TUCS

Sudeep Kanur | Johan Lilius | Johan Ersfolk

Detecting Data-Parallel Synchronous Dataflow Graphs

Turku Centre *for* Computer Science

TUCS Technical Report
No 1184, September 2017

Detecting Data-Parallel Synchronous Dataflow Graphs

Sudeep Kanur
Åbo Akademi University, Informationsteknologi
Domkyrkotorget 3, 20500 Turku, Finland
skanur@abo.fi

Johan Lilius
Åbo Akademi University, Informationsteknologi
Domkyrkotorget 3, 20500 Turku, Finland
jolilius@abo.fi

Johan Ersfolk
Åbo Akademi University, Informationsteknologi
Domkyrkotorget 3, 20500 Turku, Finland
jersfolk@abo.fi

TUCS Technical Report
No 1184, September 2017

Abstract

Synchronous Dataflow (SDF), a popular subset of the dataflow programming paradigm, gives a well structured formalism to capture signal and stream processing applications. With data-parallel architectures becoming ubiquitous, several frameworks leverage the SDF formalism to map applications to parallel architectures. But, these frameworks assume that the Synchronous Dataflow graphs (SDFGs) under consideration already are data-parallel. In this paper, we address the lack of mechanisms required to detect if an SDFG can be executed in a data-parallel fashion. We develop necessary and sufficient conditions that an SDFG must satisfy for its data-parallel execution. In addition, we develop methods that detect and transform SDFGs that cannot be determined to be data-parallel through visual graph inspection alone. We report on a prototype implementation of the developed conditions as a compiler pass in PREESM framework and test them against some useful applications expressed as an SDFG.

Keywords: Synchronous Dataflow graphs (SDFGs), PREESM, Data-Parallelism

TUCS Laboratory
Embedded Systems Laboratory

1 Introduction

Modern heterogeneous multi-cores are packed with a diverse mix of architectural features such as vector-based units, graphics processing cores, and special functional units with the aim to exploit data-parallelism of an application. Manually developing optimized applications for heterogeneous multi-cores is difficult due to complex partitioning and synchronization rules associated with the application and the architectures. Describing an application using the dataflow paradigm allows us to automate scheduling and mapping of applications across multi-cores. In addition, the dataflow paradigm allows portable application to architecture mapping that can be optimized for various goals such as latency, throughput and/or energy consumption.

Several tools and compiler techniques have been developed around Synchronous Dataflow (SDF) [1], a subset of the dataflow paradigm, making it a popular choice for describing signal processing, multi-media and other stream processing applications [2]. Frameworks such as StreamIT [3], DIF-GPU [4], DAL [5], etc. allow describing an application as an Synchronous Dataflow graph (SDFG) and provide scheduling and mapping of the graphs on architectures such as graphics processing units and multi-core central processing units. However, the scheduling and mapping techniques assume that the SDFG under consideration is data-parallel. Thus, SDFGs that have limited data-parallelism due to application structure or legacy reasons cannot be handled by these frameworks.

A first step towards automatic detection of data-parallel regions in an SDFG is to determine when an SDFG can be executed in a data-parallel fashion. Automatic detection rules can pave a way for compilers that can not only detect data-parallel regions within an SDFG, but also segregate them from the rest of the network and handle them efficiently. In this direction, our main contribution of this paper are summarised as follows. 1. We have formulated necessary and sufficient conditions that an SDFG must fulfil in order for it to be executed in a data-parallel fashion. 2. We report a prototype implementation of the formulated conditions as a compiler pass in PREESM framework [6] and the implementation is available as open-source software [7]. 3. We validate the implementation against two practical non-trivial SDFGs and show its capabilities for two cases – when the SDFG is data-parallel and when the SDFG cannot be executed in a data-parallel fashion.

The results obtained in this paper look very similar to that of polyhedral process networks [8], derived by applying state of the art polyhedral techniques and tools on static affine nested loops. The work in [9] goes further and develops abstractions using dataflow techniques for capturing and expressing data-parallelism on top of a polyhedral process network. The key difference between our work and the work using polyhedral techniques arrives from the fact that polyhedral techniques and by extension work in [9] derive dataflow graphs from well structured finite nested loops, restricting the structure of the dataflow graphs. SDFG on the other hand can have an arbitrary graph structure containing arbitrary delays at edges and thus, the results of [8] does not automatically apply to SDFGs.

The remainder of the paper is structured as follows. Section 2 provides necessary background on SDFGs and defines when an SDFG can be executed in data-parallel fashion. Section 3 gives the necessary and sufficient conditions required by an SDFG for data-parallel execution. We also investigate various properties of a data-parallel SDFGs through three lemmas. Section 4 reports the implementation of a compiler pass that uses the results obtained in this paper and tests on some useful SDFG graphs. Validation is carried out on both data-parallel and non-data-parallel SDFGs to show the extent of the capabilities of the compiler pass.

2 Background

2.1 Synchronous Dataflow graph (SDFG) and its Properties

Representing an application as an SDFG entails dividing up the application into a number of nodes called *actors* and connecting them with directed first-in first-out (FIFO) buffers. Each actor of the SDFG consumes data samples, referred here as *tokens*, for its execution. In effect, an actor can produce tokens to the connected FIFO that in turn can be consumed by another actor connected to the same FIFO.

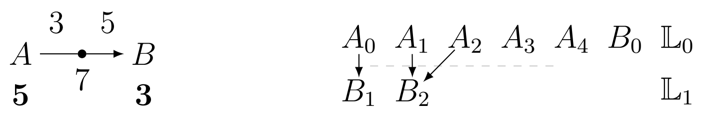
Definition 1 (Actors)**.** *An actor has a finite (possibly empty) set of input ports $\mathbb{P}_{in}$ and a finite (possibly empty) set of output ports $\mathbb{P}_{out}$ with $\mathbb{P}_{in} \cap \mathbb{P}_{out} = \emptyset$. The production and consumption of tokens for an actor is* fixed.

An actor A is termed as a *source* actor if $\mathbb{P}_{in} = \emptyset$ and can unconditionally produce output tokens, while an actor is termed as a *sink* actor if $\mathbb{P}_{out} = \emptyset$ and can unconditionally consume input tokens. In this paper, the scope of actors is restricted to be *atomic*, wherein the actors themselves do not contain other actors or SDFGs.

Definition 2 (SDFG)**.** *A Synchronous Dataflow graph (SDFG) $G = (\mathbb{V}, \mathbb{E})$ is a tuple of a finite set of actors $\mathbb{V}$ acting as vertices and a finite set of FIFO buffers $\mathbb{E}$ as directed edges between the actors. A FIFO buffer $b \in \mathbb{E}$ connects the output of an actor $A \in \mathbb{V}$ to the input of an actor $B \in \mathbb{V}$. All the FIFO buffers $b \in \mathbb{E}$ are connected to some actor in $\mathbb{V}$.*

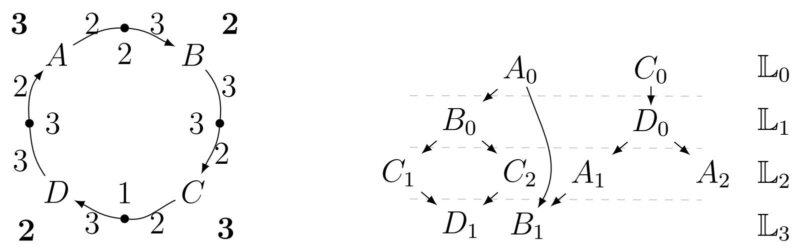
A FIFO buffer, $b \in \mathbb{E}$, connecting $A, B \in \mathbb{V}$, has a production rate ρ_b and consumption rate κ_b if the source of b A produces ρ_b tokens and destination of b, B, consumes κ_b tokens. Prior to execution of the SDFG, b can be initialized with *delay* tokens δ_b. If $\mathbb{Z}^+$ is a set of positive integers, then $\rho_b, \kappa_b \in \mathbb{Z}^+$, $\delta_b \in \mathbb{Z}^+ \cup \{0\}$.

An example SDFG containing two actors is shown in Figure 1a. A single FIFO buffer $b = (A, B)$ connects actors $A, B \in \mathbb{V}$ with rates $\rho_b = 3$, $\kappa_b = 5$ and $\delta_b = 7$.


3
5
A
B
7
5
3
A_0 A_1 A_2 A_3 A_4 B_0 $\mathbb{L}_0$
B_1 B_2 $\mathbb{L}_1$

(a) SDFG (b) DAG

Figure 1: An acyclic SDFG with two actors and its rearranged DAG.



(a) SDFG (b) DAG

Figure 2: A strictly cyclic SDFG that does not satisfy Lemma 1.

Definition 3 (Precedence, $\prec$)**.** *A node A precedes node B if there exists an edge (A, B) and is denoted by $A \prec B$. ($\prec^\star$) gives the transitive closure relation on ($\prec$). Conversely $\succ$ and $\succ^\star$ denotes successor relationship and transitive closure relation on $\succ$.*

Definition 4 (Path)**.** *A* path *is a finite sequence of nodes $\mathcal{T} = [A \prec \cdots \prec B]$ if there exits edges that eventually connect the node A to node B. A* cycle *is a path with $\mathcal{T} = [A \prec \cdots \prec A]$. A cycle is a* self loop *if $\mathcal{T} = [A \prec A]$.*

For the purpose of analysis, the actors are considered to be stateless i.e., actors do not store any state that changes across their execution. An actor with a state can be converted to a stateless actor by adding a self loop containing a delay token representing the initial state.

Definition 5 (Acyclic SDFG)**.** *An SDFG is acyclic if the SDFG contains no cycles in it. An acyclic SDFG contains at least one source actor and one sink actor.*

Definition 6 (Strictly cyclic SDFG)**.** *An SDFG is strictly cyclic if the SDFG contains exactly one cycle and all the actors of the SDF graph is a part of the cycle.*

Figure 1a is an example of an acyclic SDFG, while Figure 2a is an example of a strictly cyclic SDFG.

As the production and consumption rate of FIFO buffers are known *a priori*, we can check if a given SDFG is consistent.

Definition 7 (Consistent SDFG)**.** *A consistent SDFG has a periodic admissible schedule (PASS) with infinite sequence of actor execution that 1. can execute all* *actors $A \in \mathbb{V}$ exactly $\mathbf{q}(A)$ times without causing a* deadlock*, and 2. the size of b i.e., $|b|$ is bounded to store the intermediate tokens produced by the actors.*

The quantity $\mathbf{q}(\mathbf{A})$ is called repetition factor of $A \in \mathbb{V}$ and $\mathbf{q}$ is a vector of repetition factors of all the actors in $\mathbb{V}$. For Figure 1a, the repetition rates are $\mathbf{q}(A) = 5$ and $\mathbf{q}(B) = 3$.

2.2 Directed Acyclic Graph (DAG) and its Properties

While PASS gives a sequential schedule for an SDFG, the task level and data level parallelism of the actors in the PASS can be seen by constructing a DAG according to Definition 8. The DAG exposes the dependencies between the instances of the actors while preserving the semantics of the execution of the corresponding PASS of the SDFG. In this paper, a DAG is constructed from a corresponding SDFG and as instance level information is exposed in a DAG it is free of delay tokens. Hence, execution of an SDFG is equivalent to the execution of its corresponding DAG.

Definition 8 (DAG)**.** *A Directed Acyclic Graph (DAG) is a pair of sets, $DAG = (\mathbb{V}_{dag}, \mathbb{E}_{dag})$, where $\mathbb{V}_{dag}$ consists of $\mathbf{q}(A)$ instances $A \in \mathbb{V}$. $\mathbb{E}_{dag}$ consists of edges between all the instances of actors in G. For a buffer $b = (A, B) \in \mathbb{E}$, an edge is placed from an instance A_i to instance B_j if $\forall u, 0 \leq u < |b|$ satisfies the condition given below [10].*

$$|b| = \mathbf{q}(A) \cdot \rho_b, \quad v = \text{modulo}(\delta_b + u, |b|)$$
$$i = \lfloor \frac{u}{\rho_b} \rfloor, \quad j = \lfloor \frac{v}{\kappa_b} \rfloor$$

Let $\mathbb{I}$ be the set of entry (root) nodes of the DAG, $\mathbb{R}$ be the set containing actors of nodes in $\mathbb{I}$, and $\mathbb{X}$ be the set of exit nodes of the DAG. They are constructed according to Definition 9 given below. Further, for a non-empty DAG, $\mathbb{I}$, $\mathbb{R}$ and $\mathbb{X}$ are also non-empty.

The function $\text{actor}(A)$ returns the actor associated with the instance node A in the DAG. At this point, we clarify that the notation A, B denotes generic nodes in a graph. The type of the node is either explicitly stated or implicitly obtained by its association with a well defined set. For e.g. $A \in \mathbb{V}$ is an actor by the definition of $\mathbb{V}$ and $A \in \mathbb{V}_{dag}$ is an instance by the definition of $\mathbb{V}_{dag}$.

Definition 9 (Root and Exit nodes)**.** *$\mathbb{I}$ is defined as the set of nodes of a DAG that does not have predecessor nodes, $\mathbb{I} = \{A \mid A \in \mathbb{V}_{dag}, \forall B \in \mathbb{V}_{dag}, B \not\prec A\}$. $\mathbb{X}$ is defined as the set of nodes that does not form predecessor nodes, but forms a successor of some node, $\mathbb{X} = \{A \mid A \in \mathbb{V}_{dag}, \exists C \in \mathbb{V}_{dag}, C \prec A, \forall B \in \mathbb{V}_{dag}, A \not\prec B\}$. Lastly, $\mathbb{R} = \{\text{actor}(A) \mid A \in \mathbb{I}\}$.*

To examine dependencies between instances across actors of an SDFG, we define a predecessor branch set. A predecessor branch set, $\mathbb{B}_{X,I}$, contains all the paths originating from a node $X \in \mathbb{V}_{dag}$ to a node $I \in \mathbb{V}_{dag}$.

Definition 10 (Predecessor Branch Set)**.** *A predecessor branch set between an instance X and I is given by $\mathbb{B}_{X,I} = \{[X \succ \mathbb{V}'_{dag} \succ I] \mid \mathbb{V}'_{dag} \subseteq \mathbb{V}_{dag} - \{X, I\}, \forall A, B, C \in \mathbb{V}'_{dag}, A \succ B \wedge A \succ C \implies B = C\}$.*

Further, we define $\mathbb{B}$ as the set of all predecessor branch sets taken between all root nodes and all of exit nodes i.e. $\mathbb{B} = \{\mathcal{T} \mid X \in \mathbb{X}, I \in \mathbb{I}, \mathcal{T} \in \mathbb{B}_{X,I}\}$. $\mathcal{T}$ denotes the sequence of instances seen in a predecessor branch and $|\mathcal{T}|$ gives the number of instances seen in the *path*.

To see which instances are ready to be executed, we group the DAG into level sets such that each set contains all the instances whose input dependencies are met. In other words, there is no backward edge originating from the current level that connects any instance in the previous levels. Thus instances in one level set can be fired in any order and there are exactly $\hat{l}$ levels given by Definition 11.

Definition 11 (Level)**.** *A level of a node A in a DAG is the longest path seen from an instance A to any root node i.e. $l_A = \max(\{|\mathcal{T}| \mid \mathcal{T} \in \mathbb{B}_{A,I}, I \in \mathbb{I}, I \prec^\star A\})$. The maximum level of a DAG is given by $\hat{l} = \max(\{|\mathcal{T}| \mid \mathcal{T} \in \mathbb{B}\})$.*

Definition 12 (Level Set)**.** *A level set is a two part construction process. First we construct a set $\mathbb{L}_l = \{A \mid A \in \mathbb{V}_{dag}, l = l_A\}$. Then we construct the level set $\mathbb{L} = \{\mathbb{L}_l \mid 0 \leq l \leq \hat{l}\}$.*

The root and exit node sets according to Definition 9 for Figure 2b are given as $\mathbb{R} = \{A, C\}$, $\mathbb{I} = \{A_0, C_0\}$ and $\mathbb{X} = \{D_1, B_1, A_2\}$. The predecessor branch set $\mathbb{B}_{D_1,A_0} = \{[D_1, C_1, B_0, A_0], [D_1, C_2, B_0, A_0]\}$. The level set $\mathbb{L}_0 = \mathbb{I}$, $\mathbb{L}_{\hat{l}} \subseteq \mathbb{X}$, with $\hat{l} = 3$. For a level $l = 2$, we can compute $\mathbb{L}_2 = \{C_1, C_2, A_1, A_2\}$.

From Figure 2b, we can verify that each level set depends on the previous level set for its execution and all the nodes in $\mathbb{L}_l$ can be executed in any order within the set. We extend the property of unordered execution within a level set to define data-parallelism. As all of the input dependencies in a certain level set $\mathbb{L}_l$ are met, all the nodes in $\mathbb{L}_l$ can be executed in parallel. Hence, an actor that has all its instances contained in a unique level set can be executed in parallel. When this is true for all actors of an SDFG, then we can say that the SDFG can be executed in data-parallel fashion.

Definition 13 (Data-parallel DAG)**.** *A DAG is data-parallel if for each actor of the SDFG, all of its instances are contained in some unique level. Formally, a data-parallel DAG satisfies the condition give below.*

$$\forall A \in \mathbb{V}, \exists l \mid 0 \leq l \leq \hat{l}, \forall i, 0 \leq i < \mathbf{q}(A), A_i \in \mathbb{L}_l \tag{1}$$

We denote a DAG that satisfies (1) as DAG_{par}. Definition 13 is a stricter definition of data-parallelism and it can be easily seen that a DAG that violates Definition 13 can still be executed in data-parallel fashion. For instance, the DAG in Figure 1b violates Definition 13, while the SDFG can be executed in data-parallel fashion. However, we show in the following section that instance B_0 can be placed in $\mathbb{L}_1$, resulting in Figure 4a that maintains the Definition 13.

3 Conditions for data-parallelism

A consequence of a data-parallel SDFG is that for any given actor of the SDFG, no instance of the actor depends on another instance belonging to the same actor. If an instance of an actor $A \in \mathbb{V}$ indeed depends on some other instance of the same actor, then we cannot group all the instances of A in a unique level set, violating Definition 13. Definition 14 gives a formal definition of this condition by examining the predecessor branch set $\mathbb{B}$. If there is a dependency between two instances of any actor, then there must be at least one predecessor branch where the dependency is seen. A DAG that satisfies Definition 14 contains no such branch.

Definition 14 (Instance independent DAG). $DAG_{ind} = (\mathbb{V}_{dag}, \mathbb{E}_{dag})$ *is a DAG of a consistent SDFG when it satisfies the condition given below.*

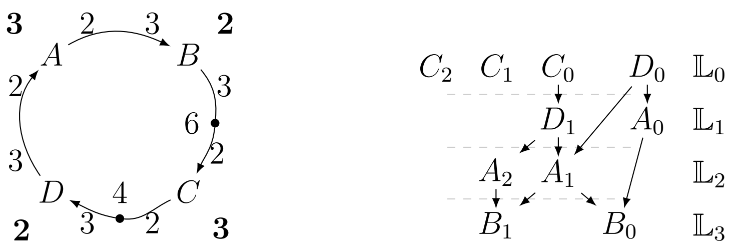
$$\forall [A \prec^{\star} B] \in \mathbb{B} \mid A, B \in \mathbb{V}_{dag}, \mathrm{actor(A)} \neq \mathrm{actor(B)} \tag{2}$$

Claim 1 formally states that any DAG that is instance independent across the same actor is also data-parallel and to show this, we develop three lemmas. Due to the lack of presentation space, the proofs are given in [11].

Claim 1. *The necessary and sufficient condition for executing all the instances of actors of a consistent SDFG graph in a data-parallel fashion is that in the corresponding DAG of the SDFG graph no instance of an actor depends on a previous instance of the same actor i.e.*

$$DAG_{ind} \iff DAG_{par}.$$

As noted in the explanation of Definition 13, SDFGs with DAG_{ind} are not DAG_{par} by default and thus, need DAG transformations. We develop transformations by analysing properties of acyclic and cyclic graphs in Sections 3.1 and 3.2 separately. The final transformation algorithm developed in the paper can be seen as a retiming transformation applied on SDFG. Retiming transformations arrange the delay tokens at different edges in the SDFG to achieve schedules optimized for various goals such as throughput and latency [12, see section 2.5.2]. We show that for a DAG that satisfies Definition 14 has a deterministic retiming transformation that results in a DAG satisfying Definition 13.



(a) SDF (b) DAG

Figure 3: A strictly cyclic SDF that satisfies Lemma 1

3.1 Data-parallelism in acyclic SDFGs

Acyclic SDFGs form the simplest case of SDFGs where the actors can always execute in a data-parallel fashion. Recall that tokens originate from the source actors and terminate at the sink actors, producing and consuming indefinite tokens respectively. Thus, we can always postpone actor execution until the required amount of tokens are available at the input of an actor such that all of its instances can be executed simultaneously. In other words, all the instances of actors in an SDFG are contained in unique level sets by construction.

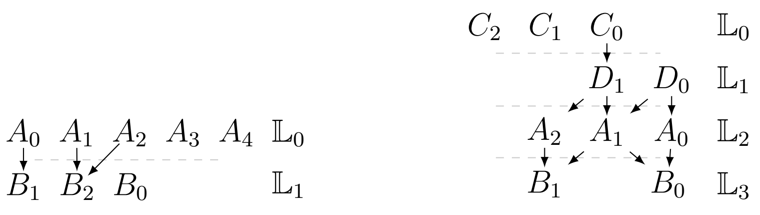
In the presence of delay tokens at some edges in an SDFG, the set $\mathbb{I}$ may contain instances from actors other than the source actors. In Figure 1b the instance $B_0 \in \mathbb{I}$ due to the presence of delays and as a consequence is in $\mathbb{L}_0$. Instead of executing B_0 at earliest opportunity, we move B_0 to $\mathbb{L}_1$ to get Figure 4a. Lemma 1 states that this can be achieved using Algorithm 1.

Lemma 1 (Rearranging Acyclic SDFG)**.** *If a DAG_{ind} corresponds to an acyclic SDFG or a cyclic SDFG that has enough delay tokens at an actor A such that all the instances of A can be executed without executing A's predecessors, then the resulting $\mathbb{L}$ of DAG_{ind} can be rearranged according to Algorithm 1 to obtain DAG_{par}.*

Lemma 1 not only covers acyclic SDFG, but also those cyclic SDFGs that has at least one FIFO with enough tokens such that all the instances of the actor following the FIFO can be executed at a time. Figure 3a is an example of a cyclic SDFG that comes under the scope of Lemma 1. Although the SDFG is similar to that of Figure 2a, due to the presence of enough tokens at $b = (B, C)$, all the instances of C can be fit in $\mathbb{L}_0$ resulting in a DAG that has no path leading from any instance of B to any instance of C. The DAG behaves as if there is no FIFO b. Applying Algorithm 1 on the original DAG, shown in Figure 3b, results in the DAG that satisfies Definition 13, as shown in Figure 4b.

Algorithm 1 Rearrange $\mathbb{L}$ of an acyclic-like SDFG

```
1:  procedure REARRANGE($\mathbb{B}, \mathbb{L}, \mathbb{I}$)
2:      for all $A \mid A \in \mathbb{I}$ do
3:          $\tilde{l} \leftarrow \max$(level of all instances of $\text{actor}(A)$)
4:          if $\tilde{l} > 0$ then
5:              $\mathbb{L} \leftarrow$SORT_INSTANCES($A, \mathbb{L}, \mathbb{B}$)
6:      return $\mathbb{L}$
7:  function SORT_INSTANCES($A, \mathbb{L}, \mathbb{B}$)
8:      for all $\mathcal{T} \mid \mathcal{T} =$ paths from $A$ to its exit nodes do
9:          for all $B \mid B$ = instances in path $\mathcal{T}$ do
10:             $\tilde{l} \leftarrow \max$(level of all instances of $\text{actor}(B)$)
11:             if $\tilde{l} >$ level of actor $B$ then
12:                 Shift $B$ to level set $\mathbb{L}_{\tilde{l}}$
13:     return $\mathbb{L}$
```


A_0 A_1 A_2 A_3 A_4 $\mathbb{L}_0$
B_1 B_2 B_0 $\mathbb{L}_1$
C_2 C_1 C_0 $\mathbb{L}_0$
D_1 D_0 $\mathbb{L}_1$
A_2 A_1 A_0 $\mathbb{L}_2$
B_1 B_0 $\mathbb{L}_3$

(a) (b)

Figure 4: DAGs of Figure 1b and Figure 3b rearranged according to Algorithm 1

3.2 Data-parallelism in strictly cyclic SDFGs

Algorithm 1 is not adequate to handle SDFGs containing a cycle that do not satisfy Lemma 1. For instance, applying Algorithm 1 on the DAG in Figure 2b results in a DAG that does not satisfy Definition 13. In this section, we consider strictly cyclic SDFGs that does not satisfy the preconditions of Lemma 1, but form an instance independent DAG and make additional observations about the associated DAGs. In particular, Lemma 3 states what happens when a cyclic SDFG with an instance independent DAG is rearranged and the observations from Lemma 2 makes rearranging such a DAG possible in the general case.

Lemma 2 (Multiple Roots for a Cyclic SDFG)**.** *If DAG_{ind} corresponds to a strictly cyclic SDFG such that no actor of the SDFG has the required delay tokens to execute all of its instances, then $|\mathbb{R}| > 1$ and ($|\mathbb{I}| \geq m \vee |\{\mathbb{B}\}| \geq m$), where $m = \min(\mathbf{q}(A)), \forall A \in \mathbb{V}$.*

The implication of Lemma 2 is that an SDFG that does not come under the purview of Lemma 1, but satisfies Definition 14 will always contain two or more

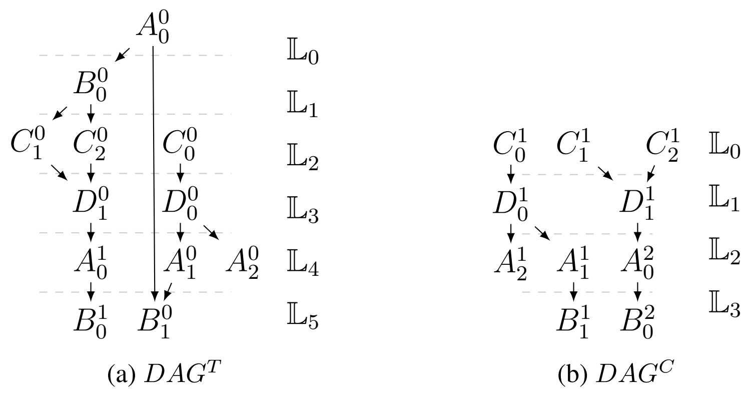


Figure 5: Transient and cyclic DAG after applying Algorithm 2. Superscripts denotes the iteration count. Note that the explicit dependency between A_0 and B_1 is transformed into an implicit one i.e. A_0^i always executes before B_1^i.

actors in its root nodes. Further, the number of instances at root will be greater than m, and/or there will be more than m paths originating from $\mathbb{I}$ to $\mathbb{X}$. The proof of Lemma 2 shows that violating these conditions will result in a DAG that has instance dependencies within an actor (see [11]).

To see Lemma 2 in action, consider Figure 2b and note that not all instances of any actor is in $\mathbb{I}$. Also note that $m = 2$, $|\mathbb{I}| = 2$ and $|\mathbb{B}| = |\{[D_1 \succ C_1 \succ^\star A_0], [D_1 \succ C_2 \succ^\star A_0], [B_1 \succ^\star A_0], [B_1 \succ^\star C_0], [A_2 \succ^\star C_0]\}| = 5$. If either $|\mathbb{I}|$ or $|\mathbb{B}| < 2$, then some instances of actor A and C will occur in different levels, but in the same precedence branch resulting in violation of Definition 14.

Lemma 3 (Rearranging Strictly Cyclic SDFG)**.** *If DAG_{ind} corresponds to a strictly cyclic SDFG, G, such that no actor has required delay tokens to execute all of its instances, then the resulting $\mathbb{L}$ of DAG_{ind} can be rearranged such that there is at least one level $\mathbb{L}_l$ that consists of all the instances of some actor in G.*

Lemma 3 directly follows from the observations from Lemma 2. As there are instances of at least two actors in the first level set, we can fix one instance of one actor and rearrange the instances of the rest according to Lemma 1. To continue with the example DAG of Figure 2b, if we fix instance A_0 to $\mathbb{L}_0$ and rearrange instances of C_0, we can see that all the instances of the actor C are contained in level $l = 2$.

3.3 Data-parallelism in generic SDFGs

The DAG obtained after rearranging a cyclic SDFG may not be data-parallel. Let $\bar{l}$ denote the level at which all the instances of some actor occur in the same level, according to Lemma 3. If $\bar{l} > 0$, as is the case in Figure 2b (in this case $\bar{l} = 2$),

Algorithm 2 Get data-parallel DAG for SDFG with DAG_{ind}

```
1: procedure DATA_PARALLEL($G$, $DAG$, $\mathbb{L}$, $\mathbb{B}$, $\hat{l}$)
2:     $DAG^C = DAG$, $DAG^T = \emptyset$
3:     $\mathbb{D} \leftarrow$ All instances of cycles in DAG
4:     if $\mathbb{D} = \emptyset$ then                                  ▷ SDFG is acyclic-like
5:         $\mathbb{L} \leftarrow$ REARRANGE($\mathbb{B}$, $\mathbb{L}$, $\mathbb{I}$)
6:         return($DAG^T, DAG^C, \mathbb{L}^T = \emptyset, \mathbb{L}^C = \mathbb{L}$)
7:     $\mathbb{L}^T \leftarrow \mathbb{L}$, $\mathbb{L}^C \leftarrow \mathbb{L}$, $DAG^T = DAG$
8:     for all $\mathbb{D}' \mid \mathbb{D}'$ = instances of a cycle do
9:         $A \leftarrow$ fix an actor from $\mathbb{R}$ of the cycle
10:        $\mathbb{I}'' \leftarrow$ rest of the instances of the cycle and instances from acyclic part of SDFG, excluding instances from source actors
11:        $\mathbb{L} \leftarrow$ REARRANGE($\mathbb{B}, \mathbb{L}, \mathbb{I}''$)
12:        $\bar{l} \leftarrow$ level at which all instances of some actor is in same level set
13:        for $l := 0$ to $\bar{l} - 1$ do
14:            for all $A \mid A \in \mathbb{L}_l$ do                       ▷ $A$ is an instance
15:                $\mathbb{P} \leftarrow$ instance of an actor preceding $A$ and is in the path of $A$ to $A$'s exit nodes
16:                Place $A$ after all $\mathbb{P}$ in $DAG^T$. Adjust the level sets
17:                Remove $A$ in $DAG^C$. Adjust the level sets
    return ($DAG^T, DAG^C, \mathbb{L}^T, \mathbb{L}^C$)
```

then the DAG is not data-parallel. In other words, rearranging levels of a DAG_{ind} is not sufficient to make it DAG_{par} when there are cycles in the SDFG.

Algorithm 2 breaks the DAG into two parts: a transient part DAG^T that is executed initially and a periodic part DAG^C that gets executed indefinitely. DAG^T retimes the delays of the original DAG such that execution of DAG^T results in DAG^C with delays at appropriate buffers. Thus, DAG^C satisfies Lemma 1 and by extension is data-parallel. Algorithm 2 safely moves some instances from DAG^C to DAG^T to make $DAG^C = DAG_{par}$. For Figure 2b, the rearranged DAGs after applying Algorithm 2 is given in Figure 5.

When an SDFG is acyclic, $\mathbb{D} = \emptyset$ and the DAG_{ind} is rearranged according to Lemma 1 to yield DAG_{par}. When an SDFG contains cycles, line 4 in Algorithm 2 is not satisfied and the algorithm proceeds to process cycles, one at a time. Line 10 filters $\mathbb{I}$ to exclude instances from the source actors and other cycles so as to not to interfere with the rearranging operation. The consequence of this is that when an SDFG contains a mix of cyclic and acyclic paths, line 11 rearranges the instances of actors in the acyclic paths on the first iteration itself.

Construction of DAG^T and DAG^C is done between line 13-17. For each instance A in the level set under consideration, we find the instances $\mathbb{P}$ that belongs to actors that precedes $\text{actor}(A)$ in the original SDFG and is in the path of A to its exit nodes. Instance A is added to DAG^T and the edges are placed between A

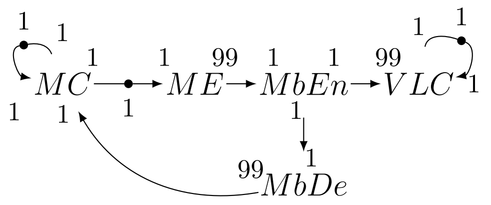


Figure 6: An SDFG of a H263 encoder. The repetition rates are given as $\mathbf{q}(MbEn) = \mathbf{q}(MbDe) = 99$ and 1 for the rest of the actors.

and all of $\mathbb{P}$. Simultaneously, the instance A and all the edges originating from it are removed from DAG^C. The level sets of both the DAGs are adjusted to reflect the addition and removal of A. Applying Algorithm 2 on the DAG of Figure 2b results in a transient and a cyclic DAG shown in Figure 5.

4 Implementation and Validation

A compiler pass to check whether a given SDFG is data-parallel according to Definition 14 and transform its DAG according to Algorithm 2 was implemented in the PREESM framework [6]. The implementation is available as a open-source software from [7]. The compiler pass accepts an SDFG, constructs a DAG and checks if the DAG satisfies Definition 14. If the DAG is found out not to be instance independent, then it reports actors that have dependent instances.

A rudimentary implementation of the pass requires computing the predecessor branch set, $\mathbb{B}$, for the entire DAG. The computational complexity of obtaining all the paths in a DAG is exponential to the number of vertices and edges, making the computation intractable for large DAGs. Instead, we follow a different approach. First, we construct $|\mathbb{I}|$ subsets of the DAG, where each subset is constructed starting from an instance $I \in \mathbb{I}$ and collecting all the instances that are reachable from I. Then, we observe that if a DAG satisfies Definition 14, then each of the subsets of a DAG, considered separately, must also satisfy Definition 14. Conversely, each of the subsets of a DAG, considered separately is also, data-parallel and thus, all the instances seen in a subset of the DAG must be contained in a unique level set. If this is not the case, then the original DAG does not satisfy Definition 14. Computation of level sets for each subset of the DAG can be done in $O(|\mathbb{V}_{dag}| + |\mathbb{E}_{dag}|)$ by traversing in topological order, making the complexity of the entire algorithm linear in time [13].

We report the validation of the compiler pass for two practical SDFG applications, that form interesting dataflow graphs. Due to the lack of the presentation space, we restrict the discussion to the dataflow graph and omit the details about the actor logic and application logic.

Figure 6 shows an SDFG representation of a H263 encoder [14]. Although

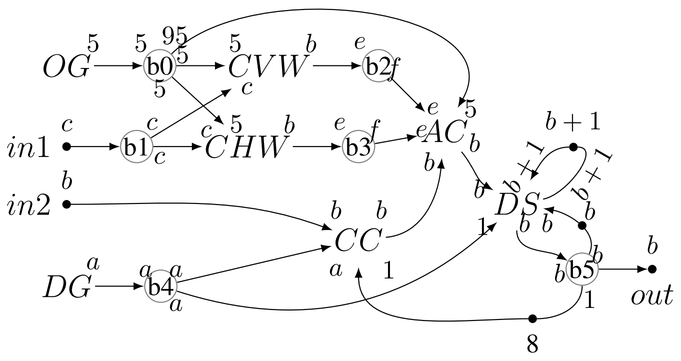


Figure 7: An SDFG of *Cost Work* hierarchical actor of stereo vision application. Circles denote *broadcast* actors and they replicate the tokens at their input port to all of its output ports. Rates are given as $a = 19$, $b = 164920$, $c = 494760$, $d = 95$, $e = 2473800$, $f = 47002200$. The repetition rates are $\mathbf{q}(CVW) = \mathbf{q}(CHW) = \mathbf{q}(b1) = 5$, $\mathbf{q}(CC) = \mathbf{q}(AC) = \mathbf{q}(DS) = \mathbf{q}(b5) = 19$ and 1 for the remaining actors.

the graph has self loops and cycles, the compiler pass correctly recognizes that the SDFG is data-parallel. The loop $\mathcal{T} = [MC, ME, MbEn, MbDe]$ in Figure 6 makes its corresponding DAG equivalent to an acyclic graph due to presence of delay token at the edge (MC, ME) and satisfies Lemma 1. As MC and VLC contains only one instance, the self loop cannot introduce instance-dependencies, thus satisfying Definition 14. As there are no more delay tokens in the SDFG, the compiler pass need not rearrange the DAG resulting in $DAG^C = DAG$.

The second application considered for validation is the stereo vision application from the PREESM application repository [15]. Figure 7 shows the SDFG of one particular hierarchical actor, *Cost Work*, that contains several self loops and cycles. The rest of the graph of the stereo vision application is acyclic and is not shown here. Unlike Figure 6, the loops does not have enough data-tokens to be executed in parallel. The compiler pass correctly determines that the graph cannot be executed in a data-parallel fashion. Further, it lists the actor $\{AC, CC, b5, DS\}$ as the actors that have dependent instances. On closer inspection of the Figure 7, we can see that these actors are part of two cycles and none of their instances are in $\mathbb{I}$.

5 Future work and Conclusion

In this paper, we presented a detailed discussion of data-parallelism arising in SDFGs and presented necessary and sufficient conditions for a data-parallel SDFG. The conditions enabled us to develop a compiler pass that can analyse arbitrary SDFGs for data-parallelism and transform them into a data-parallel representation when required. In addition, the compiler pass can also identify those actors that have dependencies between their instances, paving a way for automatic segregation of the SDFG based on data-parallelism. Extracting non-data-parallel actors from a mixed SDFG without causing a deadlock, is an interesting area for future developments.

A Appendix: Proofs

In this section we re-state the lemmas, claims and algorithms with their associated proofs.

A.1 Proof of Lemma 1

Lemma 1 (Rearranging Acyclic Synchronous Dataflow graph (SDFG))**.** *If a DAG_{ind} corresponds to an acyclic SDFG or a cyclic SDFG that has enough delay tokens at an actor A such that all the instances of A can be executed without executing A's predecessors, then the resulting $\mathbb{L}$ of DAG_{ind} can be rearranged according to Algorithm 1 to obtain DAG_{par}.*

Before we prove Lemma 1, we make another observation about Directed Acyclic Graphs (DAGs) of acyclic-like SDFGs. Note that due to the acyclic nature of the SDFG under consideration, if an instance of actor A belongs to the predecessor path of an instance of an actor B, then reverse cannot happen. Lemma 2 captures this notion in (3).

Lemma 2 (Shared instances in predecessor branch set)**.** *If a DAG_{ind} corresponds to a DAG of an acyclic SDFG or a strictly cyclic SDFG that has enough delay tokens at an actor A such that all the instances of A can be executed without executing A's predecessors, then the following condition is true:*

$$\begin{aligned}\forall X \in \mathbb{X}, \forall A \in \mathbb{R}, \forall B \in \mathbb{R} - \{A\},\\ 0 \le i, i' < \mathbf{q}(A), 0 \le j, j' < \mathbf{q}(B), i \ne i', j \ne j'\\ \forall A_i, B_j \in \mathbb{I}, \forall A_{i'}, B_{j'} \in \mathbb{V}_{dag} - \mathbb{I},\\ \textit{if } \exists B_{j'} \in \mathbb{B}_{X,A_i} \textit{ then } A_{i'} \notin \mathbb{B}_{X,B_j}\end{aligned} \tag{3}$$

Proof of Lemma 2. We prove this by contradiction. If $B'_j \in \mathbb{B}_{X,A_i}$ and $A'_i \in \mathbb{B}_{X,B_j}$, then $A_i \prec^\star B'_j$ and $B_j \prec^\star A'_i$. This violates the DAG_{ind} under consideration, as precedence can only occur in one direction in DAG_{ind} for acyclic-like SDFGs. □

Due to Lemma 2, rearranging a DAG of an acyclic SDFG using Algorithm 1 results in DAG_{par}.

Algorithm 1 Rearrange $\mathbb{L}$ of an acyclic-like SDFG

```
1:  procedure REARRANGE($\mathbb{B}, \mathbb{L}, \mathbb{I}$)
2:      for all $A \mid A \in \mathbb{I}$ do
3:          $\tilde{l} \leftarrow$ MAX_LEVEL($A, \mathbb{B}$)
4:          if $\tilde{l} > 0$ then
5:              $\mathbb{L} \leftarrow$SORT_INSTANCES($A, \mathbb{L}, \mathbb{B}$)
6:      return $\mathbb{L}$
7:  function MAX_LEVEL($B, \mathbb{B}$)
8:      $A \leftarrow \mathrm{actor}(B)$
9:      $\tilde{l} \leftarrow \max(\{|\mathcal{T}| \mid \mathcal{T} \in \mathbb{B}_{A_i,I}, 0 \leq i < \mathbf{q}(A), \forall I \in \mathbb{I}, I \prec^\star A_i\})$
10:     return $\tilde{l}$
11: function SORT_INSTANCES($A, \mathbb{L}, \mathbb{B}$)
12:     for all $\mathcal{T} \mid \mathcal{T} \in \mathbb{B}_{X,A}, X \in \mathbb{X}, A \prec^\star X$ do
13:         for all $B \mid B \in \mathcal{T}$ do
14:             $l \leftarrow |[A \prec^\star B]|$
15:             $\tilde{l} \leftarrow$MAX_LEVEL($B, \mathbb{B}$)
16:             if $\tilde{l} > l$ then
17:                 $\mathbb{L}_l \leftarrow \mathbb{L}_l - \{B\}, \quad \mathbb{L}_{\tilde{l}} \leftarrow \mathbb{L}_{\tilde{l}} \cup \{B\}$
18:     return $\mathbb{L}$
```

Proof of Lemma 1. Let, $\mathbb{I}_S = \{A \mid \forall A \in \mathbb{I}, \mathrm{in}(\mathrm{actor}(A)) = \emptyset\}$ and $\mathbb{I}' = \mathbb{I} - \mathbb{I}_S$ i.e. $\mathbb{I}_S$ is a set of instances of source actors that belong to the root of the DAG and $\mathbb{I}'$ is a set of instances of actors other than source actors.

When $\mathbb{I}' = \emptyset$, it can be trivially seen that the DAG_{ind} results in DAG_{par}. Line 4 in Algorithm 1 is never satisfied, requiring no sorting operation.

When $\mathbb{I}' \neq \emptyset$, the DAG corresponds to a DAG that satisfies Lemma 2. The function $\mathrm{max\_level}$ at line 7 calculates maximum level at which certain instance of an actor is seen. The function $\mathrm{sort\_instances}$ at line 11 sorts all the instances of actor the level seen by $\mathrm{max\_level}$. The operation at line 17 rearranges DAG_{ind} into DAG_{par}. If A is the original source instance that needs to be reassigned to a different level set, the function $\mathrm{sort\_instances}$ conditionally performs this operation for every instance seen in the path of A. Thus, reassigning does not alter the dependencies seen between the instances in the original DAG.

In addition, as the DAG satisfies Lemma 2, rearranging a source instance A containing instance from actor B will not alter the level sets when source instance B needs to be rearranged as well. □

A.2 Proof of Lemma 3

Lemma 3 (Multiple Roots for a Cyclic SDFG). *If DAG_{ind} corresponds to a strictly cyclic SDFG such that no actor of the SDFG has the required delay tokens*

Algorithm 2 Get actors that depend on each other

```
1: procedure GET_DEPENDENCY($\mathbb{R}, \mathbb{B}, \mathbb{I}, \mathbb{X}$)
2:     $\mathbb{R}' \leftarrow \mathbb{R}, \quad \mathbb{D} = \emptyset$
3:     while True do
4:         $\mathbb{D}' \leftarrow \{A, B \mid A \in \mathbb{R}, B \in \mathbb{R} - \{A\}, 0 \leq i, i' < \mathbf{q}(A), 0 \leq j, j', < \mathbf{q}(B), i \neq i', j \neq j', A_i, B_i \in \mathbb{I}, A_{i'}, B_{i'} \in \mathbb{V}_{dag} - \mathbb{I}, \forall X \in \mathbb{X}, B_{j'} \in \mathbb{B}_{X,A_i}, A_{i'} \in \mathbb{B}_{X,B_j}$
5:         if $\mathbb{D}' = \emptyset$ then
6:             break
7:         $\mathbb{D} \leftarrow \mathbb{D} \cup \{\mathbb{D}'\}$                    ▷ $\mathbb{D}$ is a set of sets
8:         $\mathbb{R}' \leftarrow \mathbb{R}' - \mathbb{D}'$
9:     return $\mathbb{D}$
```

to execute all of its instances, then $|\mathbb{R}| > 1$ *and* ($|\mathbb{I}| \geq m \vee |\{\mathbb{B}\}| \geq m$)*, where* $m = \min(\mathbf{q}(A)), \forall A \in \mathbb{V}$.

Proof of Lemma 3. We prove this by contradiction. Let DAG_{ind} exist for SDFG such that $\forall A \in \mathbb{R}, \mathbf{q}(A) > 1, 0 \leq i < \mathbf{q}(A), A_i \in \mathbb{I}, |\mathbb{I}| < \mathbf{q}(A), |\mathbb{R}| = 1, A'_i \in \mathbb{V}_{dag} - \mathbb{I}$. Thus, by construction $l_{A_i} = 0$ and $l_{A'_i} > 0$ i.e. $A_i \prec^\star A'_i$ violating DAG_{ind}. Hence $|\mathbb{R}| > 1$ and $\exists C \in \mathbb{R} \mid \mathbb{V} - A, \forall X \in \mathbb{X}, C \prec^\star X, A'_i \in \mathbb{B}_{X,C_j}$ for some $0 \leq j < \mathbf{q}(C)$.

For the second part, let $|\mathbb{I}| \wedge |\mathbb{B}| < m$. Then for some $A \in \mathbb{V} \mid A \notin \mathbb{R}, \mathbf{q}(A) \geq m$, by pigeon hole principle, at least one instance must be on a different level than the rest with a precedence on an instance of A, violating Definition 14. □

Lemma 4 (Rearranging Strictly Cyclic SDFG). *If DAG_{ind} corresponds to a strictly cyclic SDFG, G, such that no actor has required delay tokens to execute all of its instances, then the resulting $\mathbb{L}$ of DAG_{ind} can be rearranged such that there is at least one level $\mathbb{L}_l$ that consists of all the instances of some actor in G.*

Before rearranging instances from strictly cyclic SDFGs, we need to identify instances that belong to strictly cyclic SDFGs via DAG. Algorithm 2 groups all the actors present in $\mathbb{R}$ that share instances in their predecessor branch set. If two actors A, B form a loop in an SDFG such that it does not satisfy Lemma 1, then the DAG reflect this fact by having some instance of A depending on B and vice versa. Line 4 of Algorithm 2 captures this fact. Applying Algorithm 2 on Figure 2b yields the set $\mathbb{D} = \{\{A, C\}\}$. As instances of A and C are both present in their respective predecessor branch set, one can delay the predecessor branch set of C with respect to the predecessor branch set of A. At level $l = 2$ all the instances of C will belong in the same level. Lemma 1 showed that a DAG can be rearranged when $\mathbb{D} = \emptyset$. Lemma 4 shows that a DAG can also be rearranged when $\mathbb{D} \neq \emptyset$.

Proof of Lemma 4. Applying Algorithm 2 on DAG of a strictly cyclic SDFG yields $|\mathbb{D}| = 1$ as all the actors precedes each other and there is exactly one cycle. As $B_{j'} \in \mathbb{B}_{X,A_i}$, rearranging $\mathbb{D}'$ using Algorithm 1, but passing $\mathbb{I}' \leftarrow \mathbb{I} - \{A_i \mid 0 \leq i < \mathbf{q}(A)\}$ instead of $\mathbb{I}$ rearranges the DAG w.r.t A. The resulting $\mathbb{L}$ will have all instances of B at the level of B'_j. □

A.3 Retiming transformation

The DAG obtained after applying Algorithm 1 on $\mathbb{D}$ as given in the proof of Lemma 1 need not satisfy Definition 13. Let l_p denote the level at which all instances of some actor occur in the same level (given by the function $\mathrm{parallel\_level}$ in Algorithm 3). If $lp > 0$, then the DAG is not data-parallel. In other words, rearranging levels of a DAG_{ind} is not sufficient to make it DAG_{par} when there are cycles in the graph.

Recall that from Definition 8, a DAG signifies a schedule that can be executed indefinitely. Algorithm 3 breaks the DAG into two parts: a transient part DAG^T that is executed initially and a periodic part DAG^C that gets executed indefinitely. As long as DAG^C satisfies Definition 13, we can execute SDFG in data-parallel fashion. Algorithm 3 safely moves some instances from DAG^C to DAG^T to make $DAG^C = DAG_{par}$. For the Figure 2b, the rearranged DAGs after applying Algorithm 3 is given in Figure 5.

When an SDFG contains cycles, line 5 in Algorithm 3 is not satisfied and the algorithm proceeds to process cycles, one at a time. The function $\mathrm{pickelement}$ randomly chooses an element from a set. The line 11 and 15 filter $\mathbb{I}$ and $\mathbb{B}$ to contain only those elements that are relevant to the current cycle. In addition, line 11 filters $\mathbb{I}$ further so that the DAG can be rearranged w.r.t to A (see Lemma **??**). Line 13 removes instances from source actors, failing which the function $\mathrm{parallel\_level}$ will always yield $\bar{l} = 0$, in the presence of instances from source actors.

Line 16-28 iteratively shifts the instances above $\bar{l}$ to yield DAG^T and DAG^C. Consider an instance A present in a level set $\mathbb{L}_l < \mathbb{L}_{\bar{l}}$. Line 18-22 considers those instances P of $\mathrm{actor}(P)$ that precedes $\mathrm{actor}(A)$ in the original SDFG. As the loop is considering cycles, it accounts for all the actors affecting the actor of instance A. If there is no such instance, we can randomly pick an exit node to place A succeeding it. Operations from line 23-28 places an instance in DAG^T and an edge after all the instances of $\mathbb{P}$. Simultaneously, it removes the instance A from DAG^C and all the edges associated with it. The level sets and exit nodes are updated to reflect the DAG manipulation.

A.4 Proof of Claim 1

Claim 1. *The necessary and sufficient condition for executing all the instances of actors of a consistent SDFG graph in a data-parallel fashion is that in the corresponding DAG of the SDFG graph no instance of an actor depends on a*

Algorithm 3 Get data-parallel DAG for SDFG with DAG_{ind}

```
 1: procedure DATA_PARALLEL($G$, $DAG$, $\mathbb{L}$, $\mathbb{B}$, $\hat{l}$)
 2:     $\mathbb{I}_S \leftarrow \{A \mid \forall A \in \mathbb{I}, \text{in}(\text{actor}(A)) = \emptyset\}$, $\mathbb{I}' \leftarrow \mathbb{I} - \mathbb{I}_S$
 3:     $DAG^C = DAG$, $DAG^T = \emptyset$
 4:     $\mathbb{D} \leftarrow$ GET_DEPENDENCY($\mathbb{R}, \mathbb{B}, \mathbb{I}, \mathbb{X}$)
 5:     if $\mathbb{D} = \emptyset$ then                                        ▷ SDFG is acyclic-like
 6:         $\mathbb{L} \leftarrow$ REARRANGE($\mathbb{B}$, $\mathbb{L}$, $\mathbb{I}$)
 7:         return($DAG^T, DAG^C, \mathbb{L}^T = \emptyset, \mathbb{L}^C = \mathbb{L}$)
 8:     $\mathbb{L}^T \leftarrow \mathbb{L}$, $\mathbb{L}^C \leftarrow \mathbb{L}$, $DAG^C = DAG$, $DAG^T = DAG$
 9:     for all $\mathbb{D}' \mid \mathbb{D}' \in \mathbb{D}$ do
10:         $A \leftarrow \text{pickelement}(\mathbb{D}')$                                ▷ $A$ is an actor
11:         $\mathbb{I}'' \leftarrow (\mathbb{I}' - \{A_i \mid 0 \leq i < \mathbf{q}(A)\}) - \{B_j \mid 0 \leq j < \mathbf{q}(B), B \in \mathbb{D} - \mathbb{D}'\}$
12:         $\mathbb{L} \leftarrow$ REARRANGE($\mathbb{B}, \mathbb{L}, \mathbb{I}''$)
13:         $\mathbb{L}' \leftarrow \{\mathbb{L}'_l \mid \mathbb{L}'_0 = \mathbb{L}_0 - \{\mathbb{I}_S\}, \forall l > 0, \mathbb{L}'_l = \mathbb{L}_l\}$
14:         $\bar{l} \leftarrow$ PARALLEL_LEVEL($\mathbb{L}'$, $\mathbb{V}$, $\hat{l}$)
15:         $\mathbb{B}' \leftarrow \{\mathbb{B}_{X,A_i} \mid X \in \mathbb{X}, 0 \leq i < \mathbf{q}(A), A_i \in \mathbb{I}\}$
16:         for $l := 0$ to $\bar{l} - 1$ do
17:             for all $A \mid A \in \mathbb{L}_l, A \in \mathbb{B}'$ do                      ▷ $A$ is an instance
18:                 $\mathbb{P} \leftarrow \{P \mid \forall P \in \mathbb{X}, \text{actor}(P) \prec \text{actor}(A)\}$
19:                 if $\mathbb{P} = \emptyset$ then
20:                     $l_p \leftarrow \hat{l}$,    $\mathbb{P} \leftarrow \{\text{pickelement}(\mathbb{X})\}$
21:                 else
22:                     $l_p \leftarrow \max(|\{\mathbb{B}_{P,I} \mid \forall P \in \mathbb{P}, I \in \mathbb{I}\}|)$
23:                 $\mathbb{L}^T_{l_p+1} \leftarrow \mathbb{L}^T_{l_p+1} \cup \{A\}, \mathbb{V}^T_{dag} \leftarrow \mathbb{V}^T_{dag} \cup \{A\}$
24:                 $\mathbb{E}^T_{dag} \leftarrow \mathbb{E}^T_{dag} \cup \{(P, A) \mid \forall P \in \mathbb{P}\}$
25:                 $\mathbb{L}^C_{l_p+1} \leftarrow \mathbb{L}^C_{l_p+1} \cup \{A\}, \mathbb{L}^C_l \leftarrow \mathbb{L}^C_l - \{A\}$
26:                 $\mathbb{E}^C_{dag} \leftarrow \mathbb{E}^C_{dag} - \{(A, B) \mid \forall B \in \mathbb{V}_{dag}, (A, B) \in \mathbb{E}_{dag}\}$
27:                 $\mathbb{E}^C_{dag} \leftarrow \mathbb{E}^C_{dag} \cup \{(P, A) \mid \forall P \in \mathbb{P}\}$
28:                 $\mathbb{X} \leftarrow (\mathbb{X} - \mathbb{P}) \cup \{A\}$
    return ($DAG^T, DAG^C, \mathbb{L}^T, \mathbb{L}^C$)
29: function PARALLEL_LEVEL($\mathbb{L}$, $\mathbb{V}$, $\hat{l}$)
30:     for $l := 0$ to $\hat{l}$ do
31:         $\mathbb{A} \leftarrow \{\text{actor}(A) \mid \forall A \in \mathbb{L}_l\}$
32:         if $\sum_{A \in \mathbb{A}} \mathbf{q}(A) = |\mathbb{L}_l|$ then return $l$
```

previous instance of the same actor i.e.

$$DAG_{ind} \iff DAG_{par}$$

Proof of Claim 1 (Necessary condition). Let DAG = $\neg DAG_{ind}$ exist i.e.

$$\exists [A \prec^\star B] \in \mathbb{B} \mid A, B \in \mathbb{V}_{dag}, \text{actor}(A) = \text{actor}(B)$$

Further, let l_A, l_B be the levels of instances A and B respectively. As $A \prec^\star B$, $l_A < l_B$ violating Definition 13. Hence $DAG = \neg DAG_{par}$. □

Proof of Claim 1 (Sufficient condition). When an SDFG is acyclic, $\mathbb{D} = \emptyset$ and the DAG_{ind} is rearranged according to Lemma 1 to yield DAG_{par}. When an SDFG is strictly cyclic, $|\mathbb{D}| = 1$ and lines 18-28 return DAG^C that satisfies Definition 13.

When an SDFG contains a mix of cyclic and acyclic paths, line 22 rearranges the acyclic paths on the first iteration itself. This is due to the fact the line 11 retains the instances that belong to the acyclic paths of the SDFG. The rest of the cycles are rearranged one at a time by the loop at line 18.

Thus for all kinds of SDFG, Algorithm 3 transforms DAG_{ind} to $DAG^C = DAG_{par}$, proving the equivalence. □

References

[1] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," vol. C-36, no. 1, pp. 24–35, Jan 1987.

[2] S. S. Battacharyya, E. A. Lee, and P. K. Murthy, *Software Synthesis from Dataflow Graphs*. Norwell, MA, USA: Kluwer Academic Publishers, 1996.

[3] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software Pipelined Execution of Stream Programs on GPUs," in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO'09. Washington, DC, USA: IEEE Computer Society, 2009, p. 200209. [Online]. Available: http://dx.doi.org/10.1109/CGO.2009.20

[4] S. Lin, Y. Liu, W. Plishker, and S. S. Bhattacharyya, "A Design Framework for Mapping Vectorized Synchronous Dataflow Graphs Onto CPU-GPU Platforms," in *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '16. New York, NY, USA: ACM, 2016, p. 2029. [Online]. Available: http://doi.acm.org/10.1145/2906363.2906374

[5] L. Schor, A. Tretter, T. Scherer, and L. Thiele, "Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of OpenCL," in *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*, Oct 2013, pp. 41–50.

[6] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, "Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming," in *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, Sept 2014, pp. 36–40.

[7] "preesm," https://github.com/skanur/preesm, 2017.

[8] S. Verdoolaege, *Polyhedral Process Networks*. New York, NY: Springer New York, 2013, pp. 1335–1375. [Online]. Available: http://dx.doi.org/10.1007/978-1-4614-6859-2_41

[9] A. Balevic and B. Kienhuis, "A Data Parallel View on Polyhedral Process Networks," in *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '11. New York, NY, USA: ACM, 2011, pp. 38–47. [Online]. Available: http://doi.acm.org/10.1145/1988932.1988939

[10] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "A Hierarchical Multiprocessor Scheduling Framework for," in *Synchronous Dataflow Graphs, UC Berkeley UCB/ERL M95/36*, 1995.

[11] S. Kanur, J. Lilius, and J. Ersfolk, "Detecting Data-Parallel Synchronous Dataflow Graphs," Tech. Rep. TUCS Technical Reports, 1184, 2017.

[12] P. R. Schaumont, *Data Flow Modeling and Transformation*. Boston, MA: Springer US, 2013, pp. 31–59. [Online]. Available: http://dx.doi.org/10.1007/978-1-4614-3737-6_2

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[14] E. S. Group, "Sdf3 examples," http://www.es.ele.tue.nl/sdf3/download/examples.php, 2017, accessed: 2017-06.

[15] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi, "Memory Analysis and Optimized Allocation of Dataflow Applications on Shared-Memory MPSoCs," *Journal of Signal Processing Systems*, vol. 80, no. 1, pp. 19–37, Jul 2015. [Online]. Available: https://doi.org/10.1007/s11265-014-0952-6

ISBN 978-952-12-3570-2
ISSN 1239-1891