f from the graph increases its number of connected components. Discovery of graph bridges can be accomplished by established algorithms such as [43].
2) If the graph G has more than one bridge, select exactly one edge from B as the *partition edge* ρ.
3) Removing ρ leaves G in two connected components, G_c and G_s. G_c is the connected component that has the source port $source(\rho)$ of the partition edge, and G_s is the connected component with the sink port $sink(\rho)$ of the partition edge.
4) Determine degrees of redundancy $m \geq 1$ for G_s, and $n \geq 1$ for G_c. Both m and n must be positive integers.
5) Delete port $source(\rho)$ from actor $parent(source(\rho)) \in G_c$ and add m number of output IPs $p_c^1 ... p_c^m$ to $parent(source(\rho)) \in G_c$.
6) Delete port $sink(\rho)$ from actor $parent(sink(\rho)) \in G_s$ and add n number of output IPs $p_s^1 ... p_s^n$ to $parent(sink(\rho)) \in G_s$.
7) Instantiate n copies of $G_c \in A$ into α such that each $G_{c1}, G_{c2}, ..., G_{cn} \in \alpha$ are isomorphic with $G_c \in A$. The copies of G_c are referred to as the *device set*.
8) Instantiate m copies of $G_s \in A$ such that each $G_{s1}, G_{s2}, ..., G_{sm} \in \alpha$ are isomorphic with $G_s \in A$. The copies of G_s are referred to as the *server set*.
9) Instantiate and connect a new IE $e_{ji} \in \Phi$ such that $\{source(e_{ji}) = nport(G_{ci}, p_{cj}) \mid j \in \{1, 2, ..., m\}, i \in \{1, 2, ..., n\}\}$ and $\{sink(e_{ji}) = nport(G_{sj}, p_{si}) \mid j \in \{1, 2, ..., m\}, i \in \{1, 2, ..., n\}\}$

This nine-step procedure results in the VR-PRUNE-E BNG N. It is important to notice that some actors in α have IP type of ports that make them functionally differ from the actor types given in Section III-A that were allowed to exist in the

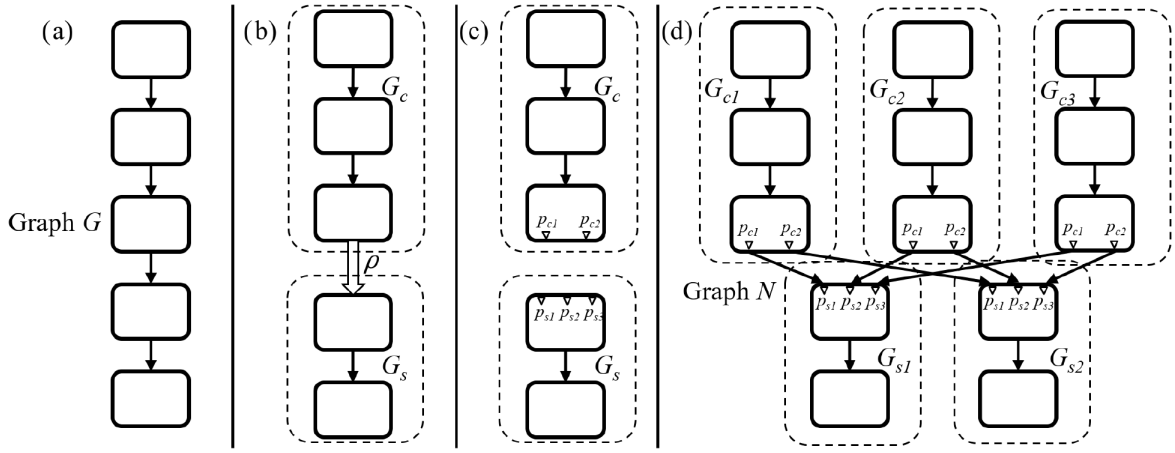


Fig. 3. Phases a) through d) of constructing the BNG N from the application graph G for a configuration of $m = 2$ and $n = 3$. Phase a) illustrates Step 1, whereas phase b) corresponds to Steps 2 and 3, phase c) corresponds to Steps 4 through 6, and phase d) Steps 7 through 9.

application graph G. Fig. 3 shows the steps of constructing a BNG N from the application graph G.

C. Faults and Error Resilience

The VR-PRUNE-E model considers two kinds of operational faults that are relevant to edge computing systems:

Connection failure can take place in Interface Edges (again, not in Regular Edges), and can last for an arbitrary period – also being permanent. Upon edge f failure, the actor $source(f)$ cannot write tokens to f, and the actor $sink(f)$ cannot read tokens from f. An edge failure can but is not required to cause data loss in edge f.

BNG resilience against connection failure. By definition, the VR-PRUNE-E model considers the occurrence of data loss to take place only within IE types of edges. Since IE edges are not allowed to exist within the application graph G, but are strictly introduced within the BNG network graph N as defined in Section IV-B, it can be concluded that IEs can exist in N only between instances of the device set and the server set. An IE f is necessarily connected to an IP type of port both at $source(f)$ and $sink(f)$. As defined previously in Section IV, actors with IPs are required to support fault-tolerant writes for output IPs, and fault-tolerant reads for input IPs. Hence, for the period when edge $f = fifo(p)$ cannot accept or provide tokens, $parent(p)$ will set $atr(p)$ to zero and continue its operation.

Node failure affects all the actors within a connected component (server set instance G_{sj} or device set instance G_{ci}) simultaneously and is considered permanent. On the VR-PRUNE-E model level, a device failure prevents all actors within the affected connected component from firing after the failure has taken place.

BNG resilience against node failure. A node failure within the server set $G_{sj}, j \in \{1, 2, ..., m\}$ prevents the actors of the affected connected component G_{sj} from consuming tokens from the attached IEs, or similarly prevents the actors from affected the connected component $G_{ci}, i \in \{1, 2, ..., n\}$ from producing tokens to the attached IEs. Since the remaining functioning connected components of α are connected to the actors of the failed component solely over IEs, the actors within each functioning component can set the $atr(p)$ to zero and otherwise continue normal operation; here, $f = fifo(p)$ is the IE between the failed component and the functioning component.

The connection failure and node failure are abstractions of corresponding real-world phenomena where a remote connection link (e.g., wireless connection) or node (physical server or endpoint device) fails. Whereas the abstract model is sufficient for theoretical analysis of system fault tolerance, it allows some freedom to practical implementations of the described behavior. Section V describes the proposed Edge-PRUNE Framework that supports the VR-PRUNE-E MoC and BNGs.

D. VR-PRUNE-E vs. VR-PRUNE

As stated earlier, the proposed VR-PRUNE-E MoC is an elaboration of the VR-PRUNE MoC [17]. VR-PRUNE-E introduces the network graph (Section IV) and the new graph elements of interface FIFO and interface port. The interface FIFO and interface port radically differ from the regular edge and port types introduced in VR-PRUNE; interface edges are unbounded and can be expected to lose tokens, whereas interface ports implement fault-tolerant reads/writes to mitigate edge and node failure consequences. A brief discussion on the future co-evolvement between VR-PRUNE and VR-PRUNE-E is presented Section VII.

V. The Edge-PRUNE Framework

The Edge-PRUNE Framework (EPRF) follows the approach of model-based design and software synthesis (similar to, e.g., [44]) for specifying the application and the underlying computing infrastructure. In addition to the network graph N, EPRF also requires an abstraction of the underlying computing platforms, which is provided in the form of an undirected *platform graph* that lists the processing units (such as CPU cores and GPUs) and specifies their interconnections. Consequently, also a *mapping file*, which assigns actors to processing units, is required. Unlike the network graph, the platform graph and the

mapping file are specific for each computing platform in the distributed system: in each platform-specific mapping file, each actor is defined either for local or remote execution.

Compiler. The most important software tool related to EPRF is the compiler, which requires as input the network graph, actor behavior files, the platform graph and a mapping file. Given this input, the EPRF compiler produces platform-specific software code, which is taken by the platform-specific compiler (for instance, `gcc`) to produce an executable binary. The EPRF compiler streamlines implementation of distributed computing: at minimum, only the mapping file needs to be modified to reflect changes in the distributed scenario. The behavior of each actor is described in a separate file using plain C or OpenCL language, or by invoking an ONNX[2] module. Each actor description has *initialization*, *firing* and *deinitialization* behaviors defined.

Explorer. A central research topic in distributed DNN inference [5], [24] has been design space exploration for endpoint/server DNN partitioning. In contrast to most frameworks, EPRF adopts a profiling-based approach: the EPRF Explorer tool indexes the $|N|$ actors of the network graph into an ascending order based on precedence, and generates $|N|$ mapping file pairs (one for the endpoint device, and one for the server) by shifting the client-server partitioning point actor-by-actor from the inference input towards the inference output; evidently, only application graph bridges provide valid partition points. In addition to the mapping files, the explorer also generates client-side and server-side scripts that enable execution-time profiling of all mapping alternatives. The Explorer tool is limited to network graph configurations with $m = n = 1$. Besides the EPRF Explorer tool, it is also possible to apply other partitioning algorithms to Edge-PRUNE. An example of this is shown in Appendix A.

Analyzer. The EPRF tools include a prototype graph analyzer, which analyzes application graph G consistency against the VR-PRUNE(-E) Design Rules.

A. EPRF Runtime

The heterogeneous parallel processing and distributed computing features of EPRF have been implemented to a compact C language library, which is compiled with the actor behavior files and the EPRF compiler-generated files into an executable, separately for the endpoint device side and for the server side.

The EPRF runtime library is built on Linux inter-process communication and networking functionalities. Each actor that has been mapped for execution on a CPU core, is instantiated as a separate thread, and actor data exchange over FIFOs is synchronized by *mutex* primitives. Leveraging GPUs or machine learning accelerators is achievable either through a computing platform specific ONNX Runtime[3], or by directly specifying the GPU/accelerator code in the OpenCL language. In both cases, memory management, data transfers and synchronization are handled by EPRF.

[2]https://github.com/onnx/onnx
[3]https://onnxruntime.ai/

TABLE II
PLATFORMS USED FOR EXPERIMENTS

Tag	CPU	Oper. Sys.
i7	Intel Core i7-12700H, 4.7 GHz, 14(20) cores	Ubuntu 22.04
i7-2	Intel Core i7-8650U, 4.2 GHz, 4(8) cores	Ubuntu 22.04
M1	ARM Cortex-A55, 1.992 GHz, 4 cores	Ubuntu 20.04
ryzen	AMD Ryzen 5955WX, 4.0 GHz, 32-core	Ubuntu 22.04

The Interface Edges (IEs) for remote connections between the endpoint devices and server(s) are implemented by Linux *sockets* such that each transmit/receive FIFO pair in an application graph receives a dedicated TCP port number.

B. Machine Learning Hardware and Software Support

EPRF supports both PyTorch and TensorFlow, as well as GPU and accelerator leverage through ONNX, which is an open standard for machine learning interoperability. In practice, the application developer specifies a (deep) neural network architecture in PyTorch or TensorFlow, and invokes a common software tool such as `torch.onnx` or `tf2onnx` to convert a neural network into a portable ONNX module that contains both the neural network architecture and weights. Recent versions of the ONNX standard have also included classical signal processing operators such as (I)FFT and window functions (Hamming, Blackman, etc.).

EPRF has in-built support for executing neural network layers that are packaged into the ONNX format. In practice, each actor within the EPRF application graph invokes an actor-specific ONNX module, delivering the input tensor data to the module from EPRF FIFO edges, and writing the ONNX module output to outgoing FIFO edges of the actor. For this purpose, Edge-PRUNE includes a software tool for decomposing an ONNX module to individual DNN layers and operators. However, keeping several ONNX nodes within a single ONNX module allows reducing the number of dataflow actors, which can be beneficial for increasing execution efficiency similar to the concept of actor merging [45].

Use of the EPRF ONNX support feature is not mandatory. Actor behavior can also be expressed in C, OpenCL, or Halide [46] languages, or using platform-specific neural network acceleration libraries such as ARM CL or OneDNN [17].

VI. EXPERIMENTAL RESULTS

The experiments cover three major areas: 1) application runtime evaluation in collaborative inference, 2) scaling of execution time as a function of device count, and 3) observations on EPRF fault tolerance. The device types used for experiments are listed in Table II. In all experiments where physically distributed processing was involved, devices were interconnected by wired 100 Mbit Ethernet.

A. CBAM ResNet-50

This experiment covers collaborative endpoint — server inference using the ResNet-50 [47] backbone amended with

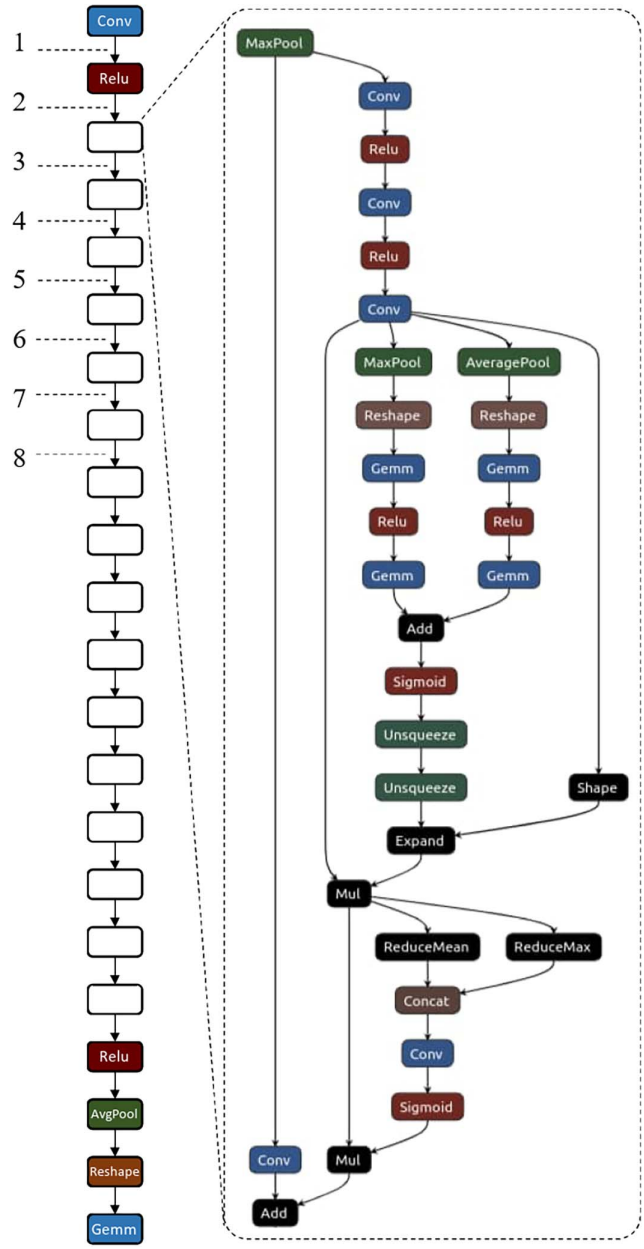


Fig. 4. The CBAM ResNet-50 dataflow graph. The 16 rounded rectangles colored in white are *bottleneck* subgraphs that contain 31 actors each. Colored nodes are dataflow actors.

Convolutional Block Attention Modules (CBAM) [48] for ImageNet 1k image classification. The dataflow graph of the CBAM ResNet-50, depicted in Fig. 4, contains 458 dataflow actors and 537 edges. Overall, the graph consists of 16 *bottleneck* subgraphs that are connected sequentially. Internally, the bottleneck blocks contain parallel edges, which prevents partitioning the CBAM ResNet-50 graph within bottleneck blocks. However, considering the edges that interconnect individual bottlenecks blocks, as well as the actors preceding and trailing the bottlenecks, altogether 21 graph bridges can be identified.

Fig. 5 shows the CBAM ResNet-50 collaborative inference execution times using the proposed EPRF framework for

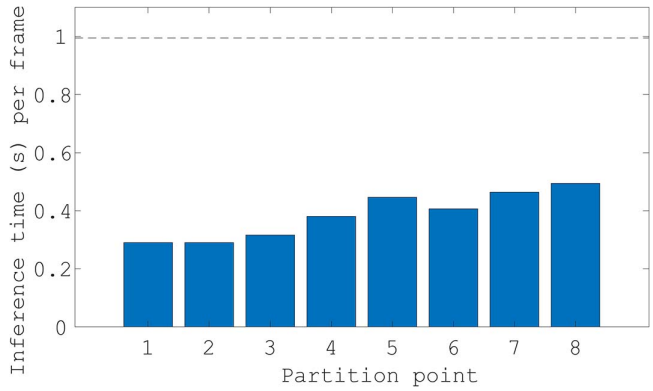


Fig. 5. EPRF endpoint device inference time as a function of partition point index for the CBAM ResNet-50. The endpoint device used was M1, and the server was the i7. The dashed line shows the inference time for endpoint-only inference. Inference time includes endpoint-server communication.

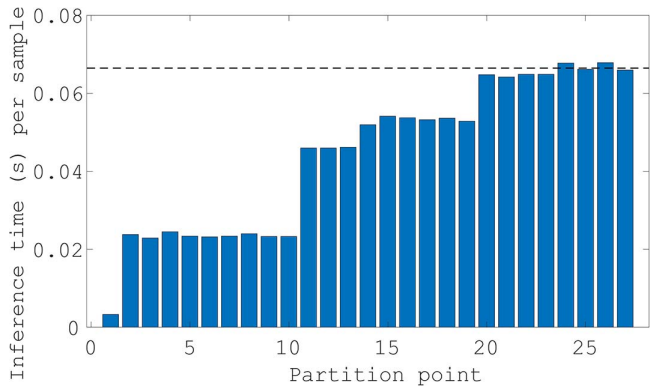


Fig. 6. EPRF endpoint device inference time as a function of partition point index for the Localization CNN. The endpoint device used was M1, and the server was the i7. The dashed line shows the inference time for endpoint-only inference. Inference time includes endpoint-server communication.

partition points 1 through 8. Endpoint-only inference on the M1 device takes 0.995 seconds (horizontal dashed line), compared to which the partition points 1 through 8 provide speed-ups between $3.4\times \ldots 2.0\times$, respectively.

B. Localization CNN

The 2nd use case for collaborative inference was the *Localization CNN* (Convolutional Neural Network) introduced in [49] that takes radar range-angle heatmaps as input for regression, to output a location estimate. The dataflow graph of this CNN is a simple chain until the 2nd last actor, containing 30 actors and 29 edges of which 28 are bridges. Fig. 6 shows the endpoint-server collaborative inference time for all the 28 partition points. Using partition point 1 provides an extraordinary speedup of $20\times$, whereas partition points 2 through 10 provide a speed-up of $2.8\times$ compared to endpoint-only inference time of 0.066 seconds. If the transmitted data needs to be protected from malicious parties [4], partition point 10 is superior to partition point 1, however the trade-off with throughput becomes significant.

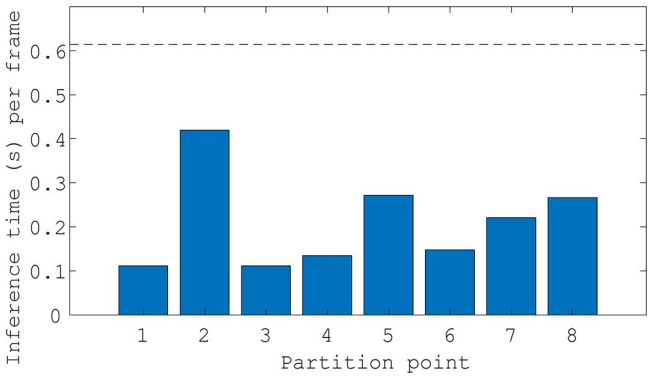


Fig. 7. EPRF endpoint device inference time as a function of partition point index for the EfficientViT-B2 transformer. The endpoint device used was M1, and the server was the i7. The dashed line shows the inference time for endpoint-only inference. Inference time includes endpoint-server communication.

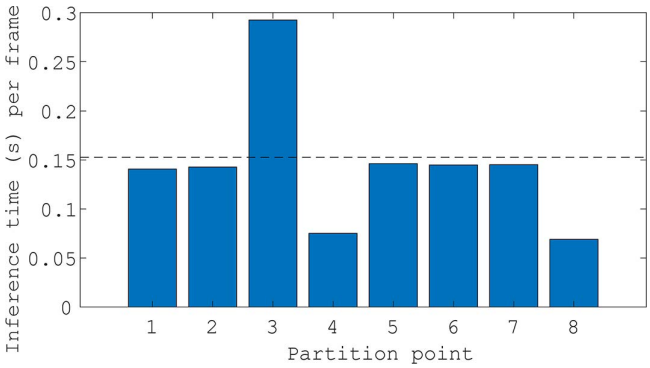


Fig. 8. EPRF endpoint device inference time as a function of partition point index for the MobileNet v1 1.0 with input frame resolution 224× 224. The endpoint device used was M1, and the server was the i7. The dashed line shows the inference time for endpoint-only inference. Inference time includes endpoint-server communication.

C. EfficientViT-B2

As an example of recent transformer-based architectures, the EfficientViT-B2 [50] network was selected. Fig. 7 shows that partition points 1 and 3 yield a 5.5× speedup compared to endpoint-only inference when EPRF is used for collaborative inference.

D. MobileNet

As an example of a mobile-oriented CNN, the MobileNet V1 [51] 224× 224 architecture was selected. Fig. 8 shows the endpoint processing time as a function of partition point. It can be seen that this mobile-oriented architecture enhances collaborative inference performance somewhat less than larger networks; still, an inference time speedup of 2.2× is achievable at partition point 8 compared to endpoint-only inference.

E. Comparison Against Previous Work

Autodidactic Neurosurgeon (ANS) [25] proposes a similar approach for collaborative inference as EPRF, and provides two

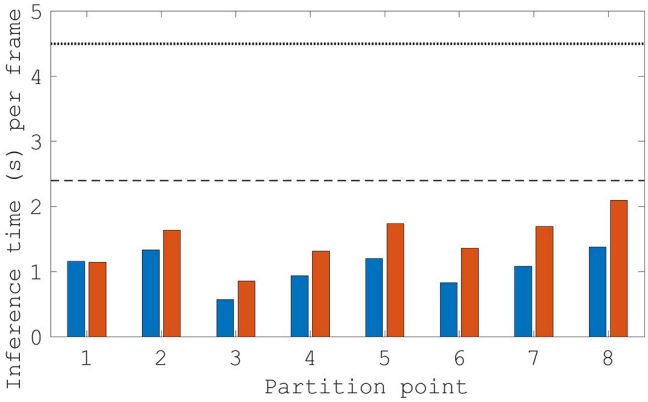


Fig. 9. Endpoint device inference time as a function of partition point index for VGG16, comparison between EPRF (proposed, in blue) and ANS (in red) [25]. The endpoint device used was M1, and the server was the i7. The dashed line shows the EPRF inference time for endpoint-only inference, whereas the dotted line is for ANS. Inference time includes endpoint-server communication.

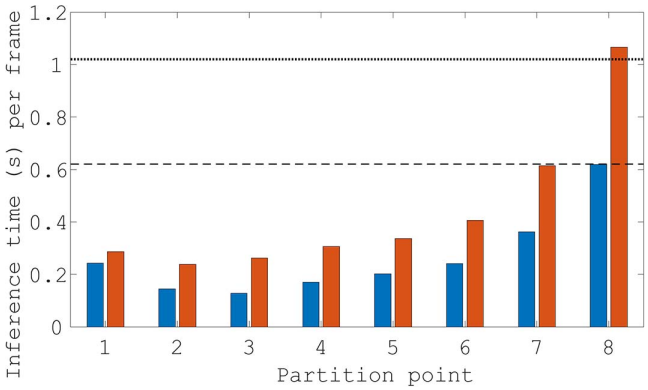


Fig. 10. Endpoint device inference time as a function of partition point index for Yolo v2 tiny, comparison between EPRF (proposed, in blue) and ANS (in red) [25]. The endpoint device used was M1, and the server was the i7. The dashed line shows the EPRF inference time for endpoint-only inference, whereas the dotted line is for ANS. Inference time includes endpoint-server communication.

neural networks, VGG16 [52] for ImageNet 1k classification and Yolo v2 [53] tiny for COCO object detection. In terms of graph structure, both in VGG and Yolo, all edges are bridges.

For performance comparison, these neural networks were implemented in EPRF, and the collaborative inference results are shown in Figs. 9 and 10. For both Yolo and VGG applications EPRF clearly outperforms ANS in collaborative and endpoint-only inference. One of the major reasons behind this is that EPRF is implemented in C/C++ and leverages ONNX whereas ANS is completely written in Python, and leverages PyTorch.

F. Fault-Tolerant Vehicle Image Classification

The last use case highlights the EPRF support for redundant endpoint and server nodes using a vehicle image classification CNN [54]. The CNN consists of 19 neural network layers interconnected by 18 edges, of which all are bridges. BNG graph alternatives with $1 \ldots 2$ servers and $1 \ldots 6$ endpoint devices

TABLE III
VEHICLE IMAGE CLASSIFICATION COLLABORATIVE INFERENCE TIME PER FRAME IN MILLISECONDS FOR $1 \leq m \leq 2$ SERVERS AND $1 \leq n \leq 6$ ENDPOINT DEVICES. UPPER HALF: ENDPOINT-SIDE TIME, LOWER HALF: SERVER-SIDE TIME. MEASURED ON THE I7-2 PLATFORM

N. OF ENDPOINTS	1	2	3	4	5	6
SINGLE-SERVER	4.7	4.9	5.2	5.4	6.2	7.0
DUAL-SERVER	4.9	5.1	5.2	5.5	6.3	7.1
SINGLE-SERVER	4.8	5.7	6.5	7.1	8.7	9.7
DUAL-SERVER	4.9	5.8	6.5	7.1	8.9	9.6

TABLE IV
VEHICLE IMAGE CLASSIFICATION LARGE-SCALE SYSTEM INFERENCE TIME MEDIAN PER FRAME IN MILLISECONDS FOR $1 \leq m \leq 2$ SERVERS AND $16 \leq n \leq 128$ ENDPOINT DEVICES. TOPMOST THIRD: ENDPOINT-SIDE TIME, MIDDLE THIRD: SERVER-SIDE TIME. BOTTOMMOST-THIRD: PER-CLIENT PROCESSING TIME OF SERVERS. MEASURED ON THE *RYZEN* PLATFORM

N. OF ENDPOINTS	16	32	64	128
CLIENT (ONE-SERVER)	2.7	2.3	*2.3*	*2.6*
CLIENT (DUAL-SERVER)	2.6	2.3	*2.4*	*2.5*
SERVER (ONE) TOTAL	121	192	*265*	*521*
SERVER (DUAL) TOTAL	105	201	*283*	*592*
SERVER (ONE) PER CLIENT	7.6	6.0	*4.1*	*4.1*
SERVER (DUAL) PER CLIENT	6.6	6.3	*4.4*	*4.6*

were established. For an experiment to explore EPRF scalability, the BNG graph alternatives were first executed as a virtual distributed system using the multiprocessing infrastructure of Ubuntu Linux: each endpoint and server node was fixed to be executed on a dedicated processor core. Table III shows that adding an endpoint increases per-node inference time by 11% on average, whereas adding a second server for fault tolerance causes insignificant processing time overhead.

Large-scale deployment performance scaling of EPRF was experimented on the 32-core *ryzen* architecture for 16, 32, 64 and 128 endpoint nodes, following a similar setting as in Table III. Table IV shows that EPRF-based systems scale well to tens of endpoint devices, and that adding a second server — expectedly — does not impose significant a performance overhead as the redundant server is an independent entity in terms of communication and computation. It is important to notice that the 64 and 128 node setup performance results are to be considered only as indicative, since with these high numbers of endpoint nodes there is more than one endpoint process executing per core.

Finally, physically distributed computing was experimented on a system consisting of an i7 device and $4\times$ M1 devices. This setup enabled experimenting with real-life behavior of EPRF in terms of connection failure (disconnecting Ethernet cable temporarily) and node failure (powering down node) for confirming EPRF fault tolerance.

Both the virtually and physically distributed setups behaved identically and confirmed that a BNG based fault-tolerant system can continue its operation in the case of permanent node or temporary connection failures.

G. BNG Efficiency and Overhead

In large-scale distributed systems communication and computation needs to be organized efficiently to minimize overhead. In the following, the BNG scalability of clients, servers, and communication channels is discussed from this point of view.

1) Each client node is an independent entity that scales only into one direction with the BNG graph: the number of servers. For the use cases we have identified, 1, 2, or perhaps 3 as realistic numbers of servers from the fault tolerance point of view. Scaling the number of servers from 1 to 2 or 3 only implies replicating client output ports, and copying the data generated by the client to one or two further outputs. The performance impact of this copying is negligible.
2) The scalability of the point-to-point connections between the clients and the servers depends on the underlying communication network bandwidth. Aspects that affect bandwidth sufficiency are application partition point (data size, l), the number of clients (n) and servers (m), and communication frequency. The bandwidth requirement of the BNG is the product nml. The BNG graph can be executed within a possible time constraint T as long as the underlying communication network has enough bandwidth to transmit nml bytes within period T.
3) Each server node is an independent entity that scales in the number of input ports as the number of clients is increased. In Table IV the BNG graph has been scaled to reach up to 128 client nodes, which implies that each server node has 128 input ports that it needs to monitor, receive the data from, and process according to the application. The scaling of the server node is clearly critical to the BNG scalability, further discussed below.

A server node within a BNG graph needs to monitor one input port for each client node in the network. The runtime behavior of a server node can be divided to three activities: 1) application processing, 2) waiting for client input, and 3) everything else. Out of these, the last category can be interpreted as *overhead* specific to the BNG server implementation. This overhead was measured for the large-scale BNG graph configurations of 16, 32, 64 and 128 client nodes. The overhead was highest for the 32-client configuration, totaling in 385 microseconds (12 μs per client) on the *ryzen* platform. Percentually, this equals to around 0.2% of the total server-side runtime.

Assuming the server overhead of 12 μs per client, each client transmitting one message per second, and max. 10% of the server node's 1 s time budget allocated to overhead, the server node could process the data of more than 8000 clients within the time budget.

VII. Discussion and Conclusion

The proposed VR-PRUNE-E MoC is an elaboration of the previously introduced VR-PRUNE MoC, which may raise the question on the future co-evolvement of the MoCs. In general, future developments, whether theoretical (e.g., new types of DPGs) or practical (software framework features) should primarily be based on VR-PRUNE-E due to its wider applicability. However, if a future contribution strictly applies to single-system computing, building on VR-PRUNE could be justifiable. Migration of VR-PRUNE applications to Edge-PRUNE is trivial, as only the mapping file (See Section V) needs to be adjusted to partition the actors from local to distributed execution across the edge computing system. To achieve the BNG fault tolerance, the application partitioning must follow the steps indicated in Section IV-B.

The experimental section of the paper showed that Edge-PRUNE provides a flexible framework for deploying neural networks across distributed systems that consist of powerful servers and low-cost endpoint devices. Besides the fault tolerance features, Edge-PRUNE also provides significant performance, as shown in the comparison to previous work [25].

In this paper we have presented VR-PRUNE-E, a model of computation for distributed signal processing and machine learning, which addresses aspects of fault tolerance. We have formally defined the VR-PRUNE-E model of computation and design rules and shown that by constructing a Bipartite Network Graph results in a distributed system that can continue its operation normally after the failure of any single computation device or communication link. The experimental section has presented Edge-PRUNE, a framework that heeds the VR-PRUNE-E model of computation. The experimental evaluation showed that the EPRF framework is capable of running deep neural networks efficiently in distributed scenarios and practically implements fault tolerance functionality on physically distributed devices. Our perception is that Edge-PRUNE provides a significant contribution to the edge computing and signal processing communities by providing a scalable, flexible and freely available software framework for deploying edge computing systems. The comparison to previous works in Table I highlighted that although several related works have been published, software releases are scarce. Besides the software release, the fault tolerance aspect of Edge-PRUNE can be expected to stir further research in the dataflow community, where fault tolerance has been little explored compared to aspects such as throughput and decidability analyses.

For future work, we regard elaboration of Edge-PRUNE towards supporting semantic communication [40] as a promising direction. Another potential direction for future work could be studying of model-based partition point selection [24], [25], [27] in the context of VR-PRUNE-E.

Appendix A
Application of Other Partitioning Methods

Previous literature on collaborative inference has proposed a multitude of approaches for client-server DNN partitioning. This section shows the results of applying the NeuroSurgeon partitioning algorithm [24, Algorithm 1] (NS) to two application examples. The NS algorithm can select the partition point (PP) for best energy or best latency. Here, we concentrate on latency.

TABLE V
Vehicle Image Classification Partition Point Selection Using the NeuroSurgeon [24] Algorithm. ΣTM Denotes Sum of Client-Side DNN Layer Latencies, ΣTC Sum of Server-Side DNN Layer Latencies, and TU Data Transfer Latency. Lowest Cost in Boldface, 2nd Best Underlined. The Server Device Used Was *i7*, Client Device *M1*, Interconnected by 100Mbit Ethernet. Column 2-4 Units Are in Milliseconds

PP Index	ΣTM	ΣTC	TU	NS Cost
1	1.9	3.7	11.1	**16.7**
2	21.7	2.6	118.0	142.3
3	25.8	2.0	118.0	145.8
4	29.1	1.6	29.5	<u>60.2</u>
5	65.0	0.6	29.5	95.0
6	67.4	0.5	29.5	97.4
7	68.3	0.4	7.4	76.1
8	68.9	0.4	7.4	76.6
9	69.1	0.4	7.4	76.9
10	69.5	0.4	7.4	77.3
11	69.8	0.3	7.4	77.5
12	75.2	0.0	0.0	75.2
13	75.5	0.0	0.0	75.6
14	75.7	0.0	0.0	75.7
15	75.9	0.0	0.0	75.9
16	76.0	0.0	0.0	76.0

The NS algorithm PP selection introduces a cost function with three components: sum of latencies for DNN layers executed on the client side (ΣTM), sum of latencies for DNN layers executed on the server side (ΣTC), and PP specific data transfer latency (TU). The cost function is evaluated for each candidate PP, and the PP candidate with minimum cost is selected as the final PP. For computing ΣTM and ΣTC the NS algorithm leverages DNN layer-specific latency estimates, and for TU the inter-layer data (tensor) size divided by network bandwidth.

Table V shows the NS cost calculation for each partition point for the Vehicle Classification CNN (Subsection VI-F), and Table VI, respectively, for the Localization CNN (Subsection VI-B). Comparing the PP-wise cost of the Localization CNN to the PP selection provided by the EPRF Explorer tool (Fig. 6) reveals slight differences in the partitioning cost calculation: with the NS algorithm the cost steadily increases from PP 1 through PP 16, whereas the Explorer tool shows a steady cost from PP 2 through PP 10.

The EPRF Explorer tool PP selection is based on latency profiling at each PP, and hence its output reflects the true latency that can be perceived for a deployed system assuming that the compute platforms and network characteristics remain unchanged. The NS PP selection algorithm, on the other hand, computes the PP cost assuming *DNN layer-specific* latency estimates. Much of the cost calculation differences between the NS algorithm and the Explorer can be attributed to the fact that

TABLE VI
LOCALIZATION CNN PARTITION POINT SELECTION USING THE NEUROSURGEON [24] ALGORITHM. ΣTM DENOTES SUM OF CLIENT-SIDE DNN LAYER LATENCIES, ΣTC SUM OF SERVER-SIDE DNN LAYER LATENCIES, AND TU DATA TRANSFER LATENCY. LOWEST COST IN BOLDFACE, 2ND BEST UNDERLINED. THE SERVER DEVICE USED WAS *I7*, CLIENT DEVICE *M1*, INTERCONNECTED BY 100MBIT ETHERNET. COLUMN 2-4 UNITS ARE IN MILLISECONDS

PP INDEX	ΣTM	ΣTC	TU	NS COST
1	1.6	15.2	1.6	**18.4**
2	2.9	14.9	26.2	<u>44.0</u>
3	3.7	14.7	26.2	44.7
4	4.3	14.6	26.2	45.2
5	11.8	14.3	26.2	52.3
6	12.6	14.1	26.2	52.9
7	13.2	14.0	26.2	53.4
8	18.2	13.6	26.2	58.1
9	19.0	13.5	26.2	58.8
10	19.6	13.4	26.2	59.3
11	29.1	12.7	52.4	94.3
12	30.3	12.5	52.4	95.2
13	31.3	12.3	52.4	96.0
14	50.3	11.4	52.4	114.2
15	51.5	11.2	52.4	115.2
16	52.5	11.1	52.4	116.0
17	55.5	10.9	13.1	79.5
18	56.1	9.3	13.1	78.5
19	56.5	9.2	13.1	78.9
20	89.5	7.6	0.1	97.2
21	89.8	3.2	0.1	93.2
22	90.0	3.2	0.1	93.4
23	90.2	3.2	0.1	93.5
24	90.4	3.2	0.1	93.7
25	90.6	0.0	0.1	90.7
26	90.7	0.0	0.1	90.8

the NS latency model assumes that each DNN layer executes in isolation, whereas the Explorer tool's full-application profiling includes real-life factors such as code and data locality, cache effects, etc. that affect latency.

Both PP selection approaches have their strengths and weaknesses. Whereas the Explorer tool is accurate, its results are valid only as long as all system parameters remain unchanged. The model-based NS algorithm, on the other hand, can produce a PP selection decision without re-profiling if a latency estimate is available for each layer in the DNN.

ACKNOWLEDGMENT

The authors thank Alfons Václavík for technical contributions.

REFERENCES

[1] A. Elsts et al., "Enabling healthcare in smart homes: The SPHERE IoT network infrastructure," *IEEE Commun. Mag.*, vol. 56, no. 12, pp. 164–170, Dec. 2018.

[2] F. U. M. Ullah et al., "AI-assisted edge vision for violence detection in IoT-based industrial surveillance networks," *IEEE Trans. Ind. Informat.*, vol. 18, no. 8, pp. 5359–5370, Aug. 2021.

[3] T. Jing, X. Tian, H. Hu, and L. Ma, "Deep learning-based cloud–edge collaboration framework for remaining useful life prediction of machinery," *IEEE Trans. Ind. Informat.*, vol. 18, no. 10, pp. 7208–7218, Oct. 2022.

[4] Z. He, T. Zhang, and R. B. Lee, "Model inversion attacks against collaborative inference," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2019, pp. 148–162.

[5] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *Proc. Workshop Mobile Edge Commun.*, 2018, pp. 31–36.

[6] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2348–2359, Nov. 2018.

[7] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "SPINN: synergistic progressive inference of neural networks over device and cloud," in *Proc. Annu. Int. Conf. Mobile Comput. Netw.*, 2020, pp. 1–15.

[8] M. Almeida, S. Laskaridis, S. I. Venieris, I. Leontiadis, and N. D. Lane, "DynO: Dynamic onloading of deep neural networks from cloud to device," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 21, no. 6, pp. 1–24, 2021.

[9] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.

[10] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[11] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," *IEEE Trans. Signal Process.*, vol. 44, no. 2, pp. 397–408, Feb. 1996.

[12] M. Abadi et al., "TensorFlow: A system for Large-Scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2016, pp. 265–283.

[13] R. Xie, H. Huttunen, S. Lin, S. S. Bhattacharyya, and J. Takala, "Resource-constrained implementation and optimization of a deep neural network for vehicle classification," in *Proc. Eur. Signal Process. Conf.*, 2016, pp. 1862–1866.

[14] S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, *Handbook of Signal Processing Systems.* 3rd edn. New York, NY, USA: Springer, 2018.

[15] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," in *Proc. Int. Conf. Acoust., Speech, Signal Process. (ICASSP),* vol. 1, Apr. 1993, pp. 429–432.

[16] M. Mattavelli, I. Amer, and M. Raulet, "The reconfigurable video coding standard [standards in a nutshell]," *IEEE Signal Process. Mag.*, vol. 27, no. 3, pp. 159–167, May 2010.

[17] J. Boutellier, Y. Ma, J. Wu, M. Khan, and S. S. Bhattacharyya, "VR-PRUNE: Decidable variable-rate dataflow for signal processing systems," *IEEE Trans. Signal Process.*, vol. 70, pp. 1819–1833, 2022.

[18] J. Boutellier, B. Tan, and J. Nurmi, "Edge-PRUNE: Flexible distributed deep learning inference," 2022, *arXiv:2204.12947*.

[19] J. Boutellier, B. Tan, and J. Nurmi, "Fault-tolerant collaborative inference through the Edge-PRUNE framework," in *Proc. Int. Conf. Mach. Learn. (ICML) Workshop Dyn. Neural Netw.*, 2022, pp. 1–6.

[20] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Trans. Signal Process.*, vol. 49, no. 10, pp. 2408–2421, Oct. 2001.

[21] A. Honorat, M. Dardaillon, H. Miomandre, and J.-F. Nezan, "Automated buffer sizing of dataflow applications in a high-level synthesis workflow," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 17, no. 1, pp. 1–26, 2024.

[22] V. Venkataramani, B. Bodin, A. Kulkarni, T. Mitra, and L.-S. Peh, "Time-predictable software-defined architecture with SDF-based compiler flow for 5G baseband processing," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 1553–1557.

[23] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, "PREESM: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming," in *Proc. Eur. Embedded Des. Educ. Res. Conf.*, 2014, pp. 36–40.

[24] Y. Kang et al., "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 615–629, 2017.

[25] L. Zhang, L. Chen, and J. Xu, "Autodidactic neurosurgeon: Collaborative deep inference for mobile edge intelligence via online learning," in *Proc. Web Conf.*, 2021, pp. 3111–3123.

[26] L. Zeng, E. Li, Z. Zhou, and X. Chen, "Boomerang: On-demand cooperative deep neural network inference for edge intelligence on the industrial internet of things," *IEEE Netw.*, vol. 33, no. 5, Sep./Oct. 2019.

[27] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive DNN surgery for inference acceleration on the edge," in *Proc. INFOCOM or IEEE Conf. Comput. Commun.*, 2019, pp. 1423–1431.

[28] A. Banitalebi-Dehkordi, N. Vedula, J. Pei, F. Xia, L. Wang, and Y. Zhang, "Auto-split: A general framework of collaborative edge-cloud AI," in *Proc. ACM SIGKDD Conf. Knowl. Discovery & Data Mining*, 2021, pp. 2543–2553.

[29] Z. Gao, S. Sun, Y. Zhang, Z. Mo, and C. Zhao, "EdgeSP: Scalable multi-device parallel DNN inference on heterogeneous edge clusters," in *Proc. Int. Conf. Algorithms Archit. Parallel Process.,* New York, NY, USA: Springer, 2021, pp. 317–333.

[30] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2017, pp. 328–339.

[31] C.-H. Tu, Q. Sun, and M.-H. Cheng, "On designing the adaptive computation framework of distributed deep learning models for internet-of-things applications," *J. SuperComput.*, vol. 77, no. 11, pp. 13191–13223, 2021.

[32] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "MoDNN: Local distributed mobile computing system for deep neural network," in *Proc. Des., Automat. Test Eur. Conf. & Exhib. (DATE)*, 2017, pp. 1396–1401.

[33] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "IONN: Incremental offloading of neural network computations from mobile devices to edge servers," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 401–411.

[34] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "JointDNN: An efficient training and inference engine for intelligent mobile cloud computing services," *IEEE Trans. Mobile Comput.*, vol. 20, no. 2, pp. 565–576, Feb. 2021.

[35] H. Li, C. Hu, J. Jiang, Z. Wang, Y. Wen, and W. Zhu, "JALAD: Joint accuracy-and latency-aware deep structure decoupling for edge-cloud execution," in *Proc. IEEE Int. Conf. Parallel Distrib. Syst. (ICPADS)*, 2018, pp. 671–678.

[36] B. Zhang, T. Xiang, H. Zhang, T. Li, S. Zhu, and J. Gu, "Dynamic DNN decomposition for lossless synergistic inference," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst. Workshops (ICDCSW)*, 2021, pp. 13–20.

[37] M. Rocklin et al., "DASK: Parallel computation with blocked algorithms and task scheduling," in *Proc. SciPy*, 2015, pp. 126–132.

[38] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele, "Scenario-based design flow for mapping streaming applications onto on-chip many-core systems," in *Proc. Int. Conf. Compilers, Archit. Synthesis Embedded Syst.*, 2012, pp. 71–80.

[39] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing*, J. L. Rosenfeld, Ed. Stockholm, Sweden: North Holland, 1974, pp. 471–475.

[40] H. Xie, Z. Qin, G. Y. Li, and B.-H. Juang, "Deep learning enabled semantic communication systems," *IEEE Trans. Signal Process.*, vol. 69, pp. 2663–2675, 2021.

[41] Y. Sun et al., "Multi-functional RIS-assisted semantic anti-jamming communication and computing in integrated aerial-ground networks," *IEEE J. Sel. Areas Commun.*, vol. 42, no. 12, pp. 3597–3617, Dec. 2024.

[42] K. An et al., "Exploiting multi-layer refracting RIS-assisted receiver for HAP-SWIPT networks," *IEEE Trans. Wireless Commun.*, vol. 23, no. 10, pp. 12638–12657, Oct. 2024.

[43] R. E. Tarjan, "A note on finding the bridges of a graph," *Inf. Process. Lett.*, vol. 2, no. 6, pp. 160–161, 1974.

[44] J. Castrillon, R. Leupers, and G. Ascheid, "MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs," *IEEE Trans. Ind. Informat.*, vol. 9, no. 1, pp. 527–545, Feb. 2013.

[45] J. Boutellier, J. Ersfolk, J. Lilius, M. Mattavelli, G. Roquier, and O. Silvén, "Actor merging for dataflow process networks," *IEEE Trans. Signal Process.*, vol. 63, no. 10, pp. 2496–2508, May 2015.

[46] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[48] S. Woo, J. Park, J.-Y. Lee, and I. S. Kweon, "CBAM: Convolutional block attention module," in *Proc. Eur. Conf. Comput. Vis. (ECCV),* 2018, pp. 3–19.

[49] Y. Xu et al., "Tracking the occluded indoor target with scattered millimeter wave signal," *IEEE Sensors J.*, vol. 24, no. 22, pp. 38102–38112, Nov. 2024.

[50] H. Cai, J. Li, M. Hu, C. Gan, and S. Han, "Efficientvit: Lightweight multi-scale attention for high-resolution dense prediction," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2023, pp. 17302–17313.

[51] A. G. Howard et al., "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.

[52] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2015, pp. 1–14.

[53] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR),* 2017, pp. 7263–7271.

[54] H. Huttunen, F. S. Yancheshmeh, and K. Chen, "Car type recognition with deep neural networks," in *Proc. IEEE Intell. Vehicles Symp. (IV)*, 2016, pp. 1115–1120.