Automated level-based clustering of dataflow actors for controlled scheduling complexity

Ophélie Renaud, Hugo Miomandre, Karol Desnos, Jean-François Nezan

Automated Level-Based Clustering of Dataflow Actors for Controlled Scheduling Complexity ⋆

Ophélie Renaud[a], Hugo Miomandre[a], Karol Desnos[a], Jean-François Nezan[a]

[a] *Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164, Rennes, France, first.last@insa-rennes.fr,*

Abstract

Dataflow Models of Computation (MoCs) significantly enhance parallel computing by efficiently expressing application parallelism on multicore architectures, unlocking greater performance and throughput. However, the complexity of graphs within dataflow-based systems can result in a time-consuming resource allocation process. To address this issue, a solution is to cluster computations to ease heuristic solving. The information encompassing the context of computations and the constraints of the architecture plays a crucial role in determining application performance. This paper presents an automated approach that leverages this information to control graph complexity prior to the resource allocation process. Experiments demonstrate that the proposed method, driven by clustering, not only yields improved throughput but also provides better mapping decisions and data transfer efficiency, achieving a throughput up to 1.8 times higher than state-of-the-art techniques.

Keywords: Dataflow model, Hierarchy, Clustering, Pipeline

1. Introduction

With the exponential growth of data and the increasing demand for faster processing, the challenge of parallel programming has become more important than ever. Parallel programming involves enabling computer programs to use multiple Processing Elements (PEs) simultaneously, which can improve the performance and efficiency of complex tasks.

⋆This work was supported by DARK-ERA (ANR-20-CE46-0001-01).

Parallel programming paradigms provide a framework for understanding how applications behave on target systems, guiding the development of high-performance parallel software. Among these paradigms is the dataflow Model of Computation (MoC) [1], which can naturally express parallelism making it well-suited for a wide range of data-processing applications. A dataflow model is a representation of an application, consisting of a graph $G = (V, E)$, where V is the set of vertices, also known as actors, and E is the set of edges. The actors represent computations, while the edges represent First In First Out queues (FIFOs) buffers.

In order to expand the capabilities of the dataflow MoC, a hierarchical structure has been introduced, promoting composability and modularity [2]. These properties foster the creation of complex programs by assembling smaller, self-contained, and reusable components. Various hierarchy-based dataflow MoCs exist, and one of them is the State-Aware Parameterized and Interfaced DataFlow (SPiDF) MoC [3]. In this context, the term *hierarchy* refers to an organizational framework that allows actors to be defined by subgraphs instead of relying on standard code.

In dataflow programming, mapping a program to PEs can be challenging for complex dataflow graphs with a large number of actors and FIFOs resulting in a time-consuming and resource-intensive process [4]. Modern applications, especially those with significant data parallelism, often exhibit a high number of actors within their dataflow graphs. This poses a challenge as allocating computational, communication, and memory resources for executing a dataflow graph is an NP-complete problem, with complexity largely dependent on the graph size and degree of parallelism exposed. Hence, it is essential to propose techniques for reducing graph complexity and pre-allocating resources, such as clustering. The concept of clustering dataflow actors involves grouping actors together based on their inherent relationships, enabling the newly created cluster to be considered as a single entity within the dataflow graph. This approach simplifies the overall graph structure and streamlines the resource allocation process, reducing complexity and effort.

When establishing effective actor clustering, it is crucial to consider two key aspects: maintaining parallelism and preserving graph consistency. Addressing the first point, uncontrolled clustering reduces complexity but transforms the graph into a sequential structure, with only a single actor remaining active. In addition to parallelism, preserving graph consistency is essential. Grouping two or more actors into a single equivalent hierarchical actor can potentially alter the behavior of the application or even lead to deadlocks. To

mitigate these issues, clustering rules have been introduced in [5]. These rules serve as guidelines to ensure that the grouping process adheres to specific criteria, preventing unintended consequences in the behavior of the application.

Previous efforts have therefore proposed a method of controlled actor clustering, with the number of processing elements in the architecture automatically aligning the degree of parallelism of the application. In this context, the initial work, as presented in [6], focuses on reducing data parallelism on the target platform. The subsequent work, [7], extends the first by introducing pipeline parallelism to the sequential portions of the graph, matching pipeline stages to the target architecture. Both of these methods are parameterized, with the parameter corresponding to the hierarchical level at which parallelism is adapted. The lower levels are grouped coarsely, and these methods yield a set of actor clustering configurations, the number of which corresponds to the number of hierarchical levels in the input graph. While both methods have effectively simplified graphs by producing transformed graph configurations based on the adjusted level, it is worth noting that certain configurations from the latter study proved non-functional when pipelines were introduced inside cycles, as reported in [8].

This paper presents a clustering method that automatically adjusts the granularity of an application to the target architecture. Unlike previous methods, the proposed approach incorporates the hierarchical context of the dataflow graph and ensures graph consistency across various architectures. This method effectively reduces the solution space to the best. The rest of this paper is organized as follows: Section 2 presents the SPiDF MoC, the standard mapping and scheduling method and the state-of-the-art clustering heuristics. Section 3 describes the proposed method. Section 4 outlines the experimental evaluation of the clustering method with respect to graph complexity, resource allocation process time, throughput, and parallelism setup. Finally, Section 5 concludes this paper.

2. Context and related work

2.1. SPiDF MoC

The SPiDF MoC [3] illustrated in Figure 1 is a recent extension of the Synchronous Dataflow (SDF) MoC offering significantly enhanced flexibility and capability in signal processing design. The two features of interest of the model exploited in this paper are state awareness and the hierarchy of a graph.

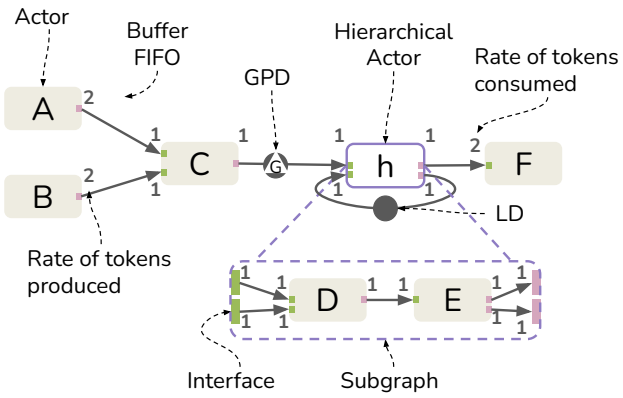

Figure 1: Illustration of the SPiDF MoC

State awareness in the SPiDF MoC refers to the capacity of the model to take into account the internal state of actors within a dataflow graph. The internal state of a dataflow graph can be represented using initial tokens in FIFOs, which are also known as delays. Delays are used to create an offset on the token consumption by an actor in the graph. By introducing initial tokens, it is possible to break dependencies between the connected actors that can impact scheduling opportunities. The SPiDF model introduces two types of delay mainly used in dataflow application modeling, distinguished by their level of persistence: Local Delay (LD) and Globally Persistent Delay (GPD).

- The LD delayed data tokens are preserved within the scope of a unique graph iteration. Indeed, LD are initialized by an optional *Setter* actor at the beginning of each iteration and the final token is retrieved by a *Getter* actor at the end of the iteration.

- The GPD are initialized at the beginning of the program and the value of the delay at the end of a graph iteration is used as the initial value for the next iteration.

The two main uses of delays are cycles and pipelines [9]. In dataflow MoC, a cycle refers to a sequence of actors characterized by one or more feedback loops. This means that after consuming the initial tokens, the first actor in this sequence becomes dependent on the tokens produced at the output of the last actor. At this point, a precedence constraint comes into play, significantly

restricting parallelism. In order to increase parallelism opportunities, one approach is to partially unroll the cycle by exploiting its sequential nature and inserting delays to create pipeline stages, as discussed in [7]. It is important to note that this cycle unrolling method is only applied to the LD persistence cycle initialized by a setter actor at each graph iteration. Applying this method to other cycle persistence can lead to additional overhead in terms of processing and memory usage [10], [11].

Example. Considering the graph illustrated in Figure 1, a FIFO feedback loop is formed around the hierarchical actor h, resulting in a cycle in the graph composed of actors D and E. This particular FIFO incorporates a LD.

Pipelining a dataflow graph consists in dividing a graph into pipeline stages, with an explicit delay used to separate stages [12]. During the first iteration of the graph, the application runs the first stage and retains the final token value, which is then used during the second iteration as the initial token for the second stage, and so on.

Example. Considering the graph G illustrated in Figure 1, a GPD is inserted between actor C and actor h creating a stage composed of actors A, B, C and another one composed of actors D, E, F.

In other words, once the graph iteration has reached the pipeline depth, the previous iteration stages can be executed concurrently. It is recommended not to introduce a pipeline within a cycle, as this would inevitably result in token shifts, thus changing the behavior of the dataflow graph. Only the developer can determine if the result is correct. Currently, this distinction cannot be automated. Therefore, to ensure consistency of results, it is advisable not to automatically insert pipelines into a dataflow cycle [13].

2.2. Standard flattening method

In order to generate a multicore implementation of an application, a scheduling process must be completed beforehand. This procedure entails assigning and ordering actor firings to available PE. The typical static scheduling process for SPiDF [14] involves four main tasks, namely flattening, Single rate Directed Acyclic Graphs (SrDAG) transformation [15], mapping, and scheduling. The *flattening* task replaces all hierarchical actors with their contents, which implies bringing actors of different granularities to the top-level graph. The *SrDAG transformation* reveals parallelism by converting

the flattened graph into a single-rates graph where consumed and produced rates are equal on each FIFO and cycles are unrolled. During this transformation, actors are duplicated based on their associated coefficients from the Repetition Vector (RV), which brings data dependencies to the forefront. The calculation of the RV, denoted as $\mathbf{q}$, where each coefficient indicates the minimum number of firings required for each actor to restore the graph to its original state [1]. After the *SrDAG transformation*, each actor in the SrDAG is individually *mapped* and *scheduled* using complex heuristics. This process involves exploring numerous possibilities, making it computationally intensive. Furthermore, as the number of actors in the application and the available processing elements in the architecture increase, the mapping and scheduling tasks become even more intricate. In fact, this process is known to be NP-hard, indicating its high level of computational complexity, as discussed in [4].

2.3. Cluster of SDF actors

This section reviews the state of the art in dataflow actor clustering methods. Clustering SDF actors is an efficient method for reducing graph complexity upstream of the standard resource allocation process. Among the various techniques, hierarchical dataflow-based clustering methods are commonly employed, with one notable branch being agglomerative clustering. Agglomerative clustering operates as a bottom-up method, systematically merging actors based on their similarities. This technique revolves around identifying actors with comparable attributes and behaviors within a dataflow graph, ultimately grouping them to form subgraphs with shared features. Subsequently, a sub-code is generated for each subgraph, encapsulating the behavior of the grouped actors. This sub-code replaces the hierarchical actor, transforming it into a standard actor defined by the newly created code. The utilization of agglomerative clustering techniques results in the conversion of hierarchical actors into standard actors, streamlining dataflow graph processing and enhancing overall efficiency.

To address potential challenges arising from clustering, a set of clustering rules has been introduced in [5], with detailed illustrations encompassing four distinct clustering techniques. These rules are designed to ensure that the behavior of the application is preserved during the clustering process and to prevent the emergence of deadlocks. The authors in [5] also present four clustering techniques. The first method is a manual clustering technique.

This approach relies on senior developer analysis to understand the application and enable effective optimizations. However, the involvement of senior developers is time-consuming and error-prone. This task is especially challenging when the applications increase. Automatic strategies like the one presented in this paper enable the safe and efficient handling of complex application programs. The second method consists of clustering SDF subgraphs for as long as possible. Similar to the previous method, it also depends on the developer's judgment to stop the iteration of subgraph clustering which, if uncontrolled, leads to a sequential application. The third method is the Unique Repetition Count (URC) clustering technique. This method consists of creating an SDF subgraph of at least two sequential actors with identical RV coefficient and no internal state as defined in Definition 1. The last method, a dynamic clustering one, falls beyond the scope of this paper, which primarily addresses static allocation. This process was originally introduced in [16].

Definition 1. Let $G = (V, E)$ be a directed acyclic graph representing a well-ordered, URC SDF subgraph, where V is the set of vertices and E is the set of edges. The graph G satisfies the following properties:

1. **Well-Ordered:** G is well-ordered if it admits only one topological sort, ensuring a unique ordering of vertices.
2. **URC:** G is a URC SDF subgraph if all vertices in V have identical repetition counts, denoted as **q**-values.

Two other SDF graph reduction techniques are discussed in [17]. The first technique involves abstracting a Homogeneous SDF (SDF) clustering actor with identical firing rates. All the above-mentioned techniques aim to simplify the graph structure but may impact the resulting schedule. In contrast, the second technique described in [17] maintains throughput and latency by transforming the SDF graph into a smaller HSDF graph using the Max-plus algebra technique. However, this HSDF graph introduces complexity in extracting timing parameters and determining a feasible schedule due to multiple executions of a single firing. Nevertheless, the Slack-Based Merging techniques[18] address this concern by considering timing constraints when selecting clusters of dataflow actors, thereby minimizing potential schedule losses.

Another clustering technique depicted in [19], the Pairwise Grouping of Adjacent Nodes (PGAN) method involves iteratively clustering two neighboring actors, resulting in nested looped schedules. This iterative process is applied to an entire SDF graph, pairing actors until only a single cluster remains. The resulting schedule variability is influenced by the selection of coupling heuristics. Consequently, the technique offers a multitude of potential configurations, making comprehensive evaluation a complex task. Its extension, the Pairwise Grouping of Adjacent Nodes for Acyclic graph (APGAN) clustering technique introduced in [20] shows that the first pairs of actors that form a cluster with the highest RV leads to a minimum memory requirement schedule and minimizes the possible configuration. The Cluster Finite State Machines technique, as described in reference [21], effectively manages cyclic features but is not suitable for hierarchical graphs. Clustering techniques in dataflow MoC analysis aren't limited to simplifying graphs; they can also optimize code. For instance, one study [22] explores the balance between modularity and code size when automatically generating code from hierarchical SDF graphs. Another study [23] delves into the trade-off between modularity and reusability in code generation from synchronous block diagrams. Additionally, there is work [24] that combines these approaches for code generation. It introduces a compositional abstraction for composite actors, allowing for modular compilation. This work provides algorithms for the automatic synthesis of non-monolithic Deterministic SDF with shared FIFOs (FIFO) profiles for composite actors based on the profiles of their sub-actors.

2.4. Previous work: hardware-specific clustering method

This section describes the two previous methods, which are extended in this paper. The two methods, known as the Scaling up of Clusters of Actors on Processing Element (SCAPE) method, assert their ability to adjust the granularity of an application to match the available processing elements in an architecture. They achieve this by employing clustering techniques to optimize mapping and scheduling opportunities.

The first clustering method [6] aims to reduce excessive data parallelism on the number of PE. It takes as input a parameter n_c that corresponds to the number of hierarchical levels to be clustered entirely. The method analyses the hierarchy levels of the input SPiDF graph starting at the bottom. The level is a cluster as long as the current level is lower than the parameter

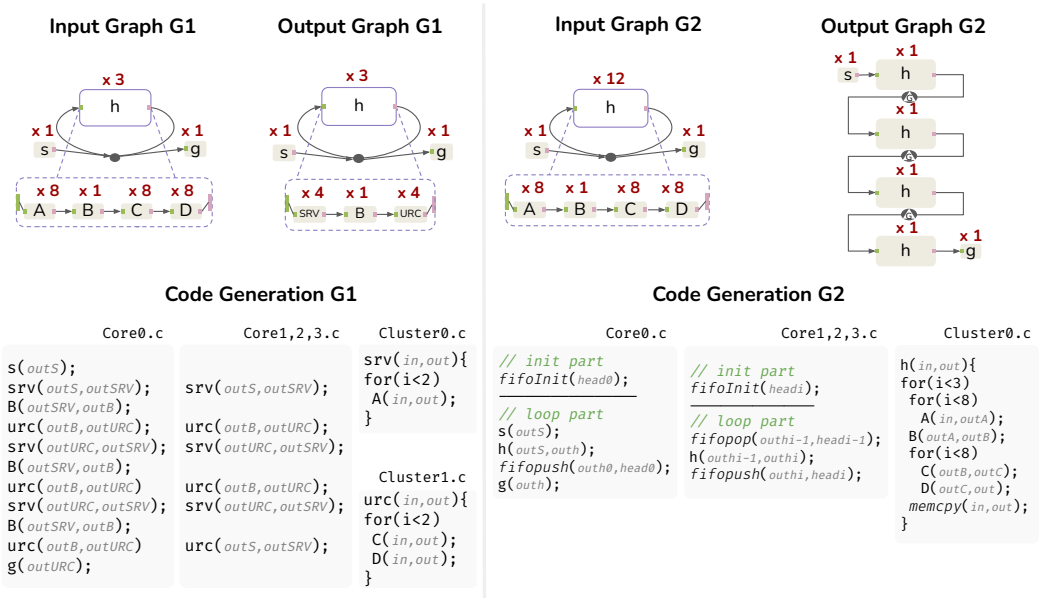

Figure 2: Illustration of cycle management using the proposed method in two cases deployed on a 4 PEs architecture. The figure on the left shows the graph transformation and the simplified generated code for a graph whose hierarchical actor repetition is greater than the number of PEs. The figure on the right shows the case where the repetition of the hierarchical actor is less than the number of PEs.

n_c. The levels above the parameter n_c are left as they are. The level equal to the parameter n_c is partially clustered by identifying interesting patterns of actors to cluster. The patterns are URC, introduced in [5], and Single Repetition Vector (SRV), introduced in [6] which is an actor with a RV greater than the number of PE. The RV of these clusters are reduced to match the number of PEs, which is called scaling. Thus, with the different values of n_c, the method offers a set of clustering configurations of decreasing granularity and parallelism according to the target architecture.
Example. Consider the graph $G1$ shown in Figure 2, along with a 4 PE architecture. This graph, which has 2 levels of hierarchy (one top and one bottom), admits 4 possible configurations:

1. All actors in the graph are clustered into one, making it sequential.
2. No clustering is performed.
3. Particular patterns are identified and clustered at the bottom level of the hierarchy.
4. Particular patterns are identified and clustered at the top level, while the bottom level is coarsely clustered.

In this specific example, the patterns are only localized at the bottom level. Actors C and D are URC candidates because they are at least 2 sequential actors sharing the same repetition; here, their repetition is 8. Actor A is a SRV candidate because it is an isolated actor with a repetition greater than the number of PEs; here, 8 is greater than 4. These two identified patterns are processed as two clusters of actors. A subgraph is generated containing actors C and D, and another subgraph containing actor A. The repetition of each subgraph is adjusted to match the number of PE; each subgraph is set to repeat 4 times, and their contents repeat twice so that $2 \times 4 = 8$, the original repetition. For each cluster, the APGAN method is applied to compute the internal subgraph schedule. Specifically,

$$S_{SRV_{G1}} = 2(A) \text{ and } S_{URC_{G1}} = 2(C\ D).$$

The subgraphs are then translated into a C file containing function calls associated with the actors in nested loops according to the calculated schedule. This translation transforms the subgraph into a standard actor. The transformed, simplified, and optimized graph is sent to the rest of the resource allocation process.

The second clustering method [7] extends the first one by adding two other patterns of actors. The patterns are *sequential* pattern which is a sequence of actors with a degree of parallelism less than the number of PEs and *Loop* pattern which is a sequence of actors where the last one is connected to the first one by a delayed FIFO incorporating a LD. The method divides the actors from the *sequential* pattern into multiple subgraphs, where the number of subgraphs is the same as the number of PEs. Each subgraph is designed to have the same latency, and a delay is introduced between each subgraph to create distinct pipeline stages. The technique also suggests partially unrolling loops to match the number of PEs. This involves dividing the original cycles into several groups, the number of which corresponds to the number of processing elements in the target. Each cluster comprises smaller loops, where the number of iterations aligns with the iteration count of the original loop divided by the number of processing elements. Additionally, delays are introduced within the FIFO loops connecting these clusters, thereby establishing pipeline stages that correspond to the target architecture. This method also provides a set of clustering configurations with greater clustering opportunities than the original SCAPE method but doesn't consider the hierarchical context state before inserting pipelines inside some subgraphs.

Example. Let's consider the graph $G2$ depicted in Figure 2, in conjunction with a 4-PE architecture. This graph, featuring 2 levels of hierarchy (one top and one bottom), offers 4 potential configurations, with the distinguishing factor being the identification of specific patterns. In this example, the patterns are localized at both the bottom and top levels. At the bottom level, actor B repeats only once, which is fewer than the number of PEs, rendering it a candidate for a *sequential* pattern. Meanwhile, at the top level, the hierarchical actor h is a candidate for looping because one of its outputs connects to one of its inputs. Regarding the clustering configuration where patterns are solely identified at the bottom level, the process automatically pipelines actor B. However, in this particular clustering configuration, where a pipeline is created automatically without considering the hierarchical context, and because the subgraph containing B is within a cycle, this results in a flawed clustering configuration. This occurs because there will be a mismatch in the order in which the data is processed. Another clustering configuration that identifies patterns only at the top level coarsely clusters the bottom level and unrolls the cycle, as illustrated in Figure 2. In this case, only 3 out of 4 configurations are not flawed.

The proposed method is an extension of SCAPE that offers better parallelism management by identifying patterns on each hierarchical level and integrating pipelines on authorized ones.

3. The proposed method

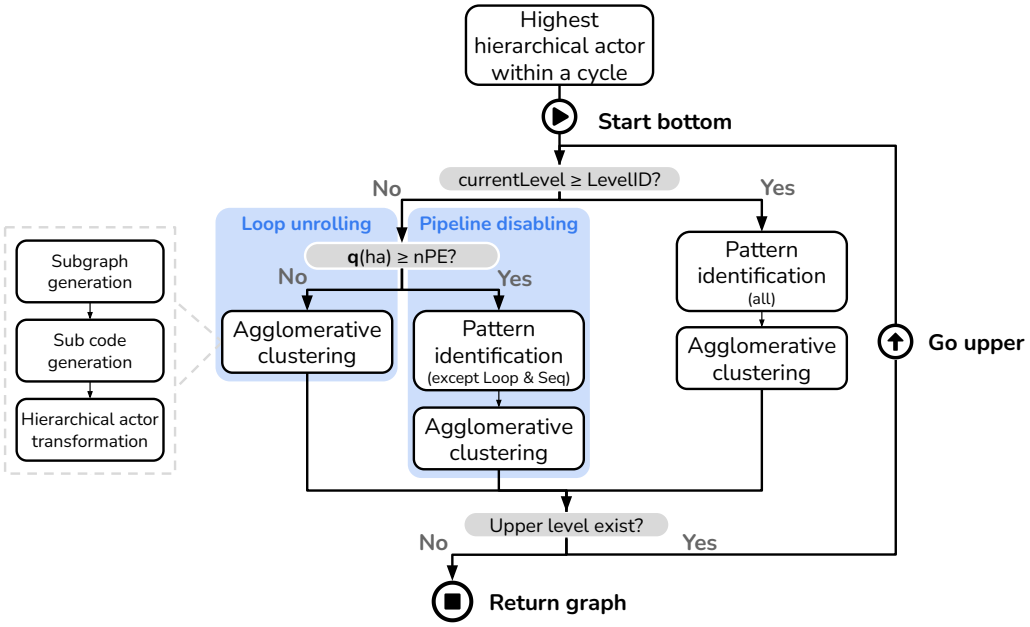

Figure 3: Algorithm of the method

The proposed clustering method simplifies the graph structure and is inserted just before the flattening process, in other words upstream of the standard resource allocation process. Unlike the previous SCAPE versions, the method returns a single well-suited clustering configuration to the architecture. The algorithm of the method is illustrated in Figure 3.

The different parts of the algorithm are detailed in the following sections. The method involves the identification of levels that impede pipeline utilization, detailed in Section 3.1. It also encompasses the recognition of four actor patterns, as expounded in Section 3.2, within authorized levels. These patterns are then replaced by hierarchical actors that encapsulate the identified actors, as described in Section 3.3. Additionally, the algorithm involves

scheduling computation for the newly formed subgraph, discussed in Section 3.4, and the subsequent replacement of the behavior of the hierarchical actor with the generated code, explained in Section 3.5.

3.1. Identification the highest hierarchical actor within a cycle

The method takes into account the limitations of prior work, where pipeline insertion in sequential portions was performed without considering the hierarchical context. This previous approach could inadvertently introduce pipelines within a cycle, potentially causing a misalignment of token order. To address this issue, the new method starts by assessing the feasibility of generating pipelines at a specific level, considering that cycles should not include pipelines. This assessment is carried out by initially traversing the graph from top to bottom and identifying the highest level where a hierarchical actor is present within a cycle.

To maximize the benefits of cycle existence and avoid inserting pipelines into cycles, the method identifies the highest hierarchical actor present within any cycles in the graph. The clustering approach chosen depends on the repetition of the cycle concerning the available processing elements:

- If the common repetition among all actors within the cycle, including the hierarchical actor, is less than the number of processing elements, the method allows all specific clusterings, as outlined in Section 2.3, that are unrelated to the pipeline on the lower levels of the hierarchical actor. In other words, only data parallelism is reduced on the child levels of the hierarchical actor.

- Conversely, if the common repetition among all actors within the cycle, with the hierarchical actor, is greater than the number of processing elements, the method coarsely groups the hierarchical levels below the hierarchical level of the actor and proceeds with partial loop unrolling.

Example. Considering the graphs $G1$ and $G2$ illustrated in Figure 2 on 2 levels. On the top level, a hierarchical actor h is self-looped and its content is a sequence of actor A, B, C, D. h_{G1} is fired three times per graph iteration and h_{G2} is fired twelve times. On both graphs, the highest hierarchical actor within the cycle are h located on the top-level graph.

3.2. Identification of dataflow actor patterns

This section details specific patterns of actors that will be identified for clustering. The choice of clustering patterns is contingent on the repetition **q** of the highest hierarchical actor within the graph ha, as solved in the preceding section, and whether the method examines the lower or upper levels concerning the hierarchical actor. The method follows a bottom-up approach, starting the recognition process at the lowest hierarchical level of the graph. Once all specific actors at the current level are clustered, the method proceeds to seek patterns at higher hierarchical levels.

Here are the patterns and corresponding scenarios:

- When the collective repetition of all actors within the cycle, including the hierarchical actor, is below the number of available processing elements and the current hierarchical level is lower than the level where the highest hierarchical actor is located, the method will pinpoint two patterns from the first method, SCAPE, which aim to reduce data parallelism according to the number of processing elements. These two patterns are URC and SRV, as introduced in Section 2.3. Additionally, in pursuit of further graph complexity reduction, the method will identify another pattern: the Low Latency Impact (LLI) pattern. This process consists of clustering actors with a negligible execution time compared to other actors on the same hierarchical level. This cluster respects the topological order of the graph and has no internal state. This case corresponds to the *Pipeline disabling* stage in Figure 3.

- Conversely, when the repetition of all actors within the cycle, including the hierarchical actor, exceeds the number of available processing elements, the method coarsely groups levels lower than the one containing the highest hierarchical actor. This scenario is representing by the *Loop unrolling* stage in Figure 3.

- As for the upper levels, the method takes into account all patterns, as it no longer faces the potential insertion of pipelines within cycles.

Example. Considering the graphs $G1$ and $G2$ Figure 2 and a 4 PE architecture. As h_{G1} is fired three times per graph iteration that is less than 4 then its content is the object of clustering URC, SRV and LLI patterns of actors. As h_{G2} is fired twelve times per graph iteration that is more than 4 then its content is the object of coarse clustering.

3.3. Subgraph transformation

Section 2.3 delved into the concept of agglomerative clustering techniques and their role in streamlining dataflow actors. Now, the focus shifts to the next phase in the proposed method, termed *Subgraph Transformation* which is a pivotal step in the hierarchical clustering of dataflow actors. This phase operates subsequent to pattern identification and entails the generation of subgraphs containing the identified actors. Within these subgraphs, the delays of all persistency are systematically extracted and interconnected with the corresponding hierarchical actors. The subgraph transformation process encompasses the addition of interfaces to the subgraph and the establishment of links between the ports on the hierarchical actor and the delay(s). Extracting these delay(s) serves a crucial purpose, enabling the computation of the internal execution order of the subgraph through the APGAN method. Moreover, it is important to highlight that at this step, The proposed method scales the URC and SRV subgraphs to align them with the target architecture. This scaling process ensures that these subgraphs iteratively match the number of PE of the target. The subsequent example provides further clarity on this scaling operation and its implications on different graphs within the context of the execution of the proposed method.

Example. Considering the graphs $G1$ and $G2$ Figure 2 and a 4 PE architecture. URC_{G1} and SRV_{G1} are scaled to iterate 4 times and their content twice per graph iteration. h_{G2} is duplicated 4 times and its content is rolled up in a loop that iterates 3 times per graph iteration so that $3 \times 4 = 12$.

3.4. Cluster code generation

Following the agglomerative clustering process, the subsequent phase entails the generation of code for the cluster. The method is implemented in the open source Parallel and Real-time Embedded Executives Scheduling Method (PREESM) framework [25]. The tool takes as input a dataflow graph where standard actors are specified in C code. Consequently, the cluster code also takes the form of C files, where a specialized function is defined. This function encapsulates the scheduled firing of actors within the subgraph, and the schedule itself is computed with the APGAN algorithm. The translation of actor firings into function calls plays a pivotal role, effectively implementing the behavior of these actors.

In the generated code, the *for loops* symbolize the RV values of the actors, allowing for the precise execution of their behaviors. Moreover, the functions

include arguments that reference the ports of the actors, and data exchange is facilitated through FIFO buffers.

In scenarios where the *loop* pattern is applied and a partial unrolling is performed, feedback mechanisms are transformed into a *memcpy* function. This function is responsible for copying data from an output buffer to an input buffer, aligning the code with the specific behavior of the actors.

Example. Considering the graphs $G1$ and $G2$ Figure 2 and a 4 PE architecture. The schedules of the clusters are calculated using the APGAN algorithm and are as follows:

- $S_{SRV_{G1}} = 2(A)$
- $S_{URC_{G1}} = 2(C\ D)$
- $S_{h_{G2}} = 3(8A\ B8(C\ D))$

The corresponding code for these clusters is illustrated in Figure 2.

3.5. Graph transformation

The behavior of the hierarchical actor parent of the subgraph is then replaced by the newly created cluster code. Depending on whether the current search level is upper than the identified level and the identified actor were the candidate of *Loop* or *Sequential* patterns then GPDs are inserted between them creating pipeline stages.

Example. One schedule of the output graph of G1 illustrated in Figure 2 can be: $S_{G1} = s\ 4(SRV)\ B\ 4(URC)\ 4(SRV)\ B\ 4(URC)\ 4(SRV)\ B\ 4(URC)\ g$ A delay is inserted between two instances of h_{G2} creating four pipeline stages. One schedule of the output graph of G2 can be: $S_{G2} = s\ 4(h)\ g$

4. Experiments

4.1. Experimental setup

The proposed method is applied to three image processing applications: OpenVVC(1), Squeezenet(2), and Stereo(3) on a SPiDF description. The OpenVVC dataflow model has been introduced in [26], the author particularly explores parallelism between tiles in the VVC decoder. The Squeezenet application is a CNN for computer vision introduced in [27]. The experimented Stereo matching algorithm has been introduced in [28] and aims

at reconstructing the disparity maps with a pair of images. OpenVVC and Stereo SPiDF model presents 1 hierarchical level whose hierarchical actor is self-looped forbidding internal pipelining. Squeezenet SPiDF model presents 3 hierarchical levels and clustering opportunities on each of them. All of these dataflow models are available in Github [1].

The proposed method, noted CC for Current Clustering, is compared to the configuration without clustering, noted NC for No Clustering, and to the Best Clustering configuration of the previous SCAPE method, noted BC. Owing the fact that the previous SCAPE method provides a set of clustering configurations, the one that offers the best tradeoff in terms of the number of SrDAG actors, resource allocation process time, also called analysis time, and throughput speedup is chosen.

Table 1 compares graph complexity based on the size of SrDAG in terms of the number of Actors, noted A, and the number of FIFOs, noted f. Figure 4 illustrates analysis time, that is the time the tool takes to map schedule and generate code to an application, and throughput speedup. Figure 5 compares two critical performance metrics: the average data transfer time per thread, and the average synchronization time per thread. The average data transfer time per thread represents the time spent by the *memcpy* function to efficiently distribute tokens among several instances of actors. This time duration exhibits variability depending on the specific thread in which these functions are executed, and the metric presented here captures the average data transfer time per individual thread. Concurrently, the average synchronization time per thread reflects the delays introduced by employing Pthreads barriers to synchronize the various threads involved in the process. Since the barriers necessitate certain threads to await the completion of others, there can be discrepancies in the waiting times across threads, thus resulting in varying synchronization times.

Given that the performance criteria heavily depend on the architecture and that the method leverages architectural details in the graph transformation process, experiments were conducted across a range of architectures, specifically those with 1 to 8 homogeneous cores. To conduct these experiments, the proposed method was implemented within the PREESM [25] rapid prototyping framework, which is part of open-source projects. The experiments were carried out on a desktop computer featuring an 8-core Intel

[1] `https://github.com/preesm/preesm-apps`

i7-8665U processor and 31.2 GB of RAM.

		Number of PEs							
		1		2		4		8	
App.	**Met.**	A	f	A	f	A	f	A	f
(1)	NC	7262	39995	7262	39995	7262	39995	7262	39995
	BC	**1**	**0**	**12**	**14**	**24**	**34**	**99**	**159**
	CC	**1**	**0**	**12**	**14**	**24**	**34**	**99**	**159**
(2)	NC	5452	17262	5452	17262	5452	17262	5452	17262
	BC	**1**	**0**	1364	3589	1376	3916	1400	4197
	CC	**1**	**0**	**124**	**260**	**188**	**480**	**316**	**920**
(3)	NC	313	1069	313	1069	313	1069	313	1069
	BC	**1**	**0**	73	125	76	143	**89**	**313**
	CC	**1**	**0**	**53**	**85**	**61**	**113**	**89**	**313**

Table 1: Comparison of the number of actors A and the number of FIFO f of the SrDAG between the No Clustering (NC) and the Best Clustering (BC) configurations of the previous SCAPE method, as well as the clustering configuration of the proposed method (CC) on 3 applications: OpenVVC(1), Squeezenet(2), and Stereo(3) on various number of PEs.

4.2. Analysis time evaluation

The experimental results in Table 1 unveil the impact of graph complexity on the performance of the proposed method. The size of the SrDAG, as indicated by the number of actors, plays a critical role in influencing the efficiency of the mapping and scheduling process. The table illustrates a key trend: larger graphs, characterized by a higher number of actors, tend to induce more time-consuming computations during the mapping and scheduling phase, as the method explores various possibilities on the target architecture. In contrast, smaller graphs lead to a simplified process. Moreover, the experimental findings highlight that the complexity of the graph generated by the proposed approach is either lower or equivalent to the best outcome achieved by the second version of the SCAPE method. Furthermore, it significantly surpasses the method without clustering, affirming its efficacy in streamlining the dataflow graph transformation process. Specifically, for the OpenVVC dataflow model, the proposed method and the previous SCAPE method both provide the same configuration, the best. The previous SCAPE method has

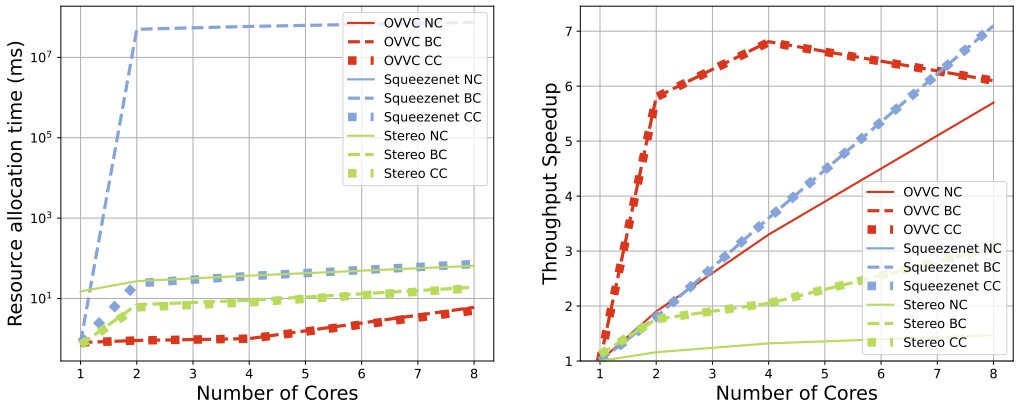

Figure 4: Comparison of analysis time and throughput speedup between the No Clustering (NC) and the Best Clustering (BC) configurations of the previous SCAPE method, as well as the clustering configuration of the proposed method (CC), across three applications and varying numbers of PEs. Squeezenet NC are not shown because they are too large.

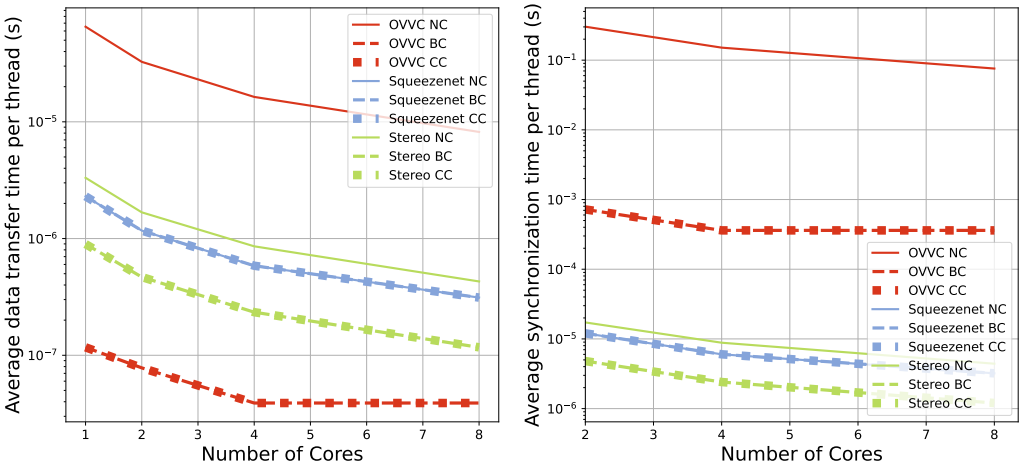

Figure 5: Comparison of average data transfer and synchronization times per thread between the No Clustering (NC) and the Best Clustering (BC) configurations of the previous SCAPE method, along with the clustering configuration of the proposed method, across three different applications and varying numbers of PEs

to evaluate all possible configurations to retrieve the best one, while the proposed method gives the best solution directly. The finding solution leads to a substantial reduction in the number of actors, from 7262 to 24, on a 4-core architecture. This configuration similarity arises particularly on dataflow graphs with a small number of hierarchical levels and few of them admit clustering opportunities. Specifically, in the case of the OpenVVC dataflow model with 1 hierarchical level and clustering opportunities localized on a single hierarchical level, both the proposed method and the previous clustering method achieve the same configuration post-clustering. This is highlighted in the Squeezenet application that has clustering opportunities on each hierarchical level the proposed method provides the least complex graphs better than all other configurations. Concerning the stereo application the method provides the least complex graphs on small architecture and then retrieves the same configuration as the previous SCAPE method. The method provides a graph complexity adapted to the number of PEs of the target, therefore the complexity of the resulting graph increases as the architecture increases until it reaches the original configuration when the degree of parallelism of the description of the origin becomes lower than the number of PEs and the pipeline gains reach the system limit.

The experimental results depicted on the right of Figure 4 compare the three use cases: in red OpenVVC, in blue Squeezenet, and in green stereo; on the three methods: in dotted line, No clustering, in dashed line, the previous SCAPE method, and in full line, the proposed method. Results concerning analysis time testify to the link between SrDAG complexity and analysis cost because the method provides a faster analysis if not equal to the previous method. The proposed method always provides on these curves an analysis time of a few seconds. The curves for Squeezenet and OpenVVC without clustering are not visible due to their high complexity, causing the analysis to reach the computational limits of the test computer after a full day of processing. This indicates that the computational capabilities of the tool have been pushed to their limits without reaching the code generation stage.

4.3. Throughput evaluation

This section investigates the performance of the method by analyzing the throughput speedup results displayed on the left side of Figure 4. These results demonstrate that the method outperforms the configuration without clustering, which is otherwise identical to the previous SCAPE configuration. Notably, the complexity of the OpenVVC and Squeezenet applications

is substantial, making direct output generation impossible without the aid of the method. In the case of OpenVVC, manual modifications have been applied to the configuration without clustering to enable performance comparison. However, such modifications are unfeasible for Squeezenet, resulting in the absence of a corresponding throughput curve for this application.

Stereo and OpenVVC, both containing cycles, benefit from the proposed method by leveraging the semi-unrolling feature. This approach brings a two-fold advantage. Firstly, it introduces pipeline stages, thereby increasing parallelism, resulting in improved speedup compared to the configuration without clustering. Secondly, as loop pipeline stages are created based on the greatest common divisor between the number of PEs and the number of repetitions of the original loop, the performance curve of the method exhibits a step-like pattern. The points where the curve reaches a ceiling correspond to the initiation of each new step in the performance improvement.

The equal performance in throughput obtained with the proposed method and the previous SCAPE method concerning Stereo and OpenVVC is due to the fact that the methods retrieve the same graph configuration. Concerning Squeezenet it is due to the fact that both configurations as similar parallelism setup which is detailed in Section4.4.

The reason why the method outperforms the OpenVVC dataflow model is a generic model independent of any target architecture. The developer's goal is to model the application once and deploy it on any available architecture, which is useful in the first phases of a project to quickly evaluate the potential of a solution. This generic model greatly limits any potential hardware-specific optimizations, which are automatically provided by the proposed method, thus achieving both the enhancement of rapid prototyping to quickly deploy an architecture-independent model of the application and the provision of hardware-specific optimizations.

4.4. Thread-level data transfer and synchronization evaluation

The results shown in Figure 5 reveal significant time savings in implementing parallelism through specific clustering. This implementation encompasses parallelism in the description, inter-thread data transfer, and synchronization processes, as detailed in [29].

Applications with a lot of communication between computations such as OpenVVC with 39995 FIFOs in Table 1 can suffer from the synchronization of dependent computations on several threads more than the data transfer time. Indeed the average data transfer time is negligible by a few nanoseconds

on CPU architectures compared to the average synchronization time of a few milliseconds.

There are two types of synchronization used in PREESM [25]: the synchronization of the beginning and end of the thread loops with the use of barriers and the synchronization between dependent computations with the use of *send* and *receive* functions on each thread between each dependent instance. The reduction of complexity thanks to clustering reduces the number of communications, from 39995 f to 14 f on 2 core architecture concerning OpenVVC, which reduces the number of synchronization and thus the average synchronization time per thread, from $302ms$ to $722us$. As the number of cores increases, the dependent computations and synchronization are distributed, reducing the gain obtained through clustering. Hence the clear acceleration in throughput on 2 cores, fades with the complexity of the architecture.

5. Conclusion

This paper presents a new automated method to reduce mapping and scheduling time while preserving and adding the parallelism of SPiDF graphs. The method consists in reducing the size of the graph by clustering actors reproducing particular patterns taking into account the graph context and the target architecture. Experimental results show that the method provides better throughput result thanks to adapting the implementation of parallelism to the target architecture. In addition, the method yields the most meaningful results on complex graphs to be deployed on complex architecture. Potential directions for future work include adapting the internal behavior of the clusters of actors to the target architecture.

References

[1] E. A. Lee, D. G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Transactions on Computers C-36 (1) (1987) 24–35. doi:10.1109/TC.1987.5009446.

[2] J. Piat, S. S. Bhattacharyya, M. Raulet, Interface-based hierarchy for synchronous data-flow graphs, 2009 IEEE Workshop on Signal Processing Systems (2009) 145–150doi:10.1109/SIPS.2009.5336240.

[3] F. Arrestier, K. Desnos, M. Pelcat, J. Heulot, E. Juarez, D. Menard, Delays and States in Dataflow Models of Computation, in: SAMOS XVIII, Pythagorion, Greece, 2018. doi:10.1145/3229631.3229645.
URL https://hal.science/hal-01850252

[4] E. Lee, S. Ha, Scheduling strategies for multiprocessor real-time dsp, in: 1989 IEEE Global Telecommunications Conference and Exhibition 'Communications Technology for the 1990s and Beyond', 1989, pp. 1279–1283 vol.2. doi:10.1109/GLOCOM.1989.64160.

[5] J. Pino, S. Bhattacharyya, E. Lee, A hierarchical multiprocessor scheduling system for dsp applications, in: Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers, Vol. 1, 1995, pp. 122–126 vol.1. doi:10.1109/ACSSC.1995.540525.

[6] O. Renaud, D. Gageot, K. Desnos, J.-F. Nezan, SCAPE: HW-Aware Clustering of Dataflow Actors for Tunable Scheduling Complexity, in: Design and Architecture for Signal and Image Processing, Springer Nature Switzerland, Cham, 2023, pp. 3–14. doi:10.1007/978-3-031-29970-4_1.

[7] O. Renaud, N. Haggui, K. Desnos, J.-F. Nezan, Automated clustering and pipelining of dataflow actors for controlled scheduling complexity, in: 2023 31st European Signal Processing Conference (EUSIPCO), 2023, pp. 1698–1702. doi:10.23919/EUSIPCO58844.2023.10290113.

[8] T. Parks, J. Pino, E. Lee, A comparison of synchronous and cyclo-static dataflow, in: Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers, Vol. 1, 1995, pp. 204–210 vol.1. doi:10.1109/ACSSC.1995.540541.

[9] E. Lee, Consistency in dataflow graphs, IEEE Transactions on Parallel and Distributed Systems 2 (2) (1991) 223–235. doi:10.1109/71.89067.

[10] X. Ye, X. Tan, M. Wu, Y. Feng, D. Wang, H. Zhang, S. Pei, D. Fan, An efficient dataflow accelerator for scientific applications, Future Generation Computer Systems 112 (2020) 580–588. doi:https://doi.org/10.1016/j.future.2020.03.023.
URL https://www.sciencedirect.com/science/article/pii/S0167739X19313986

[11] Y. Li, M. Wu, X. Ye, W. Li, R. Xue, D. Wang, H. Zhang, D. Fan, An efficient scheduling algorithm for dataflow architecture using loop-pipelining, Information Sciences 547 (2021) 1136–1153. doi:https://doi.org/10.1016/j.ins.2020.09.029.
URL https://www.sciencedirect.com/science/article/pii/S0020025520309397

[12] E. Lee, D. Messerschmitt, Pipeline interleaved programmable dsp's: Synchronous data flow programming, IEEE Transactions on Acoustics, Speech, and Signal Processing 35 (9) (1987) 1334–1345. doi:10.1109/TASSP.1987.1165275.

[13] A. Honorat, K. Desnos, M. Dardaillon, J.-F. Nezan, A Fast Heuristic to Pipeline SDF Graphs, in: Embedded Computer Systems: Architectures, Modeling, and Simulation, Embedded Computer Systems: Architectures, Modeling, and Simulation 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings, Pythagorion, Samos Island, Greece, 2020, pp. 139–151. doi:10.1007/978-3-030-60939-9_10.
URL https://hal.science/hal-02993338

[14] Z. Zhou, W. Plishker, S. S. Bhattacharyya, K. Desnos, M. Pelcat, J.-F. Nezan, Scheduling of parallelized synchronous dataflow actors for multicore signal processing, J. Signal Process. Syst. 83 (3) (2016) 309–328. doi:10.1007/s11265-014-0956-2.
URL https://doi.org/10.1007/s11265-014-0956-2

[15] G. Sih, E. Lee, Scheduling to account for interprocessor communication within interconnection-constrained processor networks., in: Proceedings of the 1990 International Conference on Parallel Processing, Urbana-Champaign, IL, USA, August 1990. Volume 1: Architecture, Pennsylvania State University Press, 1990, pp. 9–16.

[16] V. Sarkar, Partitioning and scheduling parallel programs for execution on multiprocessors (1 1987).
URL https://www.osti.gov/biblio/7043298

[17] M. Geilen, Reduction techniques for synchronous dataflow graphs, Design Automation Conference (Jul 2009). doi:10.1145/1629911.1630146.

[18] H. I. Ali, S. Stuijk, B. Akesson, L. M. Pinho, Reducing the complexity of dataflow graphs using slack-based merging, ACM Trans. Des. Autom. Electron. Syst. 22 (2) (jan 2017). doi:10.1145/2956232.
URL https://doi.org/10.1145/2956232

[19] S. S. Bhattacharyya, E. A. Lee, Scheduling synchronous dataflow graphs for efficient looping, Journal of VLSI signal processing systems for signal, image and video technology 6 (1993) 271–288. doi:10.1007/BF01608539.

[20] S. Bhattacharyya, P. Murthy, E. Lee, Apgan and rpmc: Complementary heuristics for translating dsp block diagrams into efficient software implementations, Design Automation for Embedded Systems 2 (09 1997). doi:10.1023/A:1008806425898.

[21] J. Falk, J. Keinert, C. Haubelt, J. Teich, S. S. Bhattacharyya, A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications, in: Proceedings of the 8th ACM International Conference on Embedded Software, EMSOFT '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 189–198. doi:10.1145/1450058.1450084.
URL https://doi.org/10.1145/1450058.1450084

[22] R. Lublinerman, C. Szegedy, S. Tripakis, Modular code generation from synchronous block diagrams: Modularity vs. code size, SIGPLAN Not. 44 (1) (2009) 78–89. doi:10.1145/1594834.1480893.
URL https://doi.org/10.1145/1594834.1480893

[23] R. Lublinerman, S. Tripakis, Modularity vs. reusability: Code generation from synchronous block diagrams, in: Proceedings of the Conference on Design, Automation and Test in Europe, DATE '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 1504–1509. doi:10.1145/1403375.1403736.
URL https://doi.org/10.1145/1403375.1403736

[24] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, E. A. Lee, Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs, ACM Trans. Embed. Comput. Syst. 12 (3) (apr 2013). doi:10.1145/2442116.2442133.
URL https://doi.org/10.1145/2442116.2442133

[25] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, S. Aridhi, Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming, in: 2014 6th european embedded design in education and research conference (EDERC), IEEE, 2014, pp. 36–40.

[26] N. Haggui, W. Hamidouche, F. Belghith, N. Masmoudi, J.-F. Nezan, OpenVVC Decoder Parameterized and Interfaced Synchronous Dataflow (PiSDF) Model: Tile Based Parallelism, Journal of Signal Processing Systems (2022). doi:10.1007/s11265-022-01819-7.
URL https://hal.science/hal-03884560

[27] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, K. Keutzer, Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size (2016). arXiv:1602.07360.

[28] J. Zhang, J.-F. Nezan, M. Pelcat, J.-G. Cousin, Real-time gpu-based local stereo matching method, 2013 Conference on Design and Architectures for Signal and Image Processing (2013) 209–214.
URL https://api.semanticscholar.org/CorpusID:9166047

[29] M. Pelcat, S. Aridhi, J. Piat, J. F. Nezan, Physical Layer Multi-Core Prototyping: A Dataflow-Based Approach for LTE eNodeB, Lecture Notes in Electrical Engineering, Springer-Verlag London, 2012. doi:10.1007/978-1-4471-4210-2.
URL https://hal.science/hal-00739957