SCAPE: HW-Aware Clustering of Dataflow Actors for Tunable Scheduling Complexity

Ophélie Renaud[1(✉)], Dylan Gageot[2], Karol Desnos[1], and Jean-François Nezan[1]

[1] Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164, Rennes, France
{ophelie.renaud,karol.desnos,jean-francois.nezan}@insa-rennes.fr
[2] Yubik, Rennes, France
dgageot@yubik.io

Abstract. This paper introduces a fast method to generate high performance parallelized code from a dataflow specification of an application. Dataflow Models of Computation (MoCs) are efficient programming paradigms for expressing the parallelism of an application. Traditionally, mapping and scheduling methods for dataflow MoCs rely on complex graph's transformations to explicit their parallelism which can result in complex graph for embarrassingly parallel applications. For such applications, state-of-the-art mapping and scheduling techniques are prohibitively complex, while the exposed parallelism often exceeds the parallel processing capabilities of the target architecture. We propose SCAPE, an automated method to control the complexity of the pre-scheduling graph transformation by using information from the architecture and application models. By decreasing the complexity of the graph, the mapping scheduling task is accelerated at the potential expense of the produced schedule. Our method offers a limited and controlled decrease of the schedule quality while enabling mapping and scheduling execution time between 1 and 2 orders of magnitude faster than state-of-the-art techniques.

Keywords: Dataflow model · Hierarchy · Granularity · Clustering

1 Introduction

Digital signal processing technology emerged in the 1960 s s and has grown rapidly, becoming more complex over the years, particularly with the arrival of machine learning applications a decade ago. To meet the ever-increasing need for computing power and speed of execution of these applications, developers first sought to increase the frequency of individual Processing Elements (PEs) and then turned to heterogeneous multicore embedded systems.

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

Exploiting in an optimized way the maximum parallelism of such multicore target architectures is very challenging. The development of parallel code is tedious and is not adapted to manage hardware and software upgrades during the exploitation phase of the project. Tools such as Simulink [6] and Xilinx AI Engine Technology [1] are then investigated to automate the rapid deployment of new algorithms on the computing system. Both tools are based on the dataflow approach consisting in the modeling of the algorithms by a graph in which nodes, called actors, represent the calculations and directed arcs, called First In First Out queue (FIFO) buffers, represent the data, called tokens, exchanges between nodes.

The automated generation of parallel code from such dataflow models requires solving several NP-Complete problems, especially for resource allocation. Calculations are distributed on the PEs of the target architecture and will read and write, during the execution of an application, on FIFO buffers assigned to a range of memory addresses. The resource allocation choices can be made at compile time or at runtime. The allocation at runtime leads to performance overhead and unpredictable application behavior. For these reasons, this paper investigates methods that allocate resources at compile time, during software synthesis. The software synthesis process is responsible for translating a dataflow model into a prototype.

Classic resource allocation methods involve three phases: The *mapping* consists in distributing actors on the PEs of the target. The *Scheduling* consists in ordering the execution of actors on the PEs. The *Timing* associates to each actor a start time and an end time, useful to calculate the long time average throughput, also called latency, of the application. The time required for the mapping and scheduling process grows exponentially with the number of PEs and the number of nodes and edges of the dataflow graph [7].

The Scaling up of Clusters of Actors on Processing Element (SCAPE) method is introduced in this paper which is a hierarchy-based clustering method that transforms an application to match its degree of parallelism to the parallel computation capabilities of the targeted architecture. The method offers as many clustering configurations as there are hierarchy levels in the Synchronous Dataow (SDF) input graph which gives the user the possibility to choose the required granularity for a reduced software synthesis time.

Section 2 presents dataflow MoCs, the traditional mapping and scheduling method and the state-of-the-art clustering heuristics. Section 3 describes the proposed SCAPE method and the backbone of the resulting code. Section 4 outlines experimental results on several granularity clustering configurations showing a tradeoff between design space exploration time and produced schedule latency. Finally, Section 5 concludes this paper.

2 Context and Related Work

2.1 SDF Based Dataow MoCs

The most studied dataflow MoC is the SDF [8] illustrated in the Fig. 1, in which the integer numbers on the input and output ports of actors are the rate of tokens respectively consumed and produced by actors at each execution of the actors.

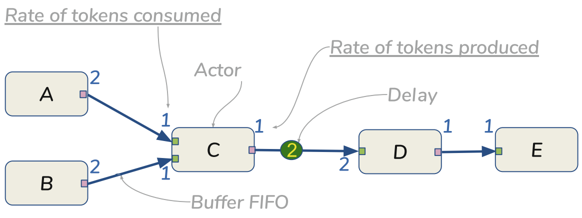


Fig. 1. SDF MoC semantics

An SDF graph is usually transformed into an equivalent Single rate Directed Acyclic Graphs (SrDAG) [12] to map and compute a periodic deadlock-free schedule iterated infinitely. A repetition of a periodic schedule is called a graph iteration. The single rate transformation consists in duplicating the actors by the number of firings specified by the schedule, and in adding special actors to distribute or gather tokens, so that the production and the consumption rates on each FIFO are equals. Then, DAG transformation consists in breaking cyclic data-paths. To be consistent, all cyclic data paths must contain at least one initial data token also called delays. Therefore, breaking a cycles means replacing FIFOs with delays by a pair of special actors which backup delayed tokens and read the backed up tokens.

An extension of the SDF model is the Parameterized and Interfaced SDF (PiSDF) model [4]. In this paper, the feature of interest of the PiSDF model is its support for hierarchy. The hierarchy feature allows the internal behavior of actors to be specified by a subgraph instead of C code. The PiSDF MoC defines interfaces of a subgraph as input and output data ports of the parent hierarchical actor. Interfaces allow the transmission of tokens between hierarchical levels. The hierarchy is used to represent different levels of granularity of the computations that compose an application, the lower levels of hierarchy being the finer granularity.

These SDF based models have two main advantages justifying their interest. The first one is to express the three types of parallelism [13], two of which are used in this paper: task and data parallelism (pipeline being the third).

– Task parallelism is expressed by two actors belonging to parallel data-paths like actors A and B in the Fig. 1. As there is no data-path between these actors, they can be fired at the same time.
– Data parallelism is expressed when several firings of a single actor are independent from each other. If enough data tokens are present in the input FIFOs, then several firings can be executed concurrently. An example is the actor C in the Fig. 1 which can be executed 2 times when A and B are executed 1 time.

The second advantage is that the model is independent of the target architecture. An application is represented once and executed on all types of architecture (single cores, multicores with shared or distributed memories, FPGA, etc.).

2.2 Classic Flattening Method

This paper focuses on tools that have chosen to allocate resources at compile time, also called static allocation. Software synthesis can be modeled by a workflow. The typical static scheduling workflow is composed of four main tasks: flattening, SrDAG transformation, mapping and scheduling, then code generation.

The *flattening* task of the workflow consists in putting all the actors of a graph at the same level which means all hierarchical actors are replaced by their subgraph. To keep the functionality of the application in Fig. 2, token rates consumed and produced in the initial subgraph are scaled up on upper levels. Tools usually flatten the whole graph to execute the rest of the process which brings the finer level of granularity to the top-level graph.

The *SrDAG transformation* task is used to reveal parallelism on flattened graph Fig. 2. It highlights 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) $\mathbf{q}$. Here, actor B is fired 4 times per graph iteration, so $\mathbf{q}(B) = 4$. It also emphasizes the interdependencies between the actors, which is useful to calculate the execution order of the actors, allowing them to be iterated infinitely without generating a deadlock. In the figure, the named nodes are the actor instances and the unnamed nodes are the special actors responsible for distributing or gathering the data tokens.

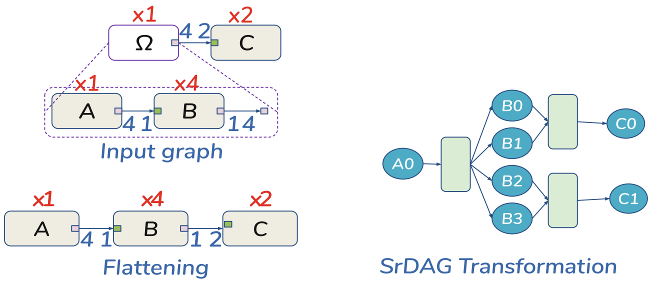


Fig. 2. Classic flattening process: 3 SDF actors turn into 10 SrDAG actors

The excessive complexity of the SrDAG increases the mapping opportunities which results in a better distribution of the computations on the different PEs and reduces the latency of the application on the target. Since the mapping opportunities are limited to the number of PEs in the architecture, exposing more parallelism than the number of PEs is unnecessary and time consuming. [10]. This is why reducing the exposed parallelism to the number of PE will most likely be sufficient to fully exploit the architecture parallelism, while being much simpler to map and schedule.

Example 1. Considering a machine learning application: *Squeezenet* neural network whose SDF model is composed of 70 actors, its SrDAG transformation results in 5452 actors. Mapping this application on an architecture composed of 8 PEs with a greedy algorithms requires considering and evaluating 8 mapping choices. Here the degree of parallelism is up to 1000 which is an needlessly fine granularity.

2.3 Cluster of SDF Actors

A way to reduce the complexity of mapping and scheduling algorithms is to reduce the number of actors to map in the srDAG, without altering behavior of the application. This reduction can be achieved using clustering techniques, wich transform the input graph by grouping actors with a particular behavior. Since grouping two or more actors into a single equivalent hierarchical actor may change the behavior of the application, or even create deadlocks, clustering rules have been introduced in [10]. These rules are illustrated in Fig. 3 where SDF graphs are represented with rectangular actors and the corresponding precedence graphs with round actors. A cluster must respect the execution order of the actors defined by the precedence rules (a), the initial tokens must be considered (b) and there must be no simple path from a node of the precedence graph to another one that contains more that one arc(c). A simple path is the one which does not visit any node along the path more than once.

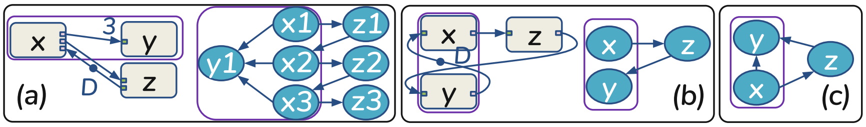


Fig. 3. (a) illustrate the violation of the first precedence shift condition, (b) illustrate the violation of the hidden delay condition, and (c) illustrate the violation of the cycle introduction condition

A method to cluster SDF actors is presented in [2] which introduced the Pairwise Grouping of Adjacent Nodes for Acyclic graph (APGAN) algorithm. Considering an acyclic SDF graph $G = \langle A, F\rangle$ where A is a set of stateless actors and F is a set of FIFO, the algorithm can be summarized as follows: A cluster hierarchy is constructed by clustering exactly two adjacent vertices at each step. At each clusterization step, the chosen pair of adjacent actors have the maximum repetition count value ρ see Definition definition 1, associated to their inter-connected edge.

Definition 1. *If Z is a subset of actors in a connected, consistent SDF graph:* $\rho(Z) \equiv \gcd(\{\boldsymbol{q}(A) \mid A \in Z\})$

APGAN candidates should respect the clustering rules that can be verified by applying a reachability matrix [3]. Then repeat the process until the end of the opportunities. Execution of hierarchical actor resulting from a clustering operation is assumed to be atomic and is thus mapped and scheduled as a whole on a single core. In order to execute such cluster actor, it is thus necessary to compute a sequential schedule of all actor firings that belong to the cluster. APGAN algorithm also provides special clustering schedules that are nested looped schedules whose specificity is to make sequential the behavior of a cluster. APGAN clustering technique focuses on single-core optimization.

Four clustering techniques are presented in [9]. The first one empowers the user to select improper groups of actors, manual methods are tedious and may introduce deadlocks. The second one consists in clustering SDF subgraphs as long as possible. The third one is the Unique Repetition Count (URC) clustering technique developed in Sect. 3.1. The last one is an adaptation of Sarkar's multiprocessor DAG scheduling heuristic [11] based on macro dataflow model in which the program is partitioned into tasks at compile time and the tasks are scheduled on processors at run time. All of these methods focus on the efficient reduction of the complexity of a graph without considering parallelism. The method introduced in this paper involves automatically generated architecture-adaptive parallel cluster instances.

3 SCAPE Method

The objective of the SCAPE method is to apply graph transformation to the SDF graph of an application, prior to its mapping and scheduling. To do so, clustering of actors within the input hierarchical SDF graph aims at reducing the complexity of the derived SrDAG used during mapping and scheduling. The SCAPE method aims at preserving the parallelism of the application so that it matches the parallel computing capabilities of the target architecture.

3.1 Design Space Exploration Optimisation

The SCAPE method is composed of three steps: configuration of the granularity, identification of particular patterns that will be the subject of clustering, and scaling up of the last clusters on the target architecture.

Configuration of the Granularity. The SCAPE method takes as input the PiSDF graph of n hierarchy levels that models the application and an integer value corresponding to the number n_c of hierarchy levels that the user wants to group coarsely. The output of the new method is a transformed graph with the RV $\mathbf{q}$ associated with the actors located in the subgraphs on $n - n_c$ level reduced to the number of PE that compose the architecture. A graph on n levels will have $n+ 2$ possible configurations of $n - n_c$ levels.

- Level 0 configuration: it is the state-of-the-art configuration where the entire graph is flattened before producing the SrDAG for scheduling.
- level $n+ 1$ clustering configuration: it is grouping the entire graph into a single actor, thus resulting in a mono-core schedule.

- level 1 clustering configuration: it corresponds to generate groups on the bottom levels and reduce the RV **q** associated with the actors located in the subgraphs on this level to the number of PE.
- level $l \mid l \in [2, n+1[$ clustering configuration: it corresponds to coarsely grouping bottom levels, generate groups on the just upper levels and reduce the RV **q** of actors on this level to the number of PE Fig. 4.

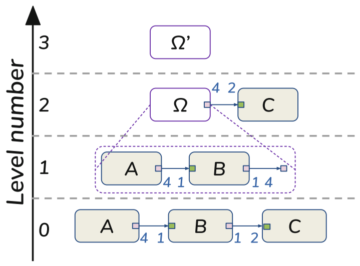


Fig. 4. Configuration based on clustering levels

Identification of Particular Patterns that will be the Subject of Clustering. The SCAPE method considers two patterns:

URC pattern: It's a sequence of at least two sequential actors with the same repetition count value ρ see Definition definition 1 without internal state. Such a pattern can be the object of a cluster if the consistency of the graph is preserved that can be verify by applying a reachability matrix.

Example 2. Considering the graph G shown in Fig. 5 which contains a sequence of actors B, C, D, and each FIFO connecting these actors presents the same repetition count value ρ, $\rho(A, B) = gcd(1, 4) = 1$ and $\rho(\Omega, C) = gcd(1, 2) = 1$. The method transforms the graph G by replacing this identified group with a hierarchical actor whose behavior is specified by a subgraph containing the identified group. This way the newly created hierarchical actor executes once per iteration and the elements it contains keep their initial execution number. Thus the SrDAG transformation of this piece of graph which would have resulted in $3 \times 8 = 24$ actors is presently 1 actor.

Single Repetition Vector (SRV) pattern: It's a single actor that does not belong to an URC candidate, with a RV **q** greater than or equal to the number of PEs of the target architecture.

Example 3. We consider the graph G shown in Fig. 5 which contains an actor E with a RV **q** equal to 16 and a target architecture which contains 4 cores. The method transforms the graph G by replacing this identified actor by a hierarchical actor whose behavior is specified by a subgraph containing the identified actor. This way the hierarchical actor executes once per iteration and the element it contains keeps its native execution number. Thus the SrDAG transformation of this piece of graph which would have resulted in 16 actors is presently 1 actor.

These two identified patterns will drastically reduce the size of the SrDAG and consequently make the application intrinsically sequential.

Scaling up of Cluster. The final step, called the *scaling*, consists of creating clusters of actors with a RV $\mathbf{q}$ matching the target architecture. The scaling is done on the hierarchical actors located on the subgraphs at the level n_c input integer value. Level 0 and n+2 clustering are not subject to scaling. According to [8] to preserve the consistency of a graph G, on each FIFO f the rates of consumed and produced tokens *cons* and *prod* and the RV $\mathbf{q}$ of the source and sink actors *src* and *snk* are linked by the equation:

$$\mathbf{q}(src(f)) \times prod(f) = \mathbf{q}(snk(f)) \times cons(f) \tag{1}$$

To calculate the scaling, the RV of the hierarchical actor $\mathbf{q}(h_a)$ shall be equal to the greatest common divisor of the RVs of the actors of the subgraph C flattened just above the number of PE n_{PE}.

$$\mathbf{q}(h_a) = gcd(\mathbf{q}(a \in C) \,|\, \mathbf{q}(h_a) \geq n_{PE} \tag{2}$$

In case the hierarchical actor contains a FIFO with a number of delay D, special care must be taken when *scaling* the actor. In particular, if the hierarchical actor is directly connected to a delayed FIFO or indirectly via a special actor or an interface connected to a delayed FIFO. If one condition holds true then the calculation of the scaling is indexed on the delay value such as the rates of consumed tokens on the delayed FIFO $cons(f_{h_{a_d}})$ has to be less than or equal to the delay value D.

$$\mathbf{q}(h_a) = gcd(\mathbf{q}(a \in C)) \,|\, cons(f_{h_{a_d}}) \leq D \tag{3}$$

In order to keep the consistency of the graph, the rates of tokens consumed and produced on the input and output ports by the hierarchical actor $in(h_a)$ and $out(h_a)$, $_f$ for final and $_i$ for the initial value, are scaled as follow:

$$\begin{cases} in(h_a)_f = in(h_a)_i \times \mathbf{q}(h_a)_i/\mathbf{q}(h_a)_f \\ out(h_a)_f = out(h_a)_i \times \mathbf{q}(h_a)_i/\mathbf{q}(h_a)_f \end{cases} \tag{4}$$

Thus the actors from the subgraph are executed $\mathbf{q}(a \in C)/\mathbf{q}(h_a)$ times.

Example 4. We consider the graph G shown in Fig. 5 and a target architecture with 4 PEs. As the RV $\mathbf{q}$ of the URC cluster is $\mathbf{q}(a \in URC) = 8$, then the scaling will be $gcd(8, 4) = 4$. Thus, the hierarchical actor URC executes 4 times and the subgraph elements twice per iteration. Respectively as the RV $\mathbf{q}$ of the SRV cluster is $\mathbf{q}(a \in SRV) = 16$, then the scaling will be $gcd(16, 4) = 4$. Thus, the hierarchical actor SRV executes 4 times and the subgraph elements 4 times per iteration. From the input graph, the classic "flattening" approach obtains a size of the SrDAG of 50 actors, the "SCAPE" approach obtains a size of the SrDAG of 10 actors. The method reduces both the number of actors related to calculations and the number of special actors related to data transfers on the different instances. The complexity of the graph has been divided by 5, which considerably reduces the mapping and scheduling time of the tool without compromising the parallelism.

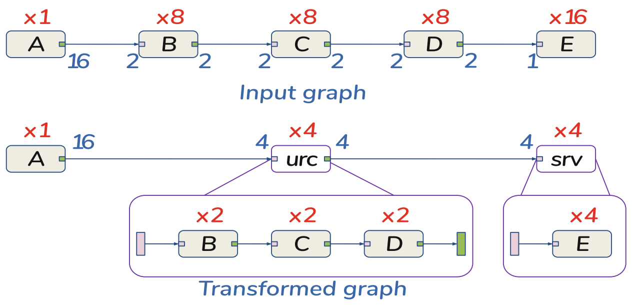
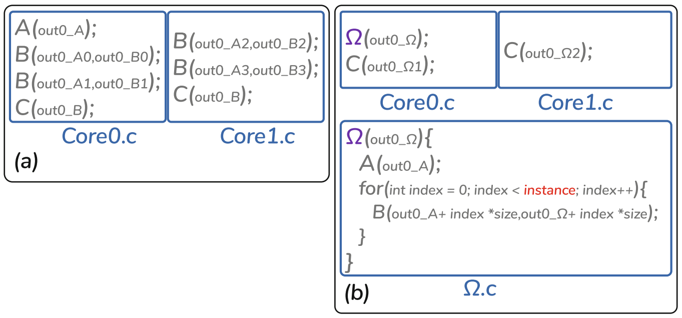


Fig. 5. SCAPE method

3.2 Code Generation

The code generated by the classic flattening approach in our tool [5] takes the form of a specific C file for each target PE. Every file contains first of all a part dedicated to the initialization of the application which includes the definition of the allocated buffers, actors and FIFOs initialization functions such as delay initialization. The second part of these files is a loop representing the thread containing the scheduled firing of actors. It is a function call implementing the behavior of the actor. Up until now, the tool did not support code generation for optimized actor groups. A plugin has been created to answer this new constraint. Thanks to this plugin, a cluster of actors is translated by nested function calls depending on whether the group contains other groups and the firing instances of the group elements are translated by "for" loops Fig. 6.



Core0.c:
```
A(out0_A);
B(out0_A0,out0_B0);
B(out0_A1,out0_B1);
C(out0_B);
```

Core1.c:
```
B(out0_A2,out0_B2);
B(out0_A3,out0_B3);
C(out0_B);
```

Core0.c:
```
Ω(out0_Ω);
C(out0_Ω1);
```

Core1.c:
```
C(out0_Ω2);
```

Ω.c:
```
Ω(out0_Ω){
  A(out0_A);
  for(int index = 0; index < instance; index++){
    B(out0_A+ index *size,out0_Ω+ index *size);
  }
}
```

Fig. 6. Considering the graph from Fig. 4 and a two-core architecture (a) illustrates the code generation on clustering configuration with level 0, and (b) illustrates the code generation on clustering configuration on 2 levels

4 Experiments

The purpose of this section is to show that the proposed method offers a trade-off between reducing mapping and scheduling time, also called analysis time, while preserving the latency of applications in comparison to the classic flattening method. The proposed method has been implemented in open source projects into Parallel and Real-time Embedded Executives Scheduling Method (PREESM) rapid prototyping framework. Compared to other mapping and scheduling frameworks, the absolute analysis time may seem high for DAGs with a few thousand actors. This time is due to the language used, Java, and heavyweight implementation choices made by PREESM. Nevertheless, the comparison of the evolution of analysis times, which relates to their complexity, would remain valid with faster implementations of both the state-of-the-art corresponding to "Level 0" and the proposed technique. The experiments are performed on a desktop computer with an 8-core Intel i7-8665U processor and 31,2 GB of RAM.

4.1 Experimental Setup

Figure 7 presents the "analysis" and latency measured for the stereo application. The application has 2 levels of hierarchy, so there are 4 possible clustering configurations as explained in the section Sect. 3.1. Three image processing application use-cases such as *Stereo, Stabilization and Squeezenet* were used to conduct the experiments on architectures with a 1, 2, 4, 8 or 16 of homogeneous cores summarized in the Table 1. These models were chosen because they do not contain too many delays, which impends the scaling opportunities of the SCAPE method. These applications have between 2 and 3 levels of hierarchy. For each number of cores, only the result giving the best latency was kept, among all levels of the SCAPE method.

4.2 Analysis Time Evaluation

The experimental results depicted in Fig. 7 compare in red the state-of-the-art configuration and the different shades of blue for the different levels of clustering configuration up to 0. The analysis time curves are shown on the left side of Fig. 7. The curves representing the clustering configuration on different levels are between two extremes. The highest curve named "Level 0" represents the more complex graph with a time that increases with the number of cores due to the time that the mapping and scheduling algorithm takes to map, schedule and allocate memory to each of the elements of the SrDAG. The lowest curve: the fully cluster configuration remains constant and fast whatever the number of cores but no parallelism.

4.3 Latency Evaluation

The latency curves are shown on the right side of figure Fig. 7. There are still two extreme curves: the "level 0" curve whose complexity allows to distribute the actors on the different cores. That's why the latency decreases with the number of cores. The level 3 clustering configuration, because it is sequential, has the longest latency and is architecture-independent. Thus the different clustering

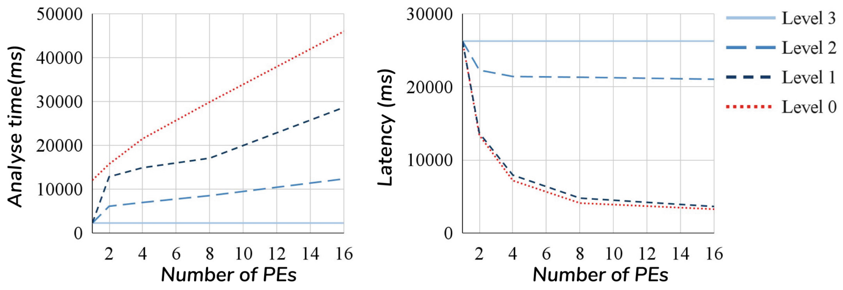


Fig. 7. Comparison of analysis time and latency between the classical flattening approach and three configurations of SCAPE method on *Stereo* application on several architectures (Color figure online)

configurations offer graphs of different levels of granularity and provide a tradeoff between the analysis time and the latency of an application.

Results depicted in Table 1 correspond to the ratio between the times obtained on the state-of-the-art configuration, "level 0", and those obtained on the clustering configuration that offers the best compromise between analysis time and latency. A value greater than 1 is a speedup. The values obtained on the *Squeezenet* application on level 0 configuration are estimated with an exponential function. The process exceeded the RAM memory capacity of the used machine, due to the large number of actors in the SrDAG on the state-of-the-art configuration, and was unable to complete after 48 h. Hence the relevance of the method that allows to provide analysis and executable code even on very complex applications.

Table 1. Comparison of analysis time and latency between the classic flattening approach and best configurations of SCAPE method on three use-cases

Application	SDF	Level	SrDAG	Relative time	Number of PEs				
					1	2	4	8	16
Stereo	28	2	187	analysis	5.3	1.2	1.4	1.7	1.6
				execution	1.0	0.9	0.9	0.8	0.8
Stabilization	22	3	98	analysis	1.5	0.5	0.5	0.6	0.7
				execution	1.0	1.0	0.7	0.7	0.8
Squeezenet	98	3	5452	analysis*	203.5k	100.5	94.5	84.2	68.8
				execution*	1.0	1.0	1.0	1.0	1.0

*Estimated values

5 Conclusion

This paper presents a new method to reduce mapping and scheduling time while preserving the parallelism of SDF graphs. It consists in reducing the size of the graph by clustering actors reproducing particular patterns and then reducing the firing instances of these clusters on the target architecture. The method allows the user to choose the potential expense of the produced schedule and reduce the analysis time accordingly. Experimental results show that for a significantly improved analysis time we obtain a slightly deteriorated latency of the generated code. In addition, the methods enable mapping and scheduling massively parallel applications which were too complex for state-of-the-art approaches. Potential directions for future work include identifying and clustering more complex patterns and automating the search for the optimal level of clustering, without needing to try all configurations.

References

1. Alok, G.: Architecture apocalypse dream architecture for deep learning inference and compute-versal ai core. Embedded World (2020)
2. Bhattacharyya, S., Murthy, P., Lee, E.: APGAN and RPMC: complementary heuristics for translating DSP block diagrams into efficient software implementations. Des. Autom. Embedded Syst. **2**, 33–60 (1997)
3. Bhattacharyya, S.S., Lee, E.A.: Scheduling synchronous dataflow graphs for efficient looping. J. VLSI Signal Process. Syst. **6**(3), 271–288 (1993)
4. Desnos, K., Heulot, J.: PiSDF: parameterized & interfaced synchronous dataflow for MPSoCs runtime reconfiguration. In: 1st Workshop on Methods and Tools for Dataflow Programming (METODO). ECSI, Madrid, Spain (2014)
5. Heulot, J., et al.: An experimental toolchain based on high-level dataflow models of computation for heterogeneous mpsoc. In: Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing, pp. 1–2 (2012)
6. Klikpo, E.C., Khatib, J., Munier-Kordon, A.: Modeling multi-periodic simulink systems by synchronous dataflow graphs. In: 2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 1–10 (2016)
7. Lee, E., Ha, S.: Scheduling strategies for multiprocessor real-time DSP. In: 1989 IEEE Global Telecommunications Conference and Exhibition 'Communications Technology for the 1990s and Beyond', vol 2, pp. 1279–1283 (1989)
8. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **C–36**(1), 24–35 (1987)
9. Pino, J., Bhattacharyya, S., Lee, E.: A hierarchical multiprocessor scheduling system for DSP applications. In: Conference Record of The 29th Asilomar Conference on Signals, Systems and Computers, vol. 1, pp. 122–126 (1995)
10. Pino, J., Bhattacharyya, S., Lee, E.: A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs (1995)
11. Sarkar, V.: Partitioning and scheduling parallel programs for execution on multiprocessors (1987)
12. Sih, G., Lee, E.: Scheduling to account for interprocessor communication within interconnection-constrained processor networks, pp. 9–16 (1990)
13. Zhou, Z., Desnos, K., Pelcat, M., Nezan, J., Plishker, W., Bhattacharyya, S.: Scheduling of parallelized synchronous dataflow actors (2013)