GOURD: Tensorizing Streaming Applications to Generate Multi-Instance Compute Platforms

Patrick Schmid, Paul Palomero Bernardo, Christoph Gerum, and Oliver Bringmann, *Member, IEEE*

***Abstract*—In this article, we rethink the dataflow processing paradigm to a higher level of abstraction to automate the generation of multi-instance compute and memory platforms with interfaces to I/O devices (sensors and actuators). Since the different compute instances (NPUs, CPUs, DSPs, etc.) and I/O devices do not necessarily have compatible interfaces on a dataflow level, an automated translation is required. However, in multidimensional dataflow scenarios, it becomes inherently difficult to reason about buffer sizes and iteration order without knowing the shape of the data access pattern (DAP) that the dataflow follows. To capture this shape and the platform composition, we define a domain-specific representation (DSR) and devise a toolchain to generate a synthesizable platform, including appropriate streaming buffers for platform-specific tensorization of the data between incompatible interfaces. This allows platforms, such as sensor edge AI devices, to be easily specified by simply focusing on the shape of the data provided by the sensors and transmitted among compute units, giving the ability to evaluate and generate different dataflow design alternatives with significantly reduced design time.**

***Index Terms*—Data orchestration, hardware platform generation, multidimensional data flow, synchronization.**

I. Introduction

WITH the advent of domain-specific architectures in the edge and cloud computing, there exists an increasing demand to rethink the dataflow processing paradigm to a higher level of abstraction by providing an expressive dataflow formalization and transformation approach for automated generation of multi-instance compute and memory platforms with interfaces to I/O devices (sensors and actuators).

The different compute instances (NPUs, CPUs, DSPs, etc.) and I/O devices do not necessarily have compatible interfaces. The automatic translation between the incompatible interfaces on the protocol level as well as the implementation of the 1-D data-flow on the top of the interfaces is well understood and can be realized by generating of protocol adaptors and the use of FIFO buffers.

In the multidimensional case, however, it is inherently difficult to reason about buffer sizes and iteration order without knowing the shape of the *DAP*, i.e., its tensorization of data, that the data-flow follows. To implement compatible communication between the aforementioned platform components or between the compute and memory instances, either the functional workload at the compute instances or the interconnect structure needs to be adapted.

While a lot of research is being put into efficient data orchestration from and to a compute instance (Buffets [1], Stash [2], Patch Memory [3], Stream-Dataflow Acceleration [4], Accelerator Store [5]), and the generation or implementation of efficient datapaths for compute instances (Rubick [6], Spatial Tensor Accelerator [7], TENET [8], UltraTrail [9], ShiDianNao [10], and Versa-DNN [11]), we have not yet observed a lot of work put into the interconnects of multiple compute instances with complete or partial incompatible DAPs.

To our best knowledge, existing multi-instance compute systems can generally be put into two categories: 1) memory-based system designs and 2) peer-to-peer system designs. Systems of the first category communicate with a memory and a controller to orchestrate the data movement (e.g., [1], [2], [4], and [5]). Systems of the second category usually communicate in a streaming-like manner, where platform components provide the necessary data to their interconnected components (e.g., [12] and [13]). In both the categories, the issue with incompatible DAPs exists.

Furthermore, due to the growing demand for multi-instance compute systems in embedded systems, a low-overhead and scalable solution that allows for rapid and easy generation of multicompute instance systems is necessary.

To this end, we introduce *GOURD* which defines a DSR of connecting components and their DAP. *GOURD* also provides an *automatic* translation tool that generates the translation units called *Streaming Buffers* when incompatible DAPs need to be connected. A high-level representation of the issue and our solution to this is shown in Fig. 1.

Our main contributions are as follows.

1) Designing a DSR of compute instances and their DAPs.
2) Derivation of appropriate *Streaming Buffers* between the components with incompatible DAPs.
3) Implementing a tool to automatically translate the DSR into synthesizable RTL.

II. Related Work

Several approaches have been presented to design and implement highly efficient datapaths for the components or data orchestration between them and a memory.

Manuscript received 7 August 2024; accepted 10 August 2024. This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the Projects Scale4Edge under Reference Number 16ME0129, and in part by the MANNHEIM-FlexKI under Reference Number 01IS22086H. This article was presented at the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Patrick Schmid.)*

The authors are with the Department of Computer Science, University of Tübingen, 72074 Tübingen, Germany (e-mail: patrick.schmid@uni-tuebingen.de; paul.palomero-bernardo@uni-tuebingen.de; christoph.gerum@uni-tuebingen.de; oliver.bringmann@uni-tuebingen.de).

Digital Object Identifier 10.1109/TCAD.2024.3445810

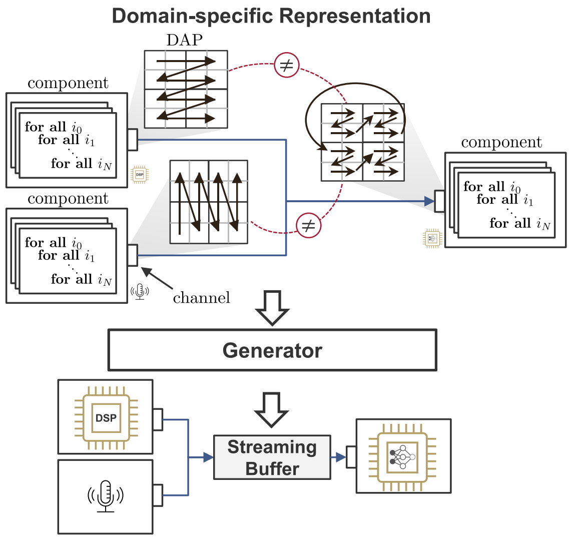


Fig. 1. Overview of *GOURD*, showcasing our proposed DSR that can describe a platform and the DAPs of its components. Our representation allows to identify incompatible connections due to the components' DAPs. *GOURD's* proposed hardware generator converts the DSR to *Streaming Buffer* translation unit, enabling connection components with incompatible DAPs.

Buffets [1] are hierarchically composable hardware components that implement an explicit decoupled data orchestration between the arbitrary compute instances and a global memory system. This approach provides an efficient data orchestration. Compared to our work this work focuses only on efficient communication with higher level memory hierarchies.

Stash [2] are hardware components that unify the benefits from the local scratchpads and caches. They can globalize the local scratchpad memory space by an user-defined translation function. Stash focused on a highly efficient data orchestration functionality for single compute instances, but neglected the influence of the underlying data shape on the design.

Patch Memory [3] provides a scratchpad-like memory for image processing. They consider a domain-specific shape of the data, but do not apply their approach to out-of-domain DAP and only consider single compute instances.

Unified Buffers [14] implement a HalideIR-based compilation scheme onto a custom compute and generated memory instances that use novel abstraction for push memories. They show the potential of their approach by scheduling machine learning and image processing applications with Halide and mapping them to a custom CGRA with push memory abstraction. Our work focuses on reusing highly efficient existing components and enabling a connection between them while also providing flexibility on the memory architecture.

Stream Dataflow [4] present a programming paradigm on data streaming workloads for CGRA architectures by introducing configurable control components that manage the dataflow from memory to CGRA. They highlight the usefulness of the data streaming approach on CGRA compute instances. Compared to design, we focus on multi-instance compute platforms and adapting the dataflow in between them.

Accelerator Store [5] is a shared memory space between compute instances. For this, they developed a system that allows individual compute instances to request memory in a shared memory space which is freed when the instance does not need it anymore. With their contribution, they focus on the memory scale-ability issue of multicompute instances. In our work, we focus on the interconnectivity between multicompute instances.

Dutta et al. [12] presented a translation scheme for window synchronous data flows [15] that uses a multidimensional FIFO system to interconnect compute instances. In comparison to that, we do not require a fixed memory architecture and also allow DAP on the consuming interface to *define loop reordering* in the form of *data access pattern reordering (DAPR)*. In addition, our generation DSR provides enough domain information to automatically generate hardware without transforming it into a data flow graph description for further analysis.

Although existing designs and systems excel within their respective domains, a gap exists in seamlessly interconnecting components and integrating comprehensive system information. The design and implementation of such systems necessitate expert knowledge and substantial engineering effort. GOURD addresses this challenge by *automatically* generating optimized hardware based on the design intent, while still maintaining the flexibility to incorporate the existing expertise, thereby drastically facilitating the development process.

III. Motivation and General Approach

Data streaming applications are an established part of nowadays embedded platforms. These applications range from classical examples, such as audio and video processing to the state-of-the-art topics of efficient implementation of neural network architectures [16], [17]. Platforms usually consists of multiple components with carefully engineered communication-channels and dataflows. Changing components require manual adaptation of the platform. We observe that DAP in data streaming applications are often described as a multidimensional iteration spaces. For example, an image sensor iterates over three dimensions when transmitting an RGB image: 1) the width; 2) the height; and 3) the channels. Another example is a neural network accelerator that reads several input features in an application-defined order [9], [10], [18], [19]. These iteration spaces are often defined as the nested `for`-loops. The iteration space can be interpreted as a tensor, where each dimension is defined by the length of the `for`-loops.

Data-producing components often transmit data *linearly* within their iteration space. For example, the previously described image sensor is a data-producing component that iterates its 3-D iteration space in a predefined application-specific order. It could first iterate over the *channel* dimension before moving to the next pixel in the *width* dimension and finally in the *height* dimension. On the other hand, data-consuming components can read data multiple times or skip over some data. For example, in convolutional neural networks, data-consuming components often read the data following the iteration order of a moving kernel.

Interconnected components have to be designed such that they can 1) understand the incoming data or 2) build a translation unit that makes the dataflow of incompatible interfaces compatible. Designing a set of compatible components often requires a lot of engineering effort. To tackle this issue, we introduce a DSR that captures the tensor-shaped data and their iteration order. With this representation, we can easily test if interfaces define incompatible dataflows and automatically generate configurable *Streaming Buffers* that perform the data translation. An overview of the solution (*GOURD*) is shown in Fig. 1. The generator is implemented within the MLIR [20]/CIRCT [21] environment.

Terminology As shown in Fig. 1, we consider each component as a black box that communicates with other components through channels. A *channel* can be either a *port* or a *streaming interface*. A *port* is a simple end-to-end connection. A *streaming interface* can be extended with the tensor shape of the data that it transmits. In the scope of this article, a DAP defines the shape of the iteration domain and the iteration order of it as shown in Fig. 1. A DAP is described by a set of nested `for`-loops. The shape of the loops are summarized as *window access pattern (WAP)*. A formal definition will be provided in Section III-A. Streaming interface channels can define multiple DAPs which we consider exclusively active alternatives that can be selected during the system's runtime, by reconfiguring the respective component. Channels are connected via *connections*. A connection logical associates a number of channels on the left side (lhs) that produce the data to a number of channels on the right side (rhs) that consume the data. Connections define a many-to-many relationship between channels, but only one producing channel can send the data at a time. We require connecting components whose interfaces define DAPs to define at least one connectable pattern. DAPs are *connectable* when the DAP of a consuming interface only accesses *iteration domain instance vectors (IVs)* which are in the iteration space spanned by the DAP of the producing interface. In our framework, we require that a component with a streaming interface can be stalled by the subsequent component when the receiving component is not ready. A data-consuming component can be stalled until the data from a data producing component is valid. Finally, we introduce a semantic that allows change of the iteration order of a DAP from a producing to a consuming interface.

The generator of *GOURD* is split into multiple steps that are outlined in Fig. 2. As depicted in Fig. 1, the generator takes the DSR as input and generates *Streaming Buffer* hardware components. The generated *Streaming Buffer* ensures that a consuming interface receives the required data in the correct order. The steps outlined in Fig. 2 successively refine the generated Streaming Buffer into a syntheziable RTL description. These steps will be further explained throughout the remainder of this chapter. The labels in the figure correspond with the headlines in Section III.

A. GOURD Abstraction Level and Definitions

This section introduces a formal definition of the data structures required by our dialect and elaborates on the terminologies we introduced in Section III.

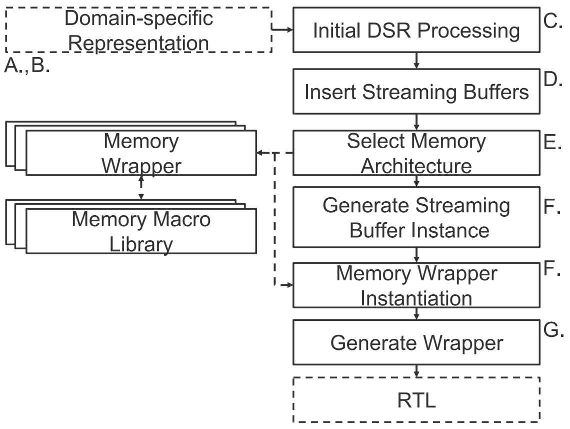

Domain-specific Representation
A.,B.
Initial DSR Processing
C.
Insert Streaming Buffers
D.
Memory Wrapper
Select Memory Architecture
E.
Memory Macro Library
Generate Streaming Buffer Instance
F.
Memory Wrapper Instantiation
F.
Generate Wrapper
G.
RTL

Fig. 2. Steps that *GOURD* performs to automatically generate hardware. The letters correspond to the sections in Section III

Algorithm 1 Loop Nest Example

```
   // Adjust UBs so that WAPs do not walk out of bounds
1: for all $i_0 \in \{0, 4 - W_1[UB][0] + 1\}$ in stride of 1 do
2:     for all $i_1 \in \{0, 4 - W_1[UB][1] + 1\}$ in stride of 1 do
3:         for all $i_2 \in \{0, 2\}$ in stride of 1 do
4:             for all $i_3 \in \{0, 2\}$ in stride of 1 do
5:                 DATA$[i_0 + i_2][i_1 + i_3]$
6:             end for
7:         end for
8:     end for
9: end for
```

A WAP is defined through an M-dimensional loop nest with the $3M$ parameters. Each `for`-loop consists of a set of three parameters: 1) *lower bound* (*LB*); 2) *upper bound* (*UB*); and 3) *stride* (*ST*), which are termed *configuration*. The loop counts from LB to $UB - 1$ in steps of ST. A WAP W is defined as

$$W = \begin{pmatrix} | & | & | \\ LB & UB & ST \\ | & | & | \end{pmatrix} \subseteq \mathbb{N}^{M\times 3} \tag{1}$$

where the 0th row represents the configuration of the outermost loop and the $M-1$th row is the configuration of the innermost loop. The dimension $\dim W = M$ describes the number of loops in a WAP. A column of a WAP can be accessed with $W[LB|UB|ST][i]$ for $0 \le i < M$. A valid WAP requires that $W[LB][i] < W[UB][i]$ and $0 < W[ST][i] \quad \forall i$.

Multiple WAPs can be combined into a DAP DP that visits a region in the iteration space multiple times

$$DP = \begin{pmatrix} W_0 & \cdots & W_{N-1} \end{pmatrix} \subseteq \mathbb{N}^{(M\times 3)\times N} \tag{2}$$

where $W_{0\le j<N}$ are WAPs that all have the same dimensions. We require that every WAP W_i does not leave the geometric area defined by it predecessor W_j, with $j < i$. To access the jth WAP we use the notation $DP[j]$. The dimension $\dim DP = N$ of a DAP equals the number of referenced WAPs within it.

For example, a $DP_{\mathtt{base1}}$ that describes a 2×2 convolution within a 4×4 windows is represented as

$$DP_{\mathtt{base1}} = \left(\underbrace{\begin{pmatrix} 0 & 4 & 1 \\ 0 & 4 & 1 \end{pmatrix}}_{W_0} \quad \underbrace{\begin{pmatrix} 0 & 2 & 1 \\ 0 & 2 & 1 \end{pmatrix}}_{W_1} \right). \tag{3}$$

The resulting loop nest is shown in Algorithm 1.

Given a set of DAPs $\mathcal{D}$ we define the *union DAP* to be

$$DP_{\mathcal{D}} = \bigcup_{DP\in\mathcal{D}} DP. \tag{4}$$

For example, the union of the DAP

$$DP_{\texttt{base2}} = \left(\begin{pmatrix} 0 & 4 & 1 \\ 0 & 4 & 1 \end{pmatrix}\right) \tag{5}$$

and $DP_{\texttt{base1}}$ (3), $\mathcal{D} := \{DP_{\texttt{base1}}, DP_{\texttt{base2}}\}$ can be written as

$$\begin{aligned} DP_{\mathcal{D}} &= DP_{\texttt{base1}} \cup DP_{\texttt{base2}} \\ &= \left(\begin{pmatrix} 0 & 4 & 1 \\ 0 & 4 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 2 & 1 \\ 0 & 2 & 1 \end{pmatrix}\right). \end{aligned} \tag{6}$$

An IV can be defined w.r.t. WAP or DAP. Given a WAP W with dimension M and a set of loop indices $i_0, \ldots, i_{M-1}$ for each loop of W, an IV of WAP $\mathcal{I}_W$ is defined as

$$\mathcal{I}_W = (i_0, \ldots, i_{M-1})^\intercal. \tag{7}$$

For example, the IVs of the example in Algorithm 1 can be written as

$$\mathcal{I}_{W_0} = (i_0, i_1)^\intercal, \qquad \mathcal{I}_{W_1} = (i_2, i_3)^\intercal. \tag{8}$$

The IV $\mathcal{I}_{DP}$ of a DAP with dimension N is obtained by applying the function m

$$\begin{aligned} m : \mathbb{N}^{M\times N} &\to \mathbb{N}^M \\ (\mathcal{I}_{W_0}, \ldots, \mathcal{I}_{W_{N-1}}) &\mapsto \begin{pmatrix} \sum_{j=0}^{N-1} \mathcal{I}_{W_j}[0] \\ \vdots \\ \sum_{j=0}^{N-1} \mathcal{I}_{W_j}[M-1] \end{pmatrix} \end{aligned} \tag{9}$$

to the tuple $(\mathcal{I}_{W_0}, \ldots, \mathcal{I}_{W_{N-1}})$, where $W_j \in DP$ with $\dim W_j = M \quad \forall j$. For example, the IV in Algorithm 1 is

$$\mathcal{I}_{D_{\texttt{base1}}} = m\big((\mathcal{I}_{W_0}, \quad \mathcal{I}_{W_1})\big) = \begin{pmatrix} i_0 + i_2 \\ i_1 + i_3 \end{pmatrix}. \tag{10}$$

The set of all IVs spans the iteration domain. A value of an IV can be accessed with $\mathcal{I}[k]$, $0 \leq k < \dim \mathcal{I}$. The lexicographical order of two IVs $\mathcal{I}_{lhs}$ and $\mathcal{I}_{rhs}$ with dimension $\mathbf{d} = \dim \mathcal{I}_{lhs} = \dim \mathcal{I}_{rhs}$ is

$$\begin{aligned} \mathcal{I}_{lhs} < \mathcal{I}_{rhs} :=\ & (\mathcal{I}_{lhs}[0] < \mathcal{I}_{rhs}[0]) \vee \\ & \bigvee_{i=1}^{\mathbf{d}-1} \left[\mathcal{I}_{lhs}[i] < \mathcal{I}_{rhs}[i] \wedge \bigwedge_{j=0}^{i} \mathcal{I}_{lhs}[i] = \mathcal{I}_{rhs}[i] \right]. \end{aligned} \tag{11}$$

Comparing IVs is only possible when the indices of the values in both IVs are in the same order. A different order can be present when a DAP defines a DAP reordering (DAPR). A DAPR is as an index-permutation function $\pi : \mathrm{Ind}(\mathcal{I}_{DP}) \to \mathrm{Ind}(\mathcal{I}_{DP})$ of IV $\mathcal{I}_{DP}$ with $\mathrm{Ind}(\mathcal{I}_{DP}) := \{0, \ldots, \dim \mathcal{I}_{DP} - 1\}$, which is defined w.r.t. an incoming data stream. This means that DAPRs are only allowed to be defined by a data-consuming component's interface. For example, assuming that $DP_{\texttt{base2}}$ (5) is defined by a consuming component's interface and defines a reordering that swaps the order in its IV in relation to the IV derived from the DAP of a producing component's interface. In this case, the DAPR is translated into a function

$$\begin{aligned} \pi : \mathrm{Ind}(\mathcal{I}_{DP_{\texttt{base2}}}) &\to \mathrm{Ind}(\mathcal{I}_{DP_{\texttt{base2}}}) \\ i &\mapsto \begin{cases} 0, & i = 1 \\ 1, & i = 0. \end{cases} \end{aligned} \tag{12}$$

Intuitively, this DAPR swaps the index positions in $\mathcal{I}_{D_{\texttt{base2}}}$:

$$\mathcal{I}_{D_{\texttt{base2}}} = \begin{pmatrix} i_0 \\ i_1 \end{pmatrix} \rightleftarrows \begin{pmatrix} i_1 \\ i_0 \end{pmatrix}. \tag{13}$$

To this end, we formalize the definition of two connectable DAPs, DP_{lhs} and DP_{rhs} as follows.

1) If DP_{rhs} does not define a DAPR, every $\mathcal{I}_{DP_{rhs}}$ must be in the iteration domain spanned by $\mathcal{I}_{DP_{lhs}}$.
2) If DP_{rhs} defines a DAPR, the reversed permutation of $\mathcal{I}_{DP_{rhs}}$ must be in the iteration domain spanned by $\mathcal{I}_{DP_{lhs}}$.

B. GOURD Domain-Specific Representation

We implement *GOURD's* DSR as a custom dialect within the MLIR [20]/CIRCT [21] environment. We opted for this environment as it already provides a comprehensive set of functions for dialect manipulation and offers the possibility for SystemVerilog (RTL) hardware generation. Furthermore, expressing the DSR as MLIR/CIRCT dialect forms a simple-to-write and easy-to-understand layer top of the formalism introduced in Section III-A that is exposed to the users of *GOURD*. An example of the dialect is shown in Listing 1.

As can be seen, components and their channels are described with simple intermediate representation (IR) constructs. Streaming interfaces can implicitly describe their DAP by defining it in terms of nested WAP. DAPRs are implemented with the existing MLIR/CIRCT language constructs. When describing a system with the IR, the interfaces of two connected components have to define at least one connectable DAP. During hardware generation, we only consider all the pairs of every connectable DAP. Also, as mentioned at the beginning of Section III, we restrict the dimensionality of a DAP of data-producing component's interface to one, as they usually iterate their DAP in a linear order.

The data structures of our generator rely on the abstraction formalism introduced in Section III-A. For example, the DAP of the component `comp1` from interface `in` in the example (Listing 1) is formally written as shown in (3). The DAPR defined by the DAP `base2` [formally shown in (5)] from interface `in` of `comp1`, in the example (Listing 1), is formally written as shown in (12).

We will use the example defined in Listing 1 as example to showcase the behavior of the generator during the hardware generation process.

C. Initial DSR Processing

The initial processing of the DSR splits connections between interfaces whose DAPs are not equal.

Then, we annotate connections whether their DAPs are equal or not. This step is illustrated in Fig. 3(a).

D. Insert Streaming Buffers

A *Streaming Buffer* is a component inserted between the previously marked *unequal* connections of streaming

```
// Component declaration (IR and HDL identifier)
gourd.component @comp0 "comp_0" {
  // Ports
  gourd.port @clk in 1 clock {}
  gourd.port @rst in 1 reset {}
  // Streaming Interfaces
  gourd.interface @in 32 in {}
  gourd.interface @out 32 {
    // Data access pattern
    // (single window access pattern)
    gourd.window @base [[0, 4, 1], [0, 4, 1]]
  }
}
// Component declaration (IR and HDL identifier)
gourd.component @comp1 "comp_1" {
  // Ports
  gourd.port @clk in 1 clock {}
  gourd.port @rst in 1 reset {}
  // Streaming Interfaces
  gourd.interface @in 32 {
    // Data access pattern, option 1
    gourd.window @base0 [[0, 4, 2], [0, 4, 2]]
    // Data access pattern, option 2
    gourd.window @base1 [[0, 4, 1], [0, 4, 1]] @conv
    gourd.window @conv [[0, 2, 1], [0, 2, 1]]
    // Data access pattern, option 3 that
    // defines a Data Access Pattern Reordering
    gourd.window @base2 [[0, 4, 1], [0, 4, 1]] affine_map
          <(i, j) -> (j, i)>
  }
  gourd.interface @out 32 out {}
}
// Connections between components
gourd.connection [@comp0::@out] [@comp1::@in]
```

Listing 1. Example of our custom MLIR [20]/CIRCT [21] dialect *GOURD* that implements our DSR. The example shows two components with multiple ports and interfaces. The interfaces define the shape of their data as DAPs which are represented as nested WAPs.

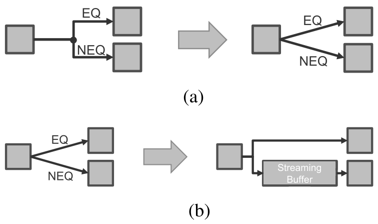


Fig. 3. Initial DSR processing and Streaming Buffer insertion. (a) Connection splitting when DAPs defined by a component's interfaces (shown as gray boxes) are not equal. (b) Streaming Buffers are inserted in between those connections between interfaces with unequal DAPs.

interfaces, as shown in Fig. 3(b). It handles the data transfer according to the defined DAPs, for which it may require access to a memory space of size `#mem`. Currently, *GOURD* does not provide data preloading. This means that $\mathcal{I}_{lhs}$ of the producing side interface must be smaller or equal than $\mathcal{I}_{rhs}$ of the consuming side interfaces. When this condition is not fulfilled, the Streaming Buffer tells the producing interface to stop transmitting via the corresponding ready signal. This condition will be formalized in Section III-F4. The Streaming Buffer has to store data which was transmitted by the producing interface but is still required by the consuming interfaces in later steps. For example, consider the second case of the data transmission example in Fig. 4. DP_{lhs} transmits the data in row-major manner while DP_{rhs} reads the data following the order of a 2-D convolution. The second data element that is transmitted is required in the first iteration of $W_{c,1}$ as well as

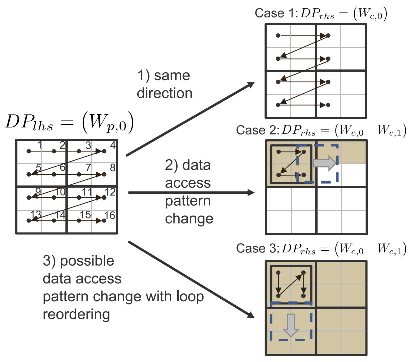

Case 1: $DP_{rhs} = (W_{c,0})$
1) same direction
$DP_{lhs} = (W_{p,0})$
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Case 2: $DP_{rhs} = (W_{c,0} \quad W_{c,1})$
2) data access pattern change
Case 3: $DP_{rhs} = (W_{c,0} \quad W_{c,1})$
3) possible data access pattern change with loop reordering

Fig. 4. Different combinations of DAPs that can occur when connecting interfaces. These DAPs also visually represent the three DAP-alternatives of component's `comp1` interface `in` shown in listing 1.

in the second iteration of it. The data element must therefore be kept in memory. In addition to that, the subsequent data elements (3–6) must also be kept in memory as they are also required in later steps. The exact amount of data that requires buffering can be computed by virtually iterating the DAPs of the connected components while keeping the track of how many elements of DP_{lhs} need to be stored before they are not required any longer in later steps of DP_{rhs}. The runtime of virtually iterating does not scale well with increasing dimensionality of the WAP and the DAP. Consider a system constituted of a producing interface with a 1D-DAP and a consuming interface with a 3D-DAP. Each WAP consists of five dimensions, whose maximum UB value is 2^8. Determining the exact memory requirements of such a system can be extremely computationally intensive. For example, our Python implementation took several hours to complete this process. Therefore, instead of trying to estimate the exact memory requirement, we propose to compute an UB of the memory space size with the DAP of the producing interface ($DP_{lhs} = (W_{p,0})$) and the DAP of the consuming interface ($DP_{rhs} = (W_{c,0} \; W_{c,1} \; \cdots)$). The UBs are derived from the geometrical shape of the DAPs, their dimension and whether a DAPRs is present or not. We divide the calculation of the size `#mem` into three cases, which represent the different combinations of DAPs that can occur when connecting interfaces. An illustration of all the three cases is presented in Fig. 4. Since the movable area of a WAP is confined to the geometrical area of the WAP at the dimension above, it is sufficient to estimate the required memory space with the WAP ($W_{c,1}$) on the first dimension of DP_{rhs}.

1) Same Direction: In the first case shown in Fig. 4, the DAPs have the same dimension, and DP_{rhs} only has a different lower/upper bound or different stride values. This means the Streaming Buffer does not require access to the memory space as it is sufficient to forward the data from a producing interface according to the definition of the DAP on the consuming interface. Therefore, the required memory space `#mem` is 0.

2) No DAPR But Change in Data Access Pattern: In the second case shown in Fig. 4, the DAP of the consuming interface must have a dimension greater than 1 and no DAPR is defined. To compute the UB we make use of the geometric shape of DP_{rhs} and the iteration order of it. Because *GOURD* currently does not include loops bounds that are dependent on the state of other loops in the loop nest, we can statically infer that some of the transmitted data is not used anymore in subsequent iterations, whenever $W_{c,1}$ is moved. This can be nicely seen in the second case of Fig. 4. The first data element of DP_{lhs} is accessed once in the first iteration of DP_{rhs}. In subsequent steps of DP_{rhs}, this field is not accessed again. Since DP_{rhs} does not define a DAPR in this case, the window of $W_{c,1}$ cannot jump backward but continuously follows the direction layouted by DP_{lhs}. For the computation of the UB, we consider the case where the minimum amount of data is freed in each move of $W_{c,1}$ and the producing interface cannot send more data ($\mathcal{I}_{lhs} > \mathcal{I}_{rhs}$). The required memory space is estimated as

$$\texttt{\#mem} = W_{c,1}[UB][0] \tag{14}$$

when the IV have only a single dimension, and

$$\begin{aligned}\texttt{\#mem} = &\underbrace{(W_{c,1}[UB][0]-1)\cdot\prod_{i=1}^{\dim\mathcal{I}_{DP_{lhs}}-1} W_{p,0}[UB][i]}_{\text{count the transmitted until the last iteration of } W_{c,1}}\\ + &\underbrace{W_{c,1}[UB][1]\cdot\prod_{i=2}^{\dim\mathcal{I}_{DP_{lhs}}-1} W_{p,0}[UB][i]}_{\text{count the remaining data in the last iteration of } W_{c,1}}\end{aligned} \tag{15}$$

in any other cases. For example, the required memory space of the consuming interface `in` defined in Listing 1 configured with the DAP `base1`, which is shown in (3) and the DAP

$$DP_{\texttt{base}} = \left(\begin{pmatrix} 0 & 4 & 1\\ 0 & 4 & 1\end{pmatrix}\right) \tag{16}$$

of the producing interface `out` is

$$\begin{aligned}\texttt{\#mem} &= 1\cdot\prod_{i=1}^{2-1} DP_{\text{base}}[0][UB][i] + 2\cdot\prod_{i=2}^{2-1} DP_{\text{base}}[0][UB][i]\\ &= 1\cdot 4 + 2\cdot 1 = 6.\end{aligned} \tag{17}$$

This is also represented by the golden colored fields in the second case of Fig. 4.

3) DAPR and Possible Change in Data Access Pattern: In this case, the consuming interface defines a DAPR and may have a different DAP as the producing interface. The data transmitted by the producing interface must be stored for the dimensions with misaligned IV indices. For example, the left interface produces an RGB image following a DAP that iterates in the order *p-width*, *p-height*, and *p-channel*. The consuming interface processes the input using a 3D-kernel where the kernel, i.e., W_{c_1} iterates in the order *w-channel*, *w-height*, and *w-width*. We have misaligned indices in *width* and *channel*. Here, one has to store all the transmitted data from *p-width* and *p-channel*. Informally, the required memory space can be written as *p-width* · *p-channel* · *p-height*. An illustration of this scenario is shown in Fig. 5 case 1.

Given a DAPR function π, we define

$$\mathbf{k} = \min_{0\leq j<\dim\mathcal{I}_{DP_{rhs}}} j \neq \pi(j) \tag{18}$$

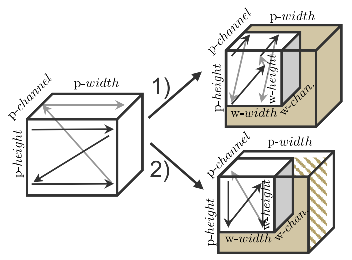


Fig. 5. Example memory size of a 3-D DAPs with different DAPRs.

as the lowest dimension index where the indices start to misalign. The memory size can then be computed as

$$\texttt{\#mem}_0 = \underbrace{\prod_{i=\mathbf{k}}^{\dim\mathcal{I}_{DP_{lhs}}-1} W_{p,0}[UB][i]}_{\text{compute the size of the initial estimate}} \tag{19}$$

by multiplying the outer-most UBs of the producing DAP starting from $\mathbf{k}$. In the example shown in Fig. 5 case 1, the misaligned index is the outer-most index, i.e., $k = 0$; hence the entire incoming data need to be stored. The required memory space is illustrated with golden colors. However, this is not sufficient to fully compute the required memory space. Consider the previous example with slightly different iteration order, where the producer iterates in the same order, but the consumer iterates in the order *w-height*, *w-width*, and *w-channel*, as illustrated in Fig. 5 case 2. There, the outer-most dimensions align, i.e., $k = 1$. Thus, the required memory size would be *p-height* · *p-width*. This is sufficient as long as *w-channel* $= 1$ but fails when the kernel window moves into the *w-channel* dimension. Therefore, the previously computed required memory size in (19) has to be scaled accordingly. We define

$$\mathbf{s} = \begin{cases}\min\limits_{0\leq j<\mathbf{k}} W_{c,1}[UB][j] > 1 & \mathbf{k} > 0\\ 0\end{cases} \tag{20}$$

as the lowest dimension where $W_{c,1}$ requires to moves into and

$$\texttt{\#fac} = \begin{cases} W_{c,1}[UB][\mathbf{s}]\cdot\prod_{i=\mathbf{s}+1}^{\mathbf{k}-1} W_{p,0}[UB][i] & \mathbf{s} > 0\\ 1\end{cases} \tag{21}$$

as the scaling factor. The factor multiplies the UB values of each dimension where $W_{c,1}$ moves into. We use $W_{c,1}[UB][\mathbf{s}]$ in the lowest dimension as it results in a lower estimated value for the memory space. The final required memory space is estimated with

$$\texttt{\#mem} = \texttt{\#mem}_0\cdot\texttt{\#fac}. \tag{22}$$

Considering the previously described case with *w-channel* > 1. In this case `#fac` would assume the value *w-channel*. It is not necessary to scale $\texttt{\#mem}_0$ with *p-channel* because the outer dimensions are not reordered. Visually this is represented by $W_{c,1}$ being confined to the golden colored and golden striped area shown in Fig. 5 until it has to move to the next field in the channel dimension.

For example, the required memory space of the consuming interface `in` (Listing 1) configured with $DP_{\texttt{base2}}$ (5) with the

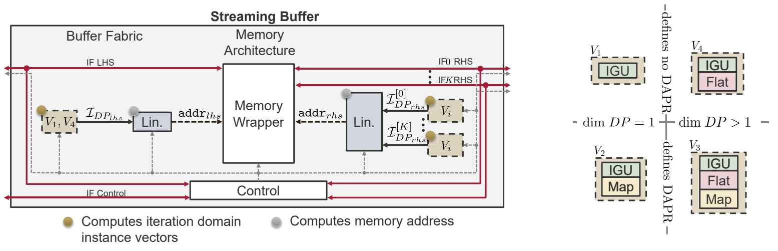


Fig. 6. Streaming Buffer macro-architecture and different templates that define different ways to compute the IVs of the consuming interface and producing interfaces. The hardware generation process of GOURD *automatically* decides which template to implement.

DAPR (12), coupled with the DAP of the producing interface $DP_{\mathtt{base}}$ (16) can be written as

$$\#\mathtt{mem} = \prod_{i=0}^{2-1} DP_{\mathtt{base}}[0][UB][i] \cdot 1 = 4 \cdot 4 = 16. \qquad (23)$$

Visually, this is represented by the golden colored fields in the third case of Fig. 4. The total required memory space for a Streaming Buffer is calculated by iterating over all the combinations of lhs and rhs DAPs connected to the buffer, computing the required memory space and taking the maximum value. This value is annotated to the Streaming Buffer, along with the maximum bitwidth of all the connecting interfaces.

E. Select Memory Architecture

The fabric of a Streaming Buffer as shown in Fig. 6, contains a memory wrapper that determines the memory architecture of the buffer. The implementation of a memory architecture uses the generated control signals, which will be described in Section III-F. Optionally, it can use a set of memory macros that are loaded from a memory macro library. A memory wrapper can have parameters that affect its memory architecture implementation. For instance, if a buffer needs access to a memory space, the memory wrapper could define that its memory architecture implements the memory space as a set of split memory macros, wherein the split factor is parameterizable. Due to the memory architecture of a memory wrapper, it might be necessary to increase the memory space. This can be due to the lack of fitting the memory macros or implementation details. This concept of memory wrappers also allows to adapt the existing highly efficient data orchestration systems, such as Buffets [1], Stashes [2], Accelerator Store [5], etc. The memory architecture of a memory wrapper connects the signals of the interfaces connected to the Streaming Buffer accordingly. During this step, we select the memory wrapper that fits best for this Streaming Buffer based on a locally optimized scheme. The scheme selects a memory wrapper based on an area and energy estimate that is computed over all the memory wrappers and their memory architecture. The memory architecture is selected at this stage to provide information about the actual size of the total addressable memory space of the buffer, which is required to generate signals with the correct bitwidth in subsequent steps.

F. Generate Streaming Buffer Instance

This section describes the proposed macro-architecture of a *Streaming Buffer* shown in Fig. 6. A buffer is responsible to receive and transmit the data according to the DAPs defined by its connecting interfaces. A consuming interface $\mathtt{IF}i$ $\mathtt{RHS}$ with DAP DP_{rhs} is able to receive the data from the producing interface $\mathtt{IF\ LHS}$ with DAP DP_{lhs} when both DAPs are connectable and $\mathcal{I}^{[i]}_{DP_{rhs}} \leq \mathcal{I}_{DP_{lhs}}$. The first condition must be fulfilled by definition. The second condition ensures that the data required in the current instance of $\mathcal{I}^{[i]}_{DP_{rhs}}$ is already transmitted by the producing interface. The producing interface can transmit data as long as the precedence relation $\mathcal{I}_{DP_{lhs}} \leq \mathcal{I}^{[i]}_{DP_{rhs}}$ is not violated. A Streaming Buffer has a single lhs and multiple rhs. The lhs is connected to *one* of the producing interfaces at a time. The connected interface can be changed during the runtime of the system. The rhs has as many interfaces as consuming interfaces are connected to the Streaming Buffer. The currently required data element is described by the current state of the IVs. These vectors are also used to generate the control signals which will be explained in Section III-F4. The computation scheme of the vectors $\mathcal{I}_{DP_{lhs}}, \mathcal{I}^{[i]}_{DP_{rhs}}$ changes depending on the defined DAP. For example, the computation of an IV for a DAP with dimension 1 requires no summation (see (9) with $\mathrm{N} = 1$); hence no hardware unit that computes the summation is required. Another example is a DAP that defines a DAPR. In this case, a hardware unit is required that reverses the DAPR in the IV. The necessary hardware units are selected based on previously described condition. The conditions are also shown on the rhs in Fig. 6. A producing interface cannot, per definition, define a DAPR. This reduces the computation schemes to V_1, V_4 for these interfaces. During hardware generation, *GOURD* automatically selects the minimum necessary hardware units to compute a IV. The following enumeration provides a brief introduction of the components in a computation scheme.

1) *Index Generation Unit (IGU):* Iterates the iteration domain for each loop dimension according to the *LB*, *UB*, and *ST* values of a DAP, similar to the example in Algorithm 1.
2) *Flattening Unit:* Computes the IV of a DAP according to (9).
3) *Mapping Unit:* Reorders the values in a IV of a consuming interface to reverse the DAPR, so that the order of the indices matches the order of IV of the producing interface.

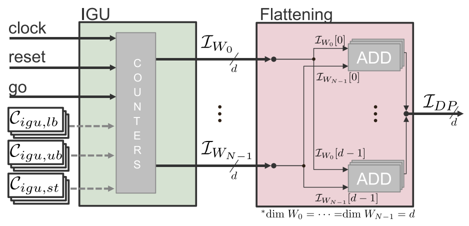


Fig. 7. Macro-architecture of the IGU with flattening unit. This also represents the architecture of template V_2 shown in Fig. 6.

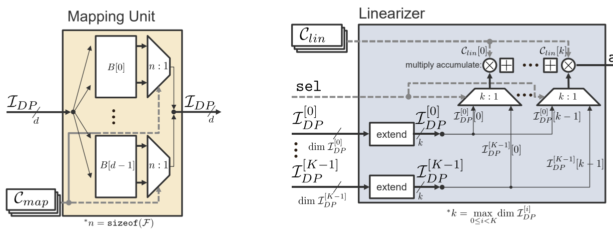


Fig. 8. Macro-architectures of the mapping unit and the linearizer. (a) Mapping unit that reorders an IV according to a DAPR. (b) Linearizer that computes an address from a set of IV.

The IVs are used to compute linearized memory addresses ($\texttt{addr}_{lhs}$ and $\texttt{addr}_{rhs}$). The linearization constants are computed using the UB values of the currently selected left DAP (DP_{lhs}). DP_{lhs} is used for the linearization of the IVs for the left and rhs to ensure addressing the correct address space even when the UB of a DAP of the rhs is smaller. The addresses are computed according to the following equation:

$$\texttt{address}(\mathcal{I}) = \sum_{j=0}^{\dim\mathcal{I}-2} \mathcal{I}[j] \cdot \overbrace{\prod_{k=j+1}^{\dim\mathcal{I}-1} DP_{lhs}[0][UB][k]}^{\text{linearizing constant}} + \mathcal{I}[\dim\mathcal{I} - 1] \tag{24}$$

where DP_{lhs} denotes the selected DAP of the currently selected data-producing interface. For example, the linearized address of $\mathcal{I}_{DP_{\texttt{base1}}}$ (9) using $DP_{\texttt{base}}$ (16) is

$$\begin{aligned} &\texttt{address}\left(\mathcal{I}_{DP_{\texttt{base1}}}\right) \\ &= \mathcal{I}_{DP_{\texttt{base1}}}[0] \cdot DP_{\texttt{base}}[0][UB][1] + \mathcal{I}_{DP_{\texttt{base1}}}[1] \\ &= (i_0 + i_2) \cdot 4 + (i_1 + i_3). \end{aligned} \tag{25}$$

The entire Streaming Buffer is configurable through a set of configuration registers which are only generated when required and can be configured with an APB interface $\texttt{IF Control}$, also shown in Fig. 6. These registers are

$$\begin{aligned} \mathcal{C}_{lhs,igu,[lb,ub,st]} &\coloneqq \text{bounds for the IGU on the left side} \\ \mathcal{C}_{rhs,igu,i,[lb,ub,st]} &\coloneqq \text{bounds of the } i\text{-th IGU on the right side} \\ \mathcal{C}_{rhs,map,i} &\coloneqq \text{mapper-config of the } i\text{-th right side} \\ \mathcal{C}_{lin} &\coloneqq \text{the constants for the linearizer.} \end{aligned}$$

The following sections provide an in-depth description of the macro-architecture of the components and the macro-architecture of the linearization unit, as well as an explanation to the necessary control signals of the Streaming Buffer.

1) Index Generation Unit and Flattening Unit: The IGU is responsible to generate the loop counters for every DAP of an interface. The counter behave similarly to the loop counters shown in Algorithm 1. For every DP in the set $\mathcal{D}$ of all DAP of an interface defines, a total amount of $\sum_{W \in DP_{\mathcal{D}}} \dim W$ counters are required. The output of the IGU represents the IV of all WAP in $DP_{\mathcal{D}}$. Fig. 7 shows the macro-architecture of the IGU. Each counter has configurable lower and UBs values and strides. The configuration is stored in the previously introduced registers. Equivalently to (9), the output values of the IGU have to be flattened to obtain $\mathcal{I}_{DP_{\mathcal{D}}}$. For this, we introduce a flattening unit that computes $\mathcal{I}_{DP_{\mathcal{D}}}$ from the output of the IGU, similar to the example shown in (10). The macro-architecture of the flattening unit is also shown in Fig. 7. The summation is computed by a set of adders. The output value of the flattening unit represents $\mathcal{I}_{DP_{\mathcal{D}}}$. It is not always necessary to generate a flattening unit. Consider the case where $\dim(DP_{\mathcal{D}}) = 1$ (see (9) with $N = 1$). In this case, it is sufficient to forward the output of the IGU to obtain $\mathcal{I}_{DP_{\mathcal{D}}}$.

Interfaces with multiple DAPs that have different dimensionality can disable counters which are not needed by current selected DAP $\in \mathcal{D}$.

2) Mapping Unit: The control signals that will be introduced in Section III-F4 are derived from the relation of IVs $\mathcal{I}_{DP_{lhs}}, \mathcal{I}^{[i]}_{DP_{rhs}}$ shown in Fig. 6. In the presence of DAPR π_i, the position of an entry within $\mathcal{I}^{[i]}_{DP_{rhs}}$ differs to the position in $\mathcal{I}_{DP_{lhs}}$. This means, for example, that it is not straight forward to compute the lexicographical order as shown in (11). It would be necessary to compare the kth position in $\mathcal{I}_{DP_{lhs}}$ with the $\pi_i(k)$th position in $\mathcal{I}^{[i]}_{DP_{rhs}}$. This increases the complexity of hardware generation. To circumvent this, we introduce a mapping unit that reverts the permutation introduced by connecting every $\pi_i(k)$th in $\mathcal{I}^{[i]}_{DP_{rhs}}$ position to the kth position of the outgoing IV. The introduced configuration registers select the current active DAPR. Note that, reversing the permutation of an IV does not change the iteration order defined by the DAP. Formally, the mapping unit reorders the values in a IV of a consuming interface according to all DAPR $\pi \in \mathcal{F}$, where $\mathcal{F}$ denotes the set of all DAPRs defined in the DAPs of a consuming interface. For completeness, if $\mathcal{F} \neq \emptyset$, we associate every DAP without DAPRs with an identity DAPR. If the domain of an DAPR has fewer dimensions than the incoming IV, we extend the DAPR by mapping the values of the indices that are outside of the domain to a default value. The macro-architecture of the mapping unit is shown in Fig. 8(a). The algorithm that generates the mapping unit is defined in Algorithm 2. The algorithm works as follows: For each DAPR in $\mathcal{F}$ it creates a list that stores which indices of the incoming IV have to be mapped to which indices of the outgoing IV. Thereafter, the list lengths are equalized by appending a default value to the list to ensures that DAPRs with different dimensions can be used. This synthetically raises the dimension of DAPRs with a lower dimension to a higher one. The previous process is repeated for each remaining DAPR in $\mathcal{F}$. Finally, a connection between the correct incoming and outgoing indices is created. When multiple DAPR exists the algorithm, creates a set of MUXes that multiplexes between the different DAPRs. The control

Algorithm 2 Create Mapping Unit

```
Input: List of DAPRs: $\mathcal{F}$,
       IV: $\mathcal{I}$
Output: $\mathcal{I}_{reordered}$
 1: $\mathcal{I}_{reorded} = ()$; $B = \{\}$// Init. output with empty vector, init. B as dictionary
 2: for all $n, \pi : \mathbb{N} \to \mathbb{N} \in$ enumerate($\mathcal{F}$) do
      // Map indices that are outside of the domain of $\pi$ to the default value zero
 3:   for all $i \in 0, \ldots,$ sizeof($B$) do
 4:     if sizeof($B[i]$) $< n$ then
 5:       $B[i].append(\mathbf{0})$
 6:     end if
 7:   end for
      // Gather the indices from $\mathcal{I}$ and map it the correct index
 8:   for all $i \in 0, \ldots, d$ do
 9:     $B[i].append(\mathcal{I}[\pi(i)])$
10:   end for
11: end for
    // Create output multiplexers to the output instance vector
12: for all $i \in 0, \ldots,$ sizeof($B$) do
13:   $\mathcal{I}_{reordered} = (\mathcal{I}_{reordered},$ create_multiplexer($B[i]$)$)^{\intercal}$
14: end for
```

signal of the MUX is connected to the introduced configuration register.

3) Linearizer: A linearizer has an input of K IVs and computes an address from one IV according to (24). Its macro-architecture is shown in Fig. 8(b). The linearization constants $\mathcal{C}_{in}$ are computed from the selected DAP of the producing interface during hardware generation of the Streaming Buffer. IVs with different dimensions are extended by appending a constant values to them. Which IVs is used is decided with a control signal, that will be introduced in Section III-F4. The result of the linearizer is a memory address that can be used by the memory wrapper to address the memory space.

4) Control Signals: The control signals for the previously introduced components are generated from a set of rules defined by the *relation* of the IV $\mathcal{I}_{DP_{lhs}}$ of the producing interface and the IVs $\mathcal{I}^{[i]}_{DP_{rhs}}$ of K consuming interfaces shown in Fig. 6. We define the following four relations.

1) The Streaming Buffer must read from its memory space when the current data element for a consuming interface was already transmitted by the producing interface
$$\psi_i = \left(\mathcal{I}^{[i]}_{DP_{rhs}} < \mathcal{I}_{DP_{lhs}}\right). \tag{26}$$
2) The Streaming Buffer can forward a data element when the states of a consuming and producing interface are equal
$$\phi_i = \left(\mathcal{I}_{DP_{lhs}} = \mathcal{I}^{[i]}_{DP_{rhs}}\right). \tag{27}$$
3) The Streaming Buffer has to write to its memory space when a data element transmitted by the producing interface is not yet required by the consuming interface
$$\varphi_i = \left(\mathcal{I}_{DP_{lhs}} < \mathcal{I}^{[i]}_{DP_{rhs}}\right). \tag{28}$$
4) The Streaming Buffer must only read from its memory space for one consuming interface i:
$$\rho_i = \psi_i \texttt{ if } i = 0 \texttt{ else } \psi_i \wedge \neg\rho_{i-1}. \tag{29}$$

A Streaming Buffer generates a single address for the rhs ($\texttt{addr}_{rhs}$) as shown in Fig. 6. Multiple accesses to its memory space have to be scheduled to prevent memory conflict. The fourth condition is a logical argument that extends the condition defined in (26). The condition ρ_i is set, when ψ_i of the ith consuming interface is set and ψ_j of all the other interface j with lower index ($j < i$) are not set. It is used to select which $\mathcal{I}^{[i]}_{DP_{rhs}}$ is connected to the right linearizer in Fig. 6. After the Streaming Buffer loaded the data from a consuming interface, it continuous loading the data for another consuming interface when the corresponding ρ_i condition is set. *GOURD* gives data transmission to consuming interfaces precedence over data transmission from producing interfaces. This property is implemented by observing if a Streaming Buffer requires to read from its memory space. For this we derive four state variables for each consuming interface from the current value of ρ_i and it values in the previous time step ρ_i^*:

$$\begin{array}{rc|c|c|l} & \multicolumn{2}{c}{\rho_i} & & \\ \cline{3-4} \mathit{wants\ load} & & wl_i & nl_i & \mathit{not\ loading} \\ \cline{3-4} \mathit{contious\ load}\ \rho_i^* & \Big| & cl_i & sl_i & \mathit{stops\ load} \\ \cline{3-4} \end{array}$$

As long as the Streaming Buffer does not load any data from memory it signals the producing interface its readiness to receive data. Incoming data can be directly forwarded to a consuming interface when ϕ_i is set. When the Streaming Buffer has to load data from memory, it signals the producing interface to stop sending data. Simultaneously, the Streaming Buffer starts to access the memory space and forwards the data to the correct consuming interface. The *wl*, *cl*, and *sl* are used to generate the necessary interface and control signals when the Streaming Buffer accesses the memory space.

5) Memory Wrapper Instantiation: We further instantiate the memory architecture of the selected memory wrapper described in Section III-E. Then, the buffer fabric is concluded by an APB bus interface which allows reconfiguration of the registers (introduced in Section III-F) at runtime.

G. Generate Wrapper

The final step generates a wrapper that connects all the Streaming Buffers and components. Streaming buffers with multiple producing interfaces use an MUX and a control register to select one interface. One-to-many connections use a merging unit to combine signals. Ready flags of interfaces connected to an MUX or merging unit are disabled when inactive. An APB interface is added to the wrapper, connecting the Streaming Buffers and control registers.

IV. Testing and Validation

We functionally evaluated *GOURD* by verifying that the generated Streaming Buffers behave according to the definition in the IR. The design is tested at RTL and postsynthesis level and compared to our Python model. We use the tools Xcelium [22] 23.09 and Genus [23] 22.14 by Cadence design systems. The synthesis target technology is 22FDX from GlobalFoundries [24] with standard cells from Synopsys [25]. We use a memory wrapper with a memory architecture that allows the implementation of the memory space of a Streaming Buffer with $0 < 2^k$ split memory macros. We verified that the equations to estimate the memory space UB in Section III-D are correct by comparing the estimated value of randomly

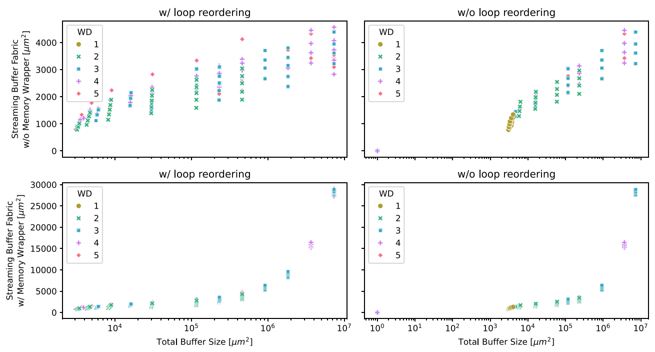


Fig. 9. Relation between the size of a Streaming Buffer fabric with and without memory wrapper to the total Streaming Buffer size. The markings and color represent the different analysed WAP dimensions (WD). The x-axis is scaled in log-scale.

generated DAPs with our Python model that virtually iterates the entire iteration space.

A. DAP, WAP, and DAPR Influence on Streaming Buffer

To analyse the influence of different component parameters we devised an experiment that shows the change in area given different parameters. To this end, we created a system of two connected components and varied the following parameters: DAP-dimension $\in \{1, \ldots, 4\}$, WAP-dimension $\in \{1, \ldots, 5\}$, the *UB* values of the WAP $\in \{2^1, \ldots, 2^8\}$, and the DAPR (every permutation). The transmitted bitwidth is set to 32 bit. The result is shown in Fig. 9. The markings and color represent the different analyzed WAP-dimensions (WDs). The x-axis in log-scale shows the total buffer size in μm^2. The y-axis displays the Streaming Buffer fabric area in μm^2 with and without the memory wrapper. We can see that the growth without the memory wrapper is logarithmic with increasing total Streaming Buffer fabric size for both DAPs with and without loop reordering. We also observed that a single reordering does not affect the fabric size much. This can be explained because the mapping unit which is generated in Section III-F2 is inferred without MUXes. After all, only a single loop reordering exists. As can be seen in the upper-right plot in Fig. 10, without loop-reording some buffer fabric sizes are zero. This happens when the DAPs of the producing and consuming interfaces are equal.

In the fabric size with the memory wrapper, we observe that the size briefly rises logarithmical before continuing to rise linearly. This comes from the hardware architecture of the memory wrapper. From this figure, we conclude that the general buffer fabric presents an implementation of a scalable solution that does not result in an exceeding area overhead. However, we also observe that the necessary buffer sizes depend heavily on the DAPs, which should be considered during the component selection. Finally, we remark, that more efficient memory wrapper architectures helps decrease the overhead introduced by it.

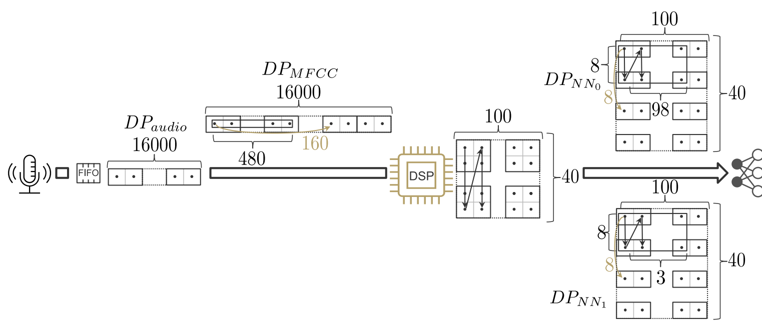
B. Audio Processing Pipeline

Systems for classification or recognition of environmental audio signals are important issues in the study of audio signal/speech recognition [16], [26]. Applications that work with the audio data often consist of two main stages: 1) extract features from the Mel frequency Cepstral coefficients (MFCCs) and 2) classification with a neural network [27], [28]. Hardware implementations of these systems typically require extensive engineering to optimize the components, data flow, and interconnections. Changes in the data flow can drastically affect the interconnects, necessitating manual system adjustments. *GOURD* simplifies the process by hiding the engineering overhead of changing the data flow through its hardware generation capability. Thus, *GOURD* can be used to rapidly narrow down the feasible system configurations while determine the necessary Streaming Buffer hardware. The pipeline includes black box models for a microphone, FIFO, MFCC module, and neural network accelerator. The microphone sends audio samples to the FIFO, the MFCC module samples 16000 audio samples from the FIFO, computes 40 MFCC coefficients with a hop size of 160, and sends the output to the accelerator for convolutional neural network processing. The accelerator has two possible DAP. For the sake of simplicity, we only focus on the data-transmitting interfaces of the components. An overview of the modules and their DAPs is shown in Fig. 10. The MFCC module has DAP DP_{MFCC} and reads the incoming audio data in chunks of 480 before it computes 1×40 output channels. Thereafter, the inner WAP shifts by the hop size and the next iteration begins. The accelerator computes a convolution with a 3×1 kernel over the incoming input channels. Configured with DAP DP_{NN_0}, the accelerator is in a weight stationary configuration, where it reads the incoming data as chunks of 98×8 data and computes an intermediate output value. In the case where the accelerator is configured with DAP DP_{NN_1}, it computes the output in an output stationary manner, where a partial output value after each chunk of size 3×8 is transmitted.

With our approach, we can easily explore the different system compositions and their characteristics as shown in Table I. The table shows the estimated buffer sizes according to the equations defined in Section III-D and the actual fields of the memory macros selected by the memory wrapper. The difference in fields is due to the available memory macros. In addition, we compare the number of steps it takes to transmit the data according to the DAPs in the ideal case to the simulative measured case. As shown, the ideal number of steps is smaller than the number of steps determined by RTL simulation. This is because a memory macro requires a step to

TABLE I
BUFFER EVALUATION FOR AUDIO PROCESSING PIPELINE

	FIFO → MFCC	MFCC→ Acc.(DP_{NN_0})	MFCC→ Acc.(DP_{NN_1})
Est. Buffer $\#mem$	480	3888	88
Actual Buffer $\#mem$	512	4096	128
Fabric size [μm^2]	5856.486	30035.387	4034.629
Steps ideal	47040	14863	11823
Steps actual	47137	14878	12313



$$DP_{audio} = ((0 \quad 16000 \quad 1))$$

$$DP_{MFCC_{in}} = ((0 \quad 16000 \quad 160)\ (0 \quad 480 \quad 1))$$

$$DP_{MFCC_{out}} = \left(\begin{pmatrix} 0 & 100 & 1 \\ 0 & 40 & 1 \end{pmatrix}\right)$$

$$DP_{NN_0} = \left(\begin{pmatrix} 0 & 100 & 1 \\ 0 & 40 & 8 \end{pmatrix} \begin{pmatrix} 0 & 98 & 1 \\ 0 & 8 & 1 \end{pmatrix}\right)$$

$$DP_{NN_1} = \left(\begin{pmatrix} 0 & 100 & 1 \\ 0 & 40 & 8 \end{pmatrix} \begin{pmatrix} 0 & 3 & 1 \\ 0 & 8 & 1 \end{pmatrix}\right)$$

Fig. 10. Example of an audio processing pipeline with configurable neuronal network accelerator and its internal DAP model in GOURD. With GOURD, it is simple to generate the resulting Streaming Buffers; hence providing feedback to the designer which neural network accelerator configuration to choose.

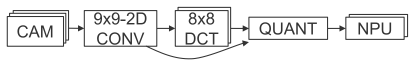


Fig. 11. Structure diagram of a video/image processing pipeline. We use *GOURD* to rapidly evaluate the impact of different design choices.

load the required data. Consider the second column Table I, the difference in steps is $14\,863 - 14\,878 = 15$, which corresponds to the number of times the Streaming Buffer starts to load the data from memory. The generated control signals allow for no additional loading step when continuously accesses the internal memory space.

C. Video Processing Pipeline

Video and image processing applications on embedded devices often include downstream tasks using neural networks [29], [30]. Examples include vehicle and pedestrian detection [31], medical anomaly detection [32] and many more. Designing these systems include a wide range of the tuneable system and component parameters to optimize the system with respect to the application constraints. An example video processing pipeline for compressed-domain deep learning applications is depicted in Fig. 11 (cf. [32] and [33]).

The tuneable parameters are camera resolution, number of pipeline stages to optionally include a DCT component between the 2D-CONV for image preprocessing and a quantization component. The DCT is used for compressed feature extraction. The pipeline implementations influences the complexity of the NPU stage at the end of the pipeline. We show how *GOURD* can be used in the design process of such a system. Given these parameters, we design three versions of the video processing pipeline. Version 1 does not use the DCT component. Version 2 includes the DCT. Version 3 features a different computation scheme of the DCT, where the feature extraction is split into horizontal and vertical feature extraction. We use *GOURD* to easily describe the different versions and the corresponding DAPs for each black box component models. In addition, we explore the influence of different camera resolutions on the pipeline versions by implementing a sweep ranging from a resolution from 340×340 to 1280×1280. The resolution values are tuned such that the downstream components can work with the dimension of the video. For every system configuration, we synthesized the generated Streaming Buffers. This totals to 480 data points per pipeline version. Due to space limitations we

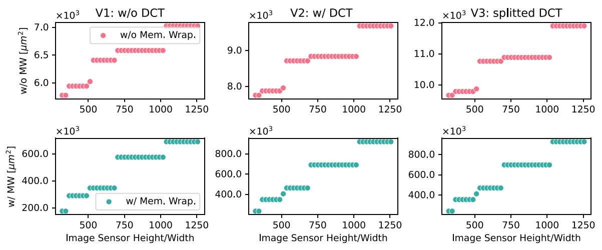


Fig. 12. Evaluation of the Streaming Buffers generated for the video/image pipeline in Fig. 11 for different image sensor resolutions. The green data points show the total size of all Streaming Buffers fabrics with the memory wrapper (MW). The red data points show only the logic size of the Streaming Buffer fabric.

cannot show the DAPs for each data point. We aggregate the total Streaming Buffer fabric size with and without memory wrapper. The result is shown in Fig. 12. *GOURD* implements two Streaming Buffers for version 1, three for version 2, and four for version 3. There are regions where the different resolutions only lead to minor changes in the resulting area. Since the register and signal sizes of a Streaming Buffer are derived from the dimension of the DAP, they may be generated with the same bitwidth. For example, a counter register that counts up to 513 or 1023 requires the same bitwidth. Similar behavior is observed regarding the size of the memory wrappers. This is caused by the used memory wrapper architecture, which only allows a power of two split of memory macros. Using *GOURD* it is possible to easily evaluate the influences of different system and component parameters. At the same time, *GOURD's* hardware generation capability also provides synthesizable RTL hardware for the Streaming Buffers; hence limiting additional engineering hours and thus improving the productivity.

D. Prior Work

To the best of our knowledge *GOURD* represents a step in a novel research direction. Current generator-based systems for streaming the data applications, such as HLS systems, allow to adding manually scaled FIFO components as the data buffers [17], [34]. This means that the DAP of all the components used must be compatible with each other. As far as we observed, these systems cannot generate smart components, such as our proposed *Streaming Buffers* based on a simple dataflow description. The other two current state-of-the-art research domains we observe consider: 1) efficient data transmission from and to hardware components and 2) optimization/generation of efficient hardware accelerators. *GOURD* does not focus both the domains, but provides a step combining both the domains toward an optimized multi-instance approach for the heterogeneous systems.

V. Conclusion

Our paper proposes an approach for generating architectures tailored to the data streaming applications. Using a minimal DSR, we capture both the components and their DAPs. This representation suffices to automatically generate synthesizable RTL of Streaming Buffers. We demonstrate how our approach applies to real-world applications, allowing rapid analysis of different system parameters on the generated hardware. Our approach is able to accommodate prevalent data orchestration methodologies. They can be integrated into *GOURD* by adapting their architecture into a memory wrappers. This allows for rapid evaluation of the system alternatives with minimal implementation overhead. Future work will include incorporating component latency for better system evaluation.

References

[1] M. Pellauer et al., "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proc. ASPLOS*, 2019, pp. 137–151.

[2] R. Komuravelli et al., "Stash: Have your scratchpad and cache it too," in *Proc. ISCA*, 2015, pp. 707–719.

[3] J. Clemons, C.-C. Cheng, I. Frosio, D. Johnson, and S. W. Keckler, "A patch memory system for image processing and computer vision," in *Proc. MICRO*, 2016, pp. 1–13.

[4] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *Proc. ISCA*, 2017, pp. 416–429.

[5] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks, "The accelerator store: A shared memory framework for accelerator-based systems," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 1–22, 2012.

[6] Z. Luo et al., "Rubick: A synthesis framework for spatial architectures via dataflow decomposition," in *Proc. DAC*, 2023, pp. 1–6.

[7] L. Jia, Z. Luo, L. Lu, and Y. Liang, "Automatic generation of spatial accelerator for tensor algebra," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 6, pp. 1898–1911, Jun. 2023.

[8] L. Lu et al., "TENET: A framework for modeling tensor dataflow based on relation-centric notation," in *Proc. ISCA*, 2021, pp. 720–733.

[9] P. P. Bernardo, C. Gerum, A. Frischknecht, K. Lübeck, and O. Bringmann, "UltraTrail: A configurable ultralow-power TC-ResNet AI accelerator for efficient keyword spotting," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 4240–4251, Nov. 2020.

[10] Z. Du et al., "ShiDianNao: Shifting vision processing closer to the sensor," in *Proc. ACM/IEEE ISCA*, 2015, pp. 92–104.

[11] J. Yang, H. Zheng, and A. Louri, "Versa-DNN: A versatile architecture enabling high-performance and energy-efficient multi-DNN acceleration," *IEEE Trans. Parallel Distribut. Syst.*, vol. 35, no. 2, pp. 349–361, Feb. 2024.

[12] H. Dutta, F. Hannig, M. Schmid, and J. Keinert, "Modeling and synthesis of communication subsystems for loop accelerator pipelines," in *Proc. IEEE ASAP*, 2010, pp. 125–132.

[13] D. Giri, K.-L. Chiu, G. Di Guglielmo, P. Mantovani, and L. P. Carloni, "ESP4ML: Platform-based design of systems-on-chip for embedded machine learning," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2020, pp. 1049–1054.

[14] Q. Liu et al., "Unified buffer: Compiling image processing and machine learning applications to push-memory accelerators," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 2, pp. 1–26, 2023.

[15] J. Keinert, C. Haubelt, and J. Teich, "Modeling and analysis of windowed synchronous algorithms," in *Proc. IEEE ICASSP*, 2006, pp. 892–895.

[16] M. Alam, M. D. Samad, L. Vidyaratne, A. Glandon, and K. M. Iftekharuddin, "Survey on deep neural networks in speech and vision systems," *Neurocomputing*, vol. 417, Dec. 2020, pp. 302–321.

[17] G. Sohn, N. Zhang, and K. Olukotun, "Implementing and Optimizing the scaled dot-product attention on streaming dataflow," 2024, *arXiv:2404.16629*.

[18] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. ACM/IEEE ISCA*, 2016, pp. 367–379.

[19] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," in *Proc. ASPLOS*, 2018, pp. 461–475.

[20] C. Lattner et al., "MLIR: Scaling compiler infrastructure for domain specific computation," in *Proc. IEEE/ACM CGO*, 2021, pp. 2–14.

[21] "CIRCT-circuit IR compilers and tools." Accessed: Dec. 2023. [Online]. Available: https://circt.llvm.org/

[22] "Xcelium logic simulator." 20234. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html

[23] "Genus synthesis solution." Accessed: Mar. 2024. [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html

[24] "FDXTM FD-SOI." Global Foundries. Dec. 2023. [Online]. Available: https://gf.com/technology-platforms/fdx-fd-soi/

[25] "Synopsys inc." Accessed: Mar. 2024. [Online]. Available: https://www.synopsys.com/

[26] W. Mu, B. Yin, X. Huang, J. Xu, and Z. Du, "Environmental sound classification using temporal-frequency attention based convolutional neural network," *Sci. Rep.*, vol. 11, no. 1, Nov. 2021, Art. no. 21552.

[27] A. Aswad, E. Alghannam, and Q. Zhang, "Developing MFCC-CNN based voice recognition system with data augmentation and overfitting solving techniques," in *Advances in Artificial Systems for Medicine and Education VI.* Cham, Switzerland: Springer, 2023. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-24468-1_11

[28] A. Sehgal and N. Kehtarnavaz, "A convolutional neural network smartphone app for real-time voice activity detection," *IEEE ACCESS*, vol. 6, pp. 9017–9026, 2018.

[29] K. P. Seng and L.-M. Ang, "Embedded intelligence: State-of-the-art and research challenges," *IEEE Access*, vol. 10, pp. 59236–59258, 2022.

[30] Y. Ming et al., "Action recognition in compressed domains: A survey," *Neurocomputing*, vol. 577, Apr. 2024, Art. no. 127389.

[31] L. Chen et al., "Deep neural network based vehicle and pedestrian detection for autonomous driving: A survey," *IEEE Trans. Intell. Transp. Syst.*, vol. 22, no. 6, pp. 3234–3246, Jun. 2021.

[32] T. Fernando, H. Gammulle, S. Denman, S. Sridharan, and C. Fookes, "Deep learning for medical anomaly detection–a survey," *ACM Comput. Surv.*, vol. 54, no. 7, pp. 1–37, Jul. 2021.

[33] Y. Dong and W. D. Pan, "A survey on compression domain image and video data processing and analysis techniques," *Information*, vol. 14, no. 3, p. 184, Mar. 2023.

[34] "Vitis HLS." Accessed: Jul. 2024. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html