Numerical Representation of Directed Acyclic Graphs for Efficient Dataflow Embedded Resource Allocation

Florian Arrestier, Karol Desnos, Eduardo Juarez, Daniel Menard

Numerical Representation of Directed Acyclic Graphs for Efficient Dataflow Embedded Resource Allocation

Florian Arrestier
Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164
Rennes, France
florian.arrestier@insa-rennes.fr

Karol Desnos
Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164
Rennes, France
karol.desnos@insa-rennes.fr

Eduardo Juarez
Universidad Politécnica de Madrid, CITSEM
Madrid, Spain
eduardo.juarez@upm.es

Daniel Menard
Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164
Rennes, France
daniel.menard@insa-rennes.fr

ABSTRACT

Stream processing applications running on Heterogeneous Multi-Processor Systems on Chips (sHMPSoCs) require efficient resource allocation and management, both at compile-time and at runtime. To cope with modern adaptive applications whose behavior can not be exhaustively predicted at compile-time, runtime managers must be able to take resource allocation decisions on-the-fly, with a minimum overhead on application performance.

Resource allocation algorithms often rely on an internal modeling of an application. Directed Acyclic Graphs (sDAGs) are the most commonly used models for capturing control and data dependencies between tasks. DAGs are notably often used as an intermediate representation for deploying applications modeled with a dataflow Model of Computation (MoC) on HMPSoCs. Building such intermediate representation at runtime for massively parallel applications is costly both in terms of computation and memory overhead.

In this paper, an intermediate representation of DAGs for resource allocation is presented. This new representation shows improved performance for run-time analysis of dataflow graphs with less overhead in both computation time and memory footprint. The performances of the proposed representation are evaluated on a set of computer vision and machine learning applications.

1 INTRODUCTION

Dataflow Models of Computation (sMoCs) are commonly used to model stream processing applications in many domains such as video and audio processing, telecommunications, and computer vision. Dataflow MoCs and related languages are increasingly popular due to their advanced analyzability and their natural expressiveness of parallelism. The recent specialized dataflow-based programming language TensorFlow [1] is an evidence of this popularity in the context of neural networks implementation on massively parallel hardware architectures. In the computer vision applications field, the OpenVX [12] standard aims at providing high performances on heterogeneous architectures, also leveraging on a dataflow MoC. An application described with a dataflow MoC is a graph composed of processing entities, called *actors*, connected through First-In First-Out Queues (sFifos). In a dataflow graph, Fifos are used to convey data tokens between actors and the execution of an actor, also called *firing* of an actor, depends on the number of data tokens available on the input Fifos of the actor.

In an embedded context, taking fast and efficient decisions also require an efficient intermediate representation of the application. Using compact and expressive dataflow MoCs, such as the Cyclo-Static Dataflow (CSDF) [6], the Schedulable Parametric Dataflow (SPDF) [10] or the Interfaced Based Synchronous Dataflow (IB-SDF) [19] allows for a high-level description of an application. However, the more compact and expressive the representation, the more costly it can be to extract information. For instance, extracting fine-grain dependencies information from a Directed Acyclic Graph (DAG) is straightforward whereas it is first necessary to compute model transformations on a CSDF-based application to do so. The more expensive stages of expressive model analysis have led to the more frequent use of DAG-based models in programming frameworks. Frameworks such as StarPU [3], XKaapi [11], OpenVX [12] or TensorFlow [1] rely on DAGs dataflow MoCs. DAGs efficiently model directed workflows with task-level parallelism. However, complex structures such as loops are cumbersome to model with DAGs due to the fact that the entire loops have to be unrolled.

There is a paradox between developing more expressive and more compact dataflow MoCs, and the fact that analysis methods often depend on the need of expanding expressive graphs into DAGs. Some works, however, try to take advantage of the expressiveness of the original MoC [8] or to limit the expansion of graphs and accelerate analysis [23].

Construction of the intermediate DAG representation at runtime is a costly step that needs to be repeated multiple times in the context of dynamic applications. In this paper, we propose a numerical modeling of the expanded DAG representation of the Synchronous DataFlow (SDF)-based MoC and some of its extensions which avoids having to build the intermediate DAG completely, thus improving significantly the performance of embedded runtimes. Our representation allows using DAG oriented analysis methods while maintaining the compactness and the expressiveness of the targeted dataflow MoC. We implemented our numerical modeling of DAGs in the Spider tool [13] on three different platforms ranging from

EMSOFT 2019, October 13–18, 2019, New-York, United-States
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

a medium laptop to a low power embedded platform. Our experiments show a significant reduction of the overhead of the Spider embedded runtime both in terms of execution time and memory footprint of the runtime.

Dataflow MoCs are presented in Section 2, followed by a presentation of existing runtimes that use DAG representation and methods that aim at avoiding the full expansion of DAG in Section 3. Then, our numerical representation of the DAG is presented in Section 4. Section 5 presents experimental results of the implementation of our contribution into the Spider tool [13] on signal processing applications. Finally, Section 6 concludes this paper.

2 CONTEXT: MODELS OF COMPUTATION

In this section, we first present the SDF MoC [16], one of the most popular static specialization of the Dataflow Process Network (DPN) MoC [17]. Then we present the Parameterized and Interfaced Synchronous DataFlow (πSDF) MoC [9] which is the MoC used by the Spider tool used in our experiments. Finally, we present the Single-Rate Directed Acyclic Graph (SR-DAG) specialization of SDF and the related transformation between an SDF Graph (SDFG) and an SR-DAG.

2.1 SDF Model of Computation

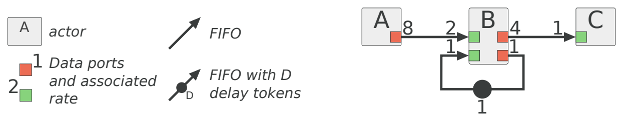

Figure 1: SDF graphical semantics and a graph example.

An application described with the SDF MoC is defined with a directed graph, whose nodes are called actors and edges Fifos. Firing rules of the SDF MoC define data token production and consumption rates of actors as fixed scalars, meaning that rates are set at design time and are fixed for the entire execution of the application. The graphical semantics of the SDF MoC and an example of SDF graph are presented in Figure 1.

Formally, an SDF graph $G = (\mathbb{A}, \mathbb{F})$ contains a set of actors $\mathbb{A}$ that are interconnected through a set of Fifos $\mathbb{F}$. An actor $a \in \mathbb{A}$ reads data tokens from its input ports and produces data tokens on its output ports. The execution of an actor is called a firing and for an actor to fire, enough data tokens need to be available on all of its input ports. In the graph of Figure 1, actor B can only fire when 2 data tokens are present on the Fifo ($\vec{AB}$) and 1 data token is present on its self-loop. The initial data tokens of a Fifo $f \in \mathbb{F}$ are called delays. The value n of the delay is the number of initial data tokens of f.

The popularity of the SDF MoC comes from its great analyzability. Indeed, using static analyses, the consistency and liveness properties of an SDF graph can be verified. When an SDF graph is schedulable, i.e it is consistent and live, a minimal sequence of firings of the actors exists for achieving an infinite execution with bounded memory. Such minimal sequence is called a *graph iteration* and the number of firings of each actor is given by the coefficients of the Repetition Vector (RV) of the graph. Figure 1 presents an SDF graph that is consistent and live. For each graph iteration, actor A is executed 1 time, actor B 4 times, and actor C 16 times.

The *consistency* property of an SDFG means that no data token will indefinitely accumulate in any Fifo of the graph. Consistency is checked through the analysis of the topology matrix Γ associated with an SDF graph [16]. Formally, $\Gamma(i, j)$ is the number of data tokens produced or consumed by actor i on Fifo j. $\Gamma(i, j)$ is a positive number if the actor i produces data tokens on the Fifo j and a negative number if the actor consumes data tokens. The graph is consistent if $rank(\Gamma) = |A| - 1$, with $|A|$ the number of connected actors in the graph. The RV, noted q, is defined as the smallest non-zero integer vector verifying $\Gamma * q = 0$. An efficient algorithm for computing the repetition vector of an SDFG is given in [5].

Static extensions to the SDF MoC have been proposed to enforce its expressiveness and conciseness while maintaining the same level of analyzability and predictability. The CSDF MoC [6] has the same expressiveness as the SDF MoC but is more concise. In CSDF, data rates change according to static cycles defined at the creation of the graph. The IBSDF MoC [19] enforces the compositionality and expressiveness of the SDF MoC by adding explicit and well-defined levels of hierarchy. In an IBSDF graph, actors can be defined by another IBSDF graph. However, changes made inside the subgraph definition do not influence the analysis of the parent graph that contains it, hence the compositionality of the IBSDF MoC. The Parameterized DataFlow (PDF) [4], SPDF [10], and πSDF [9] are reconfigurable extensions of the SDF that enforce dynamic reconfigurations of dataflow graphs. The next sub-section details semantics of the πSDF MoC as it is the reference MoC used in this work.

2.2 πSDF Model of Computation

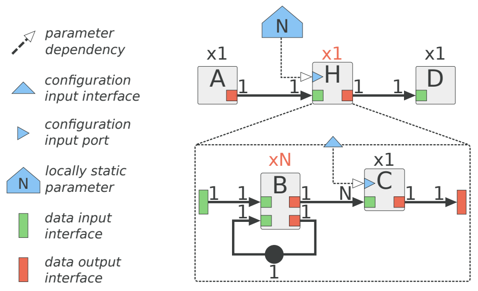

Figure 2: πSDF graphical semantic and a graph example.

The πSDF MoC [9] is a hierarchical and dynamically reconfigurable extension of the SDF MoC. In a πSDF graph, a hierarchical actor is an actor whose internal behavior is defined by a πSDF graph. Figure 2 presents an example of a πSDF graph with the associated graphical semantics. Actor H is a hierarchical actor defined by the subgraph formed by actors B and C.

Formally, a πSDF graph $G = (\mathbb{A}, \mathbb{F}, I, \Pi, \Delta)$ contains in addition to a set of actors $\mathbb{A}$ and a set of Fifos $\mathbb{F}$, a set of hierarchical interfaces I, a set of parameters Π, and a set of parameter dependencies Δ. The hierarchical interfaces of the πSDF MoC [9] are directly inherited from the IBSDF MoC [19] and the reader is invited to read both reference papers for more details on it. This direct inheritance of the interfaces make the πSDF MoC a compositional MoC which means that the internal specification of the actors composing a graph do not influence its analyzability. In Figure 2, the definition of the subgraph formed by actors B and C does not impact the analysis performed on the top-level graph. Using the compositional property of a dataflow MoC, it is possible to perform hierarchical analysis of dataflow graphs [8]. Deroui et al. show that using the hierarchy and the compositional property of the IBSDF MoC, it is possible to perform faster throughput analysis compared to state-of-the-art approaches using equivalent SR-DAG transformation of the original IBSDF graph.

Parameters $\pi \in \Pi$ are associated with parameter values $v \in \mathbb{N}$. Parameter values can either be statically defined or dynamically set by actors at runtime. Reconfigurability of the πSDF MoC comes directly from parameters whose values are used to influence different properties, namely the computation of an actor, the rates of the data ports of an actor, the value of another parameter and the number of delays in a Fifo. In Figure 2, parameter N controls the number of firings of actor B inside the hierarchical actor H but does not affect the analysis of the top-level graph.

2.3 Single-Rate Directed Acyclic Graph

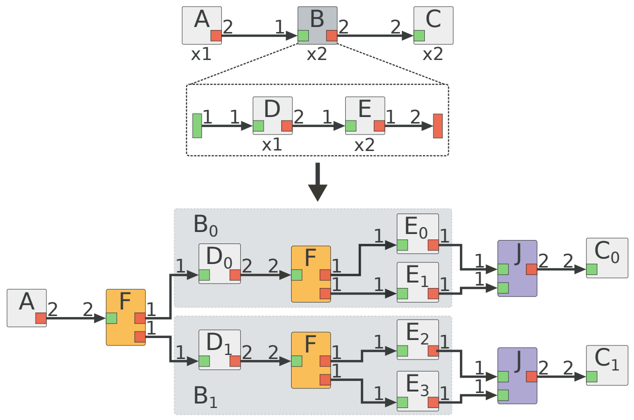

Figure 3: A πSDF to SR-DAG transformation example.

A Single-Rate Directed Acyclic Graph (SR-DAG), also called Acyclic Precedence Expansion Graph (APEG) in the literature [16], is a specialization of an SDFG. An SR-DAG does not contain any cycle and all the data rates on the edges composing the graph are unitary which means that for every edge, the production and consumption rates are equal. Figure 3 shows the transformation of a πSDF graph, in the upper part of the figure, to the equivalent SR-DAG, in the lower part of the figure. Under each πSDF actor of Figure 3 are noted their repetition vector value relative to their containing graph. Actors D and E have repetition vector values of 1 and 2, respectively, within 1 iteration of actor B but a global repetition value of 2 and 4, respectively. In the SR-DAG, all actors have a repetition value strictly of 1.

In our work, SR-DAG is considered to respect SDF dataflow semantics. Particularly, one data port can only be connected to a unique edge. Thus, in order to respect this constraint, special actors are introduced. Fork actors split a given edge into multiple edges such as $\sum_{j=0}^{n-1}(p_j) = P_F$, where p_j is the production rate of the split edge j of the Fork actor and P_F is the production rate of the original edge. In Figure 3, three Fork actors are added for the edges $(\vec{AB})$ and $(\vec{DE})$ during the SR-DAG transformation. Symmetrically, Join actors merge multiple edges into one edge, with $\sum_{j=0}^{n-1}(c_j) = C_J$, where c_j is the rate of merged edge j and C_J is the consumption rate of the obtained merged edge. In Figure 3, two Join actors are added for the edge $(\vec{BC})$ of the original πSDF graph, which becomes an edge $(\vec{JC})$ after the SR-DAG transformation.

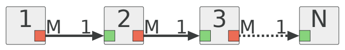

Figure 4: An SDF graph resulting in $O(M^N)$ SR-DAG actors.

Building the SR-DAG of an SDFG is a way of explicitly exposing dependencies across all actor firings of the original SDFG. The SR-DAG "exposes" all information a scheduler needs to take decisions. However, once that the SR-DAG is built, the scheduler no longer benefits from the compact and expressive representation of the original MoC used to describe the application. For instance, using the πSDF representation of the graph in Figure 3, a scheduler could easily perform hierarchical scheduling of actor B, whereas using the SR-DAG representation this information is lost. Having a tunable intermediate representation where information is already pre-processed helps to make simpler and faster scheduling algorithms. Finally, the complexity of building the SR-DAG on graphs with a high degree of parallelism grows exponentially with the repetition values of the actors and so does the complexity of the scheduling algorithm. An example of a graph with such exponential growth is given in Figure 4 where each actor is executed M times relative to its predecessor. Building the SR-DAG representation of an SDFG is therefore not well-suited for embedded runtimes where scheduling needs to be done on-the-fly.

3 CHALLENGES AND RELATED WORK

In this section, we first present the different challenges the proposed contribution addresses. Then, we present existing techniques and frameworks where these challenges are partially addressed.

3.1 Run-Time Challenges

In the context of this work, we consider a reconfigurable dataflow MoC such as the πSDF [9] or the SPDF [10]. Here, reconfigurable means that application graphs may evolve at runtime with changes in data rates or in the graph topology itself. Reconfigurable dataflow MoC imply that full static analysis of an application is not always possible at compile-time and needs to be handled at runtime. SPDF and πSDF MoCs allow for a quasi-static schedule to be derived at compile-time, removing a part of the runtime overhead. However, we will only consider the case where quasi-static schedules are not derived at compile-time as it is the worst case scenario for these models.

When dealing with dynamic behavior such as graph reconfiguration, the first challenge is to perform graph analysis and scheduling of the application with as low overhead as possible relative to the application execution time. Ideally, the time allowed for those analyses should always be negligible compared to computation time.

A second concern of matter should be the memory footprint of the runtime manager. Some analysis techniques require storing additional information that is only used for analysis purpose. For instance, in the Kalray Massively Parallel Processor Array (MPPA) ®, memory is a great concern. The MPPA ® architecture features 16 clusters composed of 16 VLIW processing core each. Each of the cluster has a local memory of 2MB and, although it has access to a bigger shared-memory, reading and writing to this memory is expensive and should be avoided as much as possible. In such a context, storing additional information for analysis purpose only can result in more frequent access to the shared-memory and thus in a downgrade of overall performances.

3.2 Existing Solutions

3.2.1 Existing runtimes. HMBE Integrated HTGS (HI-HTGS) [22] is a design tool that aims at automating analysis and optimizations of Windowed Synchronous DataFlow (WSDF) [14] graphs. HI-HTGS provides a lock-free and race-condition-free scheduler that dynamically adapts to changes in actor execution times and cope with non-deterministic characteristics of thread-based execution. HI-HTGS works in two distinct phases: a compile-time phase and a run-time phase. During the compile-time phase, HI-HTGS builds the SR-DAG representation of the WSDF user graph and performs various analyses that will be used during the run-time phase. At run-time, HI-HTGS uses the built SR-DAG and additional information of the compile-time phase to perform dynamic scheduling on multi-core processors. Due to the compile-time construction of the SR-DAG, HI-HTGS only handle static applications.

Spider [13] is a runtime manager designed for the execution of reconfigurable πSDF [9] applications on HMPSoCs platforms. Spider takes a high-level πSDF graph description of an application as input. Due to the reconfigurable nature of the πSDF MoC, Spider derives an SR-DAG and performs graph optimizations, mapping and scheduling of the application at runtime, as opposed to HI-HTGS [22]. The transformation to SR-DAG may take non-negligible time on reconfigurable applications with high-degree of task and data parallelism and with low complexity computation kernel, hence the need for a more compact representation of the SR-DAG.

The OpenVX [12] standard is a graph-based Application Programming Interface (API) proposed by the Khronos group for developing and deploying computer vision applications on embedded platforms. The MoC used by OpenVX is an SR-DAG specialization of the SDF MoC [16]. As seen in Section 2.3, SR-DAGs are less-expressive and more restrictive than SDFGs but allow for global high-level optimization. However, they limit the data-parallelism opportunities due to the fact that in OpenVX each node is supposed to be an atomic computer vision, or deep-learning, computation kernel. In SDFGs, non-unitary data rates between actors favor data parallelism allowing for each computation kernel to be further parallelized. Hence, OpenVX standard relies mostly on task-parallelism.

Other runtimes such as StarPU [3] or XKaapi [11] are task-graph based runtimes. Similarly to OpenVX, StarPU and XKaapi use a DAG dataflow model to schedule the different tasks. However, StarPU and XKaapi mainly focus on High Performance Computing (HPC) on heterogeneous architectures composed of multi-core CPUs and GPUs whereas OpenVX main focus are computer vision applications on embedded platforms. It is important to note that contrary to OpenVX, StarPU schedules the application graph at the same time it is constructed, thus limiting its vision of the full application for resource allocation decisions but allowing for dynamic reconfiguration of the application.

3.2.2 Avoiding graph expansion. Building the SR-DAG of an SDF application might not always guarantee the best performance. The resulting graph often contains more parallelism than what can actually be exploited by the targeted architecture. Moreover, this exponential growth of the SR-DAG with respect to the original SDFG increases the complexity of scheduling algorithms for HMPSoCs platforms. To limit the explosion of nodes in the SR-DAG transformation, the clustering of the original SDFG is proposed in [20], where four clustering criteria are identified. These clustering criteria provide sufficient condition for checking the introduction of deadlocks in resulting clustered graph. Pino et al. then propose a hierarchical scheduling algorithm and show that clustered SDFGs result in faster scheduling with very low impact on the obtained makespan compared to scheduling the full SR-DAGs. Using a MoC that is hierarchical and compositional by nature, as in the IBSDF [19] or the πSDF [9], removes the need of the clustering step and the hierarchical scheduling algorithm may be used directly.

Another approach to avoid the full-expansion of an SR-DAG is called the vectorization of SDFGs [21]. In [21], the optimal vectorization of an SDFG is achieved by multiplying the rates of the original graph by integers resulting in less invocation of the actors of the SDFG. Partial Expansion Graphs (sPEGs) [24] formulation provides a framework in which the vectorization of actors is integrated efficiently for multiprocessor scheduling context. Zaki et al. use Particle Swarm Optimization (PSO) to find and adjust the amount of expansion, or vectorization, of the actors of the graph.

Schedule-Extended SDFGs [7] are another class of SDFGs that aims at providing a more compact representation for throughput analysis and buffer sizing than SR-DAGs. Damavandpeyma et al. show that encompassing scheduling information directly into the original SDFG significantly reduce time for iterative throughput and buffer sizing analysis. Additionally, authors show that SR-DAG representation may lead to overestimated required buffer sizes compared to applying the same buffer sizing technique on schedule-extended SDFGs. The authors also mention that the construction time of the SR-DAG is very low compared to the analysis time. Although this is true in the context of static analysis at compile-time, the same assumption can not be made when the construction of SR-DAG is performed at runtime. Experiments in Section 5 show that in the Spider tool [13] and for all applications and platforms, the overhead induced by the construction time of the SR-DAG alone is significantly higher than the scheduling time of the SR-DAG.

Most of the existing work presented in this section show that using an SR-DAG transformation for scheduling and analysis of dataflow graphs is the most classical approach. SR-DAG offers a complete exposure of task and data parallelism available in the application. However, most of the presented work use static dataflow MoCs and SR-DAG computation time is neglected, as it can be computed at compile-time. In the context of a reconfigurable MoC such as the πSDF MoC [9], embedded runtimes need to compute SR-DAG on-the-fly and it may have a significant impact on application performance, especially in the context of embedded platforms.

In this paper, we show that it is possible to use SR-DAG information without having to pay the actual cost of building and storing it. In Section 5, the results of the implementation of our contribution in the Spider [13] tool show significant gain both in term of memory footprint and computation time overhead.

4 NUMERICAL MODELING OF SR-DAG

In this section, we show how it is possible to numerically model an SR-DAG by the equations of dependencies that define it. Then, we show that it is possible to further tune those equations in order to encompass the hierarchy semantics of the πSDF MoC.

4.1 Dependency Representation for SDFGs

In this section, a numerical representation of the dependencies of an SDFG is presented. First, the use of an SR-DAG is illustrated with an example, then the numerical model of dependencies is developed. In Sections 4.2 and 4.3, this model is extended to take into account the specificity of the πSDF MoC.

In the following, we refer to the *firing* a_i of actor a as being the i^{th} invocation of actor a during 1 iteration of the graph containing it. The last firing of actor a is a_{q_a-1}, with q_a being the repetition vector value of actor a. In the original work of Lee et al. [16], SR-DAG is depicted as a step needed for scheduling an SDFG.

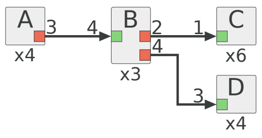

Figure 5: SDF graph with overlapping dependencies.

The SR-DAG removes all the cycles and exposes the precedence relationship between the different firings of all the actors within 1 iteration of a given SDFG. Figure 5 shows an example of a simple SDFG in which there is some overlapping in dependencies for the execution. We refer to overlapping dependencies as the fact that multiple firings of a same actor depend on the same firings of another actor. For example, in the graph of Figure 6, firings D_0 and D_1 both depend on firing B_0. The SR-DAG of Figure 6 unravel all the dependencies of the graph of Figure 5 both for scheduling the execution of the graph and for the memory allocation of the different Fifos. For instance, D_0 depends only on B_0 but D_1 depends on both B_0 and B_1. On the other hand, every two firings of actor C depends on only one firing of actor B meaning that a scheduler minimizing memory allocation could schedule two successive firings of C between two firings of B so that the allocated buffer of the Fifo ($\vec{BC}$) is reused. Importantly, the added Fork and Join actors are necessary in the SR-DAG transformation to explicit the shared dependencies but they are not necessary to model those dependencies and thus will not appear in the proposed numerical representation.

However, building the SR-DAG is not necessary to have the information of the dependencies. All dependencies between firings of actors can be derived numerically by analyzing the production

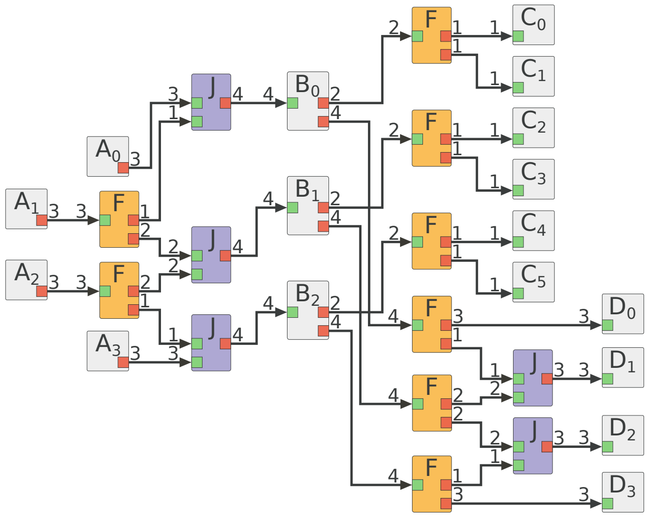

Figure 6: SR-DAG of πSDF graph of Figure 5.

and consumption of the different edges of the graph. We define the dependency matrix Δ_a of an actor a in Equation (1a). Dimensions of Δ_a are $N_{in} \times q_a$, with N_{in} the number of input edges of actor a and q_a the repetition vector value of a. There is one row for each input edge e_j of actor a and one column per firing k of a. Each value of Δ_a, noted $\delta_{j,k}$ (Equation (1b)), is a sub-matrix of size 1×2 that corresponds to an interval of dependency for edge e_j and instance k of actor a.

The first value of $\delta_{j,k}$, noted $\delta^0_{j,k}$, correspond to the first firing of $src(e_j)$ needed for the firing of a_k, with $src(e_j)$ being the actor producing data tokens on e_j. The second value of $\delta_{j,k}$, noted $\delta^1_{j,k}$, correspond to the last firing of $src(e_j)$ needed for the firing of a_k. In other words, $\delta_{j,k}$ represent the interval of firings of $src(e_j)$ on which firing k of actor a depends to execute. Since dependencies are necessary in increasing order, the first and the last firing number of $src(e_j)$ are sufficient to define completely the dependency interval.

$$\Delta_a = \text{edges}\big\downarrow \overset{\text{firings of } a \longrightarrow}{\begin{bmatrix} \delta_{0,0} & \delta_{0,1} & \cdots & \delta_{0,q_a-1} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_{N_{in}-1,0} & \delta_{N_{in}-1,1} & \cdots & \delta_{N_{in}-1,q_a-1} \end{bmatrix}} \tag{1a}$$

$$\delta_{j,k} = \begin{bmatrix} \delta^0_{j,k} & \delta^1_{j,k} \end{bmatrix} \tag{1b}$$

Taking the example of Figure 5, Equation (2) gives the corresponding dependency matrix of actor D. Firing 0 of actor D, D_0, depends on the firings 0 to 0 of actor B, i.e D_0 can be fired as soon as B_0 is finished. Similarly, D_1 depends on firings 0 to 1 of actor B. Hence, D_1 can be fired if and only if B_0 and B_1 have finished their execution.

$$\Delta_D = \begin{array}{cc} & \begin{array}{cccc} D_0 & D_1 & D_2 & D_3 \end{array} \\ \vec{BD} & \begin{bmatrix} [0\ 0] & [0\ 1] & [1\ 2] & [2\ 2] \end{bmatrix} \end{array} \tag{2}$$

Theorem 1
Let G be a consistent and live SDF Graph, and $\mathbb{A}$ be the associated set

of actors. If and only if G is consistent, there exist a repetition vector q of size $|\mathbb{A}|$. For any firing k of an actor $a \in \mathbb{A}$, and for any input edge $e_j \in a$, there exists a dependency interval $\delta_{j,k} = \left[\delta^0_{j,k}\ \delta^1_{j,k}\right]$ with $\delta^0_{j,k}, \delta^1_{j,k}$ the first and last dependencies of e_j, respectively.
We have:

$$\delta^0_{j,k} = \left\lfloor \frac{k * c_j - d_j}{p_j} \right\rfloor \tag{3a}$$

$$\delta^1_{j,k} = \left\lfloor \frac{c_j * (k+1) - d_j - 1}{p_j} \right\rfloor \tag{3b}$$

where c_j, p_j and d_j are the consumption rate, the production rate and the number of initial delays on the edge e_j, respectively.

Proof of Equation (3b). Let $b \in \mathbb{A}$ be the actor producing data tokens on input edge e_j of actor a and q_b and q_a be the repetition vector values of b and a, respectively. If and only if G is consistent, then the sum of all data tokens produced by actor b is equal to the sum of all data tokens consumed by actor a. Equation (4) formalizes this property.

$$\sum_{l=0}^{l=q_a-1} (c_j) = \sum_{i=0}^{i=q_b-1} (p_j) \tag{4}$$

For any firing k of actor a to execute, the sum of all the data tokens consumed by firings of actor a up to k must be less or equal to the sum of all the data tokens produced by actor b and the initial delays of the edge e_j. Formally, for any a_k, $k \in [0; q_a[$, there exists a given positive integer $m \in [0; q_b[$ verifying Equation (5).

$$\sum_{l=0}^{l=k}(c_j) \le \sum_{i=0}^{i=m}(p_j) + d_j \tag{5}$$

We search the minimal value m_0 of m such that Equation (5) holds. In other words, we search the minimal value m_0 for which the sum of all the data tokens produced by actor b and the initial delays is greater or equal to the sum of data tokens consumed by actor a up to its firing k. Consequently, this means that for $m_0 - 1$, the sum of all data tokens produced by actor b and the initial delays is strictly inferior to the sum of the data tokens consumed by actor a up to firing k which translates in Equation (6).

$$\sum_{l=0}^{l=k}(c_j) > \sum_{i=0}^{i=m_0-1} (p_j) + d_j \tag{6}$$

By developing the sums in Equation (5) comes:

$$(k+1) * c_j \le (m_0 + 1) * p_j + d_j \tag{7a}$$

$$\frac{(k+1) * c_j - d_j}{p_j} \le m_0 + 1 \tag{7b}$$

Developing Equation (6):

$$(k+1) * c_j > m_0 * p_j + d_j \tag{8a}$$

$$\frac{(k+1) * c_j - d_j}{p_j} > m_0 \tag{8b}$$

And using the fact that $\lceil x \rceil = n,\ n \in \mathbb{N}$ if and only if $n \ge x > n - 1$:

$$m_0 + 1 = \left\lceil \frac{(k+1) * c_j - d_j}{p_j} \right\rceil \tag{9a}$$

$$m_0 = \left\lceil \frac{(k+1) * c_j - d_j}{p_j} \right\rceil - 1 \tag{9b}$$

Finally, since k, c_j and p_j are positive integers, we have:

$$m_0 = \left\lfloor \frac{(k+1) * c_j - d_j - 1}{p_j} \right\rfloor \tag{10}$$

■

To prove Equation (3a), we search for the minimal value n_0 such that the sum of the initial delays and of all data tokens produced by a given actor b is greater than the sum of all data tokens consumed by a given actor a, up to its k^{th} firing. This definition translates to Equation (11). The rest of the developments are similar to the proof of Equation (3b) and are omitted due to space limitations.

$$\sum_{l=0}^{l=k}(c_j) < \sum_{i=0}^{i=n_0}(p_j) + d_j \tag{11}$$

Having delays on a Fifo may result in negative values for $\delta^0_{j,k}$ and $\delta^1_{j,k}$. If $\delta^0_{j,k}$ or $\delta^1_{j,k}$ is negative, this means that firing k of actor a depends on initialization tokens coming either from previous graph iteration or from a setter actor setting those initial tokens [2].

4.2 Taking hierarchy into account

Equations (3a) and (3b) hold in the general case of SDF graphs. However, to take into account the hierarchical specificity of the πSDF and IBSDF MoCs, it is necessary to define additional equations to define the behavior of interfaces. In this section, only the interfaces are discussed as the other actors inside a subgraph behave the same way as in a SDFG, meaning that Equations (3a) and (3b) apply to them. As defined in [9, 19], input and output interfaces act as a "frontier" between a hierarchical actor and its inner subgraph definition. All data tokens of an input interface must be consumed at least once during an iteration of a subgraph. If more data tokens are consumed, due to repetition vector values, then the interface behaves like a circular buffer producing the same data tokens as many times as needed. Symmetrically, an output interface only outputs the last data tokens produced by the actor connected to it and discards the rest. Importantly, interfaces have a repetition vector value of 1.

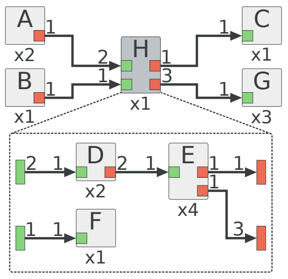

Figure 7: Hierarchical πSDF graph example.

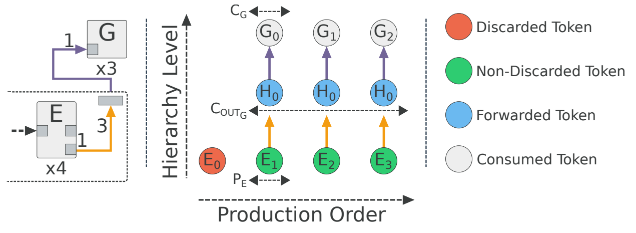

Figure 8: Behavior of the output interface connecting the subgraph H to actor G in Figure 7. Tokens are named after the corresponding firing of the actor producing them.

Figure 8 illustrates the behavior of the output interface connecting the subgraph H to the actor G in Figure 7. In Figure 7, the actor E is executed 4 times within the subgraph H, producing 4 data tokens on its output data port connected to the output interface itself connected to the actor G in the upper-graph. The output interface only consumes 3 data tokens, meaning that only the last three executions of the actor E are used for this interface, as showed in Figure 8 with the first data token produced by actor E being discarded.

Equations (12) give the dependency interval definition for an output interface o_{if} of a given subgraph.

$$\delta^0_{o_{if}} = q_P - \left\lceil \frac{c_{o_{if}}}{p_{o_{if}}} \right\rceil \tag{12a}$$

$$\delta^1_{o_{if}} = q_P - 1 \tag{12b}$$

where q_P is the repetition vector value of the actor producing data tokens on o_{if}, $c_{o_{if}}$ the consumption rate of the interface o_{if} and $p_{o_{if}}$ the production rate on the interface o_{if}. In Figure 8, $p_{o_{if}}$ corresponds to $P_E = 1$, $c_{o_{if}}$ corresponds to $C_{outG} = 3$ and q_P corresponds to $q_E = 4$. Applying Equations (12a) and (12b) to Figure 8 gives the first dependency on E equal to $\delta^0_{outG} = 1$ and the last dependency on E equal to $\delta^1_{outG} = 3$. Note that delays on FIFOs connected to output interfaces do not impact Equations (12a) and (12b) due to the behavior of the interface to only output the last data tokens produced on it. Dependencies on output interfaces also give to the scheduling algorithm the earliest time at which a hierarchical actor can be considered to have finished its internal execution.

Equation (12b) comes directly from the definition of the output interfaces [19]. If the interface only outputs the last data tokens produced on it, then the last dependency of the interface is necessarily the last firing of the actor producing data tokens on it. Equation (12a) is derived using a similar development to the one of Equation (3a). The aim is to find the minimum number of firings N of the actor producing data tokens on output interface o_{if} such as:

$$\sum_{i=1}^{N} p_{o_{if}} \geq c_{o_{if}} \tag{13a}$$

$$\sum_{i=1}^{N-1} p_{o_{if}} < c_{o_{if}} \tag{13b}$$

Using the developments of Equation (3a), it comes:

$$N = \left\lceil \frac{c_{o_{if}}}{p_{o_{if}}} \right\rceil \tag{14}$$

The first dependency of the output interface is then defined by:

$$\delta^0_{o_{if}} = q_P - N \tag{15a}$$

$$\delta^0_{o_{if}} = q_P - \left\lceil \frac{c_{o_{if}}}{p_{o_{if}}} \right\rceil \tag{15b}$$

which corresponds to Equation (12a).

Input interfaces inherit the dependencies of the hierarchical actors to which they belong. This comes directly from the definition of input interfaces that state that input interfaces can start executing as soon as the hierarchical actor is ready to fire in its parent graph. Therefore, actors connected to input interfaces can start their execution as soon as the subgraph starts and the only dependency to check is related to the presence of delays.

4.3 Relaxed execution model for πSDF

In [8], a relaxed model of execution is used on the IBSDF MoC to maximize the throughput of an application containing multiple levels of hierarchy. The relaxed execution model allows for actors contained inside an IBSDF subgraph to start their execution without having to wait for data tokens on all interfaces of their containing hierarchical actor. For example, in the graph of Figure 7, and with a relaxed execution model, actor F can execute directly after the execution of actor B independently of the executions of actor A. We apply the same relaxed execution model to the πSDF MoC.

Taking into account the relaxed constraint in the numerical model of the SR-DAG adds some complexity to the previously proposed equations. We will first investigate the case of the output interfaces. Relaxing the execution model of the πSDF leads to extend the dependency resolution problem of an actor depending on a hierarchical actor from its level of hierarchy to the subgraph level. For example, in the graph of Figure 7 dependencies of actor C are now 2-dimensional. Indeed, actor C depends on executions of actor H and for each firing of actor H, depends on executions of actor E. The objective is thus to combine Equations (3a) and (3b) to Equations (12a) and (12b), respectively.

We note $\delta^N_{a|j,k}$ the sub-matrix of size 1×2, with N the total number of levels of hierarchy the firing k of an actor a depends on for its input edge e_j. $\delta^N_{a|j,k}$ is a generalization of $\delta_{j,k}$ introduced in Section 4.1. Equation (16) gives the general definition of $\delta^N_{a|j,k}$.

$$\delta^N_{a|j,k} = \left[\begin{bmatrix} \delta^{0,0}_{a|j,k} & \cdots & \delta^{0,N-1}_{a|j,k} \end{bmatrix} \begin{bmatrix} \delta^{1,0}_{a|j,k} & \cdots & \delta^{1,N-1}_{a|j,k} \end{bmatrix} \right] \tag{16}$$

Similarly to the definition of $\delta_{j,k}$ given in Section 4.1, $\delta^N_{a|j,k}$ represents the interval of dependencies of firing k of actor a. The main difference is that $\delta^0_{j,k}$ and $\delta^1_{j,k}$ are now defined as sub-matrices of size $1 \times N$. $\delta^{0,n}_{j,k}$ is the first dependency of the kth firing of actor a at level n of hierarchy. Similarly, $\delta^{1,n}_{j,k}$ is the last dependency of a_k at the level n of hierarchy. The generalized definitions of $\delta^{0,n}_{j,k}$ and

$\delta^{1,n}_{j,k}$ are given in Equations (17a) and (17b), respectively.

$$\delta^{0,n}_{a|j,k} = \begin{cases} \delta^{0}_{a|j,k}, & n = 0, \text{ see Equation (3a)} \\ q_{p_n} - \left\lceil \frac{C^{0,n}_{a|j,k}}{P_n} \right\rceil, & n \in [1; N[\end{cases} \tag{17a}$$

$$\delta^{1,n}_{a|j,k} = \begin{cases} \delta^{1}_{a|j,k}, & n = 0, \text{ see Equation (3b)} \\ q_{p_n} - \left\lceil \frac{C^{1,n}_{a|j,k}}{P_n} \right\rceil, & n \in [1; N[\end{cases} \tag{17b}$$

where:

- q_{p_n}, the repetition vector value of the actor producing data tokens on the output interface at level n of hierarchy.
- P_n, the production rate on the output interface at level n of hierarchy.
- $C^{0,n}_{a|j,k}$, the updated consumption rate of the output interface at level n of hierarchy for the first dependency.
- $C^{1,n}_{a|j,k}$, the updated consumption rate of the output interface at level n of hierarchy for the last dependency.

$C^{0,n}_{a|j,k}$ and $C^{1,n}_{a|j,k}$ correspond to the updated consumption rates of the output interface at the level n of hierarchy for the first dependency and the last dependency, respectively. For each level n of hierarchy, the updated consumption rate of the corresponding output interface depends on the one of the level $n-1$, up to the consumption rate of the edge e_j of actor a at the top level of hierarchy. The definitions of $C^{0,n}_{a|j,k}$ and $C^{1,n}_{a|j,k}$ are given by Equations (18a) and (18b), respectively.

$$C^{0,n}_{a|j,k} = \begin{cases} ((\delta^{0}_{a|j,k} + 1)) * p_j + d_j - k * c_j, & n = 1 \\ C^{0,n-1}_{a|j,k} - (q_{p_{n-1}} - (\delta^{0,n-1}_{a|j,k} + 1)) * P_{n-1}, & n \in [2; N[\end{cases} \tag{18a}$$

$$C^{1,n}_{a|j,k} = \begin{cases} ((\delta^{1}_{a|j,k} + 1)) * p_j + d_j - (k+1) * c_j + 1, & n = 1 \\ C^{1,n-1}_{a|j,k} - (q_{p_{n-1}} - (\delta^{1,n-1}_{a|j,k} + 1)) * P_{n-1}, & n \in [2; N[\end{cases} \tag{18b}$$

where:

- c_j, the consumption rate of edge e_j of actor a.
- p_j, the production rate of edge e_j.
- d_j, the initial delay of edge e_j.
- k, the firing of actor a for which dependencies are computed.

Due to space limitation, the full development of these equations is not given in this paper. However, we provide the concept used to derive the equations. A more complete development is given in the joint technical report. A multi-level hierarchical πSDF graph is presented in Figure 9 and Figure 10 shows the corresponding data tokens dependency analysis. Figure 10 shows the direct data dependencies across the different levels of hierarchy.

For instance, the first data token consumed by A_1 is produced by E_2 during the second firing of the subgraph B (B_1), in the first firing of the subgraph H (H_0). Examples of relaxed and non-relaxed dependencies are also given in Figure 10 for A_0. With non-relaxed dependencies, A_0 depends only on H_0, then the output interface of H depends on B_1 and finally the output interface of B depends on E_2. In other words, with non-relax dependencies, A_0 has to wait for the complete execution of the second firing of the subgraph B and the corresponding firings of actor E before it can be fired. With relaxed dependencies, A_0 depends directly on E_2, from B_0 and H_0, and the dependencies due to the interfaces of the different levels of hierarchy are omitted.

By analyzing the distribution of the different data tokens and how the consumption rate of the output interfaces is influenced across the different levels of hierarchy, it comes a direct relationship linking the level n to the level $n-1$ that is expressed in the Equations (18a) and (18b) with the terms $C^{0,n-1}_{a|j,k}$ and $C^{1,n-1}_{a|j,k}$, respectively. The subtraction term of the Equations (18a) and (18b) corresponds to the offset that should be applied in order to have the actual consumption rate of the output interface of the next level of hierarchy. This subtraction comes from the inverse behavior of the output interfaces. For instance, in Figure 10, the real consumption rate of A_0 on the output interface of B_0 is equal to 1 and not 3. Similarly, the consumption rate of A_2 on the output interface of H_1 is 2 instead of 3 which will make A_2 dependent on B_1 and not B_0.

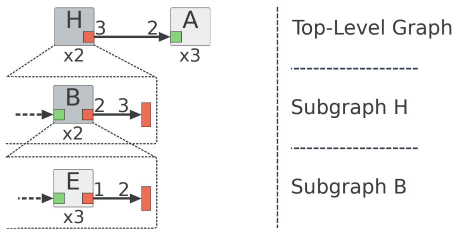

Figure 9: Multi-Level Hierarchical πSDF graph example.

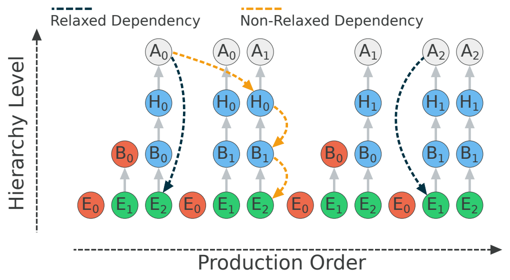

Figure 10: Dependency analysis of the graph of Figure 9. The graphical formalism is the same as in Figure 8

In Section 4.1, we explained that the definition of an interval is sufficient to define the full dependencies of an actor for a given input edge due to the fact that there can be no discontinuity in the dependencies. In other words, if an actor A depends on an actor B with the following interval $[B_0\ B_2]$, then actor A must also depends on B_1. This property is also applicable to the hierarchy case. This means that if an actor A depends on the following dependency interval $[[G_0\ H_0]\ [G_1\ H_1]]$, it must depend on all firings of G and H that fall in between except for the discarded firing of actors due to the behavior of the interfaces. Using Equations (16) , (17a) and (17b) on the example graph of Figure 9 we obtain the following dependency intervals for actors A.

$$\delta^{0}_{A|0,1} = [0\ 1] \tag{19a}$$

$$\delta^{1}_{A|0,1} = [[0\ 1]\ [1\ 0]] \tag{19b}$$

$$\delta^{2}_{A|0,1} = [[0\ 1\ 2]\ [1\ 0\ 2]] \tag{19c}$$

Equation (19a) corresponds to the dependency interval of A_1 at the top level of hierarchy, Equation (19b) corresponds to the dependency interval of A_1 in the subgraph H and Equation (19c) corresponds to the fully relaxed dependency interval of A_1. Equation (19a) shows that A_1 depends on H_0 to H_1 and Equation (19b) shows that A_1 depends on B_1 from H_0 to B_0 from H_1. Finally, Equation (19c) shows that A_1 depends on E_2 from $[H_0\ B_1]$ to E_2 from $[H_1\ B_0]$. It is possible to individually tune the number of hierarchy levels for which the execution of an actor is relaxed which gives flexibility to the scheduling algorithm. In this paper, we will only consider the cases of no-relaxation and full-relaxation.

It is important to note that for dynamic applications with parameter changes in a hierarchical actor H, it is necessary to store the different values of the parameters of each instance of H as it may influence the repetition vector in a particular instance of H and change the dependencies for actors depending on H. This constraint is not necessary for the non-relaxed execution model as the subgraph is hidden from any actor depending on H.

For the case of input interfaces, as stated in Section 4.2 no special equations have to be derived. Actors depending on interfaces directly inherit dependencies of the corresponding input edge of the containing hierarchical actor. This inheritance goes up to the top level of hierarchy. However one particular case has to be considered, the case of an actor consuming more data tokens on an input interface that the interface produces. Since, interfaces have a repetition vector value strictly equal to 1, a special actor, called a *duplicate* actor, is introduced. A duplicate actor has one input port and one output port and duplicates the tokens received on its input port as many times as needed to respect the consistency property. Duplicate actors are automatically inserted during graph analysis.

4.4 Resource allocation

The proposed numerical approach is compatible with dataflow MoCs derived from the SDF MoC and can be used to derive a schedule the same way DAG would. Indeed, it is possible to build an API that emulates accesses to an SR-DAG using the proposed numerical model. From the user point of view, the emulated SR-DAG behaves as a standard SR-DAG, the only difference being that dependencies are computed on-the-fly instead of having a pre-built graph. Therefore, any resource allocation algorithm that uses an SR-DAG can be based on the proposed numerical model instead.

The main advantage of our proposed method is to remove the costly step of building and storing the SR-DAG. However, using our method may result in an increase of the complexity of the original resource allocation algorithm, compared to using the SR-DAG, due to the computation of the dependencies done on-the-fly.

In the experiments of Section 5, to demonstrate the capacity of our model to be used in a real resource allocation algorithm and evaluate the performance gain over the SR-DAG representation, a naive *greedy scheduling* algorithm is used. The performance of the greedy algorithm is not the focus of this paper. The chosen greedy algorithm works as follows:

(1) Create a list with all actors of the graph.
(2) Find an actor a in the list that can be scheduled.
(3) Map actor a onto an available processor.
(4) Remove actor a from the list.
(5) If no more actors, exit scheduling. Else go back to step 2.

The main difference between the SR-DAG-based greedy scheduler and the numerical one comes from the input graph representation used. Using the SR-DAG, the SR-DAG itself is used and the greedy scheduler directly goes through the actor list of the graph to find the first actor that can be scheduled. Using the numerical model, the original πSDF representation of the application is used and the greedy scheduler goes through the πSDF actor list, then for each actor it computes on-the-fly the dependencies of the actor, and for the current firing of the actor, to check if it can be scheduled.

5 EXPERIMENTS

5.1 Experimental Setup

Table 1: Experimental platform characteristics

Platform	Processor	Cores	RAM	GCC
Laptop	Intel®Core™i7-7820HQ	4	32GB DDR4	7.3.0
Jetson TX2	ARM Cortex™-A57 + NVIDIA Denver 2	4 + 2	8GB LPDDR4	5.4.0
ODROID-XU3	Samsung Exynos 5422	4 + 4	2GB LPDDR3	4.9.2

The different experiments are conducted on 3 different platforms ranging from an x86 laptop with medium processor to a very low power ODROID-XU3 platform. The characteristics of these platforms are summarized in Table 1. The Spider library was compiled with `O3` level of optimizations on all platforms. Four applications from the official repository[1] of the Preesm tool [18] have been used to conduct the experiments. These applications are state-of-the-art AI, and computer vision applications. The four applications feature different levels of hierarchy and task and data parallelism, as summarized in Table 2 where $|G_{\pi SDF}|$ corresponds to the number of actors in the πSDF representation, N_{levels} corresponds to the number of hierarchical levels and $|G_{SR\text{-}DAG}|$ corresponds to the number of actors in the SR-DAG representation of the application. The number of edges for both MoCs is noted in the N_{Edges} column of the corresponding MoC.

In the applications used for our experimentation, all parameter values are changing at each graph iteration, thus triggering a complete rescheduling of the application. Although unrealistic, this behavior was forced, even in case of static parameter values, in order to emphasize the most dynamic, and thus the most complex scenario for the runtime allocation of resources. In the case of a more static behavior, both the DAG-Based and numerical model-based solutions can benefit from optimizations to conserve information between successive graph iterations, which is out of the scope of this paper.

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

Table 2: Applications description

	πSDF			SR-DAG	
Application	$\lvert G_{\pi SDF} \rvert$	N_{Edges}	N_{levels}	$\lvert G_{SR\text{-}DAG} \rvert$	N_{Edges}
SqueezeNet	108	272	2	5436	17248
Reinforcement Learning	188	459	3	417	1114
Stabilization	20	41	2	101	325
Sobel-Morpho	6	7	0	65	85

5.2 Results

In this section, we present the different experimental results obtained for our implementation of the presented numerical model into the Spider tool and compare them to the reference implementation that uses an SR-DAG model. Two configurations of the proposed numerical model are compared to the reference implementation. The first configuration is referred to the *relaxed* configuration and corresponds to the use of the relaxed execution model presented in Section 4.3, and the second configuration is referred to the *standard* configuration and corresponds to the non-relaxed execution model of the πSDF MoC. In these experiments, the metrics used for comparing the different configurations are the computation time and the memory footprint of the runtime manager performing the scheduling and mapping of a πSDF application onto multi-cores processor platforms. The scheduling algorithm used in these experiments is the greedy scheduling algorithm described in Section 4.4. Despite being a rather simple algorithm, this scheduling algorithm allows to rapidly demonstrate the feasibility of our proposed models. Nevertheless, the results show that using the direct numerical model gives overall great improvements both in terms of computational complexity and memory footprint. The measured application performance may be further optimized with a smarter scheduling algorithm [15], which would reduce the scheduling time of all experiments, but would not change the memory nor the construction time overhead of the SR-DAG based runtime.

5.2.1 *Memory footprint.*

In this section, we present the memory footprint of the different representations for the scheduling and mapping. No differentiation is made between the *relaxed* and *standard* configurations of the numerical model as both configurations share the same memory footprint.

Table 3 shows the total memory footprint of Spider during the scheduling and mapping of the applications. The gains expressed in Table 3 represent, as a percentage, the amount of memory saved with the numerical model compared to the reference SR-DAG implementation. Results show significant memory reduction with up to 98.63% of memory reduction for the reinforcement-learning application and an average memory reduction of 97.34%. This high memory reduction is due to the fact that in our proposed implementation, we only store the current firing value and the total number of firings of all πSDF actors during the mapping and scheduling phase as all dependencies are computed on-the-fly when needed.

In addition to the memory needed for the different representations, there is memory used to store information about the schedule execution. The memory used for the schedule execution is similar in both the SR-DAG and the numerical model and is comprised in the values of Table 3. In other words, there exists an upper bound to the potential memory footprint reduction that depends on the memory used for the schedule execution. Figure 11 shows the relative memory footprint of the numerical model and the SR-DAG representation over the total memory footprints of Table 3, hence highlighting the relative memory footprint of the schedule execution information. Values of Figure 11 show that actual memory used by the numerical models to perform scheduling and mapping only account for 0.93% to 11.15% of the total memory footprint of Spider whereas in the case of the SR-DAG representation, actual memory used for the scheduling and mapping is greater than 92% of the total memory footprint. Hence, Figure 11 emphasizes the low memory footprint overhead of the proposed approach on the runtime over the reference SR-DAG representation.

Table 3: Memory footprint of the representations

Application	Reference (SR-DAG)	Numerical Model	Gain (%)
SqueezeNet	8405.9 KB	**515.3 KB**	93, 87
Reinforcement Learning	5183.7 KB	**70.9 KB**	98.63
Stabilization	782.8 KB	**11.8 KB**	98.49
Sobel-Morpho	404.5 KB	**6.8 KB**	98.32

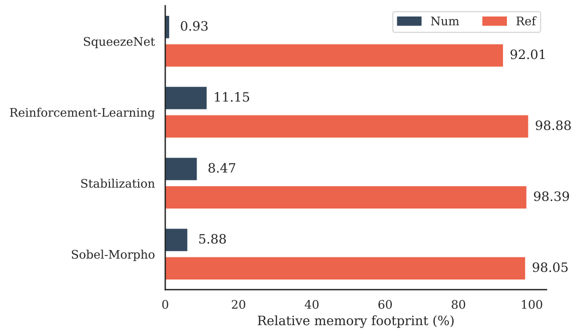

Figure 11: Relative memory footprint of representations over total memory footprint.

5.2.2 *Execution time.*

In this section, the execution times of the different configurations of the numerical model (*relaxed* and *standard*) are compared to the reference implementation. Then, a comparison of the schedule latency, i.e execution time for one graph iteration, for the two configurations of the numerical model is performed, highlighting a potential trade-off between execution time and schedule latency.

Table 4: Intermediate Representation building time in ms

	Laptop		Jetson TX2		ODROID-XU3	
Application	Ref	Num	Ref	Num	Ref	Num
SqueezeNet	7.105	**0.221**	39.43	**0.664**	79.77	**1.90**
Reinforcement Learning	0.868	**0.180**	6.03	**0.551**	12.41	**1.71**
Stabilization	0.138	**0.017**	0.67	**0.059**	1.70	**0.19**
Sobel-Morpho	0.061	**0.005**	0.23	**0.017**	0.69	**0.06**

Table 4 presents the execution times taken by the construction phase of the intermediate representations. In the case of the SR-DAG (Ref column), this time corresponds to the construction of the SR-DAG and the initialization of the schedule execution information. In the case of the numerical model (Num column), the value of Table 4 corresponds to the initialization of the schedule execution information and the allocation of the arrays used to store firing information during the scheduling and mapping phase. Note that the construction phase is shared for both *relaxed* and *standard* configurations, thus no difference is made between them in Table 4. On all three platforms, building the numerical model is significantly faster than building the SR-DAG representation, with a maximum speedup of 59.39 for the SqueezeNet application on the Jetson TX2.

Table 5 shows the resource allocation execution times for the three compared configurations. In Table 5, *Num-R* and *Num-S* refer to the *relaxed* and the *standard* configurations of the numerical model, respectively. The results show significantly lower scheduling times for the *standard* configuration over the two others. This is explained by the hierarchical nature of the *standard* execution model and the greedy scheduling algorithm used. Indeed, the greedy algorithm iterates over the actors of a graph until it finds an actor that can be scheduled. In the numerical model configurations, the algorithm is thus much faster, as it iterates over the πSDF graph which contains fewer actors than the SR-DAG one (see Table 2). Moreover, in the *standard* configuration, actors located in nested levels of hierarchy are not tested until the corresponding hierarchical actor can be scheduled reducing furthermore the number of tested actors per iteration of the greedy algorithm.

Interestingly, Table 5 shows that the *relaxed* configuration has overall higher resource allocation times than the reference configuration. Contrary to the *standard* configuration, in the case of relaxed execution, every actor of the πSDF is tested per iteration of the greedy algorithm. Moreover, the complexity of fetching the dependencies of an actor located in a deep level of hierarchy is significantly higher than when dealing with same level of hierarchy dependencies. This effect is particularly visible with the SqueezeNet application which possesses a high number of dependencies between actors belonging to separate subgraphs. However, the case of the relaxed execution could be improved in future implementations by storing hierarchical dependencies, thus avoiding their re-computation at the cost of an increased memory footprint. Another way of improving the relaxed execution model would be to perform graph analysis before the first graph iteration to simplify the πSDF hierarchy whenever it is possible.

Finally, Table 6 gives the relative difference of the obtained schedule latency when scheduling with the numerical models compared to the reference implementation. A value of 0% means that the obtained schedule latency is equal to the one of the reference. Small relative differences in latency (inferior to 5%) are explained by two factors. Firstly, in the SR-DAG representation, Fork and Join actors are explicitly scheduled due to the fact that they are part of the resulting graph whereas they are not in the numerical representation. Secondly, Spider performs several passes of optimizations on the SR-DAG to reduce the number of special actors (Fork, Join, Broadcast and Roundbuffer actors) introduced during the transformation. However, optimizations do not necessarily remove all special actors introduced during the transformation. Importantly, optimizations passes may also remove special actors that are part of the original πSDF graph which can further improve the obtained schedule latency which is not the case for the numerical representation where no optimizations are performed on the πSDF graph.

Table 6 shows no clear improvement of the schedule latency of the relaxed execution model over the standard one on 3 out of the 4 tested applications. For the Sobel-Morpho application, this is explained by the absence of hierarchy, thus there is no need for relaxation. In the case of the Stabilization application, the obtained latency is limited by the topology of the graph itself with synchronization points that can not be reduced. However, in the case of the Reinforcement-Learning application, a significant gain with a difference up to 24.1 percentage points can be achieved using the relaxed execution model at the cost of higher scheduling time.

Figure 12 shows the relative total execution time for the three configurations and for the three different platforms. The total execution time is the sum of the intermediate representation building time (Table 4) and the scheduling time (Table 5). The relative total execution time is the relative difference of the total execution time of the numerical representations with the total execution time of the reference. Figure 12 shows that even with higher scheduling time for the relaxed configuration, a minimum reduction of 47.11% of total execution time is achieved when considering the total execution time spent in the resource allocation phase of Spider.

For the SqueezeNet application, a reduction of up to 94.5% of the total execution time is achieved on the Jetson platform with the standard execution model with 0.22% of increase on the obtained schedule latency (Table 6). By comparison, the relaxed configuration reduces the execution time of 75.53% on the Jetson platform with a negligible impact on the obtained schedule latency (0.11%). On the other hand, for the reinforcement learning application, there is a non-negligible difference in the obtained schedule latency for the Jetson and Odroid platforms (19.15 and 24.1 percentage points, respectively) with a difference inferior to 10 percentage points of execution time between the relaxed and the standard execution models. Therefore, depending on the application graph topology and the targeted platform, there is a trade-off between better scheduling performance and execution time. Finally, it is important to note that the execution time of the relaxed configuration could be improved with additional optimizations of the implementation in Spider, which would reduce the gap with the standard configuration in terms of raw execution time performance.

6 CONCLUSION

In this paper, we proposed a numerical representation of dependencies relationship between actors first for the SDF MoC and then extended to the πSDF MoC. We showed that numerical representation is better suited for fast resources allocation of application than DAG-based methods due to the cost of building and storing DAG. Experiments on various computer vision and machine learning applications showed significant gains compared to DAG-based methods both in scheduling time and memory overhead. Future work will investigate hierarchy scheduling algorithm and the integration with other state-of-the-art scheduling methodology and algorithms.

Table 5: Resource allocation execution time in ms of the different configurations.

	Laptop			Jetson TX2			ODROID-XU3		
Application	Ref	Num-R	Num-S	Ref	Num-R	Num-S	Ref	Num-R	Num-S
SqueezeNet	2.491	4.856	**1.043**	22.51	14.50	**2.76**	23.10	23.19	**4.49**
Reinforcement Learning	**0.105**	0.327	0.120	0.50	0.91	**0.35**	0.81	1.47	**0.71**
Stabilization	0.020	0.055	**0.019**	0.08	0.13	**0.06**	0.18	0.24	**0.12**
Sobel-Morpho	0.012	0.010	**0.009**	0.05	0.03	**0.03**	0.11	0.06	**0.05**

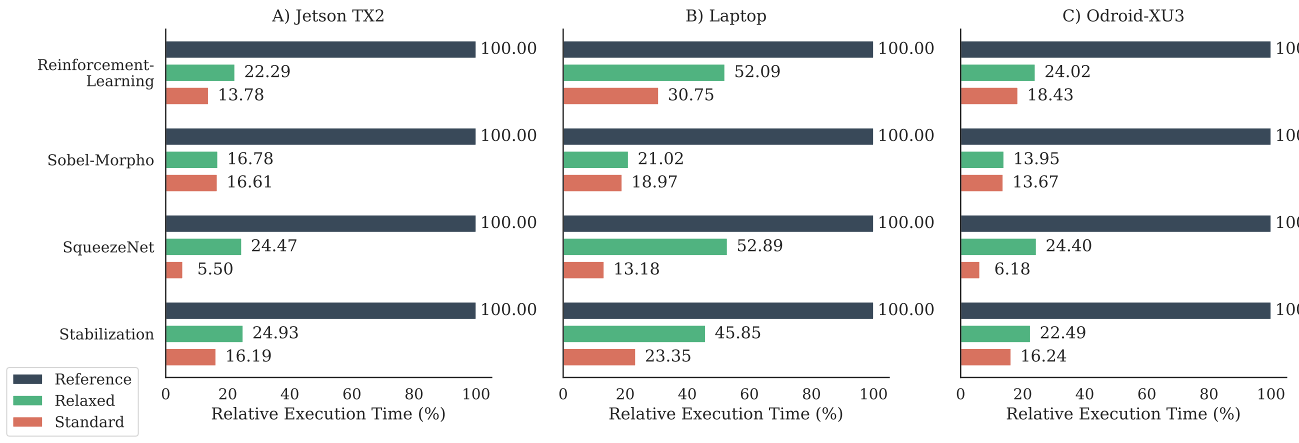

Figure 12: Relative total execution time, intermediate representation building time + scheduling time, for the 3 platforms.

Table 6: Relative change in schedule latency (%) for the different configurations.

	Laptop		Jetson TX2		ODROID-XU3	
Application	Num-R	Num-S	Num-R	Num-S	Num-R	Num-S
SqueezeNet	**0.11**	0.22	**-0.05**	0.22	**0.07**	4.30
Reinforcement Learning	**1.36**	8.19	**3.10**	22.25	**1.61**	25.71
Stabilization	5.45	5.45	4.55	4.55	9.20	9.20
Sobel-Morpho	−2.23	−2.23	4.86	4.86	0.00	0.00
Average	**1.17**	2.91	**3.12**	7.97	**2.72**	9.80

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 732105. and from the French Agence Nationale de la Recherche under grant ANR-15-CE25-0015 (ARTEFaCT project).

REFERENCES

[1] Matin Abadi et al. 2016. TensorFlow: A system for large-scale machine learning. 265–283.
[2] Florian Arrestier, Karol Desnos, Maxime Pelcat, Julien Heulot, Eduardo Juarez, and Daniel Menard. 2018. Delays and states in dataflow models of computation. In *Proceedings of the 18th International Conference on Embedded Computer Systems Architectures, Modeling, and Simulation - SAMOS '18*. ACM Press, Pythagorion, Greece, 47–54. https://doi.org/10.1145/3229631.3229645
[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. (2009), 16.
[4] Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya. 2001. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing* 49, 10 (2001), 2408–2421.
[5] Shuvra S Bhattacharyya, Edward A. Lee, and Praveen K. Murphy. 1996. *Software Synthesis from Dataflow Graphs.* Kluwer Academic Publishers, Norwell, MA, USA.
[6] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. 1996. Cycle-static dataflow. *IEEE Transactions on Signal Processing* 44, 2 (Feb. 1996), 397–408. https://doi.org/10.1109/78.485935
[7] Morteza Damavandpeyma, Sander Stuijk, Twan Basten, Marc Geilen, and Henk Corporaal. 2013. Schedule-Extended Synchronous Dataflow Graphs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 10 (Oct. 2013), 1495–1508. https://doi.org/10.1109/TCAD.2013.2265852
[8] Hamza Deroui, Karol Desnos, Jean-François Nezan, and Alix Munier-Kordon. 2017. Relaxed Subgraph Execution Model for the Throughput Evaluation of IBSDF Graphs. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS).*
[9] Karol Desnos, Maxime Pelcat, Jean-François Nezan, Shuvra S. Bhattacharyya, and Slaheddine Aridhi. 2013. Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on.* IEEE, 41–48.
[10] Pascal Fradet, Alain Girault, and Peter Poplavko. 2012. SPDF: A schedulable parametric data-flow MoC. In *Proceedings of the Conference on Design, Automation and Test in Europe.* EDA Consortium, 769–774.
[11] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. 2013. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on.* IEEE, 1299–1308.
[12] Kronos Group. 2013. The OpenVX API for hardware acceleration. In *http://www.khronos.org/openvx.*
[13] Julien Heulot, Maxime Pelcat, Karol Desnos, Jean-François Nezan, and Slaheddine Aridhi. 2014. Spider: A synchronous parameterized and interfaced dataflow-based rtos for multicore dsps. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in.* IEEE, 167–171.
[14] J. Keinert, C. Haubelt, and J. Teich. 2006. Modeling and Analysis of Windowed Synchronous Algorithms. In *2006 IEEE International Conference on Acoustics Speed and Signal Processing Proceedings*, Vol. 3. IEEE, Toulouse, France, III–892–III–895.

https://doi.org/10.1109/ICASSP.2006.1660798

[15] Y.-K. Kwok. 1997. *High-performance algorithms of compile-time scheduling of parallel processors*. Ph.D. Dissertation. Hong Kong University of Science and Technology. Advisor(s) Ahmad, Ishfaq.

[16] Edward A. Lee and David G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[17] Edward A. Lee and Thomas M. Parks. 1995. Dataflow process networks. *Proc. IEEE* 83, 5 (1995), 773–801.

[18] Maxime Pelcat, Karol Desnos, Julien Heulot, Clément Guy, Jean-François Nezan, and Slaheddine Aridhi. 2014. Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*. IEEE, 36–40.

[19] Jonathan Piat, Shuvra S. Bhattacharyya, and Mickaël Raulet. 2009. Interface-based hierarchy for synchronous data-flow graphs. In *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*. IEEE, 145–150.

[20] José Luis Pino, Shuvra S. Bhattacharyya, and Edward A. Lee. 1995. *A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs*. Electronics Research Laboratory, College of Engineering, University of California.

[21] Sebastian Ritz, Matthias Pankert, V. Zivojinovic, and Heinrich Meyr. 1993. Optimum vectorization of scalable synchronous dataflow graphs. In *Application-Specific Array Processors, 1993. Proceedings., International Conference on*. IEEE, 285–296.

[22] Jiahao Wu, Timothy Blattner, Walid Keyrouz, and Shuvra S. Bhattacharyya. 2018. A design tool for high performance image processing on multicore platforms. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Dresden, Germany, 1304–1309. https://doi.org/10.23919/DATE.2018.8342215

[23] George F. Zaki, William Plishker, Shuvra S. Bhattacharyya, and Frank Fruth. 2012. Partial Expansion Graphs: Exposing Parallelism and Dynamic Scheduling Opportunities for DSP Applications. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, Delft, Netherlands, 86–93. https://doi.org/10.1109/ASAP.2012.14

[24] George F. Zaki, William Plishker, Shuvra S. Bhattacharyya, and Frank Fruth. 2017. Implementation, Scheduling, and Adaptation of Partial Expansion Graphs on Multicore Platforms. *Journal of Signal Processing Systems* 87, 1 (April 2017), 107–125. https://doi.org/10.1007/s11265-016-1107-8