Integrating Kahn Process Networks as a Model of Computation in an Extendable Model-based Design Framework

Omair Rafique and Klaus Schneider
Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany

Keywords: Model-based Synthesis, Kahn Process Networks, Runtime System.

Abstract: This work builds upon an extendable model-based design framework called SHeD that enables the automatic software synthesis of different classes of dataflow process networks (DPNs) which represent different kinds of models of computation (MoCs). SHeD proposes a general DPN model that can be restricted by constraints to special classes of DPNs. It provides a tool chain including different specialized code generators for specific MoCs and a runtime system that finally maps models using a combination of different MoCs on cross-vendor target hardware. In this paper, we further extend the framework by integrating Kahn process networks (KPNs) in addition to the so-far existing support of dynamic and static/synchronous DPNs. The tool chain is extended for automatically synthesizing the modeled systems for the target hardware. In particular, a specialized code generator is developed and the runtime system is extended to implement models based on the underlying semantics of the KPN MoC. We modeled and automatically synthesized a set of benchmarks for different target hardware based on all supported MoCs of the framework, including the newly integrated KPN MoC. The results are evaluated to analyze and compare the code size and the end-to-end performance of the generated implementations of all MoCs.

1 INTRODUCTION

1.1 Motivation and Problem Setting

A model of computation (MoC) precisely determines *why, when and which atomic component of a system is executed*. Dataflow process networks (DPNs) (Karp and Miller, 1966; Dennis, 1974) can be used to define such MoCs. In general, a DPN is a system of autonomous processes that communicate with each other via dedicated point-to-point channels having First-In-First-Out (FIFO) buffers. Each process performs a computation by firing where it consumes data tokens from its input buffers and produces data tokens for its output buffers. The firing of a process is generally triggered by the availability of input data. While the general model of computation does not impose further restrictions, many different classes of DPNs (Kahn and MacQueen, 1977; Engels et al., 1995; Buck, 1993; Lee and Messerschmitt, 1987; Lee and Parks, 1995) have been introduced over time. These classes mainly differ in the kinds of behaviors of the processes which affects on the one hand the expressiveness of the DPN class as well as the methods for their analysis (predictability) and synthesis (efficiency). These behaviors are precisely described based on the underlying semantics of how each atomic process is triggered for an execution, and how each execution of a process consumes/produces data, in particular, whether a statically or dynamically determined amount of data is consumed and produced.

Design tools for modeling like Ptolemy (Brooks et al., 2010) and FERAL (Kuhn et al., 2013) support the modeling and simulation of behaviors based on different MoCs, including particular classes of DPNs. These frameworks are used to model and to design parallel embedded systems using well-defined and precise MoCs. In (Golomb, 1971), Golomb discussed models and their relationship to the real world, and famously stated that *"you will never strike oil by drilling through the map"*. Of course, this does not reduce the importance and great value of a map. We therefore appreciate the convenient use of these well-established frameworks to study and analyze different MoCs at the design level. However, we also encounter a lack of emphasis on automatically synthesizing models to real implementations, to analyze and evaluate the artifacts exhibited by particular MoCs in the real world.

Rafique, O. and Schneider, K.
Integrating Kahn Process Networks as a Model of Computation in an Extendable Model-based Design Framework.
DOI: 10.5220/0010260500870099
In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2021)*, pages 87-99
ISBN: 978-989-758-487-9

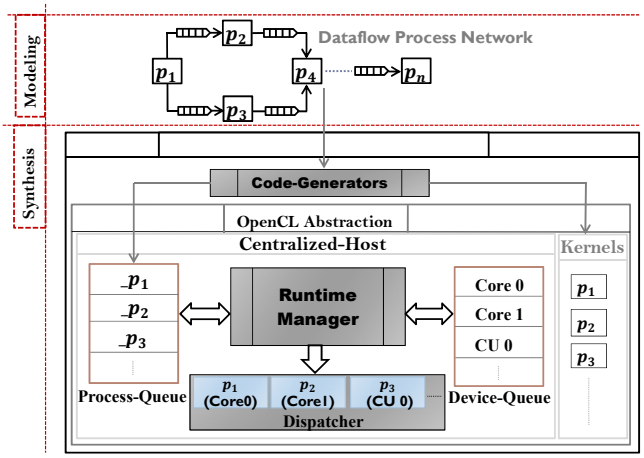

Figure 1: The basic building block diagram of the framework.

The existing design tools for synthesis are usually limited to specific classes of DPNs, i.e., each tool is dedicated to a particular DPN class. These frameworks provide specialized tool chains, in particular, a specialized code generator for a specific MoC. Each framework therefore allows one to model and to implement systems based on a specific MoC, i.e., the underlying DPN class. For instance, a design tool that only supports a synchronous (static) DPN class can be used for the modeling and synthesis of synchronous behaviors. Similarly, a design tool based on a dynamic DPN class can be employed for dynamic and asynchronous behaviors.

The overall motivation is therefore to enable the modeling as well as the automatic software synthesis of systems using different well-defined and precise MoCs (classes of DPNs) under the supervision of a common extendable model-based design framework.

1.2 Previous Work and Challenges

In (Rafique and Schneider, 2020b), we presented an extendable model-based design framework called $\mathbb{S}$HeD which essentially enables the modeling as well as the automatic software synthesis of systems based on different classes of DPNs. The overall design flow is systematically organized in two phases of modeling and synthesis as shown in Fig. 1. First, the modeling phase proposes a common general DPN model that relies on an abstract notion of a process. A process is composed of a finite set of actions where each action can perform a computation by consuming tokens from input buffers and producing tokens to output buffers. This general model is used with specific constraints and definitions to specify the precise classes of DPNs. The underlying modeling language of the proposed DPN model is the CAL actor language (Eker and Janneck, 2003). The framework currently supports two different classes of DPNs, namely the dynamic dataflow (DDF) MoC and the synchronous (static) dataflow (SDF) MoC. The DDF MoC allows processes whose actions consume different numbers of inputs and produce different numbers of outputs while the actions of processes in the SDF MoC all consume the same number of input tokens and produce the same number of output tokens.

Second, the synthesis part provides a tool chain that consists of various essential tools including different specialized code generators and runtime systems for different MoCs. Using the open computing language (OpenCL) (Stone et al., 2010), it incorporates a standard hardware abstraction for cross-vendor heterogeneous hardware architectures. OpenCL offers a programming model consisting of a host and several kernels where the host is a centralized entity that is connected to one or more computing devices and is responsible for the execution of kernels (Rafique and Schneider, 2020a). Each kernel is a C function that actually implements one instance of the behavior of a system or part of a system. The framework adopts this idea of host and kernels for the synthesis as shown in Fig 1. The synthesis phase uses a combination of different code generators which generates an OpenCL kernel for each process in the network based on the underlying class of that process. The runtime system, in particular, is organized in a centralized host and kernels architecture, built under the OpenCL abstraction. The host accommodates different essential components along with the *Runtime-Manager*. The Runtime-Manager exploits other components of the host and provides different low-level implementations to finally execute the modeled DPNs (kernels) on the target hardware.

As presented in (Rafique and Schneider, 2020b), the SDF MoC proved to be the most effective one in terms of the generated code size, the build time, and the end-to-end performance. This is mainly because the SDF MoC only supports statically schedulable systems and therefore generates very succinct code for static processes. In contrast, the implementations based on the DDF semantics accommodate additional code for enabling the dynamic evaluation of actions at runtime. The DDF MoC offers semantics to model static as well as dynamic behaviors, but at the cost of the additional runtime overhead. Therefore, it exhibits a trade-off between flexibility and overall performance. The main challenge is therefore to integrate a MoC in the framework that is able to capture dynamic behaviors and that allows one to generate succinct code.

1.3 Idea and Contributions

Kahn process networks (KPNs) (Kahn and MacQueen, 1977) are dynamic DPNs that exhibit latency-insensitive deterministic behaviors that do not depend on the timing or the execution order of the processes. The KPN MoC is typically specified with the following restrictions and properties: (**1**) Processes are not allowed to test input buffers for the existence of tokens. (**2**) Reading from input buffers is blocking, and writing to output buffers is non-blocking. (**3**) Processes must implement deterministic sequential functions. (**4**) Processes do not need all of their inputs to get triggered for execution. Based on these restrictions/properties, it can be implied that in contrast to the DDF MoC, the KPN MoC only triggers a process for execution if the exact information on inputs required to produce the output is available. This avoids the generation of additional code for processes to dynamically evaluate actions at runtime, and hence, avoids the runtime overhead associated with the DDF MoC. In contrast to the SDF MoC, the KPN MoC can capture both static as well as dynamic behaviors. Altogether, the integration of the KPN MoC in the framework will potentially provide more efficient system implementations by combining the flexibility to support dynamic processes and the ability to produce succinct kernel code for processes.

In this paper, we therefore integrate the KPN MoC based on the proposed general model of DPN and the runtime system of the framework. Since channels/buffers with unbounded capacity cannot be realized in real implementations, the proposed DPN model only supports blocking write. However, a KPN semantics-preserving implementation can use bounded buffers using blocking read and write (Parks, 1995). In summary, we make the following contributions in this paper:

- We integrate the KPN MoC in an extendable model-based design framework, mainly with the aim to produce more efficient system implementations in terms of end-to-end performance compared to the already supported MoCs (DDF and SDF) of the framework.
- We formally describe the integrated KPN MoC and its corresponding code generator based on the proposed general model of DPN through structural operational semantics (SOS).
- We present the extended runtime system for finally implementing KPN models on the target hardware.
- We designed a set of simple benchmarks involving static as well as dynamic behaviors. Each benchmark is modeled and automatically synthesized three times (when possible): First, based on the DDF MoC, second using the SDF MoC, and finally using the newly integrated KPN MoC. Each generated implementation is evaluated on two different target hardware platforms. The results are presented to analyze and to compare the code size and the end-to-end performance of all generated implementations.

2 RELATED WORK

Design tools for modeling like Ptolemy (Brooks et al., 2010) and FERAL (Kuhn et al., 2013) support various MoCs including different classes of DPNs. These frameworks use software components called *directors* that control the semantics of the execution of processes as well as the communication between them. However, Ptolemy provides also a preliminary code generation facility[1]. It requires the supporting helper code for each process to generate a general C program. This helper code is required to be provided manually using a fairly complex procedure for each process. Currently, only specific processes have supporting helper code.

Another well known design tool called System-CoDesigner (Haubelt et al., 2008) supports different classes of DPNs. It incorporates an actor-oriented behavior built on top of SystemC. System-CoDesigner currently supports only hardware synthesis based on a commercial tool named Forte's Cynthesizer.

Design tools for synthesis are moreover limited to particular MoCs where each framework usually only

[1] http://ptolemy.berkeley.edu/ptolemyII/ptII10.0/ptII10.0.1/ptolemy/cg/

supports a specific DPN class. We present a few examples of dataflow-oriented synthesis frameworks:

- The framework presented in (Schor et al., 2013) introduces a design flow for executing applications specified as SDF graphs on heterogeneous systems using OpenCL. The main focus of this work is to develop features to better utilize the parallelism in heterogeneous architectures.
- The DAL framework (Schor et al., 2012) presents a scenario-based design flow for mapping streaming applications onto heterogeneous systems. Behaviors are modeled based on Kahn process networks (KPNs) (Kahn and MacQueen, 1977) and a finite state machine.
- The work presented in (Lund et al., 2015) translates DPN models to programs using OpenCL. The methodology incorporates static analysis and transformations and confined to the synthesis of SDF models. Similarly, another dataflow oriented framework (Boutellier et al., 2018) proposes a MoC as a symmetric-rate dataflow, a restriced form of SDF where the token production rate and the token consumption rate per FIFO channel is symmetric.

Thus, the existing frameworks based on DPNs generally support the modeling and synthesis of systems based on a particular MoC, i.e., a specific DPN class. In contrast, our approach enables the modeling as well as the automatic software synthesis of systems using different classes of DPNs, including their heterogeneous combinations under the supervision of a common extendable framework.

3 THE GENERAL MODEL OF DPN

A general dataflow process network is a triple $\aleph = (D, F, \mathcal{P})$, consisting of a data type D, FIFO buffers F and processes $\mathcal{P}$. The data type D always includes a special symbol $\perp \in D$ to indicate the absence of a value. The semantics of FIFO buffers is a pair of $\{i, o\} \times \mathbb{N}$ where i and o denotes input and output buffers, respectively. Each FIFO buffer $f \in F$ provides a channel to a sequence of tokens over D, termed as a stream, denoted by X_D^f. Following that, $\vec{X}_D^F$ is the set of all finite sequences of tokens over D. For convenience, we simply denote the contents of buffers by X_D^*. $\mathcal{P}$ is the finite set of processes. Each process $p = (F_{in}, F_{out}, Act) \in \mathcal{P}$ consists of a list of input buffers F_{in}, a list of output buffers F_{out}, and an associated set of actions Act. The input and output buffers of a process are always mutually exclusive, i.e., $F_{in} \cap F_{out} = \{\}$. We further organize each process in a set of what we call actions Act.

3.1 The Basic Structure of Actions

Each action $act = (F_i^{act}, F_o^{act}, \vec{\mathcal{V}}_i^{act}, \vec{\mathcal{V}}_o^{act}) \in Act$ consists of a list of input buffers $F_i^{act} \subseteq F_{in}$, a list of output buffers $F_o^{act} \subseteq F_{out}$, a list of sequences of input token variables $\vec{\mathcal{V}}_i^{act}$ of F_i^{act}, and a list of sequences of output token variables $\vec{\mathcal{V}}_o^{act}$ of F_o^{act}. All sequences $\in \vec{\mathcal{V}}^{act} = (\vec{\mathcal{V}}_i^{act}, \vec{\mathcal{V}}_o^{act})$ contain a list of local variables for temporary storage of data and are exclusive to a single action. Each sequence $\in \vec{\mathcal{V}}_i^{act}$ contains a list of local variables, denoting a finite sequence of input tokens that will be consumed per execution of an action from its respective input buffer $\in F_i^{act}$. Similarly, each sequence $\in \vec{\mathcal{V}}_o^{act}$ contains a list of local variables, denoting a sequence of output tokens that will be produced per execution of an action in its respective output buffer $\in F_o^{act}$. For an action with *'t'* inputs $F_i^{act} = \{F_{i_1}, F_{i_2}\ ..., F_{i_t}\}$ providing input streams $\vec{X}_D^{F_i^{act}} = \{X_D^{F_{i_1}}, X_D^{F_{i_2}}\ ..., X_D^{F_{i_t}}\}$, the corresponding sequences of local input variables are defined by $\vec{\mathcal{V}}_i^{act} = \{\mathcal{V}_{i_1}^{act}, \mathcal{V}_{i_2}^{act}, ..., \mathcal{V}_{i_t}^{act}\}$. Each sequence $\mathcal{V}_{i_j}^{act} \in \vec{\mathcal{V}}_i^{act}$ consists of a finite number of input token variables. For instance, if F_{i_j} has a sequence of *'q'* tokens per action execution, the corresponding list of local variables is defined by $\mathcal{V}_{i_j}^{act} = \{v_{i_{j\text{-}1}}^{act}, v_{i_{j\text{-}2}}^{act}, ..., v_{i_{j\text{-}q}}^{act}\}$. The number of token variables in each sequence, denoted by $\mathcal{L}(\mathcal{V}_{i_j}^{act})$, determines the token consumption rate per execution of the corresponding input buffer.

Similarly, an action with *'l'* outputs $F_o^{act} = \{F_{o_1}, F_{o_2}\ ..., F_{o_l}\}$ is defined with the corresponding sequences of local output variables $\vec{\mathcal{V}}_o^{act} = \{\mathcal{V}_{o_1}^{act}, \mathcal{V}_{o_2}^{act}, ..., \mathcal{V}_{o_l}^{act}\}$. Each sequence $\mathcal{V}_{o_j}^{act} \in \vec{\mathcal{V}}_o^{act}$ consists of a finite number of output token variables. The number of token variables in each sequence, denoted by $\mathcal{L}(\mathcal{V}_{o_j}^{act})$, determines the token production rate per execution of the corresponding output buffer. This structure of inputs and outputs of actions is illustrated in Listing 1 and Listing 2 (Lines 1 and 6). The declaration of inputs and outputs is separated by the identifier '==>'.

For an action that requires input tokens to have particular values, an additional condition can be specified using a **guard** (Lines 2 and 7, in Listing 1 and Listing 2). The inputs used for guards (guarded inputs) and their corresponding token variables are denoted by $F_\gamma^{act} \subseteq F_i^{act}$ and $\vec{\mathcal{V}}_\gamma^{act} \subseteq \vec{\mathcal{V}}_i^{act}$, respectively. In particular, a guard is composed of a list of Boolean

Listing 1: Dynamic split using the proposed model.

```
1  act_1: action F_i1 : [v_i1_1^act1], F_i2 : [v_i2_1^act1] ==> F_o1 : [v_o1_1^act1]
2    guard v_i1_1^act1 = 1
3    do
4      v_o1_1^act1 := v_i2_1^act1;
5    end
6  act_2: action F_i1 : [v_i1_1^act2], F_i2 : [v_i2_1^act2, v_i2_2^act2] ==> F_o1 : [v_o1_1^act2], F_o2 : [
       v_o2_1^act2]
7    guard v_i1_1^act2 = 2
8    do
9      v_o1_1^act2 := v_i2_1^act2; v_o2_1^act2 := v_i2_2^act2;
10   end
```

Listing 2: Dynamic merge using the proposed model.

```
1  act_1: action F_i1 : [v_i1_1^act1], F_i2 : [v_i2_1^act1] ==> F_o1 : [v_o1_1^act1]
2    guard v_i1_1^act1 = 1
3    do
4      v_o1_1^act1 := v_i2_1^act1;
5    end
6  act_2: action F_i1 : [v_i1_1^act2], F_i2 : [v_i2_1^act2], F_i3 : [v_i3_1^act2] ==> F_o1 : [v_o1_1^act2,
       v_o1_2^act2]
7    guard v_i1_1^act2 = 2
8    do
9      v_o1_1^act2 := v_i2_1^act2; v_o1_2^act2 := v_i3_1^act2;
10   end
```

expressions $\mathcal{E}_{\mathcal{B}}^{act} = \{\mathcal{E}_{\mathcal{B}_1}^{act}, \mathcal{E}_{\mathcal{B}_2}^{act}, ..., \mathcal{E}_{\mathcal{B}_n}^{act}\}$, where $n \in \mathbb{N}$. Each expression $\in \mathcal{E}_{\mathcal{B}}^{act}$ can be applied on an individual input token variable. A guard consisting of a list of Boolean expressions $\mathcal{E}_{\mathcal{B}}^{act}$ is therefore applied on a list of individual token variables denoted by $\mathcal{V}_{\mathcal{E}_\gamma}^{act} \subseteq \bigcup \vec{\mathcal{V}}_{\gamma}^{act}$.

3.2 Basic Definitions for Actions

For any action *act* in a defined behavior of a process to be fired, there exists two necessary conditions and the additional condition of a guard (if used): First, there must be enough input tokens available as specified by the sequences of input token variables $\vec{\mathcal{V}}_i^{act}$ in F_i^{act}. Second, there must be enough space available as specified by the sequences of output token variables $\vec{\mathcal{V}}_o^{act}$ in F_o^{act}. Finally, the required values on the guarded inputs must be available, i.e., the guard must be true. These conditions can be described with the following definitions:

Definition 1 (Input Constraint (δ_{1_a}))**.** For an action $act \in Act$ to be fired, each sequence of token variables $\in \vec{\mathcal{V}}_i^{act}$ must be a prefix of the sequence of tokens available on the corresponding input buffer (where $s_1 \sqsubseteq s_2$ means that s_1 is a prefix of s_2): $\forall \mathcal{V}_{i_j}^{act} \in \vec{\mathcal{V}}_i^{act}, \mathcal{V}_{i_j}^{act} \sqsubseteq X_D^{F_{i_j}}$.

Definition 2 (Output Constraint (δ_{1_b}))**.** For an action $act \in Act$ to be fired, each output $\in F_o^{act}$ must contain space for the corresponding sequence of output token variables $\in \vec{\mathcal{V}}_o^{act}$. Overall, this yields:

$$\forall F_{o_j} \in F_o^{act}, (size(F_{o_j}) - count(X_D^{F_{o_j}})) \geq \mathcal{L}(\mathcal{V}_{o_j}^{act})$$

where

- $size(f)$ returns the total size $\in \mathbb{Z}$ of the buffer f, and
- $count(X_D^f)$ returns the current count $\in \mathbb{Z}$, i.e., the available number of tokens in the buffer f.

Definition 3 (Guard Constraint (δ_2))**.** For an action $act \in Act$ to be fired, the combined evaluation of all Boolean expressions must evaluate to 1 (*true*). With $\mathcal{B} = \{0,1\}$, this yields:

$$\forall act \in Act, (\mathcal{E}_{\mathcal{B}}^{act}(\mathcal{V}_{\mathcal{E}_\gamma}^{act}) \rightarrow \mathcal{B}) \stackrel{!}{=} 1$$

Upon firing, each action performs the desired computation defined within do/end construct, as illustrated in Listing 1 and Listing 2 (Lines 3-5, Lines 8-10).

4 KPN FORMULATION AND OPERATIONAL SEMANTICS

In this section, we formulate the KPN MoC and define its operational semantics based on the proposed general model of DPN, as presented in Section 3. We introduce a state transition system of a DPN by $\Sigma = \langle Act, \mathcal{M}_F, \mathcal{M}_\mathcal{V} \rangle$. The behavior of a process is described by a set of actions $Act = \{act_1, act_2, ..., act_n\}$ where $n \in \mathbb{N}$. Each $act \in Act$ is of the form l: $\vec{\mathcal{V}}_i^{act|\vec{X}_D^{F_i^{act}}|} \rightarrow \vec{\mathcal{V}}_o^{act|\vec{X}_D^{F_o^{act}}|}$ that upon firing consumes tokens as specified by $\vec{\mathcal{V}}_i^{act}$ from the corresponding streams of F_i^{act} and produces tokens as specified by $\vec{\mathcal{V}}_o^{act}$ to the corresponding streams of F_o^{act}. The labels l of actions are mutually exclusive. For all FIFO buffers, the function $\mathcal{M}_F : F \rightarrow X_D^*$ maps each buffer f to a sequence of data values (tokens) of data type D. Similarly, for all sequences of local token variables, the function $\mathcal{M}_\mathcal{V} : \vec{\mathcal{V}} \rightarrow D$ maps each local token variable to a data value of type D. The KPN MoC formulation and operational semantics are elaborated formally based on the state transition system Σ.

4.1 Organization of Actions

Each process $p \in \mathcal{P}$ can be composed of a set of actions $\in Act$ with **guards**. The inputs and outputs and their associated number of token variables

(i.e., token consumption and production rates) can be different across different actions with the exception of guarded inputs. In particular, the inputs used for guards (guarded inputs) across all actions in a process have at least one common input. This implies that for any pair of guarded actions in a process $F_i^{act_1} \cap F_i^{act_2} \neq \{\}$. The consumption rates of guarded inputs are always same across actions in a process, i.e., $\mathcal{L}(\vec{\mathcal{V}}_\gamma^{act_1}) = \mathcal{L}(\vec{\mathcal{V}}_\gamma^{act_2})$. For convenience we simply denote all guarded inputs in a process and their token variables by F_γ^p and $\vec{\mathcal{V}}_\gamma^p$, respectively. Second, the Boolean expressions $\mathcal{E}_\mathcal{B}$ of guards are always applied on the same tokens in inputs F_γ^{act} provided by different token variables across actions. However, the Boolean expressions $\mathcal{E}_\mathcal{B}$ of guards are always exclusive, e.g., $\mathcal{E}_\mathcal{B}^{act_1} \cap \mathcal{E}_\mathcal{B}^{act_2} = \{\}$. These restrictions facilitate the imposition of sequential function implementations in processes. In particular, they ensure that for each execution of a process, the actions will never compete for an execution for any set of tokens. They enable the execution of processes with dynamic data rates and dynamic data paths, mainly dependent on which guards are enabled on each execution.

Examples. Two different dynamic processes, namely the dynamic split (d-split) and the dynamic merge (d-merge) are illustrated in Listing 1 and Listing 2, respectively. The d-split process consists of two actions (act_1 and act_2) that depending on the value of token at input F_{i_1} split the tokens from input F_{i_2} to outputs F_{o_1} and F_{o_2}. Both actions use the same input F_{i_1} for the guard, declared with the same consumption rate (Lines 1 and 6). The guard is composed of different exclusive Boolean expressions (Lines 2 and 7). Both actions declare the input F_{i_2} with different consumption rates i.e., $\mathcal{L}(\mathcal{V}_{i_2}^{act_1}) \neq \mathcal{L}(\mathcal{V}_{i_2}^{act_2})$ (Lines 1 and 6). The action act_2 has an additional output F_{o_2}. On the contrary, the d-merge process depending on the value of token at input F_{i_1} merges the tokens from inputs F_{i_2} and F_{i_3} to output F_{o_1}. Both actions in d-merge declare the guarded input F_{i_1} with the same consumption rate (Lines 1 and 6), which is used with different exclusive guard expressions (Lines 2 and 7). The action act_2 has an additional input F_{i_3}. Both actions declare output F_{o_1} with different production rates i.e., $\mathcal{L}(\mathcal{V}_{o_1}^{act_1}) \neq \mathcal{L}(\mathcal{V}_{o_1}^{act_2})$ (Lines 1 and 6).

4.2 Evaluation and Execution of Actions

As discussed, the KPN MoC does not allow processes to test input buffers for the existence of tokens. A process is only triggered for execution if the exact information on inputs required to execute an action is available. Therefore, each time a process $p \in \mathcal{P}$ is triggered for an execution, a particular action is executed, mainly dependent on which guard is enabled. The guards of actions $\in Act$ are always evaluated sequentially in the same order of their actions definitions. This execution of actions is formally described using the state transition system Σ with two specific rules. Definitions 4-7 (i.e., Rules 1-4) have already been proposed for the existing DDF and SDF MoCs of the framework in (Rafique and Schneider, 2020b).

Definition 8 (Rule 5, PREPARE (ρ_5))**.** This rule is fired to read the specified fixed number of tokens $\vec{\mathcal{V}}_\gamma^p$ without consuming them from all inputs F_γ^p used for guards in a process $p \in \mathcal{P}$. It is formally described as:

$$\frac{\varnothing}{\langle Act, \mathcal{M}_F, \mathcal{M}_\mathcal{V} \rangle \rightarrow \langle Act, \mathcal{M}_F, \mathcal{M}_\mathcal{V}^{'} \rangle}$$

with, $\mathcal{M}_\mathcal{V}^{'} = \mathcal{M}_\mathcal{V}[\vec{\mathcal{V}}_\gamma^p]$.

Therefore, a finite number of tokens designated by $\vec{\mathcal{V}}_\gamma^p$ are read (not consumed) from streams $\vec{X}_D^{F_\gamma^p}$ provided by the guarded inputs F_γ^p.

Definition 9 (Rule 6, PROCEED (ρ_6))**.** An action $act_1 \in Act$ fires for an execution if the required values on the guarded inputs are available, i.e., the guard is true. It is formally described as:

$$\frac{(act_1 \in Act,\ \delta_2)}{\langle Act, \mathcal{M}_F, \mathcal{M}_\mathcal{V} \rangle \rightarrow \langle Act, \mathcal{M}_F^{'}, \mathcal{M}_\mathcal{V}^{'} \rangle}$$

with $\mathcal{M}_F^{'} = \mathcal{M}_F[F_i^{act_1}, F_o^{act_1}]$,$\mathcal{M}_\mathcal{V}^{'} = \mathcal{M}_\mathcal{V}[\vec{\mathcal{V}}_i^{act_1}, \vec{\mathcal{V}}_o^{act_1}]$. Therefore, a finite number of tokens $\vec{\mathcal{V}}_i^{act_1}$ are consumed from streams $\vec{X}_D^{F_i^{act_1}}$ provided by the inputs $F_i^{act_1}$, the defined computation is performed, and a finite number of tokens $\vec{\mathcal{V}}_o^{act_1}$ are produced to streams $\vec{X}_D^{F_o^{act_1}}$.

4.3 Triggering Processes for Execution

The KPN MoC only triggers a process for execution if the exact information on inputs required to produce the output is available. Therefore, each process $p \in \mathcal{P}$ is triggered for an execution if there exists at least one action $act \in Act$ having: enough tokens as specified by $\vec{\mathcal{V}}_i^{act}$ in inputs F_i^{act}, enough space as specified by $\vec{\mathcal{V}}_o^{act}$ in outputs F_o^{act}, and required values on the guarded inputs F_γ^{act}. This can be formally defined as:

$$\forall p \in \mathcal{P}\ \exists act \in Act.\ (\delta_1, \delta_2) = true$$

When a process is triggered for an execution, the actions are executed based on the proposed rules ρ_5 and ρ_6.

5 KPN SYNTHESIS

The synthesis phase of the framework, as shown in Fig. 1, provides a tool chain including code generators for specific MoCs and the runtime system. These tools work together and finally implement the modeled systems on the target hardware based on the underlying semantics of the used MoCs (i.e., classes of DPNs). In particular, a code generator is designed and developed based on the underlying semantics of each used DPN class. Each code generator generates an OpenCL kernel for each process based on the underlying DPN class. Second, the runtime system is organized in a centralized host and kernels program model, built under the OpenCL abstraction. The host features different components including the *Runtime-Manager* that schedule and execute processes (generated kernels) on the target hardware.

The complete synthesis phase and the associated tool chain is presented in detail in (Rafique and Schneider, 2020b). In this section, we mainly present the tools extended to support the synthesis of KPN models. This includes the specialized code generator designed for generating kernels based on the KPN MoC, and the extended *Runtime-Manager* for finally implementing KPN models on the target hardware.

Algorithm 1: Code generation of the KPN MoC.

```
1  foreach process p ∈ P in Network ℵ do
2      PREPARE(ρ5){
3      peek(for_all_guarded_inputs_of_this_process)}
4      foreach action act ∈ Act in a process p do
5          evaluate all guard expressions E_B^act
6          if δ2 then
7              PROCEED(ρ6){
8              read(for_all_inputs_of_this_action)}
9              perform_modeled_computations()
10             write(for_outputs_of_this_action)}
11         end
12     end
13 end
```

5.1 KPN Code Generation

Prior to presenting the KPN code generation, we first introduce some additional execution primitives using the state transition system Σ:

The **read**$(F_{i_j}, \mathcal{V}_{i_j}^{act}, \mathcal{L}(\mathcal{V}_{i_j}^{act}))$ method consumes a number of tokens as specified by $\mathcal{L}(\mathcal{V}_{i_j}^{act})$ from the input buffer F_{i_j}, and stores them in the temporary input token variables $\mathcal{V}_{i_j}^{act}$:

$$\frac{read(F_{i_j}, \mathcal{V}_{i_j}^{act}, \mathcal{L}(\mathcal{V}_{i_j}^{act}))}{\langle Act, \mathcal{M}_F, \mathcal{M}_{\mathcal{V}} \rangle \rightarrow \langle Act, \mathcal{M}_F^{'}, \mathcal{M}_{\mathcal{V}}^{'} \rangle}$$

with $\mathcal{M}_F^{'} = \mathcal{M}_F[F_{i_j}]$, $\mathcal{M}_{\mathcal{V}}^{'} = \mathcal{M}_{\mathcal{V}}[\mathcal{V}_{i_j}^{act}]$.

The **peek**$(F_{i_j}, \mathcal{V}_{i_j}^{act}, \mathcal{L}(\mathcal{V}_{i_j}^{act}))$ method reads a number of tokens (without consuming them) as specified by $\mathcal{L}(\mathcal{V}_{i_j}^{act})$ from the input buffer F_{i_j}, by simply copying them into the temporary input token variables $\mathcal{V}_{i_j}^{act}$:

$$\frac{peek(F_{i_j}, \mathcal{V}_{i_j}^{act}, \mathcal{L}(\mathcal{V}_{i_j}^{act}))}{\langle Act, \mathcal{M}_F, \mathcal{M}_{\mathcal{V}} \rangle \rightarrow \langle Act, \mathcal{M}_F, \mathcal{M}_{\mathcal{V}}^{'} \rangle}$$

with $\mathcal{M}_{\mathcal{V}}^{'} = \mathcal{M}_{\mathcal{V}}[\mathcal{V}_{i_j}^{act}]$.

The **write**$(F_{o_j}, \mathcal{V}_{o_j}^{act}, \mathcal{L}(\mathcal{V}_{o_j}^{act}))$ method writes a number of tokens as specified by $\mathcal{L}(\mathcal{V}_{o_j}^{act})$ from the output token variables $\mathcal{V}_{o_j}^{act}$ into the output buffer F_{o_j}:

$$\frac{write(F_{o_j}, \mathcal{V}_{o_j}^{act}, \mathcal{L}(\mathcal{V}_{o_j}^{act}))}{\langle Act, \mathcal{M}_F, \mathcal{M}_{\mathcal{V}} \rangle \rightarrow \langle Act, \mathcal{M}_F^{'}, \mathcal{M}_{\mathcal{V}} \rangle}$$

with $\mathcal{M}_F^{'} = \mathcal{M}_F[F_{o_j}]$.

The code generation based on the underlying semantics of the KPN MoC is shown in Algorithm 1. The fundamental principle of the KPN code generation is based on the dynamic execution of actions using the proposed rules (ρ_5 and ρ_6) of the underlying KPN semantics. For each process $p \in \mathcal{P}$ in the network $\aleph$, the proposed algorithm works as as described in the following.

First, the code is generated to fire the rule ρ_5 (**PREPARE**) (Lines 2-3). To this end, the **peek** method is inserted for all guarded inputs of a process (Line 3). The algorithm then iterates through the set of modeled actions in the order of their definitions (Line 4), where for each action $\in Act$, it proceeds as follows: First, the code is generated to evaluate all the guarded Boolean expressions $\mathcal{E}_{\mathcal{B}}^{act}$ (Line 5). Next, the code is generated to evaluate and fire the rule ρ_6 (**PROCEED**) (Lines 6-11). For ρ_6, first, the code is generated to read (consume) all the inputs of an action (Line 8). For this purpose, the **read** method is inserted for each input of an action. Second, the generated code for the modeled computations is inserted, prior to generating code for writing the computed results on the outputs (Lines 9-10). For writing, the **write** method is inserted for outputs based on the modeled computations of an action.

The generated kernel for d-split as illustrated in Listing 1 is listed in Listing 3. The code generator generates generic kernel code to enable the centralized host to dispatch multiple execution (instances) of kernels on the device at a time. However, for brevity, we only list and focus on the kernel code generated based on the KPN semantics. For better correspondence, we use the same conventions for inputs/out-

Listing 3: Generated kernel for d-split.

```
1  _kernel void d-split(__global fifo_t* F_{i_1}, __global
       fifo_t* F_{i_2}, __global fifo_t* F_{o_1}, __global fifo_t
       * F_{o_2}){
2      __private uint V_{i_1}[L(V_{i_1}^{act_1})];
3      __private uint V_{i_2}[L(V_{i_2}^{act_2})];
4      __private uint V_{o_1}[L(V_{o_1}^{act_1})];
5      __private uint V_{o_2}[L(V_{o_2}^{act_2})];
6      uint* v_{i_{1,1}}^{act_1} = &V_{i_1}[0]; uint* v_{i_{1,1}}^{act_2} = &V_{i_1}[0];
7      uint* v_{i_{2,1}}^{act_1} = &V_{i_2}[0]; uint* v_{i_{2,1}}^{act_2} = &V_{i_2}[0];
8      uint* v_{i_{2,2}}^{act_2} = &V_{i_2}[1]; uint* v_{o_{1,1}}^{act_1} = &V_{o_1}[0];
9      uint* v_{o_{1,1}}^{act_2} = &V_{o_1}[0]; uint* v_{o_{2,1}}^{act_2} = &V_{o_2}[0];
10     /*PREPARE(ρ5)*/
11     peek(F_{i_1}, V_{i_1}, 1);
12     /*PROCEED(ρ6): act1*/
13     if(*v_{i_{1,1}}^{act_1}==1){
14         read(F_{i_1}, V_{i_1}, 1);
15         read(F_{i_2}, V_{i_2}, 1);
16         *v_{o_{1,1}}^{act_1} = *v_{i_{2,1}}^{act_1};
17         bytes = write(F_{o_1}, V_{o_1}, 1); }
18     /*PROCEED(ρ6): act2*/
19     else if(*v_{i_{1,1}}^{act_2}==2){
20         read(F_{i_1}, V_{i_1}, 1);
21         read(F_{i_2}, V_{i_2}, 2);
22         *v_{o_{1,1}}^{act_2} = *v_{i_{2,1}}^{act_2}; *v_{o_{2,1}}^{act_2} = *v_{i_{2,2}}^{act_2};
23         bytes = write(F_{o_1}, V_{o_1}, 1);
24         bytes = write(F_{o_2}, V_{o_2}, 1); }
25 }
```

puts as were used throughout the paper. The generated code can be explained as described in the following.

First, the arrays are declared for the sequences of input/output token variables (Lines 2-5). In each execution of a KPN process, only a particular action is executed, which depends on the enabled guards. To avoid unnecessary duplication, an array is declared for each input/output with the highest consumption/production rate of all actions. For instance, an array is declared for the input F_{i_2} with the highest consumption rate $\mathcal{L}(\mathcal{V}_{i_2}^{act_2})$ of both actions (Line 3). Second, the individual input/output token variables are declared and pointed to their respective sequences i.e., arrays (Lines 6-9). Next, the rule ρ_5 (**PREPARE**) is followed to peek the tokens from the inputs used for guards (Line 10-11). Finally, the rule ρ_6 (**PROCEED**) is invoked to execute a particular action (i.e., either act_1 or act_2) based on the activation of guard. To this end, if the guard is true for act_1, a single token is consumed from F_{i_2} and written to the output F_{o_1} (Lines 13-17). On the other hand, if the guard is true for act_2, two tokens are consumed from F_{i_2}, where the first token is written to F_{o_1}, and the other to F_{o_2} (Lines 19-24).

5.2 KPN Runtime-Manager

General Workflow. The *Runtime-Manager* is a part of the host that uses the *Process-Queue* and the *Device-Queue* as shown in Fig. 1 and provides the schedulers for triggering processes for execution based on the used MoCs, a dispatcher for mapping processes executions to devices, and a communication and synchronization mechanism between the host and kernels. The *Process-Queue* provides the desired information about each process to the Runtime-Manager such as the associated FIFO buffers, the process's status (idle, running or blocked), the associated kernel, etc. The *Device-Queue* lists all the available devices of the target hardware using the OpenCL specification. Each element of this queue provides a command queue of a device where the processes can be dispatched for execution. While a scheduler fetches a ready process from the *Process-Queue* based on the underlying MoC, the dispatcher finds the device from the *Device-Queue* with the least load and maps the fetched process on that device. The generated kernel of the dispatched process is then executed based on the used MoC. The data communication between the host (FIFO buffers) and the OpenCL device is carried out using OpenCL buffers. For each bounded FIFO buffer, an OpenCL buffer is created with the same design and size of the FIFO buffer. When all the dispatched instances of the kernel are executed, the Runtime-Manager is then automatically notified to update the components using a synchronization mechanism. During synchronization, a set of general tasks are performed including retrieving data from OpenCL buffers, updating all the FIFO buffers of the process, updating the *Process-Queue* as well as device's load and so on.

KPN Extension. The communication and synchronization mechanism is common to all DPN classes of the framework including the newly integrated KPN MoC. Second, as the main aim of this work is to analyze and evaluate the artifacts exhibited by the underlying semantics of particular MoCs in real world. The same dispatcher is used for mapping execution on devices for the KPN MoC as used for existing MoCs in (Rafique and Schneider, 2020b).

Schedulers are designed for triggering processes for execution based on the underlying MoCs, and are therefore MoC dependent. To trigger processes for execution based on the KPN semantics, as described in Section 4.3, the Runtime-Manager is extended with a specialized KPN scheduler. The KPN MoC only triggers a process for execution if the exact information on inputs/outputs required to fire an action is available. Since the host and generated kernels are in-

dependent components, this information is required to be extracted from modeled processes at compile time. The extracted information can be used by the KPN scheduler at runtime to trigger processes for execution. To this end, we propose a systematic way of extracting the information by introducing the *input-output tree wrapper* (IOT-wrapper).

5.2.1 Introducing IOT-wrapper

The IOT-wrapper wraps the exact information of inputs/outputs required to trigger a process in a standard tree structure, while taking into account the underlying semantics of the KPN MoC. For each process in a network, a wrapper is generated at compile time from the modeled behavior. The IOT-wrapper generation based on the underlying KPN semantics is shown in Algorithm 2. For each process $p \in \mathcal{P}$ in the network $\mathfrak{K}$, the proposed algorithm works as follows:

The IOT-wrapper is initialized, first, by adding the root node, and second, by generating and assigning a function *StepFunction* to the root node (Lines 2-5). The StepFunction of the root node is generated with the code specifically related to the guards (Line 5) and hence also termed as guard node. In particular, the code is generated for each action in the order in which actions were defined to first check if each input used for guard ($F_{\gamma_j} \in F_\gamma^{act}$) has enough tokens ($\mathcal{V}_{\gamma_j}^{act} \sqsubseteq X_D^{F_{\gamma_j}}$), and second to evaluate the guard ($\mathcal{E}_\mathcal{B}^{act}(\mathcal{V}_{E_\gamma}^{act}) \rightarrow \mathcal{B}$). For each action, if the guard is true, the StepFunction returns a different number *num* $\in \mathbb{Z}$ that corresponds to a specific branch in the tree originating from the root node. The algorithm then iterates through the modeled set of actions Act in the order of there definitions (Line 7). For each action $act \in Act$, the algorithm adds nodes for all non-guarded inputs ($F_i^{act} \setminus F_\gamma^{act}$) and all outputs ($F_o^{act}$). For each non-guarded input and each output in act, the algorithm proceeds as follows: First, if the current node is the root node, a new node is added at a specific branch of the root node provided by the variable *num* (Line 11), which is incremented by 1 for each action (Line 15). On the contrary, if the current node is not the root node, a new node is always added at the branch 0 (leftest) of the current node (Line 19). Second, for a non-guarded input node, the StepFunction is generated with the code to check if that input (F_{i_j}) has enough tokens ($\mathcal{V}_{i_j}^{act} \sqsubseteq X_D^{F_{i_j}}$) (Line 13). For an output node, the StepFunction is generated with the code to check if that output (F_{o_j}) has enough space ($size(F_{o_j}) - count(X_D^{F_{o_j}}) \geq \mathcal{L}(\mathcal{V}_{o_j}^{act})$ (Line 14).

The IOT-wrapper generated for d-split, as illustrated in Listing 1, is shown in Figure 2. The root node only involves the input F_{i_1} as it is the only input used for guard by d-split. The StepFunction generated and assigned to each node is shown mathematically in dashed boxes. The set of branches originating from the root node and extending up to the leaf node represents a particular action. For instance, act_2 is represented by branches originating from the root node (F_{i_1}) and extending up to the leaf node (F_{o_2}).

Algorithm 2: IOT-wrapper generation.

```
1  foreach process p ∈ P in Network 𝔎 do
2      Initialize IOT-wrapper{
3      add root (guard) node
4      for root node, generate StepFunction{
5      ∀act ∈ Act, [∀V_γj^act ∈ V⃗_γ^act, V_γj^act ⊑ X_D^{F_γj} ∧
         E_B^act(V_Eγ^act) → B] }}
6      Initialize num to 0
7      foreach act ∈ Act do
8          current node = root node
9          foreach ( (F_ij ∈ F_i^act \ F_γ^act) ∧ (F_oj ∈ F_o^act) ) do
10             if current node == root node then
11                 new node = add child node to current node
                     at branch num
12                 for new node, generate StepFunction{
13                 V_ij^act (∈ V⃗_i^act \ V⃗_γ^act) ⊑ X_D^{F_ij} ∨
14                 size(F_oj) − count(X_D^{F_oj}) ≥ L(V_oj^act) }
15                 increment num by 1
16                 current node = new node
17             end
18             else
19                 new node = add child node to current node
                     at branch 0
20                 for new node, generate StepFunction{
                     same as Lines 13-14 }
21                 current node = new node
22             end
23         end
24     end
25 end
```

5.2.2 KPN Scheduler based on IOT-wrapper

The KPN scheduler is provided with the generated IOT-wrappers of all processes in the used network. It uses a variant of depth-first search (DFS) method (Tarjan, 1972) that starts at the root of the tree, selects a branch, and traverses through that branch as deep as possible until the leaf node is reached. In general, for each node, the scheduler calls the assigned StepFunction, and only moves to the next node if the function returns *true*. In particular, the StepFunction of the root node returns a number *num* $\in \mathbb{Z}$ mainly dependent on which guard is true. This number is

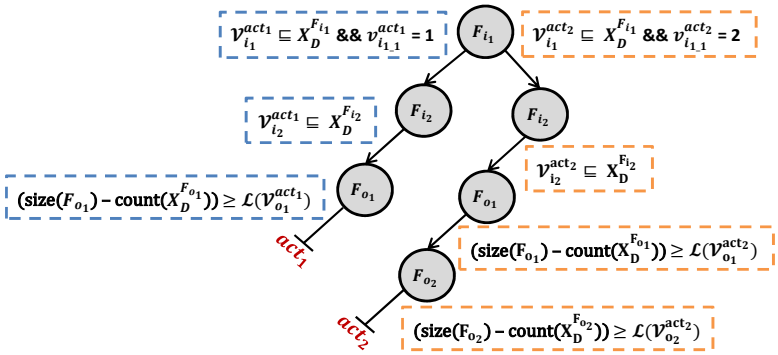

Figure 2: IOT-wrapper for d-split.

used to select a specific branch originating from the root node that directs to a specific action for which the guard is true. In case if the leaf node is reached and its StepFunction returns *true*, the scheduler triggers the process for execution. On the contrary, if the StepFunction of one of the nodes returns *false*, the process gets blocked until that node returns *true*.

6 EXPERIMENTAL EVALUATION

6.1 Benchmarks

We designed a set of simple benchmarks involving static as well dynamic behaviors. These benchmarks are typically designed to offer a variety of processes that enable the evaluation and comparison of implementations based on all three different MoCs of the framework. Each benchmark is organized in a network of three processes which are connected in a *producer-worker-consumer* setting. While the *producer* process produces the source data, the *consumer* process displays the results of the benchmark. The *worker* process performs the main operation, e.g., the d-split process illustrated in Listing 1. The list of benchmarks is shown in Table 1.

The SWITCH benchmark is designed to switch one of several input channels through to a single common output channel by the application of a control input. In contrast, the DWORKER benchmark performs the opposite operation by taking one single input channel and switching it to any one of a number of individual output channels, one at a time. The DITE benchmark is a dynamic version of the if-then-else operation that sequentially consumes data from input channels based on the value of data on a control input. In contrast, the SITE benchmark, a static version of the if-then-else operation, always consumes data from all input channels in parallel. The DMERGE benchmark is based on the d-merge process, as illustrated in Listing 2. It is designed to merge several input channels to a single common output channel by the application of a control input. In contrast, the DSPLIT benchmark, based on the d-split process as illustrated in Listing 1, splits a single input channel to a number of individual output channels. Apart from SITE, which only offers a static behavior, all other benchmarks provide dynamic behaviors.

Each benchmark is modeled and automatically synthesized (when possible) based on all three supported MoCs of the framework. Thereby, generating three different implementations, namely the DDF MoC version, the SDF MoC version and the KPN MoC version. The end-to-end performance, i.e., the total execution time of the network to process the complete input data set including initialization and termination of the program is considered as the comparison metric. The data set used has a maximum of 10,000 samples per input and the average of 50 repetitions is taken for each version.

6.2 Experimental Setup

We executed the generated versions of benchmarks on the following hardware:

- Intel i5-4460 @ 3.20GHz CPU
- AMD Radeon HD 5450 GPU
- 8GB RAM

The software environment used for execution is:

- AMD Radeon HD 5450 Driver Version 15.201.1151.1008
- Intel OpenCL SDK Version 6.3.0.1904
- Windows 10 Pro Version 1903 Build 18362.720

6.3 Evaluation

The generated implementations for each benchmark based on all three MoCs are evaluated based on their code size and the end-to-end performance.

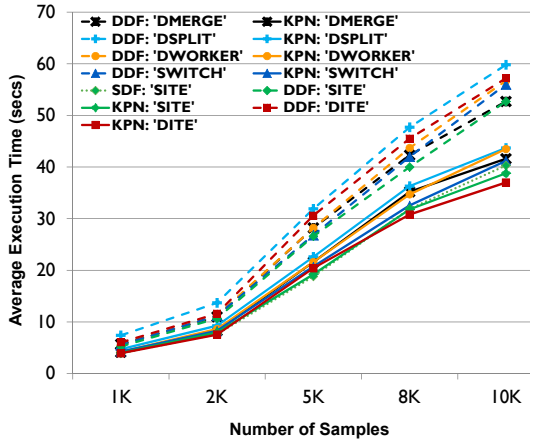

Figure 3: Performance comparison of all classes on CPU.

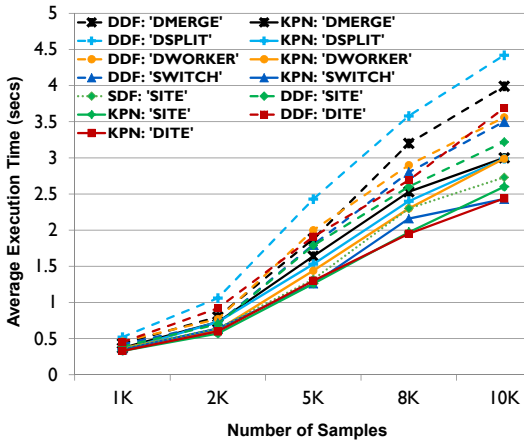

Figure 4: Performance comparison of all classes on GPU.

6.3.1 Generated Code Size

The generated code size of each benchmark for the complete network based on all three MoCs is depicted in Table 1. The code size for each generated version (implementation) of benchmark is measured as the sum of lines of code of all generated kernels for that version. Since the SDF MoC only supports static behaviors, it can only model and synthesize the SITE benchmark.

Discussion. The SDF MoC only supports static behaviors and therefore triggers a process when the data/space is available for all inputs/outputs. Hence, it generates very succinct kernel code for static processes as observed in the case of SITE. The KPN MoC also supports dynamic behaviors, however, only triggers a process for execution when the exact information on inputs/outputs required to fire an action is available. In contrast, the DDF MoC dynamically evaluates actions including their inputs/outputs when the process is triggered for execution. Therefore, it accommodates additional code for enabling the dynamic evaluation of actions within kernels at runtime. This overhead can therefore be observed from the number of lines of the generated code for each benchmark. In particular, the generated code based on the DDF MoC for SITE is 147% and 136% greater than the SDF and KPN versions, respectively. The same trend has been observed for dynamic benchmarks. The biggest difference is recorded in DITE where the generated code based on the DDF MoC is 122% greater than the KPN version.

6.3.2 End-to-End Performance

Each generated version of a benchmark is either executed on the CPU or the GPU at a time to evaluate and compare the end-to-end performance of all used

Table 1: Generated code size of each process.

Benchmarks	Lines of Code		
	SDF	DDF	KPN
SWITCH	-	148	74
DWORKER	-	124	56
DITE	-	140	63
SITE	63	156	66
DMERGE	-	142	66
DSPLIT	-	152	78

MoCs. On each target hardware, i.e., the CPU and the GPU, the average execution time of each generated version of a benchmark is measured against the number of data samples as shown in Fig. 3 and Fig. 4, respectively.

Discussion. Regardless of which target hardware is used, the average execution time of each benchmark version increases with the increase in the number of data samples. On the CPU, in general, the KPN MoC performed substantially better than the DDF MoC as shown in Fig. 3. In particular, for the only static benchmark, i.e., SITE, the KPN MoC version is about 36% faster than the DDF MoC version. For SITE, the SDF MoC performed only slightly faster than the KPN MoC for smaller number of samples. However, with the increase in the number of samples, the KPN MoC performed slightly better than the SDF MoC. For the highest number of data samples used, the KPN MoC version performed about 4% faster than the SDF MoC version. This is mainly because the SDF MoC checks for the availability of data/space for all inputs/outputs in a process and therefore induces a scheduling overhead. For all dynamic benchmarks, the KPN MoC performed at least 27% faster than the DDF MoC. The biggest difference has been recorded in the case of DITE where the KPN MoC version is about 55% faster than the DDF MoC version.

In comparison to the CPU, the average execution time of each benchmark version is highly reduced on the GPU. On average, the generated versions on the GPU executed 15x faster than on the CPU. This is mainly because the used GPU provided superior processing power over the used CPU. Similar to the CPU, the KPN MoC provided significantly improved performance for majority of the benchmarks on the GPU. In particular, for SITE, the KPN MoC version is 24% and 5% faster than the DDF MoC and SDF MoC versions, respectively. For all dynamic benchmarks, the KPN MoC performed at least 19% faster than the DDF MoC. The biggest difference has been recorded in the case of DITE where the KPN MoC version is about 51% faster than the DDF MoC version.

7 CONCLUSIONS AND FUTURE WORK

We extended a model-based design framework using different classes of dataflow process networks (DPNs) as different models of computation (MoCs) by Kahn process networks. We modeled and automatically synthesized a set of benchmarks for different target hardware architectures based on all supported MoCs of the framework, including the newly integrated KPN MoC. We evaluated all generated versions of benchmarks for their code sizes and the end-to-end performance.

Based on our evaluations, we observed that the SDF MoC generated the most succinct kernel code for static processes. The KPN MoC also supports dynamic behaviors and generated more compact kernel code than the DDF MoC. The DDF MoC used additional lines of code for dynamically evaluating actions at runtime within kernels. Furthermore, the KPN MoC provided more efficient implementations for all benchmarks in terms of end-to-end performance on all target architectures. The KPN MoC effectively performed up to 1.55x and 1.51x faster than the DDF MoC on the CPU and the GPU, respectively.

Future work aims at exploring the schemes for efficiently mapping the models on heterogeneous architectures for performance acceleration.

REFERENCES

Boutellier, J., Wu, J., Huttunen, H., and Bhattacharyya, S. (2018). PRUNE: Dynamic and decidable dataflow for signal processing on heterogeneous platforms. *IEEE Transactions on Signal Processing*, 66(3):654–665.

Brooks, C., Lee, E., and Tripakis, S. (2010). Exploring models of computation with ptolemy II. In Givargis, T. and Donlin, A., editors, *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 331–332, Arizona, USA. ACM.

Buck, J. (1993). *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, University of California, USA. PhD.

Dennis, J. (1974). First version of a data-flow procedure language. In Robinet, B., editor, *Programming Symposium*, volume 19 of *LNCS*, pages 362–376, France. Springer.

Eker, J. and Janneck, J. (2003). CAL language report. ERL Technical Memo UCB/ERL M03/48, EECS Department, University of California at Berkeley, Berkeley, California, USA.

Engels, M., Bilsen, G., Lauwereins, R., and Peperstraete, J. (1995). Cyclo-static dataflow. In *International Conference on Acoustics, Speech and Signal Processing*, pages 3255–3258, USA. IEEE Computer Society.

Golomb, S. (1971). Mathematical models: Uses and limitations. *IEEE Transactions on Reliability*, R-20(3):130–131.

Haubelt, C., Schlichter, T., Keinert, J., and Meredith, M. (2008). SystemCoDesigner: automatic design space exploration and rapid prototyping from behavioral models. In Fix, L., editor, *Design Automation Conference (DAC)*, pages 580–585, Anaheim, California, USA. ACM.

Kahn, G. and MacQueen, D. (1977). Coroutines and networks of parallel processes. In Gilchrist, B., editor, *Information Processing*, pages 993–998. North-Holland.

Karp, R. and Miller, R. (1966). Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics (SIAP)*, 14(6):1390–1411.

Kuhn, T., Forster, T., Braun, T., and Gotzhein, R. (2013). FERAL - framework for simulator coupling on requirements and architecture level. In *Formal Methods and Models for Codesign*, pages 11–22, USA. IEEE Computer Society.

Lee, E. and Messerschmitt, D. (1987). Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245.

Lee, E. and Parks, T. (1995). Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801.

Lund, W., Kanur, S., Ersfolk, J., Tsiopoulos, L., Lilius, J., Haldin, J., and Falk, U. (2015). Execution of dataflow process networks on OpenCL platforms. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 618–625, Finland. IEEE Computer Society.

Parks, T. (1995). *Bounded Scheduling of Process Networks*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California. PhD.

Rafique, O. and Schneider, K. (2020a). Employing OpenCL as a standard hardware abstraction in a distributed embedded system: A case study. In *Conference on Cyber-Physical Systems and Internet-of-Things*, Budva, Montenegro. IEEE Computer Society.

Rafique, O. and Schneider, K. (2020b). SHeD: A framework for automatic software synthesis of heterogeneous dataflow process networks. In *Euromicro Conference on Digital System Design (DSD) and Software Engineering and Advanced Applications (SEAA)*, Portoroz, Slovenia. IEEE Computer Society.

Schor, L., Bacivarov, I., Rai, D., Yang, H., Kang, S.-H., and Thiele, L. (2012). Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Compilers, Architecture, and Synthesis for Embedded Systems*, pages 71–80, Finland. ACM.

Schor, L., Tretter, A., Scherer, T., and Thiele, L. (2013). Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of OpenCL. In *IEEE Symposium on Embedded Systems for Real-time Multimedia*, pages 41–50. IEEE Computer Society.

Stone, J., Gohara, D., and Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12(3):66–73.

Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160.