Automated Clustering and Pipelining of Dataflow Actors for Controlled Scheduling Complexity

Ophélie Renaud, Naouel Haggui, Karol Desnos, Jean-François Nezan

Automated Clustering and Pipelining of Dataflow Actors for Controlled Scheduling Complexity

Ophélie Renaud*, Naouel Haggui*†, Karol Desnos*, Jean-François Nezan*
**Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164,* 35000 Rennes, France. first.last@insa-rennes.fr
†*LETI,* Sfax, Tunisia. nawel.hagui@enis.tn

***Abstract*—Dataflow models are efficient programming paradigms for expressing the parallelism of an application. Dataflow-based resource allocation methods on multicore architectures usually rely on complex graph transformations to explicit the application parallelism which can result in complex graphs for embarrassingly parallel applications. This paper presents an automated method that efficiently manages pre-scheduling graph complexity, pipelines sequential parts, and optimally adapts the dataflow model to the target architecture, striking a superior balance between application complexity and performance than existing methods. Our method surpasses state-of-the-art techniques, achieving up to 1.8 times higher throughputs in experiments. It also significantly reduces analysis time to seconds compared to the original PREESM method, which could take several days for fine-grained applications.**

***Index Terms*—Dataflow, PREESM, clustering, pipeline**

I. Introduction

A major issue in parallel programming is the granularity of the application description. Data transfers and computation times differ depending on the target architecture. Architectures with low data transfer costs like FPGAs are better suited for a fine-grain description [10]. Moreover, the granularity is also related to the number of Processing Elements (PEs) of the architecture, requiring adapting the number of actors and their size to the hardware platform.

Solutions such as OpenMP, OpenCL, and CUDA facilitate this adaptation but still require time-consuming hardware-specific programming. The dataflow parallel programming paradigm [6] investigated in this paper naturally expresses data, task, and pipeline parallelisms. Task parallelism consists in executing different computations on several PEs. Data parallelism consists in executing the same computations on different data simultaneously on several PEs. Pipeline parallelism breaks the data dependencies and divides the computations into several stages, executed simultaneously on several PEs. A dataflow model is a collection of computational units called actors exchanging data via First In First Out queues (FIFOs). Dataflow Model of Computation (MoC) is architecture independent which means a single description is portable on any type of architecture. In practice, the granularity of the dataflow models is often adapted to the architecture. Decreasing the granularity is performed by increasing the number of actors and decreasing the size of the FIFOs.

The aim of this paper is to introduce a clustering method that is based on dataflow, which addresses the two-fold challenge of decreasing graph complexity while maintaining performance on the target architecture. The complexity of an application is a critical issue in parallel programming. Excessive complexity hampers compilation due to the NP-complete nature of resource allocation. Conversely, an overly fine-grained application causes excessive data transfer, while a simple one underutilizes parallelism. Dataflow-based clustering methods have proven to be effective in automatically adapting the granularity of applications to the number of PEs of the target architecture. Whereas previous work on clustering [9] deals with task and data parallelism, this paper adds the possibility of automatically pipelining the application execution. A primary advantage of dataflow-based clustering is also to accelerate the resource allocation process. The paper focuses on the static resource allocation done at compile time. Resource allocation involves two steps: mapping and scheduling [7]. Mapping consists in distributing actors on the target PEs, and scheduling consists in ordering the execution of these actors on the PEs. Clustering simplifies the resource allocation by reducing the number of actors but this must be done in such a way as to maintain the execution parallelism. In this paper, the impact of the clustering in terms of latency, throughput, and memory footprint are also evaluated.

The rest of this paper is organized as follows: Section II presents dataflow MoCs, the classic static scheduling method, and the state-of-the-art clustering heuristics. Section III describes the proposed method. Section IV outlines the experimental evaluation of the clustering method about resource allocation process time, latency, throughput, and memory footprint. Finally, Section V concludes this paper.

II. Background

A. SDF-based dataflow MoC

Due to their high predictability that allows the development of real-time applications, Synchronous Dataflow (SDF) models, illustrated in Figure 1, are extensively employed for modeling signal and image processing applications. SDF graphs are characterized by a fixed rate of tokens consumed and produced by actors at each of their executions. FIFOs can have an initial state depicted as initial tokens, also called delays. The delays are used to create a shift of the tokens in the FIFOs and allow feedback loops in the graph or feed-forward graph cuts for example.

The State-Aware Parameterized and Interfaced DataFlow (SPiDF) [1] is an extension of the SDF model. This paper

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

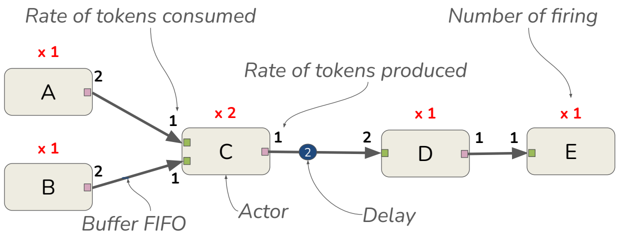


Fig. 1: SDF MoC semantics

emphasizes two key features of the SPiDF model: the hierarchy and graph persistence. The hierarchy allows specifying the internal behavior of an actor by a subgraph instead of a C. The SPiDF hierarchy feature defines interfaces to isolate the hierarchy levels and allow the composability of the model. The model introduces the persistence of a graph state with three types of delays: Local Delay (LD), Locally Persistent Delay (LPD), Globally Persistent Delay (GPD). The paper focuses on LDs to unroll cycles. The LD delayed data tokens are preserved within the scope of a unique graph iteration. In this case, an LD is not only connected to a fixed initial token number and a FIFO but it is also linked to two optional data connections. The input data connection of the delay is linked to a Setter actor in charge of initializing the data tokens of the concerned FIFO. The output data connection of the LD is linked to a Getter actor receiving the last held values by the LD. GPDs are used for pipelining because they persist across graph iterations until their next use. The advantage of SPiDF is that it allows a finer description of the application.

B. *Standard flattening method*

In order to generate a multicore implementation, a scheduling process must be performed first. The typical static scheduling process involves four primary tasks, namely flattening, Single rate Directed Acyclic Graphs (SrDAG) transformation, mapping, and scheduling.

The *flattening* task replaces all hierarchical actors with their contents, which implies bringing actors of different granularities to the top level. 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. The transformation duplicates actors by the minimal number of firings of each actor to return the graph back to its original state given by the calculation of the Repetition Vector (RV) **q** and exposes data dependencies. After the *SrDAG transformation*, each actor in the SrDAG is individually *mapped* and *scheduled*.

The code generated by the classic flattening approach in the Parallel and Real-time Embedded Executives Scheduling Method (PREESM) open source framework takes the form of a specific C file for each target PE. Every file contains first the application initialization section including the allocation of buffers, actors, and FIFOs and some initialization such as the initialization of globally persistent delays. The second part of these files is a loop containing the firing of actors scheduled at compile time.

The problem that arises is that the static scheduling time increases with the application complexity and the number of PEs in an architecture [7]. Since the mapping options are restricted to the number of PEs on the target, it is not necessary to depict the application with more parallelism than the target.

C. *SDF actor clustering methods*

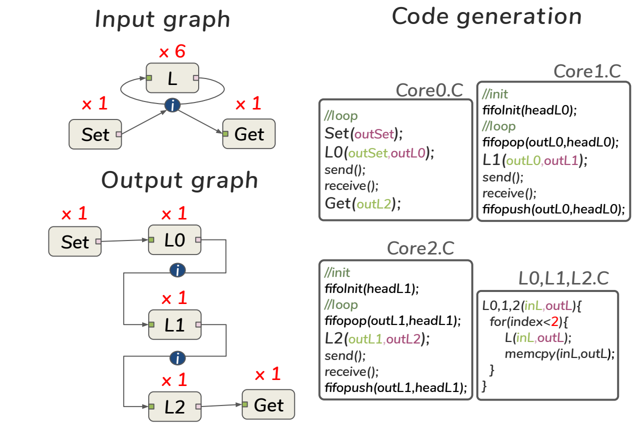
The clustering of SDF actors is an efficient method for reducing graph complexity. Since grouping several actors into a single equivalent hierarchical actor may change the behavior of the application or even create deadlocks, clustering rules have been introduced in [8] and illustrated on four clustering techniques. The first one is a manual and tedious method that enables the user to select improper groups of actors introducing deadlocks. The second one consists of clustering SDF subgraphs as long as possible. The third one is the Unique Repetition Count (URC) clustering technique. The method consists in creating an SDF subgraph of at least two sequential actors with identical RV **q** and no internal state. Considering the graph G shown in Figure 1, an URC candidate is the cluster of the actors D and E. The last one concerns dynamic resource allocation. Another clustering technique depicted in [2], the Pairwise Grouping of Adjacent Nodes (PGAN) method, consists in coupling two actors together and provides a wide choice of possible configurations that are tedious to evaluate. Its extension, the Pairwise Grouping of Adjacent Nodes for Acyclic graph (APGAN) clustering technique introduce in [3] shows that first clustering couples that form a cluster with the highest RV lead to a minimum memory requirement schedule and minimize the possible configuration. These methods focus solely on addressing one of the two challenges presented in the paper: reducing graph complexity. However, they do not take into account the second challenge, which involves considering the parallelism of the target architecture.

The Scaling up of Clusters of Actors on Processing Element (SCAPE) method [9] 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 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 and Single Repetition Vector (SRV) 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. Thus, the method offers a set of clustering configurations of decreasing granularity and parallelism. The proposed method is an extension of SCAPE that offer better parallelism management by unrolling cycles and integrating pipeline.

III. The proposed Method

The proposed clustering method will reduce graph complexity and happen just before the flattening process. Like SCAPE, the method takes as a task input a parameter n_c that corresponds to the number of levels to be coarsely clustered.

When the n_c levels are clustered, the method is about identifying four patterns of actors explained in Section III-A on the bottom level of the clustered graph, replacing them with a hierarchical actor that contains the identified actor explained in Section III-B, computing the scheduling of the newly created subgraph, generating the associated code explained in Section III-C, and replacing the behavior of the hierarchical actor with this code explained in Section III-D.



Core0.C

```
//loop
Set(outSet);
L0(outSet,outL0);
send();
receive();
Get(outL2);
```

Core1.C

```
//init
fifoInit(headL0);
//loop
fifopop(outL0,headL0);
L1(outL0,outL1);
send();
receive();
fifopush(outL0,headL0);
```

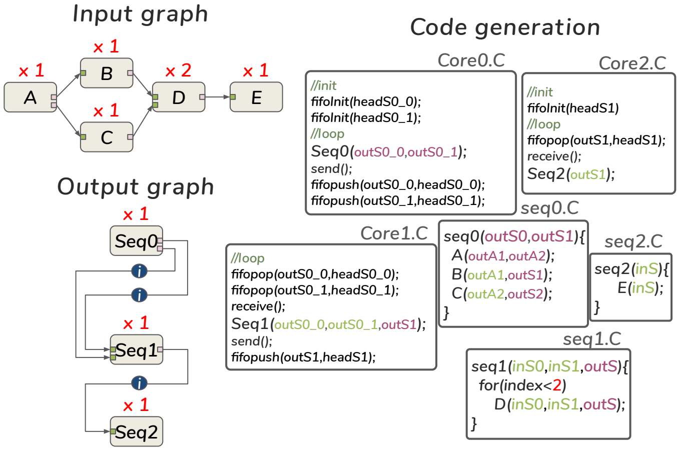
Core2.C

```
//init
fifoInit(headL1);
//loop
fifopop(outL1,headL1);
L2(outL1,outL2);
send();
receive();
fifopush(outL1,headL1);
```

L0,L1,L2.C

```
L0,1,2(inL,outL){
  for(index<2){
    L(inL,outL);
    memcpy(inL,outL);
  }
}
```

Fig. 2: Illustration of the clustering **Loop** pattern on a 3 PEs architecture



Core0.C

```
//init
fifoInit(headS0_0);
fifoInit(headS0_1);
//loop
Seq0(outS0_0,outS0_1);
send();
fifopush(outS0_0,headS0_0);
fifopush(outS0_1,headS0_1);
```

Core2.C

```
//init
fifoInit(headS1)
//loop
fifopop(outS1,headS1);
receive();
Seq2(outS1);
```

Core1.C

```
//loop
fifopop(outS0_0,headS0_0);
fifopop(outS0_1,headS0_1);
receive();
Seq1(outS0_0,outS0_1,outS1);
send();
fifopush(outS1,headS1);
```

seq0.C

```
seq0(outS0,outS1){
  A(outA1,outA2);
  B(outA1,outS1);
  C(outA2,outS2);
}
```

seq2.C

```
seq2(inS){
    E(inS);
}
```

seq1.C

```
seq1(inS0,inS1,outS){
  for(index<2)
    D(inS0,inS1,outS);
}
```

Fig. 3: Illustration of the clustering **Sequential** pattern on a 3 PEs architecture

A. *Identification of particular pattern*

The method considers the two patterns of SCAPE,URC and SRV II-C, and adds two other ones:

The **Loop** pattern allows the creation of parallelism on cyclic graphs. Illustrated in Figure 2, a cyclic part is a sequence of actors where the last one is connected to the first one by one or more FIFOs with a LD. Considering the input graph G where the input of actor L depends on its output but is initialized by actor set, the output of the 6th firing of L is stored by the get actor at each graph iteration. Given an architecture of 3 PEs. The identified actor is L.

The **Sequential** pattern allows the creation of parallelism on sequential graphs. Figure 3 is a sequential part of the graph or a part with a degree of parallelism lower than the number of PE. The method is about grouping actors in topological order so that the sum of the execution times of the actors contained in a group tends to be equally distributed. The number of groups must be equal to the number of PEs.Considering the input graph G with a degree of parallelism of 2 and architecture of 3 PEs. We assume that the sum of the executions of actors A, B, and C is equivalent to two executions of D and one execution of E on a single core. The method first identifies a first set composed of actors A, B, and C, the second set is composed of D, and the last one is composed of E.

B. *Subgraph transformation*

The second step of the method consists in generating subgraphs composed of the identified actors. The LDs are extracted from the subgraph and connected to the corresponding hierarchical actor. The operation consists of adding interfaces in the subgraph and linking the corresponding ports on the hierarchical actor to the delay(s). Extracting the delay(s) is necessary to compute the internal execution order of the subgraph via the APGAN method.

In the case of **Loop** pattern, the transformation consists of duplicating n times the hierarchical actor, where n_{loop} is the greatest common divisor of the RVs of the actors of the subgraph C flattened just above the number of PE, fixing its RV $\mathbf{q}$ to 1. The RV of the content is also scaled such as it's the result of the division.

$$n_{loop} = gcd(\mathbf{q}(a \in C)) \,|\, n_{loop} \geq n_{PE} \tag{1}$$

To preserve the consistency of the graph the FIFO buffers subsequently connected to the actors from the **Loop** are duplicated, distributed, or gathered on the different **Loop** instances. Considering the input graph G shown in Figure 2 where identified actor L has a RV $\mathbf{q}$ = 6 and an architecture of 3 PEs. Thus, $gcd(6, 3) = 2$ the method duplicates the subgraph that contains 2 instances of L 3 times. Thus the SrDAG transformation of this graph which would have resulted in $6 + 2 = 8$ actors is presently 3 actors.

In the case of **Sequential** pattern, the SrDAG transformation of graph G shown in Figure 3 which would have resulted in 6 actors is presently 3 actors.

C. *Cluster code generation*

The third step is to generate a code for the cluster. The SCAPE approach takes the form of C files where is defined a function that contains the scheduled firing of actors of the subgraph. The schedule is computed with the APGAN algorithm. The firing of actors is translated into function calls implementing the behavior of the actors. The functions contain arguments referring to the ports of the actors and exchange data via FIFO buffers.

The particularity of the code resulting from the subgraph that contains looped actors lies in the copy of delayed output(s) on delayed input(s) using *memcpy* function. Considering the input graph G shown in Figure 2, The Set function initializes a buffer on its output argument, The first looped function L_0 copies it on the input of the first L function call. Then L output buffer is copied on the L input buffer and the L function call output buffer is copied on the looped function output buffer.

D. Graph transformation

The last step is to replace the behavior of the hierarchical actor with the code generated beforehand. Then the method integrates pipelining by adding a GPD between the **Loop** and **Sequential** clustering stages. [5] describe how to create a pipeline on an SDF graph.

The clustered transformed graph is then employed by the rest of the static scheduling method able to generate code. PREESM translates GPD by a *fifoInit* function in the initialization part of the PE C File. This function is used to reset a global buffer at the start of the application. In the second part of the PE C File, the loop part, a *fifopop* precedes firing of the **Loop** and **Sequential** clustering pipeline stages loading the previously initialized GPD. The stages are succeeded by a *fifopush* function that stores the last delayed tokens of the actor for the next use. Thus, at each graph iteration, the first firing of clustering pipeline stages receives the delayed FIFO from the previous iteration.

Considering the input graph G shown in Figure 2 the method transmits an output graph to the rest of the static scheduling process composed of 3 pipeline stages, the first one containing the setter actor and the last one the getter actor. Stages are linked by GPDs. *fifoInit* function initialized global buffers $headL0$ and $headL1$ in the initial part of the program. *fifoPop* function copy the $headL0$ buffer in another buffer $outL0$ before its reading by $L1$ function call. *fifoPush* function copy the $outL0$ buffer in $headL0$ buffer after the $L1$ function process.

IV. Experiments

A. Experimental set up

The proposed method is applied to three image processing applications such as the OpenVVC dataflow model [4], SDP Evolutionary Pipeline (SEP), and Stereo on a SPiDF description summarize in Table I. Owing the fact that the proposed 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, latency, and throughput speedups, and memory footprint growth is chosen, noted BC for $Best\ Clustering$ results. This configuration is compared to the configuration without clustering, noted NC for $No\ Clustering$.

The above-mentioned applications have some limitations. Indeed, they are characterized by a large number of sequential parts that limit the mapping possibilities. Our approach solves this constraint by allowing the use of pipelining to provide a better mapping option. In addition, the proposed approach solves the problem of SrDAG explosion in some applications, especially for the OpenVVC and the SEP dataflow models. In fact, the user on both applications has to manually modify some instructions to avoid the explosion of the SrDAG graph. Without manual modification, the resource allocation process is $Computationally\ Prohibitive$, noted CP by the test desktop computer. The proposed approach makes the process fully automated.

The three exposed use cases were not relevant to the original SCAPE method because the resulting clustering configurations either present a poor degree of parallelism or are too complex and not schedulable hence the relevance of the extension.

Because the performance criteria depend on the architecture and the method exploits this information in the graph transformation, experiments have been conducted on architectures with 1 to 16 homogeneous cores. Latency and memory footprint are computed at runtime, whereas the size of the SrDAG and throughput are metrics simulated by the tool. The proposed method has been implemented in open-source projects into the PREESM rapid prototyping framework. The experiments are performed on a desktop computer with an 8-core Intel i7-8665U processor and 31,2 GB of RAM.

B. General results

Results in table I show the possible gains in terms of analysis time, latency, and throughput with the associated impact on memory footprint. The size of the SrDAG impacts the analysis time, small value leads to a fast process time. The method significantly reduces the size of the SrDAG, especially on large applications. The OpenVVC application described for a 24-tile bitstream without clustering has a size of 7262 actors which reaches the computational limit of the test PC, with the method this application on a 4-core architecture has a size of 24 actors and compiles in a few seconds. Pipelining is used to add parallelism to the application at the cost of increased latency. Actors in the first pipeline stage are executed and intermediate results are saved in the first iteration but the first output is only generated after a number of iterations equal to the number of pipeline stages. As the method matches the number of pipeline stages to the number of PEs, the latency increases as the architecture becomes more complex. That's why the stereo application on 16 cores has a speed down of 0.1 in terms of latency. In the case of the OpenVVC application, the FIFO sizing exploited by the clustering reduces the internal memory access time to the computation. Thus on a 4-core architecture, the latency speedup is 6.8 because the global computation time is reduced so that the time lost by the pipeline is insignificant. The latency loss is balanced by the gain in throughput. In sequential parts of an application without task nor data parallelisms, the pipelining provides additional parallelism for the remaining. Pipelining is only possible if the sequential computations are not part of an iterative loop with self-data dependencies. For applications exploiting the **Loop** pipelining, the gain is the most important when the number of PEs is a divisor of the number of loop iterations. For example, the loop in Stereo iterates 60 times so that the pipeline is efficient for 1, 2, and 4 PEs providing a speedup of the throughput very close to the optimal parallelism, 3.8 on 4 cores. Due to the fact that the SEP is modeled by looped hierarchical actors on its top level, pipelining of the inner levels is not allowed. Only the configuration that coarsely clusters the hierarchy up to the

Application	Output	Number of PEs									
		1		2		4		8		16	
		NC	BC	NC	BC	NC	BC	NC	BC	NC	BC
OpenVVC	SrDAG	7262	**1**	7262	**12**	7262	**24**	7262	**99**	7262	**99**
	Analysis Time	CP	**0.8s**	CP	**0.9s**	CP	**1s**	CP	**6s**	CP	**7s**
	Latency Speedup	1	1	1.9	**5.8**	3.3	**6.8**	5.7	**6.1**	**11.4**	10.7
	Throughput Speedup	1	1	1.9	1.9	**3.3**	2.9	5.7	**7.6**	11.4	**15.2**
	Memory Footprint Growth	1	1	1	1	1	1	1	1	1	1
SEP	SrDAG	4423	**1**	4423	**21**	4423	**27**	4423	**27**	4423	**27**
	Analysis Time	CP	**6s**	CP	**6s**	CP	**9s**	CP	**17s**	CP	**25s**
	Latency Speedup	1	1	**1**	0.6	**1.1**	0.4	**1.1**	0.4	**1.1**	0.4
	Throughput Speedup	1	1	1	**1.7**	1.1	**2.1**	1.1	**2.1**	1.1	**2.1**
	Memory Footprint Growth	1	**0.1**	**1.2**	1.9	**1.3**	3.9	**1.3**	3.9	**1.3**	3.9
Stereo	SrDAG	313	**1**	313	**56**	313	**61**	313	**89**	313	**109**
	Analysis Time	15s	**0.7s**	27s	**7s**	37s	**9s**	65s	**19s**	112s	**22s**
	Latency Speedup	1	1	**2**	0.6	**3.7**	0.4	**5.5**	0.2	**8.3**	0.1
	Throughput Speedup	1	1	2	2	3.7	**3.8**	5.5	**7.0**	8.3	**11.2**
	Memory Footprint Growth	1	**0.1**	1.5	**1.3**	1.7	**1.5**	**1.8**	2.3	**1.8**	3.9

TABLE I: Comparison of graph complexity, analysis time, latency, throughput, and memory footprint between the No Clustering (NC) and the Best Clustering (BC) configurations on 3 applications on various number of PEs, CP: Computationally Prohibitive

top level can be performed. The sequential description of the application justifies the use of the pipeline but quickly reaches a parallelization limit with a throughput speedup of up to 2.1. When the gains in throughput are small, the memory footprint is reduced compared to the configuration without clustering. Indeed pipelines add memory requirements since data must be stored at the end of each iteration. The application memory footprint is reduced by clustering which exploits FIFO sizing by default, that's why the Stereo application requires 10 times less memory on 1 core with clustering than without. However, if the gains in throughput are substantial, parallelism may increase memory requirements. For example, the SEP application has a throughput speedup of 1.8 times faster with clustering than without on 16 cores and a memory requirement 3 times higher. OpenVVC dataflow model cannot take advantage of the memory optimization technique provided by the tool since a significant portion of the memory is already allocated by an external library used in this project.

V. Conclusion

This paper presents a new clustering method to reduce the mapping and scheduling processing time on multicore architectures. The method consists in reducing the size of the graph by clustering actors detecting particular patterns taking into account the target architecture. The method allows the developer to choose the best granularity to optimize the latency, throughput, and memory requirements on a target architecture. This clustering method preserves the task and data parallelism of SDF graphs and automatically implements the pipelining of sequential parts of an application in order to improve its throughput. This automatic method enables developers to evaluate very quickly the increase in throughput, latency, and memory footprint associated with the use of pipelining. The automatic code generation enables to generate automatically the multicore pipelined implementation, reducing the programming time associated with this optimization. Fine grain dataflow models of applications can now be proposed without the concern of exploding the parallelism, knowing the clustering will adapt the implementation to the number of PEs of the architecture in order to reach the best acceleration optimizing throughput, latency, and memory requirements. A potential direction for future work includes the consideration of heterogeneous PEs in the selection of clusters of actors.

References

[1] F. Arrestier, K. Desnos, M. Pelcat, J. Heulot, E. Juarez, and D. Menard. Delays and States in Dataflow Models of Computation. In *SAMOS XVIII*, Pythagorion, Greece, July 2018.

[2] S. S. Bhattacharyya and E. A. Lee. Scheduling synchronous dataflow graphs for efficient looping. *Journal of VLSI signal processing systems for signal, image and video technology*, 6:271–288, 1993.

[3] S.S. Bhattacharyya, P. Murthy, and E. Lee. Apgan and rpmc: Complementary heuristics for translating dsp block diagrams into efficient software implementations. *Design Automation for Embedded Systems*, 2, 09 1997.

[4] N. Haggui, F. Belghith, W. Hamidouche, N. Masmoudi, and J.-F. Nezan. Multiple transform selection concept modeling and implementation using dynamic and parameterized dataflow graphs. *J. Signal Process. Syst.*, 94(7):709–720, jul 2022.

[5] A. Honorat, K. Desnos, M. Dardaillon, and 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, pages 139–151, Pythagorion, Samos Island, Greece, July 2020.

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

[7] E.A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time dsp. In *1989 IEEE Global Telecommunications Conference and Exhibition 'Communications Technology for the 1990s and Beyond'*, pages 1279–1283 vol.2, 1989.

[8] J.L. Pino, S.S. Bhattacharyya, and E.A. Lee. A hierarchical multiprocessor scheduling system for dsp applications. In *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 122–126 vol.1, 1995.

[9] O. Renaud, D. Gageot, K. Desnos, and J.-F. Nezan. SCAPE: HW-Aware Clustering of Dataflow Actors for Tunable Scheduling Complexity. In *Workshop on Design and Architectures for Signal and Image Processing (DASIP)*, 2023.

[10] S. Van der Vlugt, H. Alizadeh Ara, R. de Jong, M. Hendriks, R. Guerra Marin, M. Geilen, and D. Goswami. Modeling and analysis of fpga accelerators for real-time streaming video processing in the healthcare domain. *Journal of Signal Processing Systems*, 91(1):75–91, January 2019.