Embedded Software Synthesis using Heterogeneous Dataflow Models

Dissertation

Vom Fachbereich Informatik der Technischen Universität Kaiserslautern zur Verleihung des akademischen Grades Doktor der Ingenieurwissenschaften (Dr.-Ing.) genehmigte Dissertation

von

Omair Rafique

Datum der wissenschaftlichen Aussprache	10.09.2021
Dekan	Prof. Dr. Jens Schmitt
Gutachter	Prof. Dr. Klaus Schneider
	Prof. Dr. Christian Haubelt

D 386

Abstract

Dataflow process networks (DPNs) consist of statically defined process nodes with First-In-First-Out (FIFO) buffered point-to-point connections. DPNs are intrinsically data-driven, i.e., node actions are not synchronized among each other and may fire whenever sufficient input operands arrived at a node. In this original form, DPNs are data-driven and therefore a suitable model of computation (MoC) for asynchronous and distributed systems. For DPNs having nodes with only static consumption/production rates, however, one can easily derive an optimal schedule that can then be used to implement the DPN in a time-driven (clock-driven) way, where each node fires according to the schedule.

Both data-driven and time-driven MoCs have their own advantages and disadvantages. For this reason, desynchronization techniques are used to convert clock-driven models into data-driven ones in order to more efficiently support distributed implementations. These techniques preserve the functional specification of the synchronous models and moreover preserve properties like deadlock-freedom and bounded memory usage that are otherwise difficult to ensure in DPNs. These desynchronized models are the starting point of this thesis.

While the general MoC of DPNs does not impose further restrictions, many different subclasses of DPNs representing different dataflow MoCs have been considered over time like Kahn process networks, cyclo-static and synchronous DPNs. These classes mainly differ in the kinds of behaviors of the processes which affect on the one hand the expressiveness of the DPN class as well as the methods for their analysis (predictability) and synthesis (efficiency). A DPN may be heterogeneous in the sense that different processes in the network may exhibit different kinds of behaviors. A heterogeneous DPN therefore can be effectively used to model and implement different components of a system with different kinds of processes and therefore different dataflow MoCs.

Design tools for modeling like Ptolemy and FERAL are used to model and to design parallel embedded systems using well-defined and precise MoCs, including different dataflow MoCs. However, there is a lack of automatic synthesis methods to analyze and to evaluate the artifacts exhibited by particular MoCs. Second, the existing design tools for synthesis are usually restricted to the weakest classes of DPNs, i.e., cyclo-static and synchronous DPNs where

each tool only supports a specific dataflow MoC.

This thesis presents a model-based design based on different dataflow MoCs including their heterogeneous combinations. This model-based design covers in particular the automatic software synthesis of systems from DPN models. The main objective is to validate, evaluate and compare the artifacts exhibited by different dataflow MoCs at the implementation level of embedded systems under the supervision of a common design tool. We are mainly concerned about how these different dataflow MoCs affect the synthesis, in particular, how they affect the code generation and the final implementation on the target hardware. Moreover, this thesis also aims at offering an efficient synthesis method that targets and exploits heterogeneity in DPNs by generating implementations based on the kinds of behaviors of the processes.

The proposed synthesis design flow therefore generally starts from the desynchronized dataflow models and automatically synthesizes them for cross-vendor target hardware. In particular, it provides a synthesis tool chain, including different specialized code generators for specific dataflow MoCs, and a runtime system that finally maps models using a combination of different dataflow MoCs on the target hardware. Moreover, the tool chain offers a platform-independent code synthesis method based on the open computing language (OpenCL) that enables a more generalized synthesis targeting cross-vendor commercial off-the-shelf (COTS) heterogeneous platforms.

Danksagung

First of all, I would like to express my sincere gratitude to my supervisor and advisor Prof. Klaus Schneider for all his guidance, support and advice throughout the journey of my doctoral studies. I would really like to thank him for his time that he invested in correcting my papers and raising the quality of publications. Also, I would like to thank him for providing me the opportunity to work on different projects and collaborations. This has really helped me in gaining new knowledge and applying it to my research work. I have always learned a lot from him from all the inspiring and interesting discussions on both the professional and the personal level. Thank you for always being so kind, helpful and polite. I would also like to sincerely thank Prof. Christian Haubelt for accepting to review this work. I am also really grateful to Prof. Stefan Deßloch for ensuring a smooth conduct of my PhD defense as the head of the committee.

I am really thankful to Ms. Marita Stuppy who has always helped me with the administrative tasks with great kindness and concern. I would especially like to thank all my colleagues Anoop, Julius, Marc, Martin, Maximilian, Tripti and Xiao for enabling a great atmosphere in the group and helping me out with my research work whenever needed. I would also like to especially thank Dr. Yu Bai who helped me a lot with all his valuable discussions and suggestions on this research work.

Most importantly, I would like to dedicate this thesis to my parents, my wife and my whole family. Their unconditional love and support have always motivated me in my life. I truly would like to thank them for encouraging me in all of my pursuits and inspiring me to follow my dreams.

September 2021, Omair Rafique

Contents

List of Figures

Acronyms

AIF Averest intermediate format

BAS Building automation system

CAL Cal actor language

CPU Central processing unit

CSDF Cyclo static dataflow

CU Compute unit

DDF Dynamic dataflow

DLP Data level parallelism

DPN Dataflow process network

FIFO First-In-First-Out

FSM Finite state machine

GPU Graphical processing unit

HSDF Homogeneous static dataflow

KPN Kahn process network

MoC Models of computation

OpenCL Open computing language

OS Operating system

PE Processing element

SDF Static dataflow

SGA Synchronous guarded actions

SR Synchronous reactive

Symbols

p	a process
$\mathcal{P}$	a set of processes
f	a FIFO buffer
$\mathcal{F}$	a set of FIFO buffers
α	an action
$\mathsf{actions}(p)$	the set of actions of process p
γ_α	the guard of action α
$\mathsf{inBuf}(p)$	the set of input buffers of process p
$\mathsf{outBuf}(p)$	the set of output buffers of process p
$\mathsf{inAct}(\alpha)$	the subset of input buffers used by action α
$\mathsf{outAct}(\alpha)$	the subset of output buffers used by action α
$\mathsf{inGrd}(\alpha)$	the subset of input buffers used by the guard γ_α
$\mathcal{D}$	a domain of values
$\mathcal{D}^*$	the set of finite sequences on $\mathcal{D}$
$\mathcal{D}^\omega$	the set of infinite sequences on $\mathcal{D}$
$\mathcal{D}^\infty$	the union of the sets of finite and infinite sequences

Chapter 1

Introduction

1.1. Motivation and Problem Setting

The ever-increasing functionality and the non-functional constraints in modern embedded systems lead to an enormous growth in the complexity at the system level. Dealing with the resulting complexity, many different system architectures have been introduced in embedded system design that are integrating heterogeneous single-core and multi-core processors with application specific hardware and even more specific sensors and actors. Besides this heterogeneity of the platforms, also the reduced time-to-market has imposed new challenges for designers to develop products within a short period of time. The trends of new system architectures and the tight time-to-market deadlines have enforced model-based design flows not just to satisfy functional requirements but also to better address non-functional characteristics like performance, portability and scalability that are otherwise not intrinsically supported by traditional design flows and programming paradigms.

Model-based design flows for embedded systems have been introduced to allow late design changes while still keeping tight time-to-market deadlines. A model-based design is generally characterized with a hardware-agnostic abstract model based on a particular model of computation (MoC) and is equipped with a tool chain typically providing simulators, tools for verification, code generators, and tools for synthesis. A MoC precisely determines *why, when, and which atomic action of a system is executed.* The clock-driven or synchronous reactive (SR) MoC [Schn09; BeGo92; BCEH03] describes a system with an abstract notion of time (the clock). All components (modules) of a synchronous system react in parallel to their environment in a sequence of logical ticks and automatically synchronize after each tick. This abstraction imposed in the form of perfect synchrony allows deterministic semantics, which makes the SR MoC well suited for formal analysis, simulation, verification and synthesis of embedded systems. However, this synchrony hypothesis can cause an unnecessary burden in the form of synchronization overhead, in particular, for the implementation of distributed and asynchronous systems.

In contrast, dataflow process networks (DPN) [KaMi66; Denn74; Kahn74]

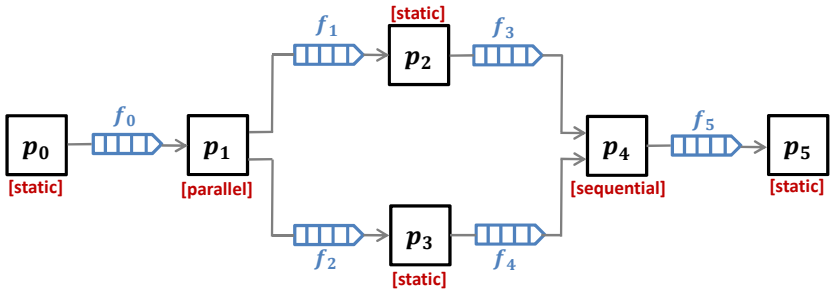

Figure 1.1.: *A simple visualization of a heterogeneous DPN. It consists of different kinds of processes characterized by static, sequential and parallel behaviors. The processes ($p_0 \dots p_5$) are connected together via FIFO buffered point-to-point channels ($f_0 \dots f_5$).*

consist of statically defined process nodes with first-in-first-out (FIFO) buffered point-to-point connections. In their natural form, DPNs are data-driven, i.e., process nodes are not synchronized among each other and may fire whenever sufficient data is available at a node. In this original form, DPNs are data-driven and therefore a suitable MoC for asynchronous and distributed systems. However, model-based designs starting from DPNs may suffer from problems like deadlocks and buffer overflows [Park95; GeBa03]. Preserving properties like deadlock-freedom and bounded memory usage in DPNs is in general not decidable. Therefore, both data-driven and clock-driven MoCs have their own advantages and disadvantages. As an alternative, *desynchronization* of synchronous models [BeCG99; Gira05a] has been recently developed that benefit both from the static analysis methods for synchronous systems and the performance of the finally synthesized asynchronous systems. Desynchronization techniques [BeCG99; PoCB06; TOGB12; Bai16] are used to convert clock-driven models into data-driven ones in order to more efficiently support distributed implementations. These techniques preserve the functional specification of the synchronous models and moreover preserve properties like deadlock-freedom and bounded memory usage that are otherwise difficult to ensure in DPNs. These desynchronized models are the starting point of this thesis.

While the general MoC of DPNs does not impose further restrictions, many different subclasses of DPNs have been considered over time like Kahn process networks (KPN) [Kahn74], static dataflow (SDF) [PaPL95] networks and Boolean dataflow (BDF) [Buck93] networks. Each class defines a specific dataflow MoC by specifying a particular execution and communication semantics [FHTZ17]. These classes mainly differ in the kinds of behaviors of the processes which affect 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 process is triggered for an execution, and how each execution of a process consumes/produces data. A process in a static DPN exhibits a static behavior where a statically determined amount of data

is consumed and produced in each execution. Second, a process in a KPN may possess a sequential behavior where a dynamically determined amount of data is consumed and produced sequentially in each execution. In contrast, a process in a dynamic dataflow (DDF) [Kosi78] network may have a parallel behavior that can consume and produce a dynamically determined amount of data in parallel. A DPN may be homogeneous, i.e, all processes possess the same kind of behavior, or it may be heterogeneous where different processes exhibit different kinds of behaviors. A heterogeneous combination of particular kinds of processes can be used to model and implement different components of a system with different kinds of processes and therefore different dataflow MoCs. A simple example of a heterogeneous DPN consisting of different kinds of behaviors of the processes is visualized in Figure 1.1.

Design tools for modeling like Ptolemy [EJLL03; BrLT10] and FERAL [KFBG13] are used to model and to design parallel embedded systems using well-defined and precise MoCs, including different dataflow MoCs. The main emphasis of these design tools is to analyse different MoCs, including their heterogeneous combinations, for modeling and designing embedded systems. Some of these modeling design tools also introduced at some stage a limited synthesis facility, supporting only platform-dependent synthesis methods usually restricted to implementations based on particular dataflow MoCs. We therefore appreciate the convenient use of these well established frameworks to study and to analyse different MoCs at the design level. However, there is a lack of automatic software synthesis methods to analyse and to evaluate the artifacts exhibited by different dataflow MoCs and especially their heterogeneous combinations.

Second, the existing design tools for software synthesis like [STST13; LKET15; BWHB18] are usually restricted to the weakest classes of DPNs, i.e., cyclo-static and static DPNs where each tool only supports a specific dataflow MoC. These tools provide specialized tool chains, in particular, a specialized code generator for a specific MoC. Each tool therefore allows one to model and to implement systems based on a specific dataflow MoC. A design tool that only supports a static dataflow MoC can be used for the modeling and synthesis of static (synchronous) behaviors. Similarly, a design tool based on a dynamic dataflow MoC can be employed for dynamic and asynchronous behaviors. A design tool like the open RVC-CAL compiler (ORCC) [YLJC13a] that is based on a more generalized dataflow MoC (DDF) can also be used to implement a behavior that precisely belongs to a more restricted dataflow MoC (e.g., SDF). However, when it comes to synthesis, in particular, the code generation and scheduling, this may lead to an inefficient implementation mainly because of the runtime overhead caused by a more relaxed semantics than needed. Moreover, for heterogeneous DPNs offering heterogeneous combinations of different kinds of behaviors of the processes, the synthesis method should exploit the heterogeneity by generating efficient implementations based on the precise dataflow MoC of each process.

Apart from efficiency, another crucial challenge is the portability of applications on different cross-vendor platforms which is not systematically handled

by the traditional design flows. In general, the final result of these design flows is more or less a set of automatically generated C programs. Consequently, a non-trivial manual effort is finally required for deploying these programs to a particular target architecture. To further automate the design process, a systematic approach is needed to implement and deploy the modeled systems on different cross-vendor target platforms in an automatic way that avoids tedious and error-prone manual steps.

The overall motivation of this thesis is to enable the automatic software synthesis of systems using different well-defined and precise dataflow MoCs including their heterogeneous combinations. The main objective is to validate, evaluate and compare the artifacts exhibited by different dataflow MoCs at the implementation level of systems under the supervision of a common design tool. Moreover, the idea is to offer an efficient synthesis method that exploits heterogeneity in dataflow networks by generating implementations based on the kinds of behaviors of the processes. Finally, this thesis also considers the challenge of systematically handling the portability of modeled systems on cross-vendor heterogeneous platforms as an integral part of the synthesis process.

1.2. Related Work

A number of model-based design tools have been presented over time for the design and development of embedded systems. This section covers a number of well-established design tools, categorized mainly from the perspective of desired goals, employed strategies and usage as given in the following sections.

1.2.1. Design Tools for Modeling

The Ptolemy project [EJLL03; BrLT10] is a design tool originally constructed in a Java-based environment to support the modeling and simulation of behaviors based on different MoCs, including particular dataflow MoCs. It provides a common platform for organizing a system into different domains characterized as directors. Each enclosing director represents a semantic model based on a specific MoC and triggers the execution of the contained components in accordance to the implemented semantics. The heterogeneous combination of MoCs is therefore realized by coupling different directors within an application scenario. The main emphasis of Ptolemy project is to analyse different MoCs, including their heterogeneous combinations, for modeling and designing embedded systems. Hence, the main focus is to study and analyse different MoCs at the design level. However, it also provides 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, the code generation facility targets implementations based on particular individual MoCs.

FERAL [KFBG13] is another framework that supports heterogeneous modeling and simulation. It is developed to provide a holistic model-based design

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

approach to enable the coupling of specialized simulators in offline scenarios, i.e., without connecting them to real hardware. This project very interestingly adopts and extends the concepts from the Ptolemy project.

The formal system design (ForSyDe) [SaJA17] tool offers a formal design methodology for embedded systems based on different MoCs including the SR MoC and two particular dataflow MoCs. The ForSyDe modeling framework is characterized as language independent where currently two versions are available as a set of libraries in Haskell[2] and SystemC[3]. Although the major focus of this design tool is the modeling framework, it also provides a hardware synthesis tool that has been mainly elaborated for translating models limited to the SR MoC into the corresponding VHDL code. Another synthesis plug-in called *f2cc*[4] has been introduced for generating GPGPU software code from models limited to the SR MoC.

The SystemC models of computation (SysteMoC) [HFKS07] is an actor-oriented dataflow programming language built on top of SystemC. It supports different precise and well-defined dataflow MoCs and is available as an open source C++ class library. Besides supporting different dataflow MoCs, it also offers the automatic MoC identification of processes (actors), which is not featured in frameworks like Ptolemy and ForSyDe. The identification of different MoCs in SysteMoC is mainly realized by analysing the communication behavior. The communication behavior of an actor is typically specified by an explicit finite state machine model. Although, the main focus of SysteMoC has been at the design level, the System-CoDesigner [HFKS07; HSKM08] framework specializes in automatic design space exploration and rapid prototyping starting from SysteMoC models. In particular, the framework offers a platform-based automatic system generation from SysteMoC models. Furthermore, it also supports the generation of Verilog/VHDL code using high-level commercial synthesis tools like Forte Cynthesizer.

SDF for free (SDF^3) [StGB06a] is a versatile experimental tool that can generate random static dataflow graphs (SDFGs), with support to analyse and visualize these graphs. It supports three different classes of static DPNs, namely the static dataflow (SDF) [LeMe87a], the cyclo-static dataflow (CSDF) [EBLP95] and the scenario aware dataflow (SADF) [SGTB11]. The tool includes an extensive library of SDFG analysis and transformation algorithms as well as functionality to visualize and simulate them.

1.2.2. Design Tools for Synthesis

Model-based design tools for synthesis in the related state-of-the-art mainly differ by their employed MoCs. A number of dataflow oriented design tools have been presented where each tool usually only supports a specific dataflow MoC. To this end, some of the inspiring model-based design tools for synthesis are presented in [STST13; LKET15; BWHB18; BBJE09; BTRM14; BoHa16;

[2] `https://www.haskell.org/`
[3] `https://www.accellera.org/downloads/standards/systemc`
[4] https://github.com/forsyde/f2cc/wiki

SBRY12] and [BoGh15] (to name a few).

The framework presented in [STST13] introduces a design flow for executing applications specified as SDF graphs on heterogeneous systems using the open computing language (OpenCL) [StGS10]. The main focus of this work is to develop and to provide features and concepts to better utilize the parallelism and thereby improving end-to-end throughput in heterogeneous architectures. However, it only supports the execution of behaviors limited to the SDF MoC.

The work presented in [LKET15] translates DPNs modeled using a subset of Cal actor language (CAL) [EkJa03], namely the RVC-CAL language [BoNy15], to parallel programs running some of the computations on OpenCL. The methodology incorporates static analysis and transformations and thus confined to the synthesis of SDF models. Similarly, another dataflow oriented framework [BWHB18] proposes a dataflow MoC as a symmetric-rate dataflow, a restricted form of SDF, where the token production rate and the token consumption rate per FIFO channel is symmetric.

The Open DataFlow (OpenDF) [BBJE09] is a compilation framework built under the Eclipse environment that models DPNs based on CAL. It uses a dedicated software back-end (CAL2C) [RWRJ08] and a dedicated hardware back-end [JMPR08] for the generation of C code and RTL descriptions (Verilog), respectively. Similarly, in [BTRM14], the HW/SW co-design methodology based on the RVC-CAL language, is built as an Eclipse plug-in on top of open RVC-CAL compiler (ORCC)[5] and OpenForge[6]. The open-source tools are used as a tool chain of the framework, capable of providing simulation and the HW/SW synthesis. The final implementations provided by these frameworks are based on a dynamic dataflow MoC.

Another approach presented in [BoHa16] is aimed to provide a dataflow programming framework for enabling the execution of behaviors based on a dynamic dataflow MoC on GPU devices. The framework targets modeling a system based on dynamic dataflow and allows the mapping of actors with a data-dependent consumption of inputs and a data-dependent production of outputs.

The distributed application layer (DAL) framework [SBRY12] presents a scenario-based design flow for mapping streaming applications onto heterogeneous on-chip many-core systems. Behaviors are modeled based on a specific dataflow MoC, namely the KPN MoC [Kahn74], and the execution scenarios are coordinated using a finite state machine (FSM), where each scenario is represented by a state. The work presented in [BoGh15] proposed the DAL based design flow to execute DPNs modeled as RVC-CAL programs on multi-core platforms. A dedicated DAL back-end based on the C back-end of ORCC is proposed to translate RVC-CAL actors to DAL processes.

[5] `http://orcc.sourceforge.net`
[6] `https://sourceforge.net/projects/openforge`

1.2.3. Design Tools in Industry

One of the most popular and commercially recognized model-based design tool Matlab[7] has introduced a variety of supporting toolkits over time. The modeling toolkit *Simulink* provides a graphical extension for modeling and simulation of systems. Similarly, the *Embedded Coder* generates C and C++ files for embedded software processors. The *Simulink Design Verifier* and *Polyspace* are introduced for the formal verification of models and code, respectively. Interestingly, Matlab Simulink introduced the dataflow domain[8] where applications can be modeled and simulated based on the SDF MoC. It automatically partitions the model, generates the static scheduler and simulates the system using multiple threads. The main objective of introducing the dataflow domain is to improve the simulation throughput with multithreaded execution.

The Signal Processing Worksystem (SPW) from Cadence Design Systems[9] supports the modeling and analysis of signal processing algorithms based on static as well as dynamic dataflow models. The actors can be modeled and organized in a hierarchical setting using several different models like SystemC, Matlab, C/C++, Verilog, VHDL, or the design library from SPW. The design flow mainly focuses on the simulation and manual refinement of modeled systems.

CoCentric System Studio (previously known as COSSAP) from Synopsys[10] is a system-level design solution consisting of tools, methodologies, and libraries that enables the design and simulation of systems-on-a-chip. It supports languages like C/C++, SystemC, VHDL, Verilog and others. The System Studio provides a wide variety of modeling capabilities to capture complex systems efficiently. The modeling paradigms can be hierarchically mixed at all levels for e.g., based on nested dataflow models and FSMs. The main emphasis of the design flow is the modeling and analysis of complex systems.

1.2.4. Summary

In general, model-based design tools for embedded systems that support heterogeneous combinations of MoCs including different dataflow MoCs have found their major interest in the modeling, simulation, and analysis of complex systems. These frameworks, developed and evolved over decades, are being conveniently used to formally analyse different MoCs for modeling and designing embedded systems. Some of the design tools in this category also introduced a synthesis facility, supporting only platform-dependent synthesis methods usually restricted to implementations based on particular MoCs. Thus, there is a lack of automatic synthesis methods to analyse and to evaluate the artifacts exhibited by different dataflow MoCs. Second, the existing design tools for synthesis are usually dedicated for automatically implementing

[7] `http://www.mathworks.com/matlabcentral/`
[8] `https://www.mathworks.com/help/dsp/ug/dataflow-domains.html/`
[9] `https://www.cadence.com/`
[10] `https://www.synopsys.com/`

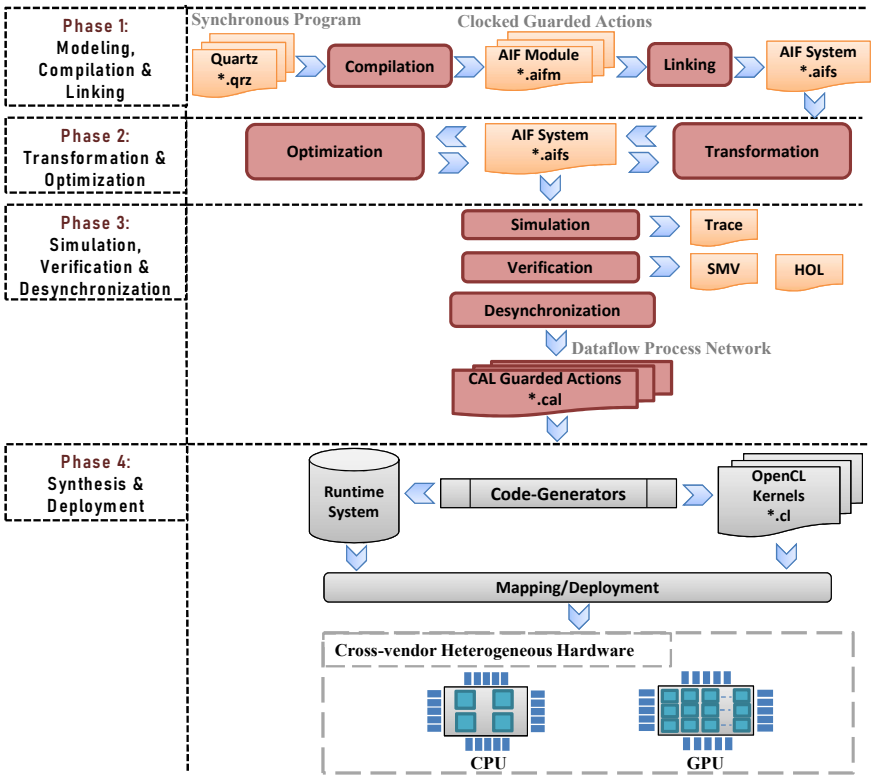

Figure 1.2.: *The overall design flow based on Averest is organized in four phases: the system behavior is modeled as a synchronous program which is compiled to an intermediate format (AIF). The AIF description is used to perform code optimizations and transformations as well as to simulate and verify the synchronous components. If verified for desynchronization, the components are transformed into a DPN which is then automatically synthesized and deployed on the target hardware.*

systems based on a specific dataflow MoC. Therefore, a common synthesis design flow is still needed that mainly focuses and emphasizes on the automatic software synthesis of systems based on different dataflow MoCs. Moreover, an efficient software synthesis method is desired that targets and essentially exploits the heterogeneity in the network by generating implementations based on the precise dataflow MoC of each process.

1.3. Contributions

As a long term project, our group developed the *Averest*[11] tool for a model-based design process starting with synchronous models. *Averest* is a framework for the specification, verification and implementation of synchronous reactive systems. Furthermore, the work presented in [Bai16] offers a desynchronization design flow using *Averest* that starts from the synchronous model of a system and then transforms it into an asynchronous model that can be

[11] `http://www.averest.org`

directly used for the synthesis of the distributed applications [BaRS21]. The complete design flow based on *Averest* can be expressed in four phases as shown in Figure 1.2.

The design flow is based on the imperative synchronous programming language *Quartz* [Schn09], which is a variant of Esterel [BeGo92]. In Phase 1, at the start of the design flow, the system behavior is specified in *Quartz* which can then be compiled to the *Averest* intermediate format (AIF). In Phase 2, the intermediate format is utilized to perform code optimization and transformations such as dead and passive code elimination [BrSB14] and static single assignment SSA format generation. In Phase 3, the system behavior is simulated and verified by directly using the intermediate description. If synchronous components are successfully verified for desynchronization, the AIF code is transformed into a DPN. The underlying language of the target DPN model is the Cal actor language (CAL) [EkJa03]. These desynchronized models (DPNs) are the starting point of this thesis.

This thesis presents a model-based synthesis design flow based on different dataflow MoCs including their heterogeneous combinations. This model-based design covers in particular the automatic software synthesis of systems from DPN models. The proposed synthesis method has been successfully used as Phase 4 of the complete design flow based on *Averest* as shown in Figure 1.2. In particular, in Phase 4, the target code is automatically synthesized from the desynchronized code for different cross-vendor target hardware. Moreover, the synthesis design flow has also been implemented in a standalone framework.

The main objective of this thesis is to validate, evaluate and compare the artifacts exhibited by different well-defined and precise dataflow MoCs at the implementation level of systems under the supervision of a common design tool. To this end, we propose a synthesis design flow that essentially enables the automatic software synthesis of systems based on different dataflow MoCs. In particular, it supports three different dataflow MoCs, namely the synchronous (static) dataflow (SDF) [LeMe87a] MoC, the Kahn process networks (KPN) [KaMa77] MoC and a deterministic variant of the dynamic dataflow (DDF) [Kosi78] MoC. The common design tool can be effectively used to generate implementations based on the individual dataflow MoCs [RaSc21; RaSc19a; RaSc18]. Moreover, in contrast to existing dataflow oriented model-based synthesis methods, the proposed design flow offers an efficient synthesis method that targets and exploits heterogeneity in dataflow networks by generating implementations purely based on the kinds of behaviors of the processes or the underlying precise dataflow MoC of each process [RaSc20b]. Hence, each process in the network is scheduled and executed based on the underlying precise dataflow MoC.

The target DPN model of our desynchronization method is based on a limited subset of CAL. The proposed synthesis design flow considers a general DPN model based on this subset of CAL that is used with specific constraints and restrictions to specify the supported dataflow MoCs. Second, it provides a comprehensive synthesis tool chain, including different specialized code generators for specific dataflow MoCs, and a runtime system that finally maps

models using a combination of different dataflow MoCs on the target hardware. The tool chain essentially offers a platform-independent code synthesis method based on the open specification language (OpenCL) abstraction that enables a more generalized synthesis targeting commercial off-the-shelf (COTS) heterogeneous architectures. In particular, this thesis focuses on mapping modeled systems on cross-vendor multi-core CPUs and many-core GPUs as depicted in Figure 1.2. We target dynamic application environments involving a number of data samples at a time. This allows us to potentially schedule and map multiple executions of dataflow models on the target devices at a time that are then executed either sequentially or in parallel depending on the underlying MoC. The proposed synthesis method therefore enables us to analyse and evaluate the artifacts exhibited by different dataflow MoCs on heterogeneous platforms consisting of CPUs and parallel architectures like GPUs.

The main contributions of this thesis can be summarized as follows:

- We proposed an automatic model-based synthesis design flow that enables us to synthesize systems using different dataflow MoCs, namely the SDF MoC, the KPN MoC and a deterministic variant of the DDF MoC.
- We implemented a platform-independent code synthesis method for CAL DPN models. In particular, we offer a synthesis tool chain that automatically synthesizes CAL models into OpenCL code which is platform independent. This enables a more generalized synthesis not restricted to devices like multi-core processors.
- We describe the supported dataflow MoCs and present their corresponding code generators based on the employed CAL DPN model.
- We offer a single back-end based on OpenCL which is comprised of different specialized code generators for specific dataflow MoCs.
- We present the centralized host and the runtime system designed under the OpenCL abstraction for finally deploying DPNs on cross-vendor COTS target hardware.

1.4. Outline

The remainder of this thesis is organized in the following chapters:

- Chapter 2 highlights the background of this thesis by presenting some important preliminaries. This includes introducing the tools, languages and specifications used as essential parts of the proposed design flow.
- Chapter 3 overviews the proposed design flow and presents the experimental tools especially developed for evaluating the feasibility of the used languages and specifications.

- Chapter 4 focuses on presenting the supported dataflow MoCs. In particular, it first introduces the considered general model of DPNs, which is then used with specific restrictions and constraints to represent the precise dataflow MoCs.
- Chapter 5 presents the complete synthesis tool chain including the specialized code generators and the runtime system. The synthesis method employed for heterogeneous DPNs is also covered in this chapter.
- Chapter 6 demonstrates the proposed design flow by presenting a number of standalone benchmarks as well as a number of interesting case studies. The results are presented for validating, evaluating and comparing the artifacts exhibited by different supported dataflow MoCs.
- Chapter 7 draws the conclusions and discusses the future prospects of the presented synthesis framework.

Chapter 2

Background

Contents

2.1. Models of Computation

Model-based design flows have been introduced to simplify the design process, to trace out specification discrepancies and design errors earlier in the development cycle, and to reduce the overall time-to-market [BTRM14; EJLL03; HSKM08; SZTK04]. These design flows are based on models of computation (MoCs) that precisely determine *why, when and which atomic action of a system is executed*. A MoC specifies in general what triggers the execution of a component, how each execution of a component consumes/produces data, and how these components communicate with each other. Depending on applications (hard or soft real-time systems) as well as on the target architectures (single-core, multi-core or distributed), these MoCs carry their own advantages and disadvantages. Thus, for an efficient model-based design process targeting different applications and architectures, it is important to have the

ability to translate parts of one MoC to another one. This section discusses in general the synchronous MoC and the dataflow process network (DPN) MoC with their advantages and disadvantages, and talks about the translation of synchronous models to DPNs targeting particular applications and architectures.

2.1.1. Synchronous Model of Computation

The clock-driven, time-driven, or synchronous reactive (SR) MoC [Schn09; BeGo92; BCEH03] organizes the behavior of a system into logical steps (ticks) with an abstract notion of time (the clock). Each logical step is referred to as a reaction of the system. All components (modules) of a synchronous system react in parallel to their environment in a sequence of logical steps. In each reaction, all inputs are read and all outputs are instantaneously computed by all modules in parallel where values are instantaneously communicated between modules. The SR MoC forces components to communicate within one step and therefore the communication requires no buffers at all. The outputs of components in a synchronous program are conceptually simultaneous with their inputs, which makes every component within the step to run instantaneously. This is called perfect synchrony. However, components can not run at any order, instead, they have to follow causal rules, i.e. the modules are partially ordered according to their data dependencies. This abstraction imposed in the form of perfect synchrony allows a simplified reasoning about time in a synchronous design. In particular, this offers many advantages like the ability to model concurrent behaviors, deterministic simulation, formal analysis, verification and synthesis of embedded systems. The SR MoC follows the same abstraction of time as synchronous hardware circuits and thus can be visualized as shown in Figure 2.1. At each logical tick of the global logical clock, all the modules (A,B and C) are triggered. Then, all inputs (x1, x2) are read, output results (y1, y2) are computed, and instantaneously communicated via signal connections or wires.

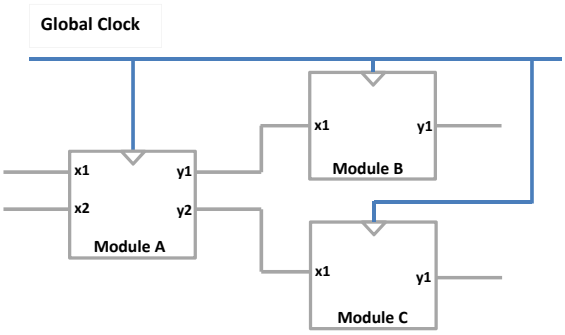

Figure 2.1.: *A synchronous network consisting of modules (A, B, and C) driven by a global clock.*

Model-based design methods based on the SR MoC have been established where either a sequential program or an application-specific hardware can be

generated from one and the same synchronous model. Synchronous models are deterministic and therefore provide in particular predictable temporal behaviors, which greatly simplifies many efforts in the verification and validation of hard real-time systems. They have proven their usefulness on both single-core and multi-core platforms in safety critical applications such as aviation [DPIC19] and other embedded system industries. However, when it comes to soft real-time applications such as streaming and signal processing [LKET15], performance and design flexibility are dominant factors over safety, and commercial off-the-shelf (COTS) heterogeneous hardware platforms are preferred from both the perspective of marketing as well as performance [BaRS21]. With the advent of multi/many core heterogeneous platforms in embedded systems, the generation of distributed implementations is often desired for such applications, where different components are mapped and executed on different computing units (devices). The SR MoC may induce an unnecessary burden in the form of synchronization overhead and excessive communication for such implementations mainly because the components would have to synchronize after each reaction step. Therefore, synchronization and communication overheads caused by the synchronous semantics often reduce the performance to an unacceptable level and is therefore not well suited for the development of such applications especially when implemented on distributed or even single-platform heterogeneous architectures involving multi/many core devices.

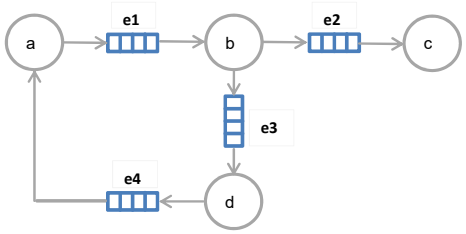

Figure 2.2.: *A dataflow process network visualized as a graph consisting of vertices (a,b,c and d) and directed edges (e1, e2, e3 and e4).*

2.1.2. Dataflow Process Networks

A dataflow process network (DPN) describes the behavior of a system in a set of statically defined process nodes with FIFO buffered point-to-point channels. As shown in Figure 2.2, a DPN can be visualized as a graph consisting of vertices (a, b, c, d) called process nodes (or simply processes) and directed edges (e1, e2, e3, e4) called channels having FIFO buffers. A process of a DPN represents a single computing unit in the network that models functionality or computations to be executed. Each channel consisting of a FIFO buffer connects at most two processes, i.e. one consumer that reads data values (tokens) from the head of the buffer and one producer that writes data values at the tail of the buffer. In general, the channels are conceptually considered to be unbounded unless otherwise stated. Each process performs a computation

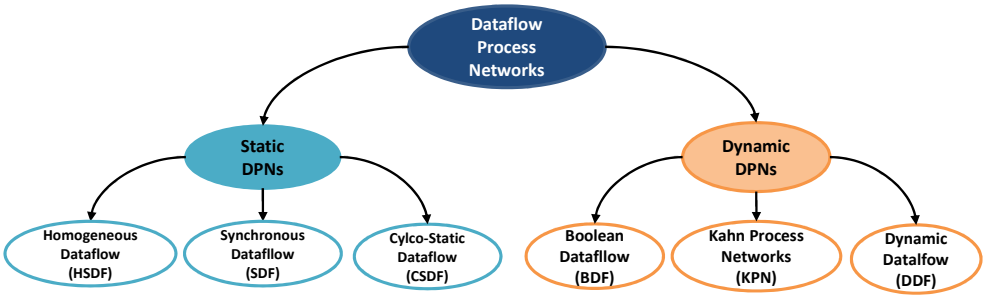

Figure 2.3.: *Categorization of various dataflow MoCs. Different classes of dataflow process networks are categorized into static and dynamic ones. The static DPNs are characterized as ones having fixed consumption and production rates. Whereas, the dynamic DPNs involve variable consumption and production rates.*

by firing where it consumes data tokens from its input buffers and produces data tokens for its output buffers. The behavior of each process is described by firing rules which are triggered by the availability of data. Hence, the processes in a DPN are autonomous in the sense that they are not driven by a global clock as done in the SR MoC. A process executes until enough tokens are available in its input buffers for performing a computation.

The general MoC of DPNs does not impose further restrictions. However, a number of different classes of DPNs representing different dataflow MoCs have been considered over time [FHTZ17]. These classes mainly differ in the kinds of behaviors of the processes which affect 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. The number of tokens consumed by a process from a particular input buffer while firing is termed as the *consumption rate*. The number of tokens produced by a process on a particular output buffer while firing is termed as the *production rate*. Based on these factors, the most commonly known classes can be categorized into *static* and *dynamic* DPNs as depicted in Figure 2.3.

The latter accommodates DPNs like Kahn process networks (KPN) [Kahn74], Boolean dataflow (BDF) [Buck93] and the dynamic dataflow (DDF) [Kosi78] networks. Whereas, the former includes DPNs like static dataflow (SDF) [LeMe87a; PaPL95], homogeneous synchronous dataflow (HSDF) [LeMe87a] and the cyclo-static dataflow (CSDF) [EBLP95] networks. Static DPNs are generally characterized as having only processes where the consumption and production rates are neither influenced by the values of the consumed tokens nor they are dependent on the points in time at which tokens arrive on the input buffers. Thus, processes in static DPNs always consume the same number of input tokens from particular input buffers and produce the same

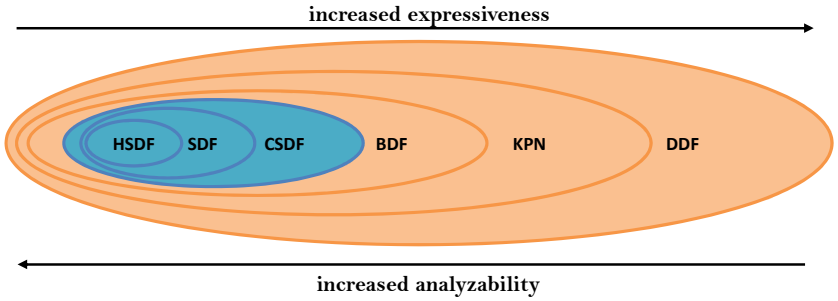

Figure 2.4.: *A hierarchy [FHTZ17] of DPNs based on expressiveness and analyzability. The hierarchy is partitioned into static (dark cyan) and dynamic (orange) DPNs. Dynamic dataflow (DDF) is the most generalized set, whereas the homogeneous static dataflow (HSDF) is the most limited subset of DPNs. As shown in this figure, the analyzability of DPNs is inversely related to their expressiveness.*

number of output tokens to particular output buffers. However, they may read different number of tokens from different input buffers and may write different number of tokens to different output buffers. On the one hand, these characteristics allow powerful design-time analysis techniques (e.g. for predictability and decidability), but on the other hand they limit the expressiveness by excluding dynamic behaviors (like select and switch nodes).

In contrast to static DPNs, where the consumption and production rates are not influenced by the values of the consumed tokens, processes in dynamic DPNs can vary their consumption and production rates in each firing mainly dependent on the history of the consumed tokens and also on the tokens to be consumed. This allows conditional or data dependent executions of processes, in particular, each process can produce and consume different number of tokens in every firing. This generalization results in higher expressiveness and flexibility, but makes the analysis more difficult.

Thus, different classes belonging to both categories exhibit a different tradeoff between their expressiveness and their analyzability as depicted in Figure 2.4. Generally, the analyzability of DPNs is inversely related to their expressiveness. There are major correctness properties like checking whether a DPN can run with bounded buffer sizes, which are decidable for less expressive DPNs but are not decidable for more expressive ones. Because of the static nature, static DPNs offer high analyzability, where one can check at compile time the correctness properties like boundedness of buffers and absence of deadlocks [Park95], as well as one can effectively determine a static schedule to run a DPN. In contrast, the dynamic DPNs offer high expressiveness that can be used to capture more flexible, dynamic and data dependent behaviors.

In general, DPNs offer a modeling paradigm well suited for the modeling of concurrent and distributed embedded systems. However, model-based design flows starting from DPNs, in particular, based on more expressive dynamic DPNs, have to deal with analyzability issues i.e., the undecidability of checking

major correctness properties like buffer boundedness and absence of deadlocks. Therefore, implementations of concurrent and distributed embedded systems from DPNs like KPNs may suffer from problems like deadlocks and buffer overflows. Verifying the existence of these problems in DPNs is in general not decidable.

2.1.3. From Synchronous MoC to DPN MoC

Based on the discussion in previous sections, it can be observed that both the SR MoC and the DPN MoC carry their own advantages and disadvantages. In particular, depending on applications and even more on the selected target architectures, these MoCs have advantages and disadvantages. As an alternative, *desynchronization* of synchronous models [BeCG99; Gira05a] has been recently developed that benefit both from the static analysis methods for synchronous systems and the performance of the finally synthesized asynchronous systems. This thesis focuses on the implementation of systems on distributed and heterogeneous target architectures. In order to more efficiently support distributed implementations, desynchronization is beneficial in the sense that starting from a synchronous model, system correctness can be easily verified, and a DPN can be generated while its correctness is maintained [PSST11a].

As a long term project, our group developed the *Averest*[1] tool for a model-based design process starting with synchronous models. The *Averest* project aims at providing a complete set of tools for the development of reactive systems. Moreover, the work presented in [BSBK14; Bai16] further presents a desynchronization design flow based on *Averest* that transforms synchronous networks into DPNs. The complete design flow based on *Averest* is illustrated in Figure 1.2. Since synchronous models are particularly well suited for analysis, the design flow starts with synchronous models, and then translates them to DPNs for the synthesis of concurrent and distributed systems. The underlying language of the target DPN model is a subset of the Cal actor language (CAL) [EkJa03]. Specifically, synchronous programs modeled in *Quartz* are compiled into synchronous guarded actions (SGAs), which represent the *Averest* intermediate format (AIF). These SGAs are then translated into CAL guarded actions. A set of SGAs representing a synchronous module is translated into a set of CAL guarded actions that form a CAL process. Each generated CAL process is stateless and depending upon the behavior of the synchronous module, it precisely belongs to a particular dataflow MoC.

These desynchronized models (DPNs) based on CAL are the starting point of this thesis. CAL is not related to any particular dataflow MoC. Instead, it offers a set of abstract notions for modeling systems based on various dataflow MoCs, from dynamic DPNs exhibiting dynamic behaviors to the more restrictive static DPNs. The design flow proposed in this thesis supports static as well as dynamic DPN MoCs. These include the static dataflow (SDF) [LeMe87a] MoC, the Kahn process networks (KPN) [KaMa77] MoC and a deterministic variant of the dynamic dataflow (DDF) [Kosi78] MoC. The representation of

[1] `http://www.averest.org`

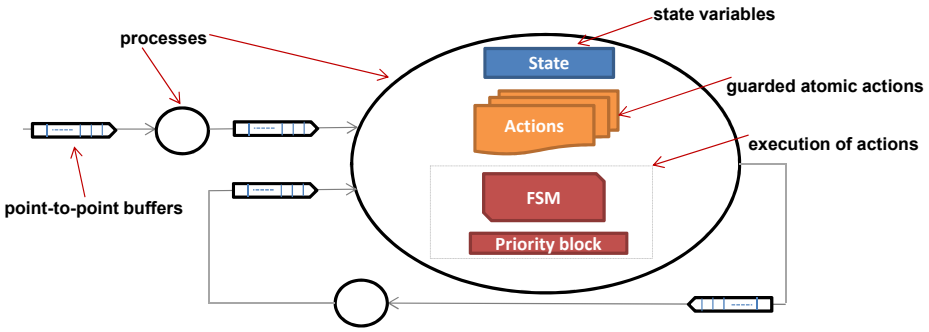

Figure 2.5.: *CAL DPN illustration: it consists of a set of processes connected via point-to-point buffers. The behavior of a process is defined in a set of actions. A process encapsulates its state using state variables and the order of the execution of actions can be handled by a finite state machine (FSM).*

different supported dataflow MoCs based on the used subset of CAL is given in Chapter 4.

2.2. Cal Actor Language (CAL)

CAL [EkJa03] is a dataflow and actor oriented language that provides useful abstractions for dataflow programming with actors (processes). CAL has been employed in a wide variety of applications and has been synthesized to hardware, software and hardware/software co-design implementations. CAL is constructed on a small set of semantic concepts which simplifies the development of a compiler to transform any program into the target language. Thus, a code generator is needed for the specific implementation language to execute a CAL program on any given platform.

2.2.1. CAL DPN

CAL models behaviors by distributing them in a set of actors (processes). The basic building block diagram of a CAL DPN is shown in Figure 2.5. It consists of a set of processes that are connected with each other through point-to-point buffers. Each process is a description of a computation on sequences of input tokens that produces sequences of output tokens as a result. CAL offers a set of essential constructs that can be used to model computations in processes. A process is a modular component that encapsulates its own state using **state variables**. The CAL processes can only interact with each other through input and output buffers. This restriction disallows any process to modify the state of another process. At the network level, the processes are generally independent and can be triggered for execution concurrently.

The behavior of a process is defined in a set of **actions**. Each action can perform a computation or part of a computation by consuming input tokens from particular inputs, modifying the internal state using state variables, and

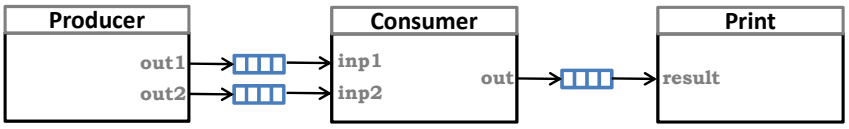

Figure 2.6.: *A simple example of a producer-consumer network.*

producing output tokens to particular outputs. The computations are generally executed sequentially within a process. In other words, the transitions of a process are purely sequential i.e., actions are triggered for execution one after the other. However, depending on the underlying dataflow MoC of a process, the actions can be mapped and executed on devices in parallel. CAL provides a number of constructs for describing the firing conditions and the selection order for actions in a process. An action is enabled for firing based on its input patterns and **guards**. The input patterns are determined by the amount of data required for the input sequences. Whereas guards are Boolean expressions that are applied on the state variables and/or on input sequences that need to be satisfied for enabling the execution of an action. The selection order of actions is determined by explicitly using an **action schedule** in the form of a **finite state machine (FSM)** and/or a **priority block**. The FSM performs the action selection according to its particular state to the presence of input tokens and to the values of the tokens evaluated by guards. A simple producer-consumer network is shown in Figure 2.6. The consumer process modeled with CAL that contains all the discussed language constructs is shown in Listing 2.1.

A process is declared with a finite set of input ports and a finite set of output ports, separated with the identifier '==>' (Lines 1-2). The state of an actor can be defined by state variables (Line 3). The behavior of a process is specified by a set of actions that are declared with a set of input and output ports (Lines 4, 5-13, 14), where each action upon execution consumes tokens from its declared input ports and may produce tokens for its declared output ports. The input and output sequences consumed and produced in each execution of an action are specified in square brackets next to the declared input and output ports, respectively. For instance, the action *forward* consumes a token each denoted by *a* and *b* from the input ports *inp1* and *inp2* respectively, and forwards them to the output port *out* (Line 14). There are two ways to implement an action: The functionality can either be implemented in the output port access definition (Lines 4 and 14), or more descriptively in the action body (Lines 8-13). The *repeat* keyword associated with a port defines the number of tokens per execution that can be consumed/produced from an input/output port (Line 5). An action may contain a guard (Lines 6-7) that specifies an additional condition to execute that action. Guard conditions can refer to state variables and the values of the input tokens.

The order in which actions are executed can be specified by an action schedule (Lines 15-19) and/or using a priority block (Lines 20-22). An action schedule is modeled as an FSM, where state transitions are triggered by the execu-

```
1 actor consumer() int(size=8) inp1, int(size=8) inp2 ==>
2                                    int(size=8) out :
3   bool state_flag := false;
4   add:action inp1:[a],inp2:[b] ==> out:[a+b] end
5   addnsub:action inp1:[a] repeat 2, inp2:[b] ==> out:[c]
6   guard
7     b > 0
8   var
9     int(size=8) c
10  do
11    c := a[0] + a[1] - b;
12    state_flag := true;
13  end
14  forward:action inp1:[a],inp2:[b] ==> out:[a,b] end
15  schedule fsm s_start :
16    s_start ( forward ) --> s_working;
17    s_working ( addnsub ) --> s_start;
18    s_working ( add ) --> s_working;
19  end
20  priority
21    addnsub > add;
22  end
23 end
```

Listing 2.1: *CAL code of the consumer process as shown in Figure 2.6.*

tion of actions. In a particular state, only the associated actions can be fired. For instance, in the state *s_start*, the action *forward* will be fired that causes a state change to *s_working* (Line 16), where the actions *addnsub* and *add* can be fired. For such a state where more than one action can be fired, a priority block can be used to specify the priorities (Line 21).

The topology of the input and output connections sets up a network of processes. The topology of the network is usually described using the functional network language (FNL) based on the XML format [BEJL11]. A simple producer-consumer dataflow network as introduced in Figure 2.6 is specified in FNL as shown in Listing 2.2. This example shows the two most basic elements of FNL, namely the *Instance* and the *Connection*. Each *Instance* field defines a process instance (Lines 2-4, 5-7 and 8-10), and possibly can even refer to another network. Each *Connection* field defines a connection between an input port and an output port of two instances (Lines 11-16).

2.2.2. Desynchronized CAL DPN Model

The desynchronization design flow based on *Averest* as illustrated in Figure 1.2 finally yields a DPN based on a limited subset of CAL. In particular, the SGAs are first verified for desynchronization and then translated into the CAL guarded actions. Since, the proposed synthesis method targets the execution and deployment of DPNs on heterogeneous platforms consisting of different types of devices including GPUs, the desynchronization method generates stateless dataflow processes. This simplifies not only the target DPN

```
 1 <XDF name="ConProd">
 2     <Instance id="producer">
 3         <Class name="cal.producer"/>
 4     </Instance>
 5     <Instance id="consumer">
 6         <Class name="cal.consumer"/>
 7     </Instance>
 8     <Instance id="print">
 9         <Class name="cal.print"/>
10     </Instance>
11     <Connection dst="consumer" dst-port="inp1"
12     src="producer" src-port="out1"/>
13     <Connection dst="consumer" dst-port="inp2"
14     src="producer" src-port="out2"/>
15     <Connection dst="print" dst-port="result"
16     src="consumer" src-port="out"/>
17 </XDF>
```

Listing 2.2: *The FNL network description for the producer-consumer example as shown in Figure2.6.*

```
 1 act1: action A:[a], B:[b] ==> Z:[z]
 2   guard a >= 0
 3   do
 4     z := b;
 5   end
 6 act2: action A:[a], C:[c] ==> Z:[z]
 7   guard a < 0
 8   do
 9     z := c;
10   end
```

Listing 2.3: *Sequential if-then-else based on the used CAL subset.*

specification for the final synthesis, but also paves the way for dynamically handling parallelization in OpenCL based synthesized implementations. The target subset of CAL simply consists of a set of guarded actions. Thereby, each generated stateless process essentially consists of a set of guarded actions where the guards are applied on the values of the input tokens. Depending on the behavior of a particular synchronous module, the generated process possesses a particular kind of behavior that precisely determines a particular dataflow MoC. To exemplify, a desynchronized version of the sequential if-then-else (SITE) operation is illustrated in Listing 2.3. The syntax and the informal semantics of all the supported dataflow MoCs based on this limited subset of CAL are presented in Chapter 4. We only briefly describe here the generated set of guarded actions for SITE:

SITE is a dynamic version of the if-then-else operation that sequentially consumes data from the input buffers B and C based on the value of data on the input buffer A. It consists of two actions act_1 and act_2, having different inputs B and C, respectively (Lines 1 and 6). Both actions use the input buffer

A for the guard with mutually exclusive guard expressions (Lines 2 and 7). Depending on which guard is enabled in each execution, either act_1 or act_2 is fired. In each execution, it therefore either consumes a token from the input buffer B or the input buffer C and writes it to the only output buffer Z (Lines 4 and 9), mainly dependent on the consumed value of a token on the input buffer A.

In general, CAL is not related to any particular MoC, instead, offers modeling a behavior based on various dataflow MoCs. The behavior once modeled can then be interpreted and synthesized to an implementation based on the underlying dataflow MoC. Therefore, this limited subset of CAL is very conveniently employed by the proposed design flow of this thesis to synthesize from modeled behaviors, implementations based on the supported dataflow MoCs. Therefore, for each synchronous module, a CAL actor (process) is generated that consists of a set of guarded actions as illustrated in the example in Listing 2.3. The network description file (xdf) describes the topology of the network as illustrated in Listing 2.2. The generated desynchronized CAL code thus consists of two parts: the CAL processes and the network description.

2.3. Open Computing Language

The open computing language (OpenCL) [StGS10] is an open specification language designed for parallel computing on cross-vendor and heterogeneous architectures. In contrast to proprietary specification languages with limited hardware choices, OpenCL allows task-parallel and data parallel heterogeneous computing on a heterogeneous collection of modern central processing units (CPUs), graphical processing units (GPUs), digital signal processors (DSPs), and other microprocessor designs organized into a single platform [LNKP15; SFSV13]. The change from proprietary programming languages to open standard facilitates the acceleration of general computation in a cross-vendor fashion. To this end, OpenCL is supported by the leading hardware vendors including Intel, Apple, AMD, and many others. It offers a low-level, high-performance, portable abstraction that gives software developers portable and efficient access to the power and the resources of these cross-vendor heterogeneous processing platforms.

OpenCL[2] has been used in a wide variety of applications, ranging from embedded and consumer software to high performance computing systems. It offers an efficient, close-to-the-target programming interface that forms the foundation layer of a parallel computing ecosystem of platform-independent tools, middleware and applications. OpenCL provides a comprehensive API for handling parallel computation across heterogeneous processors as well as it offers a cross-platform programming language with a well specified computation environment.

A primary benefit of OpenCL is a substantial acceleration in parallel processing. OpenCL supports both coarse-grained (task-level) as well as fine-

[2] `https://www.khronos.org/opencl/`

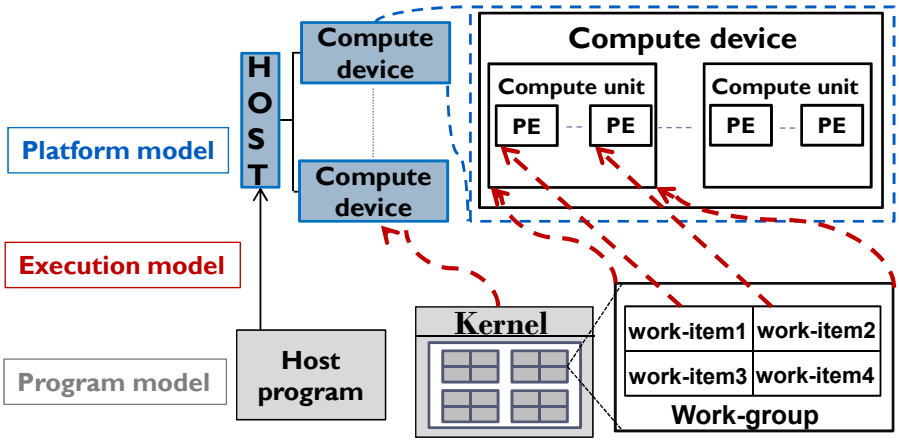

Figure 2.7.: *Overview of the OpenCL architecture: the **platform model** provides a standard abstraction of the target hardware. The **program model** specifies the behavior of a system typically organized as a host and several kernels. The **execution model** describes the mapping of the program model onto the platform model.*

grained (data-level) parallelism. Second, it provides the ability to write vendor-neutral cross platform applications. This is achieved by providing high-level abstractions to hide low-level details of implementations, such as drivers and runtime. The basic strength of this abstraction is the ability to scale code from simple embedded microcontrollers to multicore CPUs and highly parallel GPUs without revising the code. These benefits can be derived by understanding and exploiting a set of abstract models provided by OpenCL, as depicted in Figure 2.7 and Figure 2.9.

2.3.1. Platform Model

The OpenCL platform model provides users with a convenient abstraction of the target hardware. It is defined as a *host* connected to one or more compute devices, each having multiple compute units (CUs), each of which further consists of multiple processing elements (PEs).

A host is typically a CPU running a standard operating system (OS), while a compute device may be a GPU, a DSP, a further multicore CPU or any other specific microprocessor. Each device therefore consists of a collection of one or more CUs where each CU can be conceived as, for instance, a core of a CPU, or a streaming multiprocessor of a GPU. A CU is further composed of one or more PEs that execute instructions. Each PE can therefore be conceived as, for instance, a streaming core (or SIMD lane) of a GPU. An OpenCL device therefore executes the instruction computations on the PEs within the device. In particular, PEs execute instructions in a single instruction multiple data (SIMD) or a single program multiple data (SPMD) fashion. SPMD instructions are typically executed on microprocessor designs such as CPUs, while SIMD instructions require a vector processor such as a GPU or vector processing units integrated in a CPU.

2.3.2. Program Model

The OpenCL program model is comprised of two main components: the *host program* and *kernels*. The host program executes on the host, defines devices context, sets up command queues of devices and enqueues kernel's execution instances on devices.

Kernels

A kernel is a C-like function that actually implements the abstract behavior of the system or part of the system. OpenCL targets the parallel execution of a kernel on compute devices by organizing it into an N-dimensional computation domain, where N = 1, 2, or 3. For instance, performing a computation on a linear array of data would require N = 1, while processing an image would require N = 2. This computation domain is defined when a kernel is mapped for execution on the command queue. Each independent element of this domain represents the execution instance of the kernel and is termed as the *work-item*. Each work-item performs the same kernel function but on different data. The total number of work-items represents the global work size of a kernel, where each work-item is assigned a unique global ID.

Sequential C function	OpenCL kernel

void addArray (int n, const float* arrayA,
const float* arrayB, float* arrayC)
{
int i;
for(i=0; i<n; i++)
arrayC[i] = arrayA[i] + arrayB[i];
}
	__kernel void addArray (__global const float* arrayA,
	__global const float* arrayB, __global float* arrayC)
	{
	int id = get_global_id(0);
	arrayC[id] = arrayA[id] + arrayB[id];
	}

Figure 2.8.: *OpenCL kernel illustration: the addArray example performs the add operation on two linear arrays. The left hand side version is a sequential C function, whereas the right hand side version is the OpenCL kernel.*

OpenCL also allows grouping work-items together into *work-groups*, as shown in Figure 2.7. The work-group size specifies the number of work-items in a group and is termed as the local work size. All work-items in the same work-group are executed together on the same compute unit. The work-items in a group share local memory and synchronization. Each work-item also has its own private memory that allows each work-item to conveniently operate on its own assigned data. Whereas, global work-items are independent and cannot be synchronized. A simple example *addArray* that performs the add operation on linear arrays is shown in Figure 2.8. In this example, each element of two linear arrays *arrayA* and *arrayB* are added together and the result is stored in *arrayC*. The figure presents two different versions of the *addArray* example to illustrate how a kernel is implemented in OpenCL. The first version implements *addArray* as a sequential C function with a simple for-loop iterating through the elements in the arrays and then performing addition.

The OpenCL version implements a data parallel function without using a sequential for-loop: It simply reads the unique global ID for the particular kernel instance (work-item), performs the operation, and produces the output. The host program determines the fine-grained data-level parallelism by specifying the total number of work-items when the kernel is placed on the command queue.

```
1  /*Create context for the devices*/
2  context = clCreateContext(0, 1, &device, NULL, NULL, &err);
3  /*Create program from kernel source*/
4  program = clCreateProgramWithSource(context,1,(const char**)&
       kernel_source,(const size_t *)&source_size,&err);
5  /*Create memory objects*/
6  memobj_A = clCreateBuffer(context, CL_MEM_READ_ONLY |
       CL_MEM_ALLOC_HOST_PTR, buffer_size, arrayA, &err);
7  memobj_B = clCreateBuffer(context, CL_MEM_READ_ONLY |
       CL_MEM_ALLOC_HOST_PTR, buffer_size, arrayB, &err);
8  memobj_C = clCreateBuffer(context, CL_MEM_READ_ONLY |
       CL_MEM_ALLOC_HOST_PTR, buffer_size, arrayC, &err);
9  /*Create command queue*/
10 cmd_queue = clCreateCommandQueue(context,device,0,NULL);
11 /*Enqueue memory commands*/
12 err = clEnqueueWriteBuffer(cmd_queue, memobj_A, CL_TRUE, 0,
       buffer_size, (void*)a, 0, NULL, NULL);
13 err |= clEnqueueWriteBuffer(cmd_queue, memobj_B, CL_TRUE, 0,
       buffer_size, (void*)b, 0, NULL, NULL);
14 /*Set up kernel*/
15 kernel = clCreateKernel(program,"addArray",&err);
16 /*Set up kernel arguments*/
17 err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &memobj_A);
18 err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &memobj_B);
19 err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &memobj_C);
20 /*Set up the computation domain*/
21 size_t global_work_size = n;
22 size_t local_work_size = n;
23 /*Enqueue kernel execution commands*/
24 err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, &
       global_work_size, &local_work_size, 0, NULL, NULL);
```

Listing 2.4: *The host program of the addArray example as illustrated in Figure 2.8.*

Host Program

The host program resides and executes on the host and is responsible for setting up and handling the execution of kernels on the compute devices using the defined context. The context essentially sets up the environment for executing kernels and is comprised of a number of resources including OpenCL devices, program source, kernels and memory objects. To this end, the context is created with a set of devices that are used by the host to execute kernels. Second, the context defines the program source that implements a kernel or a collection of kernels. Finally, the context defines the OpenCL memory objects that are used as a source of communication between the host program and devices. After the context is created, command queues are created where the

kernels are mapped to get executed on the OpenCL devices associated with the context. Each command queue can represent a complete device (e.g., a CPU) or even a compute unit of that device (e.g., a CPU-core). Each command queue accepts the execution and memory requests for the corresponding devices and their CUs. A command queue typically accepts three types of commands: *Kernel execution commands* map kernels on devices. *Memory commands* transfer memory objects between the memory space of the host and the memory space of devices. *Synchronization commands* specify order in which enqueued commands are executed. Commands are inserted into the command queue in-order and can be either executed in-order or out-of-order. The host program illustrating the main resources for the *addArray* example as shown in Figure 2.8 is listed in Listing 2.4.

2.3.3. Execution Model

The OpenCL execution model can simply be understood as the mapping of kernels on the platform model which is implemented in the host program. Depending on the target compute device (e.g., a CPU or a GPU), kernels are mapped differently. In case of GPUs, OpenCL only allows the user to create a command queue at the level of a compute device. Hence, for a GPU, a kernel is typically allocated on a compute device, a work-group is ideally mapped on a CU, and work-items of that work-group are executed by PEs of that CU, as depicted in Figure 2.7. In contrast, for CPUs, a command queue can be created at the level of a compute device as well as at the level of a CU. For the latter, the whole kernel (all work-groups) are mapped to the same CU (i.e., a core of a CPU). Furthermore, the execution model also facilitates the usage of different scheduling schemes by allowing in-order and out-of-order execution of kernels. In in-order setting, the kernels are executed sequentially in the order they are placed into the command queue. The out-of-order setting executes kernels based on the synchronization constraints specified for the kernels.

2.3.4. Memory Model

OpenCL offers a disjoint memory model to programmers as shown in Figure 2.9. This is mainly because OpenCL targets heterogeneous platforms where most platforms utilize disjoint memory systems due to different memory requirements of different architectures. OpenCL visualizes its target as a system where data sharing between the host and compute devices is performed explicitly by a system network, such as a peripheral component interconnect (PCI) bus. The OpenCL memory model is organized in five regions consisting of *host*, *global*, *constant*, *local* and *private* memories.

The host memory is described as the region of system memory that is only directly accessible from the host processor. Any transfer of data between the host and the kernels should be done explicitly through the OpenCL global memory region typically by using the OpenCL API. The global and constant memories are shared between all devices including the host within a given context. However, local and private memories are always associated with

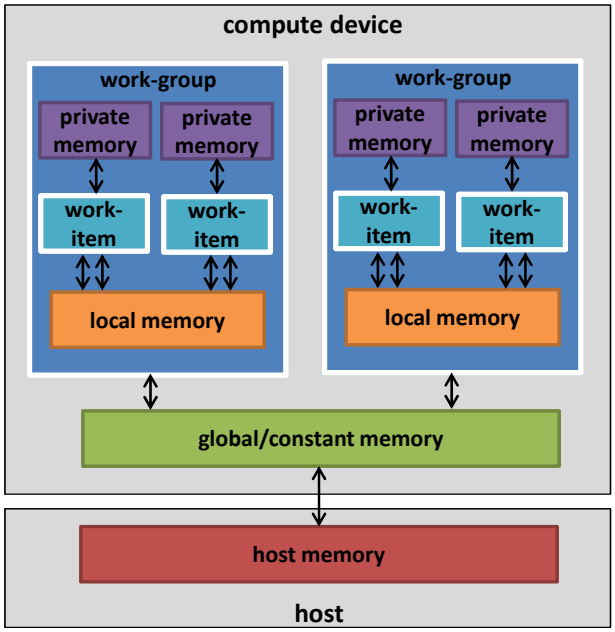

Figure 2.9.: *OpenCL memory model*[3] *consists of five regions: The* ***host memory*** *is only accessible to the host processor. The* ***global memory*** *is accessible to both the host and device. The* ***constant memory*** *is fully accessible to the host and write-protected for the device. The* ***local memory*** *is only visible to the host and is local to a single compute unit. The* ***private memory*** *is private to an individual work-item executing within an OpenCL processing element*

a particular device. In particular, global memory offers a region where all work-items and work-groups have read and write access on both the compute device and the host. The host can only allocate this memory region at runtime. Constant memory resides in the global memory and offers the memory region that is constant throughout the execution of kernels. While work-items can only read, the host is permitted to read and write from this memory region. Local memory offers a region as a shared memory to work-items in a work-group. All work-items belonging to the same work-group have both read and write access. Finally, private memory offers a memory region only accessible to one work-item.

For heterogeneous architectures consisting of multiple devices integrated on a single platform, host memory and device memory are independent of one another. The memory management needs to be handled explicitly to allow the data communication between the host and the device. This requires the explicit handling of data from host memory to device memory and back to host. OpenCL offers an API dedicated to manage the data sharing between host and devices in a number of ways. This includes the data communication using explicit read/write functions, memory mapping and others.

[3] `https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_API.html`

Chapter 3

The Model-based Design Flow

Contents

3.1. Feasibility Evaluation and Selection Criteria

We envisaged a common model-based synthesis framework that enables the synthesis of a system using different well-defined and precise dataflow models of computation (MoCs). Second, it integrates the standard hardware abstractions using the open computing language (OpenCL) to promote the use of vendor-neutral heterogeneous architectures. Altogether, we envisioned an automatic synthesis that maps models using a combination of different dataflow MoCs on heterogeneous platforms. To this end, a limited subset of Cal actor language (CAL) is employed as the target language for DPNs. CAL innately supports the use of concurrent nodes and is potentially suitable for modeling parallel and distributed systems as DPNs. Second, OpenCL is employed as an integral part of the synthesis tool chain to systematically handle the portability of modeled systems on various commercial off-the-shelf (COTS) heterogeneous hardware. OpenCL allows task-parallel and data-parallel heterogeneous computing on a variety of modern CPUs, GPUs, DSPs, and other microprocessor designs [StGS10].

As a part of this thesis work, two experimental tools are designed and developed to systematically evaluate the feasibility of employing CAL and OpenCL for the proposed model-based synthesis framework. One of the tools primarily focuses on presenting a systematic approach for evaluating OpenCL as a hardware abstraction layer for distributed embedded systems. This tool is designed

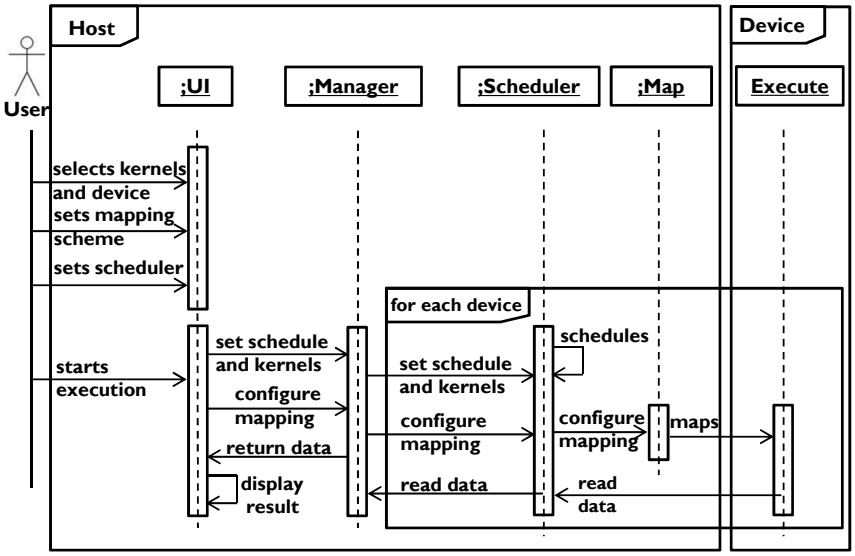

Figure 3.1.: *Overview of experimental tool-1: the user interface (UI) is used to intuitively set up the evaluation environment. The runtime manager (Manager) invokes the execution based on the selected environment. The evaluation results are read back by the Manager that are then displayed at the UI.*

to enable the user to utilize the OpenCL supported computing resources by using and exploring different scheduling and mapping schemes. The second tool offers a translation scheme for generating efficient parallel OpenCL code from DPNs modeled using a subset of CAL, namely RVC-CAL [BoNy15]. This tool provides an experimental evaluation environment for evaluating the feasibility of employing CAL for modeling parallel and distributed systems. Second, it also evaluates OpenCL for heterogeneous parallel computing and cross-vendor portability.

3.1.1. Experimental Tool-1

This experimental tool [RaSc20] presents a systematic approach that employs OpenCL as a standard hardware abstraction to explore and to evaluate the utilization of parallel computing resources using different scheduling and mapping schemes. Hence, enabling the user to potentially exploit both coarse-grained task-level parallelism and fine-grained data-level parallelism, respectively.

As discussed, OpenCL distinguishes between a host and kernels where the host is a centralized entity that is responsible for managing the execution of the kernels on the available target devices. This experimental tool adopts this idea in that it enables the user to utilize the available computing resources using a combination of different scheduling and mapping schemes. An overview of the experimental tool is depicted in Figure 3.1.

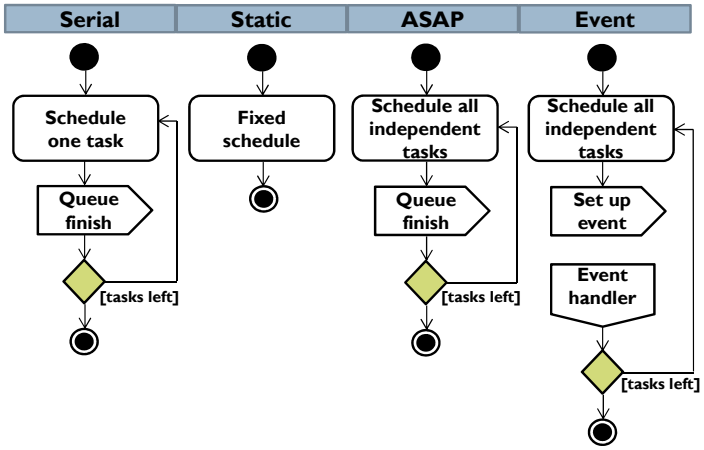

Figure 3.2.: *Available scheduling schemes in experimental tool-1. Apart from the static scheduler that implements a static scheme, all other schedulers follow dynamic data driven schemes.*

User Interface and Runtime Manager

A user interface (UI) is designed to allow the user to intuitively set up and to configure the desired evaluation environment. A user can select OpenCL kernels from an extendable list of available kernels along with their data dependencies. The UI also displays a list of available target devices where the user can select a particular device for execution. Furthermore, a user can configure different mapping schemes mainly by selecting a combination of different OpenCL parameters such as number of work-items, work-groups, etc. Finally, the UI provides a list of available scheduling schemes where a particular scheme can be selected that will be used in combination with the already chosen mapping configuration. However, depending on the underlying scheme of the selected scheduler, it may not allow all mapping combinations. Upon invoking a start button, the runtime manager (Manager) starts the execution based on the selected evaluation environment.

Based on that, the Manager sets up and initializes the kernels, sets up the OpenCL mapping configuration, and finally invokes the selected scheduler. Once the complete execution of the selected evaluation environment is finished, the Manager reads back the evaluated results and supplies it to the UI. Intuitively, the UI is capable of displaying and plotting the results on the fly. In addition, the results are logged in a csv file for offline analysis and monitoring.

Scheduling and Mapping

The chosen scheduler schedules the selected kernels (tasks) based on the underlying scheduling scheme. To this end, four distinctive scheduling schemes are developed and made available to the user as shown in Figure 3.2. The *serial scheduler*, as the name suggests, realizes a serial execution of tasks, i.e., one task at a time. It implements a dynamic scheduling scheme that enqueues

one task at a time for execution and waits until it is completely executed. The *ASAP scheduler* also relies on a dynamic scheduling scheme that exploits task-level parallelism by enqueuing all independent tasks at a time. It then waits until all enqueued tasks are completely executed. The *static scheduler* implements a static scheme that follows a predefined fixed schedule of tasks. The current implementation of the *static scheduler* (for the case study) is simply a static version of the *ASAP scheduler*, however, it avoids the runtime overhead of dynamically searching the independent tasks. Finally, the *Event scheduler* also implements a dynamic scheduling scheme, and is similar to the *ASAP scheduler*, except that it does not wait for the complete execution of enqueued tasks. Instead, the *Event scheduler* sets up an event for all scheduled (enqueued) tasks, and therefore will be automatically notified when the execution terminates. Hence, it provides a more flexible scheme by allowing itself to perform other operations (if needed) while the already scheduled tasks are being executed.

Regardless of which scheduler is selected by the user, each scheduled task is then mapped on the target device for execution based on the configured mapping scheme. A mapping scheme is configured by selecting a combination of different parameters including the numbers of data samples, work-items, work-groups, etc. This enables the user to potentially exploit the data-level parallelism of the scheduled task by using the desired mapping scheme. A combination of different scheduling and mapping schemes can be used to exploit both the coarse-grained task-level parallelism and the fine-grained data-level parallelism, respectively. Hence, the experimental tool systematically facilitates the substantial acceleration in parallel computing.

Evaluation

Using *experimental tool-1*, we evaluated OpenCL for the utilization of parallel computing resources in a distributed embedded system, namely the *ConceptCar*[1]. The *ConceptCar* features seven different electronic control units (ECUs). Specifically, OpenCL is evaluated for embedded computing on a particular ECU of the ConceptCar. The computations are provided by an extendable set of related advanced driving assistance system (ADAS) [KTPB18] applications built under a common application setup. This application set is mapped and executed using different scheduling schemes in conjunction with various OpenCL mapping configurations.

Application Setup A number of related ADAS applications are designed and developed under a common application setup as shown in Figure 3.3. The main focus of this work is not to propose efficient algorithms for ADAS applications. Instead, the idea is to develop a set of related applications under a common system using a minimal set of sensors, mainly for initial lab testing and performance evaluations.

[1] `https://es.cs.uni-kl.de/research/applications/concept-car/`

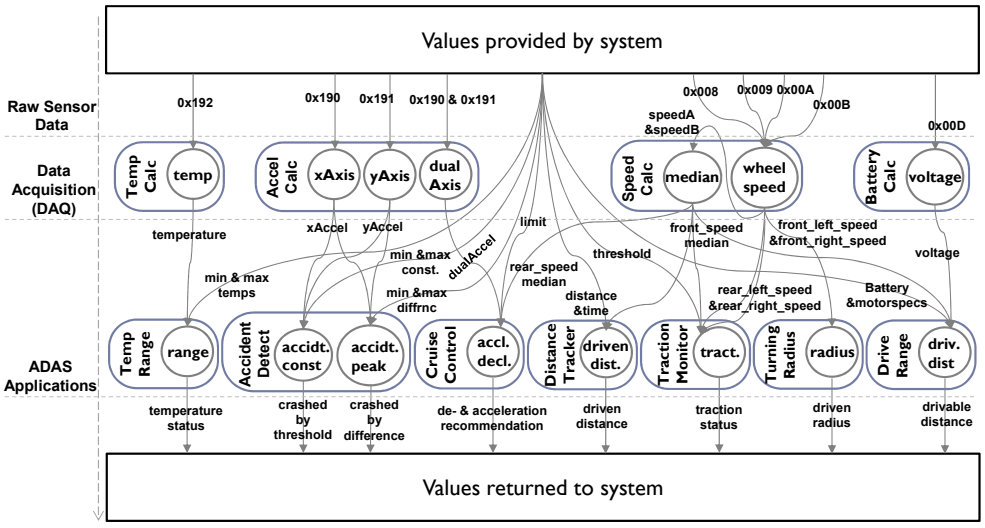

Figure 3.3.: *The application setup for evaluation using experimental tool-1. It is organized in a three-level design: the first level retrieves raw sensor data from the CAN bus. The second level features applications that perform data acquisition on sensor data. The third level implements the designed ADAS applications.*

The proposed application setup can be understood in a three-level design where the system first retrieves raw data of sensors from the controller area network (CAN) bus which is forwarded to the second level. Seven different sensors have been used including a temperature sensor, a dual-axis accelerometer, wheel speed sensors, and others. The second level implements data acquisition (DAQ) applications for signal conditioning and digital value conversions of the used sensors. Specifically, four different DAQ applications are implemented to provide the converted numeric values and other optimized results for temperature, acceleration, wheel speed, and battery voltage. The third level, fed by the applications from the second level, implements different ADAS applications. The application setup is presented in detail in [RaSc20].

Evaluation Environment The application setup, as shown in Figure 3.3, is evaluated on the target platform, i.e., the *multicore ECU* of the ConceptCar. The multicore core ECU features a heterogeneous computing platform with the following hardware specification:

- CPU: 1.2 GHz quad-core ARM Cortex A53
- GPU: 400 MHz Broadcom VideoCore IV
- 1 GB LPDDR2-900 SDRAM

Furthermore, the software environment used for the evaluation is summarized as follows:

- OpenCL CPU implementation: portable computing language (POCL) v1.3
- OpenCL GPU implementation: VideoCore IV OpenCL conformant to OpenCL v1.2
- Operating system (OS): Debian GNU/Linux 9.8

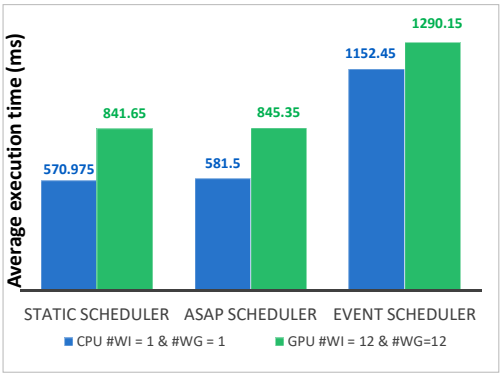

Figure 3.4.: *Experimental tool-1 results illustrate the top three combinations of scheduling and mapping schemes on each device. The results on the CPU are depicted by the blue bars, whereas, the GPU results are depicted by the green ones.*

The application setup is either executed on the CPU or the GPU at a time with different combinations of scheduling and mapping schemes for evaluating the performance on individual target devices. The average execution time of the complete application setup to process the desired number of sensors data samples is used as the performance metric. A maximum of one million forty-four thousand data samples are used and the average of fifty repetitions is taken for each evaluation.

The complete evaluations are thoroughly presented in [RaSc20], where three individual OpenCL mapping parameters are considered, namely the workload, the work-items and the work-groups. We discuss here the main conclusions regarding the evaluation of OpenCL.

Results: Conclusions We derived the top three combinations of scheduling and mapping schemes on each device that provided the fastest execution times, as shown in Figure 3.4. Based on the results, all schedulers achieved their best on the CPU with a single work-item and a single work-group. However, on the GPU, all schedulers achieved their fastest execution times with a total number of twelve work-items, and a total number of twelve work-groups.

On the CPU, the *static scheduler* proved to be the fastest, in particular, more than twice as fast as the *Event scheduler*, and about 1.9% faster than the *ASAP scheduler*. Similarly, on the GPU, the *static scheduler* and the *ASAP scheduler* performed comparably fast, with the *static* one only slightly faster than the *ASAP*. Both *static* and *ASAP* schedulers are approximately 53% faster than the *Event scheduler*. This is mainly because the underlying scheme of the *static scheduler* is especially devised for the proposed application setup at compile time. The *serial scheduler* does not exploit any task-level parallelism, and therefore performed slowest of all schedulers. The *ASAP scheduler* and the *Event scheduler* induced the overhead of dynamically scheduling tasks at runtime. Additionally, the *Event scheduler* further caused the overhead of setting up events and calling handlers at runtime. These overheads therefore

resulted in elevating the execution times. Finally, the *static scheduler* being the fastest, performed about 47% faster on the CPU than on the GPU. It is important to observe that based on the used application setup, the task-level parallelism clearly dominated over the data-level parallelism. Therefore, the application setup on the GPU understandably performed slower than on the CPU because the communication overhead of OpenCL on GPU is higher than the performance gain of executing instances in parallel on many cores.

Using *experimental tool-1*, we therefore evaluated that OpenCL can be exploited for targeting coarse grained task-level as well as fine grained data-level parallelism in a distributed embedded system. In particular, different scheduling schemes and OpenCL mapping configurations can be used together to efficiently exploit heterogeneous parallel computing on different types of devices. Especially, it is observed that on a GPU the best performance is achieved when both task-level as well as data-level parallelism have been exploited. On the contrary, on a CPU, the best performance in terms of average execution time is achieved when only task-level parallelism is exploited. In a nutshell, the main observation of this evaluation is that OpenCL is an open specification that needs to be utilized systematically to achieve the desired performance on different types of devices. Hence, OpenCL can be employed as a hardware abstraction in a model-based design targeting the synthesis of systems on heterogeneous platforms.

3.1.2. Experimental Tool-2

This experimental tool [RaKS19] evaluates the feasibility of employing the combination of RVC-CAL and OpenCL for modeling and implementing parallel and distributed systems. In particular, it evaluates RVC-CAL as a potential candidate for effectively modeling parallel and distributed systems as DPNs. Second, it evaluates OpenCL as a hardware abstraction for implementing DPNs on various COTS target hardware. Altogether, this tool offers the generation of parallel software from RVC-CAL models based on the potential parallelism of modeled behaviors. The approach considers both the coarse-grained (task-parallel) execution of processes using multithreading and the fine-grained (data-parallel) execution of their actions using OpenCL. This tool analyzes the effectiveness of using the combination of RVC-CAL and OpenCL by evaluating RVC-CAL benchmarks on OpenCL abstracted hardware platforms. The experimental results are evaluated for efficiency (performance) in comparison with a pure multithreaded C++ approach and a well-known reference framework, namely the open RVC-CAL compiler (ORCC) [YLJC13a; BTRM14] framework. ORCC supports parallel execution of DPNs at the coarse-grained level using a multithreading concept offered by an operating system.

Overview

Experimental tool-2 is presented in detail in [RaKS19]. The overall approach employs the multithreaded execution of RVC-CAL processes and the paral-

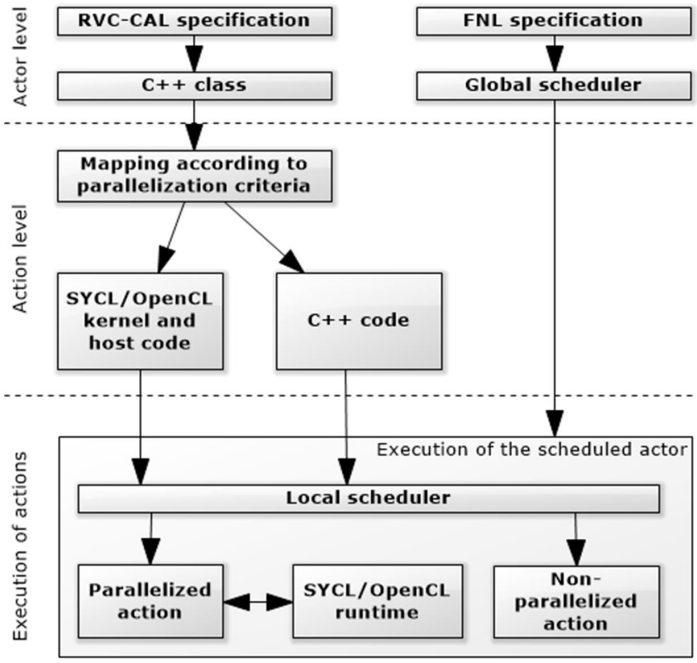

Figure 3.5.: *Overview of experimental tool-2: it is organized in a three level approach. The approach typically targets the multithreaded execution of actors using C++ and the parallelization of their actions using OpenCL.*

lelization of their actions with OpenCL. It is systematically organized into different levels of code generation and the final execution, as shown in Figure 3.5.

Actor level. At this level, RVC-CAL processes (actors) and the network description are taken into account where each process is translated into a C++ class and a global scheduling routine is generated for scheduling processes, respectively. The global scheduling routine provides the scheduler at the process level. The processes are executed concurrently by a fixed number of threads, generally equal to the available number of compute units. Each thread calls the global scheduling routine initially and when a scheduled process terminates.

Action level. At this level, all actions of a process are either translated into pure C++ code or to OpenCL code for parallel execution. If the criteria for parallelization with OpenCL are not met, the actions are translated to C++ member functions and are executed without any further parallelization.

Execution of actions. During execution, the global scheduler decides which processes to execute next in the calling threads. Each process has its own local scheduling routine that manages the execution of actions based on the modeled behavior. Upon execution of a process, its local scheduler is called. Depending on the generated code, the local scheduler either calls an action that is parallelized by OpenCL or directly executes it in the thread. Each time a local scheduler terminates, the control flow returns to the global scheduler and the next process is scheduled in the current thread.

Parallelization Criteria

The proposed concept of executing actions within processes in parallel mainly relies on the fine-grained (data-level) parallelism of actions using OpenCL. Hence, the basic idea is to execute multiple instances (work-items/threads) of an action, operating on different data, in parallel. However, this can only be achieved if each work-item knows exactly where to read/write its input/output data. Based on this fundamental concept, the proposed criteria for deciding whether a process can further be parallelized with OpenCL or not is depicted in Algorithm 1.

Algorithm 1: Pseudo code to determine whether processes in the network can be parallelized

```
1  foreach process p in the network do
2      if NoStateVariables(p) then
3          if NoFSM(p) then
4              if NoGuardCondtion(p) then
5                  create a kernel for each action of p
6              end
7              else if IsSynchronous(p) then
8                  create one kernel for p
9              end
10         end
11         else if UnicyclicFSM(p) then
12             create a kernel for complete cycle of the action schedule of p
13         end
14     end
15 end
```

The algorithm first checks whether a process has state variables (Line 2). As the same state variable can be used by different actions, executing them in parallel without ensuring the correct access order can lead to non-deterministic implementation. Therefore, processes only without state variables are qualified for further investigation.

Next, the algorithm checks whether a process has an FSM or not (Line 3). If an actor has no FSM, the algorithm checks if there are actions with guard conditions (Line 4). If there is no guard condition, for each action, a separate OpenCL kernel and the corresponding host code is generated (Line 5). Each kernel can be executed in a number of parallel instances (work-items) depending on the availability of tokens. If a process consumes/produces a fixed number of tokens in each firing, it can be parallelized even with guard conditions as each work-item would know exactly where to read/write its input/output data and each work-item can evaluate the guard conditions independently. In this case, a single OpenCL kernel is created for the process containing all actions (Line 8). Hence, the guard conditions are evaluated in the kernel and the enabled actions are executed.

If a process has an FSM, the algorithm first checks if it consists of exactly one cycle (Line 11). In this case, the resulting action schedule is clear because in each state there is only one state transition possible. Therefore, a single

OpenCL kernel is created for a complete cycle of the action schedule (Line 12). However, if there are multiple state transitions possible in a single state of an FSM, no parallelization is possible because a scheduling decision is required before the execution of each action.

Evaluation

We selected two benchmarks, namely a FIR filter and a ZigBee multi-token transmitter from the ORCC samples repository[2], mainly because of the following reasons: First, the applications are entirely modeled with RVC-CAL code. Second, a C++ program can be generated from RVC-CAL code and can be executed to get the desired output.

For each selected benchmark modeled with RVC-CAL, two different software versions are generated, namely the C++/OpenCL based version and a pure C++ version that only uses multithreading for the concurrent execution of processes. The total execution time of the network to process the complete input data set, including initialization and termination of the program, is used as the comparison metric. Based on that, the execution of the pure C++ version and the versions generated by the ORCC framework are compared against the OpenCL based versions. The data set used has a maximum of thirty-seven million samples and the average of fifty repetitions is taken for each version.

Experimental Setup We executed the benchmarks on the following hardware:

- **Platform 1**
  - **CPU-P1**: Intel i5-7200U
  - **GPU-P1**: NVIDIA GTX 950M
  - 8GB RAM
- **Platform 2**
  - **CPU-P2**: Intel i7 7700HQ
  - **GPU-P2**: NVIDIA GTX 1050
  - 16GB RAM

For the execution of the benchmarks we used the following software environment:

- NVIDIA GeForce graphics driver version 22.21.13.8792
- Intel OpenCL SDK Version 7.0.0.2519
- Windows 10 pro version 1803 build 17134.407

Evaluation of the FIR Filter The low level finite impulse response (FIR) filter is shown in Figure 3.6. Based on the parallelization criteria as illustrated in Algorithm 1, the actions of the processes *source*, *sink* and *delay* cannot be parallelized with OpenCL. Apart from that, the actions of all other processes are parallelized as they meet the criteria. This benchmark is either executed on **CPU-P1** or **GPU-P1** at a time to evaluate the performance

[2]https://github.com/orcc/orc-apps

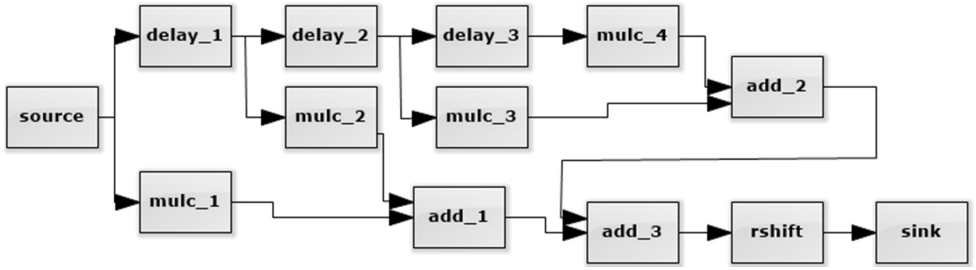

Figure 3.6.: *The first benchmark considered for evaluation using experimental tool-2 is the digital FIR filter.*

on individual target devices. The performance comparison of the considered implementations is shown in Figure 3.7.

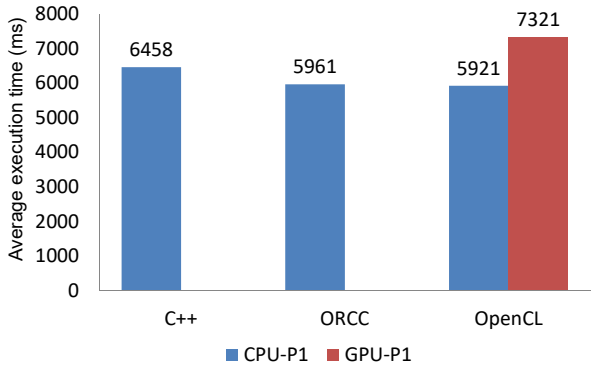

Figure 3.7.: *The results evaluated for the FIR filter using experimental tool-2 are shown in this figure. The results measured on the CPU are depicted by blue bars and the results on the GPU are depicted by the red ones.*

Discussion. Although the parallelized actions only perform simple operations like addition and subtraction, the generated OpenCL version on the CPU shows the best performance in comparison to all other versions. Considering the fact that only few simple actions are parallelized, the OpenCL CPU version shows a promising performance, slightly better than the ORCC version and considerably better than the C++ version. The OpenCL version on the GPU understandably performs low because the communication overhead of OpenCL on GPU is higher than the performance gain of executing rather small kernels in parallel on many cores. In contrast to the OpenCL CPU where the host and the kernels reside on the same device, in the case of GPU, the data has to be transferred to the GPU and back to the main memory (host). This overhead therefore contributes in elevating the total execution time.

Evaluation of the ZigBee Multi-token Transmitter The ZigBee multi-token transmitter benchmark is shown in Figure 3.8. For this benchmark only the actions of the processes *chipMapper* and *qpskMod* can be parallelized with OpenCL because all other processes do not meet the proposed criteria. This

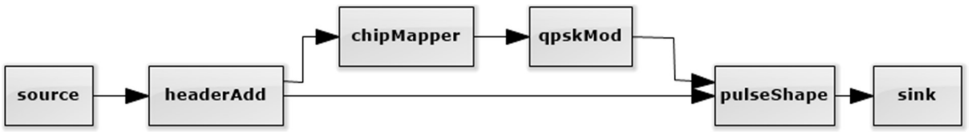

Figure 3.8.: *The second benchmark considered for evaluation using experimental tool-2 is the ZigBee multi-token transmitter.*

benchmark is either executed on **CPU-P2** or **GPU-P2** at a time to evaluate the performance on individual target devices. The performance comparison of the considered implementations is depicted in Figure 3.9.

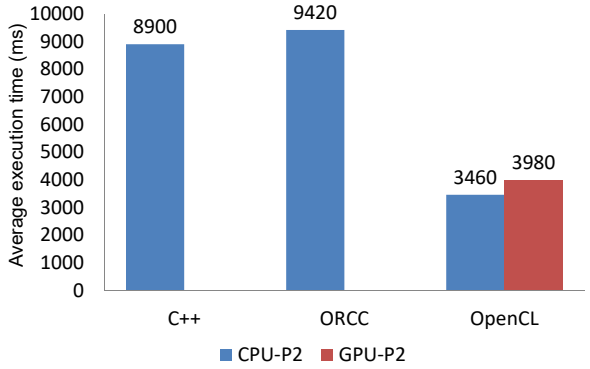

Figure 3.9.: *The results evaluated for the ZigBee benchmark using experimental tool-2 are shown in this figure. The results measured on the CPU are depicted by blue bars and the results on the GPU are depicted by the red ones.*

Discussion. As the parallelized actions perform more complex operations, the generated implementations showed promising results. The OpenCL CPU version again performed the best of all other versions, in particular, about 2.7× faster than the ORCC version and almost 2.6× faster than the parallel C++ version. With the increased complexity of parallelized actions, the OpenCL GPU version also performed substantially better than the ORCC and C++ versions. Precisely, it is 2.36× and 2.23× faster than the ORCC and C++ versions, respectively. However, the communication overhead caused by the data transfer between the GPU kernels and the host (main memory) still makes it slower than the OpenCL CPU.

Conclusions

Experimental tool-2 allowed us to evaluate the feasibility of employing the combination of RVC-CAL and OpenCL for modeling and implementing systems on different COTS target platforms. First, we evaluated RVC-CAL as a language for modeling such systems as DPNs. We identified that RVC-CAL DPNs explicitly offer parallelism both at the level of processes and at the level of actions. To efficiently execute these models on a target hardware, it is important to generate parallel code based on the entire parallelism provided by

these two levels. Second, OpenCL also provides means to exploit task-level and data-level parallelism.

Based on the evaluation, the proposed approach has shown promising results with significant improvement in the end-to-end performance. In particular, even with the selection of applications where only few actions could further be parallelized based on the parallelization criteria, the approach has substantially improved the performance up to 2.7× in comparison with the reference approaches. Furthermore, we also observed that the applications can be ported to a variety of heterogeneous platforms consisting of devices of different types from different vendors without revising the code (kernels) at all. Employing OpenCL does not only allow us to target vendor-neutral heterogeneous architectures, but also provides a standard abstraction for better resource utilization in parallel architectures.

In a nutshell, we can conclude that systematically incorporating the combination of RVC-CAL and OpenCL effectively contributes in modeling the parallel and distributed computing systems as DPNs and in implementing them on various heterogeneous COTS hardware. Based on these evaluations, we therefore systematically employ a limited subset of CAL and OpenCL as an integral part of the synthesis design flow proposed in this thesis.

3.2. The Proposed Synthesis Design Flow: Overview

This thesis proposes a model-based design flow for automatic software synthesis of systems from DPN models. The design flow is implemented in an extendable model-based design framework. In contrast to existing design tools for synthesis, the proposed framework enables us to automatically synthesize systems based on different well-defined and precise dataflow MoCs including their heterogeneous combinations. We therefore offer a framework for automatic software synthesis that maps models using a combination of different dataflow MoCs on cross-vendor COTS target hardware. The framework supports three different dataflow MoCs, namely the synchronous (static) dataflow (SDF) [LeMe87a] MoC, the Kahn process networks (KPN) [KaMa77] MoC and a deterministic variant of the dynamic dataflow (DDF) [Kosi78] MoC. The KPN and DDF MoCs allow processes whose actions consume different numbers of input tokens and produce different numbers of output tokens 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. The SDF MoC is the most restrictive amongst the supported dataflow MoCs in the sense that it only supports processes having static behaviors. The KPN MoC can capture static as well as sequential behaviors. The DDF MoC further supports processes with parallel behaviors.

The overall design flow can be understood in two phases i.e., the modeling phase and the synthesis phase as shown in Figure 3.10. In general, the starting point of this work is a desynchronized model. As explained in detail in Chapter 2.2.2, the desynchronization design flow based on our *Averest* tool finally generates a CAL DPN based on the supported dataflow MoCs. To

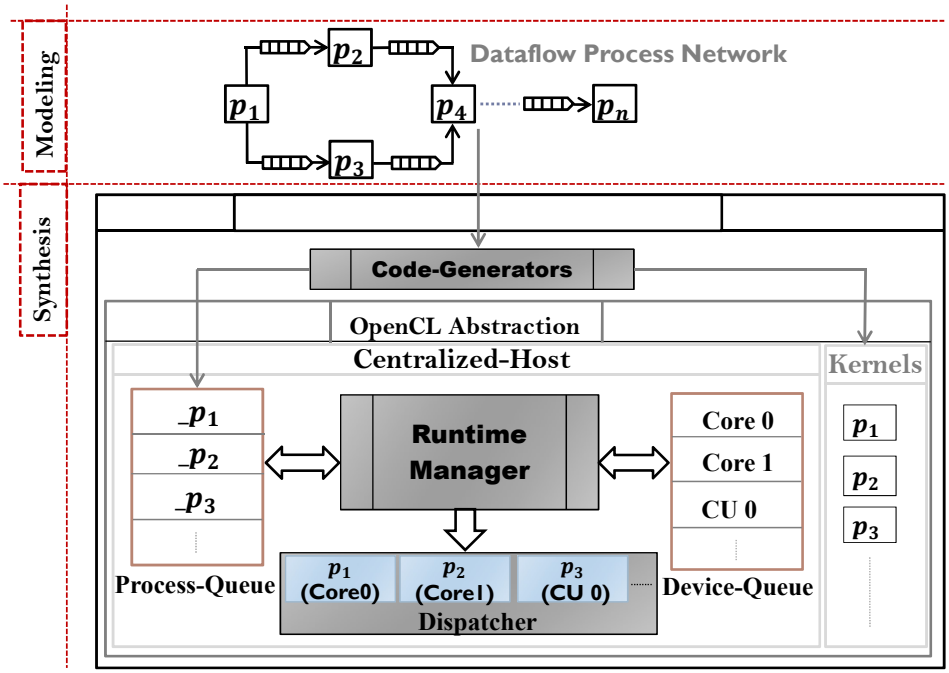

Figure 3.10.: *The basic building block diagram of the proposed framework. It can be understood in two phases: the modeling phase is in general provided with the desynchronized CAL DPN models. The synthesis phase employs the OpenCL abstraction and features a tool chain involving the specialized code generators and the runtime system that finally executes and maps the desynchronized models on the OpenCL abstracted target hardware.*

this end, a general CAL DPN model is considered 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 restrictions to specify the precise dataflow MoCs. Each supported dataflow MoC interprets the behavior of the CAL process in two parts: **(i)** process's triggering or scheduling behavior and **(ii)** process's execution behavior. The triggering behavior determines the conditions under which the dataflow MoC triggers a process for an execution. Whereas, the execution behavior determines how a process consumes/produces data when it is triggered for an execution. The general DPN model based on the used CAL subset and the supported dataflow MoCs are described in Chapter 4.

Depending on the modeled synchronous behavior, the generated CAL DPN may be heterogeneous in the sense that it consists of processes offering different kinds of behaviors. Thus, different components of the system can be based on different kinds of dataflow processes, and therefore also different precise dataflow MoCs. The generated desynchronized CAL code consists of two parts: the CAL processes and the network description that specifies the network topology. Moreover, since the proposed design flow is implemented in a

standalone framework, a system can also be directly modeled as a CAL DPN based on the supported dataflow MoCs. The desynchronized or modeled DPN consisting of CAL processes and the network specification is provided as input to the synthesis phase.

The synthesis phase as shown in Figure 3.10 provides a comprehensive tool chain, including a single back-end that offers different specialized code generators for different dataflow MoCs, and a runtime system which finally executes DPNs on the target hardware. Using OpenCL [StGS10], it incorporates a standard hardware abstraction for cross-vendor heterogeneous hardware architectures. The proposed framework conceptually employs OpenCL as an operating system (OS) in the sense that it provides: common services for managing the target hardware, software resources and the implementation of modeled systems based on the supported dataflow MoCs. Thus, the framework logically incorporates the OpenCL specification as an OS mainly because of two reasons: first, it provides an abstraction for heterogeneous hardware, and second, the framework uses this abstraction in the composition of the synthesis where different components implement different low-level details. As discussed, 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 [RaSc20].

The framework adopts this idea of host and kernels for the synthesis as shown in Figure 3.10. 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 dataflow MoC of that process. In particular, the generated kernel implements the execution behavior of the process. A single back-end based on OpenCL is developed that provides different specialized code generators for specific dataflow MoCs. Each code generator generates kernel code strictly based on its underlying dataflow MoC. The syntactical representation of each dataflow MoC is used by the back-end to ascertain the MoC of each process in the network and enables the synthesis phase to automatically realize the implementations based on the kinds of behaviors of the processes. The runtime system systematically employs OpenCL as an integral part of the synthesis and manages the scheduling of processes and their communication based on the precise dataflow MoC of each process. A scheduler each, is designed for each dataflow MoC that schedules a process based on the triggering semantics of the underlying MoC. The runtime system 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.

The framework supports the synthesis of heterogeneous DPNs by using the combination of different specialized code generators. Moreover, the synthesis tool chain provides a common environment that dynamically schedules and executes the generated kernels at runtime based on their underlying dataflow MoCs. This dynamic environment is designed to enable the synthesis of het-

erogeneous composition of different kinds of processes under the supervision of a common framework. The synthesis tool chain along with all its essential components is presented in detail in Chapter 5. In particular, the back-end comprising of different code generators is explained. The runtime system and the associated host based on OpenCL is also discussed in detail in Chapter 5.

Chapter 4

Modeling: Dataflow Models of Computation

Contents

As discussed, the starting point of this thesis is the desynchronized deterministic dataflow models. In particular, the synchronous guarded actions (SGAs) are verified for desynchronization and then translated into the Cal actor language (CAL) guarded actions [BSBK14; BaRS21]. The target dataflow process network (DPN) model is based on a limited subset of CAL that is comprised of stateless processes having guarded actions. The main purpose of this chapter is not to present the formal specification of dataflow models of computation (MoCs) as this has been thoroughly considered in the literature [Faus82; GeBa03; Star87]. Instead, the main idea here is to informally illustrate how the limited subset of CAL is used to specify a general model of DPN and how this general model is used with specific constraints and restrictions to specify the precise dataflow MoCs. We therefore first present the syntax and the

informal semantics of the general DPN model based on the used CAL subset and then illustrate the constraints to specify the supported dataflow MoCs.

4.1. The General Model of DPN

A DPN is a set of processes $\mathcal{P} = \{p_0, \ldots, p_{m-1}\}$ with static point-to-point connections via FIFO buffers $\mathcal{F} = \{f_0, \ldots, f_{n-1}\}$. We also assume a total order $\preceq$ on the FIFO buffers so that we can unambiguously switch from sets to tuples of FIFO buffers by simply ordering the corresponding set to a tuple. For this reason, we often ignore the difference between sets and the corresponding tuples. For any tuple $t = (t_0, \ldots, t_\ell)$, we denote its components as $t_i = \mathsf{proj}_i(t)$. Processes of the DPN communicate with each other by consuming data tokens from their input buffers and adding data tokens to the output buffers. Therefore, we define for each process $p \in \mathcal{P}$, the tuple of its input buffers $\mathsf{inBuf}(p)$ and its output buffers $\mathsf{outBuf}(p)$.

In the following, we informally present and elaborate the syntax and the semantics of the general model of a process based on the used subset of CAL.

4.1.1. Syntax

The syntax of a process $p \in \mathcal{P}$ based on the used subset of CAL is illustrated with an abstract example as shown in Listing 4.1. A process generally consists of a set of input and output buffers and several actions.

```
 1 actor ex() <Type> X_1,...,<Type> X_M ==> <Type> Y_1,...,<Type> Y_N :
 2 label_α: action X_1:[x_{1,1},...,x_{1,p_1}],...,X_m:[x_{m,1},...,x_{m,p_m}] ==>
 3                 Y_1:[y_{1,1},...,y_{1,q_1}],...,Y_n:[y_{n,1},...,y_{n,q_n}]
 4   guard γ
 5   do
 6     y_{1,1} := e_{1,1};
 7     ⋮
 8     y_{1,q_1} := e_{1,q_1};
 9     ⋮
10     y_{n,q_n} := e_{n,q_n};
11   end
12 ⋮
13 label_α: action X_1:[x_{1,1},...,x_{1,p_1}],...,X_m:[x_{m,1},...,x_{m,p_m}] ==>
14                 Y_1:[y_{1,1},...,y_{1,q_1}],...,Y_n:[y_{n,1},...,y_{n,q_n}]
15   guard γ
16   do
17     y_{1,1} := e_{1,1};
18     ⋮
19     y_{1,q_1} := e_{1,q_1};
20     ⋮
21     y_{n,q_n} := e_{n,q_n};
22   end
23 endactor
```

Listing 4.1: *Abstract example of a process based on the general CAL DPN model.*

4.1.2. Semantics

The abstract example of a process as shown in Listing 4.1 illustrates the general model based on the used subset of CAL. The head of a process $p \in \mathcal{P}$ specifies the input buffers $\mathsf{inBuf}(p) = (X_1, \ldots, X_M)$ and output buffers $\mathsf{outBuf}(p) = (Y_1, \ldots, Y_N)$, including the type of tokens communicated via the buffer (Line 1). The used CAL subset provides three data types: Boolean, integer and real numbers. The behavior of every process $p \in \mathcal{P}$ is determined by a set of actions $\mathsf{actions}(p) = \{\alpha_1, \ldots, \alpha_h\}$. Actions are preceded by action labels which in the general model need not to be unique, i.e. the same label can used for more than one action (Lines 2 and 13). The head of an action $\alpha \in \mathsf{actions}(p)$ specifies for the input buffers in $\mathsf{inBuf}(p)$ the number of data tokens to be read (Line 2). It may or may not specify all input buffers in $\mathsf{inBuf}(p)$. If the action is fired, these data tokens are consumed from the heads of input buffers and are assigned to the variables $x_{i,j}$ such that $x_{i,1}$ is the head of the input buffer X_i. Analogously, the action interface determines for the output buffers in $\mathsf{outBuf}(p)$ the number of data tokens to be written. Thereby, the values $y_{i,1}, \ldots, y_{i,q_i}$ are added in this order to the tail of output buffer Y_i. The body of the action is therefore a sequence of statements that compute values based on expressions $e_{i,1}, \ldots, e_{i,q_i}$ and assign them to output variables $y_{i,1}, \ldots, y_{i,q_i}$ (Lines 5-11). An expression may compose of variables, values, and both arithmetic and Boolean expressions. Since only a subset of $\mathsf{inBuf}(p)$ may be used by an action $\alpha \in \mathsf{actions}(p)$, we also define $\mathsf{inAct}(\alpha) \subseteq \mathsf{inBuf}(p)$ as the subset of input buffers used by that action. Similarly, we define $\mathsf{outAct}(\alpha) \subseteq \mathsf{outBuf}(p)$ as the subset of output buffers used by the action. For an action $\alpha \in \mathsf{actions}(p)$ that requires that input tokens have particular values, an additional condition can be specified using a guard (Line 4) which is a predicate on the tokens of (some prefixes of) the input buffers in $\mathsf{inAct}(\alpha)$. Since only a subset of $\mathsf{inAct}(\alpha)$ may be used by a guard, we also define $\mathsf{inGrd}(\alpha) \subseteq \mathsf{inAct}(\alpha)$ as the subset of input buffers whose values are considered by the guard γ_α of action α.

For the semantics, we consider a domain $\mathcal{D}$ of values that may be the union of integers, booleans and real numbers. We denote the set of finite sequences on $\mathcal{D}$ as $\mathcal{D}^*$ and the set of infinite sequences on $\mathcal{D}$ as $\mathcal{D}^\omega$, and the union of both as $\mathcal{D}^\infty$, i.e., $\mathcal{D}^\infty := \mathcal{D}^* \cup \mathcal{D}^\omega$. For sequences $\sigma_1, \sigma_2 \in \mathcal{D}^\infty$, we introduce the prefix ordering $\sigma_1 \sqsubseteq \sigma_2 :\Leftrightarrow \exists \sigma_3 \in \mathcal{D}^\infty.\ \sigma_2 = \sigma_1 \cdot \sigma_3$ where $\sigma_1 \cdot \sigma_3$ means the concatenation of the sequences σ_1 and σ_3 which demands that $\sigma_1 \in \mathcal{D}^*$. The prefix ordering on sequences $\sigma_1, \sigma_2 \in \mathcal{D}^\infty$ is lifted to tuples of sequences $\sigma_1 = (\sigma_{1,0}, \ldots, \sigma_{1,\ell})$ and $\sigma_2 = (\sigma_{2,0}, \ldots, \sigma_{2,\ell})$ in that we demand $\sigma_{1,i} \sqsubseteq \sigma_{2,i}$ for all $i \in \{0, \ldots, \ell\}$.

Each process $p \in \mathcal{P}$ defines a function that maps the consumed input tokens to produced output tokens. This function is determined by a set of actions $\mathsf{actions}(p)$ of the process p where the semantics of each action $\alpha \in \mathsf{actions}(p)$ is a function of type $(\mathcal{D}^*)^m \to (\mathcal{D}^*)^n$ with the following meaning: The action consumes tokens from m input buffers and produces tokens to n output buffers, thus, $m := |\mathsf{inAct}(\alpha)|$ and $n := |\mathsf{outAct}(\alpha)|$.

Any action $\alpha \in \mathsf{actions}(p)$ as shown in Listing 4.1 is enabled iff the following

```
1.  actor nondet-merge() int X1, int X2 ==> int Y1
2.  act1: action X1: [x1] ==> Y1: [y1]
3.        guard true
4.        do
5.           y1:= x1;
6.        end
7.  act2: action X2:[x2] ==> Y1: [y1]
8.        guard true
9.        do
10.          y1:= x2;
11.       end
12. end
```

Figure 4.1.: *A simple example of a non-deterministic process in CAL. The output produced on **Y1** depends on the arrival time of tokens on the inputs **X1** and **X2**.*

conditions are all satisfied:

- each input buffer $X_i \in \mathsf{inAct}(\alpha)$ has enough tokens, i.e., X_i must have at least p_i many tokens
- each output buffer $Y_i \in \mathsf{outAct}(\alpha)$ has enough space, i.e., Y_i must have at least space for q_i many tokens
- the guard condition γ which is a condition on the input tokens $x_{i,j}$ in $\mathsf{inGrd}(\alpha)$ is satisfied

The general model of DPN does not impose further restrictions and therefore actions consisting of common inputs and/or outputs may be enabled in the same execution, as depicted in Listing 4.1. As a result, this gives rise to read and write conflicts in buffers, ultimately ending up in non-deterministic behaviors. A read conflict means that two actions are enabled in an execution that read a token from the same input. Whereas, a write conflict means that two actions are enabled in an execution that write a token to the same output. A simple example of a non-deterministic process is illustrated in Figure 4.1. It consists of two actions *act1* and *act2* that consume tokens from different inputs **X1** and **X2**, respectively, and produce tokens to the common output **Y1**. Depending on the availability of tokens on the inputs, both actions may be enabled in the same execution, and therefore may give rise to write conflict in **Y1**. Hence, the output produced on **Y1** depends on the arrival time of tokens on the inputs and therefore exhibits a non-deterministic behavior.

We therefore demand and use the general CAL DPN model with specific constraints and restrictions to specify the precise dataflow MoCs.

4.2. Static Dataflow Model

The static dataflow (SDF) [LeMe87a] MoC allows one to model static (synchronous) behaviors. It is a more restricted DPN class in the sense that the decision on whether to consume and produce tokens in each execution is statically made (fixed). Each execution of a process consumes and produces fixed number of tokens. To this end, the number of tokens consumed or produced by each node (process) must be specified a priori. In other words, the number of tokens consumed or produced on each buffer must be independent of the value as well as the arrival time of data. A process in SDF becomes enabled if and only if all its inputs have required tokens and all its outputs have required space. An enabled process may fire, and once fired, consumes the statically specified number of tokens from its inputs and produces the statically specified number of tokens to its outputs.

We demand and assume certain restrictions on the general DPN model to represent the SDF MoC. In the following, we present an abstract example of a static process based on the SDF MoC and informally illustrate its semantics.

4.2.1. Syntax

The syntax of a static process in SDF is illustrated with an abstract example as shown in Listing 4.2.

```
1  actor SDF() <Type> X_1,...,<Type> X_m ==> <Type> Y_1,...,<Type> Y_n :
2  label_α1 : action X_1 : [x_1,1,...,x_1,p1],...,X_m : [x_m,1,...,x_m,pm] ==>
3                    Y_1 : [y_1,1,...,y_1,q1],...,Y_n : [y_n,1,...,y_n,qn]
4     guard γ_α1
5     do
6        y_1,1 := e_1_1,1;
7        ⋮
8        y_1,q1 := e_1_1,q1;
9        ⋮
10       y_n,qn := e_1_n,qn;
11    end
12 ⋮
13 label_αh : action X_1 : [x_1,1,...,x_1,p1],...,X_m : [x_m,1,...,x_m,pm] ==>
14                   Y_1 : [y_1,1,...,y_1,q1],...,Y_n : [y_n,1,...,y_n,qn]
15    guard γ_αh
16    do
17       y_1,1 := e_h_1,1;
18       ⋮
19       y_1,q1 := e_h_1,q1;
20       ⋮
21       y_n,qn := e_h_n,qn;
22    end
23 endactor
```

Listing 4.2: *Abstract example of a process in SDF.*

4.2.2. Semantics

A process $p \in \mathcal{P}$ in SDF consists of a set of actions $\mathsf{actions}(p) = \{\alpha_1, \ldots, \alpha_h\}$. The action labels need to be unique, i.e. the same label must not be used for more than one action (Lines 2 and 13). For each action α_i, we define its guard γ_{α_i}. Each action $\alpha_i \in \mathsf{actions}(p)$ specifies for all input buffers $\mathsf{inBuf}(p)$ and all output buffers $\mathsf{outBuf}(p)$ the number of tokens to be read and written, respectively. Thus, the input and output buffers are always same across all actions i.e., $\mathsf{inAct}(\alpha_1) = \ldots = \mathsf{inAct}(\alpha_h)$ and $\mathsf{outAct}(\alpha_1) = \ldots = \mathsf{outAct}(\alpha_h)$. Second, the number of tokens to be consumed and the number of tokens to be produced are always same for the same input and output buffers, respectively, across all actions. This restricts the execution of processes to fixed consumption and production rates. Regardless of which action is executed, the same number of tokens are consumed and produced in the same buffers in each firing of a process. Moreover, we demand that the guard conditions should always be mutually exclusive across actions. This ensures that for each execution of a process, the actions will never compete for an execution. Hence, in each execution of a process only a specific action is fired whose guard is enabled.

Execution of Actions

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 $\mathsf{actions}(p)$ are always evaluated sequentially in the same order of their actions definitions. Since all actions in a process have same input buffers with same consumption rates, hence for any action $\alpha_i \in \mathsf{actions}(p)$, the specified fixed number of tokens are first consumed from all input buffers $\mathsf{inAct}(\alpha_i) = \mathsf{inBuf}(p)$. Finally, the enabled action is fired whose guard is true. Upon firing, the defined computations are performed and the specified fixed number of tokens are produced to all output buffers $\mathsf{outAct}(\alpha_i) = \mathsf{outBuf}(p)$.

We therefore assume that a particular guard must be true in each execution of a process. In case if none of the guards is true in an execution, it can simply be considered as an erroneous or an incomplete behavior where no functionality is specified for certain values of tokens. On the other hand, this can be tackled purely at the implementation level by using the so-called *silent* action that only fires if no other action is fired in an execution. The *silent* action may simply consume those tokens and produce some dummy output to continue with the rest of the executions or it may simply terminate the process to indicate the erroneous behavior.

Triggering Processes for Execution

Each process $p \in \mathcal{P}$ in SDF is triggered for an execution if and only if all input buffers $\mathsf{inAct}(\alpha_i)$ of an action $\alpha_i \in \mathsf{actions}(p)$ have enough input tokens and all output buffers $\mathsf{outAct}(\alpha_i)$ of that action have enough space. The process shown in Listing 4.2 is triggered for an execution iff for any action α_i, each input buffer $X_j \in \mathsf{inAct}(\alpha_i)$ has at least p_j many tokens and each output buffer

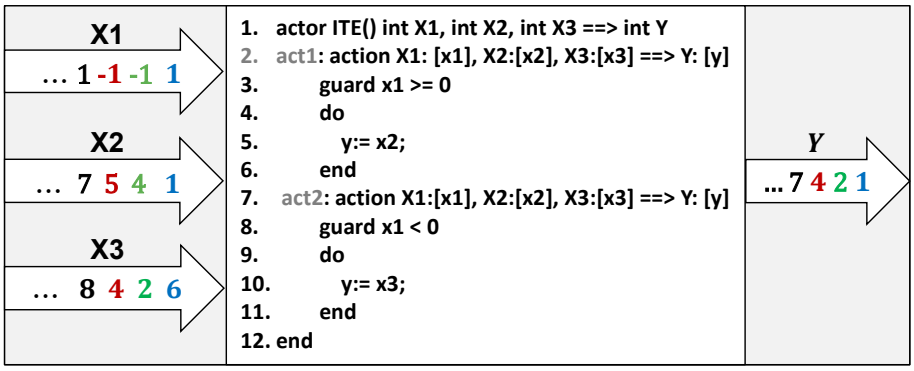

Figure 4.2.: *The static if-then-else (ITE) node: a simple example of a static process in SDF. An example behavior is illustrated with a set of input values and the computed output values as shown inside arrows.*

$Y_j \in \mathsf{outAct}(\alpha_i)$ has at least space for q_j many tokens.

SDF Process Example

A simple example of the static if-then-else (ITE) operation is illustrated in Figure 4.2. In each execution, the ITE process consumes a token each from all three inputs and produces a token to its only output. It consists of two actions i.e., *act1* and *act2*, having same inputs **X1**, **X2** and **X3**, and the same output **Y** (Lines 2 and 7). Both actions use the input **X1** for the guard with mutually exclusive guard conditions (Lines 3 and 8). In each execution, depending on which guard is enabled, either *act1* or *act2* fires for an execution. ITE is only triggered for an execution if there is a token available in all three inputs **X1**, **X2** and **X3** and if there is space available for a token to be produced at the output **Y**.

4.3. Kahn Process Networks Model

Kahn process networks (KPNs) [KaMa77] are dynamic DPNs where processes can consume and produce different number of tokens in every firing depending on the history of the consumed tokens and also on the tokens to be consumed. In other words, it supports the conditional or data dependent execution of processes where each process can vary its consumption and production rates in every firing. KPNs 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:

- processes are not allowed to test input buffers for the existence of tokens
- reading from input buffers is blocking, and writing to output buffers is non-blocking
- processes must implement deterministic sequential functions

- processes do not need all of their inputs to get triggered for execution

Based on these restrictions/properties, it can be implied that a process in KPN can be any sequential program where the firing rules can be tested in a predefined order in each execution using blocking reads [LePa95]. This reflects the ability to uniquely consume the inputs in each firing without timing information provided by the input signals. A KPN process is only triggered for execution if the exact information on inputs required to produce the output is available. A process therefore becomes enabled if the required values on inputs are available to perform the computation and produce the output. A process once enabled, may fire, and once fired, it may consume different number of tokens from different inputs based on the history of the consumed tokens. The KPN MoC can capture both static as well as sequential behaviors. Since buffers with unbounded capacity cannot be realized in real implementations, the used KPN model only supports blocking write. However, since the starting point of this work is the desynchronized models, desynchronization preserves properties like deadlock-freedom and bounded memory usage that are otherwise difficult to ensure in KPNs [BaRS21].

Next, we present an abstract example of a sequential process based on the KPN MoC and illustrate its semantics.

4.3.1. Syntax

The syntax of a sequential process in KPN is illustrated with an abstract example as shown in Listing 4.3.

```
1  actor KPN() <Type> X_1,...,<Type> X_M ==> <Type> Y_1,...,<Type> Y_N :
2  label_α1 : action X_1 : [x_1,1,...,x_1,p1],...,X_m : [x_m,1,...,x_m,pm] ==>
3                   Y_1 : [y_1,1,...,y_1,q1],...,Y_n : [y_n,1,...,y_n,qn]
4    guard γ_α1
5    do
6      y_1,1 := e_1_1,1;
7      ⋮
8      y_1,q1 := e_1_1,q1;
9      ⋮
10     y_n,qn := e_1_n,qn;
11   end
12 ⋮
13 label_αh : action X_1 : [x_1,1,...,x_1,f1],...,X_u : [x_u,1,...,x_u,fu] ==>
14                  Y_1 : [y_1,1,...,y_1,g1],...,Y_v : [y_v,1,...,y_v,gv]
15   guard γ_αh
16   do
17     y_1,1 := e_h_1,1;
18     ⋮
19     y_1,g1 := e_h_1,g1;
20     ⋮
21     y_v,gv := e_h_v,gv;
22   end
23 endactor
```

Listing 4.3: *Abstract example of a process in KPN.*

4.3.2. Semantics

A process $p \in \mathcal{P}$ in KPN consists of a set of actions $\mathsf{actions}(p) = \{\alpha_1, \ldots, \alpha_h\}$. The action labels need to be unique, i.e. the same label must not be used for more than one action (Lines 2 and 13). For each action α_i, we define its guard γ_{α_i}. Each action $\alpha_i \in \mathsf{actions}(p)$ specifies for the input buffers $\mathsf{inAct}(\alpha_i) \subseteq \mathsf{inBuf}(p)$ and the output buffers $\mathsf{outAct}(\alpha_i) \subseteq \mathsf{outBuf}(p)$ the number of tokens to be read and written, respectively. In general, the input and output buffers can be different across different actions. However, since processes in KPN consist of sequential functions, we demand that all actions in a process must have at least one common input. This implies that $\mathsf{inAct}(\alpha_1) \cap \ldots \cap \mathsf{inAct}(\alpha_h) \neq \{\}$. Moreover, we demand that the guard conditions are always mutually exclusive across actions. This ensures that for each execution of a process, the actions will never compete for an execution. Hence, in each firing of a process only a specific action is executed mainly dependent on which guard is enabled. Altogether, these restrictions ensure that for each execution of a process, the actions will never compete for an execution for any set of tokens. Second, they enable the execution of processes with dynamic consumption rates and dynamic production rates, mainly dependent on which guards are enabled on each execution.

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 $\alpha_i \in \mathsf{actions}(p)$ is executed whose guard γ_{α_i} is enabled. The enabled action α_i, once fires, consumes a finite number of tokens from the input buffers $\mathsf{inAct}(\alpha_i)$ and produces a finite number of tokens to the output buffers $\mathsf{outAct}(\alpha_i)$ as specified for that action.

Triggering Processes for Execution

Since processes in KPNs consist of sequential programs, the availability of tokens on the inputs, the availability of space on the outputs, and the guards can be evaluated sequentially in a predefined order of their actions definitions. Each process $p \in \mathcal{P}$ is triggered for an execution if there exists one particular action $\alpha_i \in \mathsf{actions}(p)$ having: enough input tokens in $\mathsf{inAct}(\alpha_i)$, required values on the guarded inputs $\mathsf{inGrd}(\alpha_i)$, and enough space in $\mathsf{outAct}(\alpha_i)$. For instance, the process shown in Listing 4.3 is triggered for an execution when for a particular action (say α_h), each input buffer $X_j \in \mathsf{inAct}(\alpha_h)$ has at least f_j many tokens, each output buffer $Y_j \in \mathsf{outAct}(\alpha_h)$ has at least space for g_j many tokens, and the guard γ_h is true. In case if one of the inputs does not have enough tokens, the process is blocked (i.e., the blocking behavior of KPN) until sufficient tokens are available on that input.

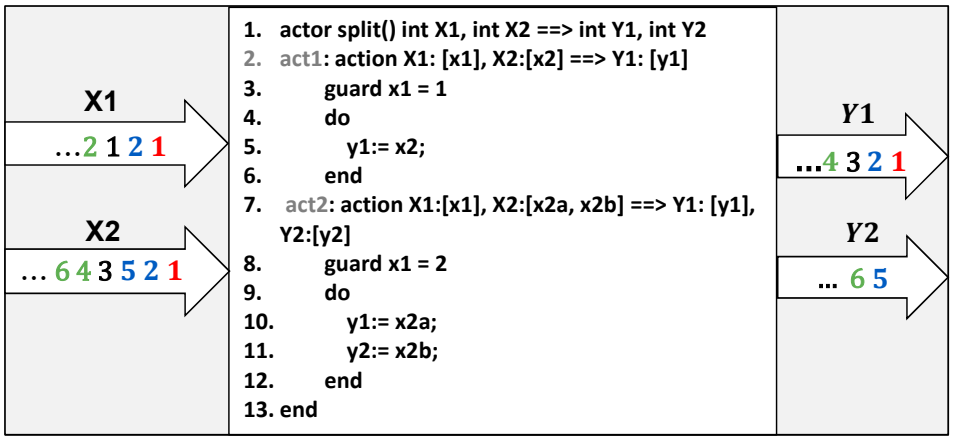

Figure 4.3.: *The split node: a simple example of a sequential process in KPN. An example behavior is illustrated with a set of input values and the computed output values as shown inside arrows.*

KPN Process Example

A simple example of a sequential process, namely the *split* node is illustrated in Figure 4.3 that splits a single input channel to a number of individual output channels. The split node consists of two actions *act1* and *act2* that depending on the value of a token at the input **X1** splits the tokens from the input **X2** to outputs **Y1** and **Y2**. Both actions share a common input, namely **X1**. The guards are composed of mutually exclusive conditions (Lines 3 and 8). Both actions declare the input **X2** with different consumption rates (Lines 2 and 7). The action *act2* has an additional output **Y2**. In each execution, depending on which guard is enabled, either *act1* or *act2* fires for an execution. In the case where *act1* fires, a single token each is consumed from **X1** and **X2**, and a single token is produced to **Y1** (Line 5). On the contrary, when *act2* fires, a single token is consumed from **X1**, two tokens are consumed from **X2** and a token each is produced to **Y1** and **Y2** (Lines 10-11). Hence, in each execution, a different number of tokens can be consumed from the input **X2** and a different number of tokens can be produced to outputs **Y1** and **Y2**. The split node is only triggered for an execution if there exists one action i.e., either *act1* or *act2*, having required number of tokens in **X1** and **X2**, required space in outputs **Y1** and **Y2**, and required values on the input **X1**.

4.4. Dynamic Dataflow Model

The dynamic dataflow (DDF) [Kosi78] also sometimes referred to as the non-determinate dataflow is a dynamic DPN class that allows one to model dynamic and asynchronous processes. It offers a more generalized data dependent and asynchronous execution semantics than the KPN MoC. In particular, the DDF MoC allows one to model processes with parallel programs consisting of concurrent and independent computations where more than one action can

```
 1 actor DDF() <Type> X_1,..., <Type> X_m,..., <Type> X_M ⟹
 2             <Type> Y_1,..., <Type> Y_n,..., <Type> Y_N :
 3 label_α1 : action X_1 : [x_1,1,..., x_1,p1],..., X_m : [x_m,1,..., x_m,pm] ⟹
 4              Y_1 : [y_1,1,..., y_1,q1],..., Y_n : [y_n,1,..., y_n,qn]
 5   guard γ_α1
 6   do
 7     y_1,1 := e_1_1,1;
 8     ⋮
 9     y_1,q1 := e_1_1,q1;
10     ⋮
11     y_n,qn := e_1_n,qn;
12   end
13 ⋮
14 label_αk : action X_m+1 : [x_m+1,1,..., x_1,pm+1],..., X_M : [x_M,1,..., x_M,pM] ⟹
15              Y_n+1 : [y_n+1,1,..., y_n+1,qn+1],..., Y_N : [y_N,1,..., y_N,qN]
16   guard γ_αk
17   do
18     y_n+1,1 := e_k_n+1,1;
19     ⋮
20     y_n+1,qn+1 := e_k_n+1,qn+1;
21     ⋮
22     y_N,qN := e_k_N,qN;
23   end
24 ⋮
25 label_αh : action
       X_1 : [x_1,1,..., x_1,f1],..., X_m : [x_m,1,..., x_m,fm]..., X_M : [x_M,1,..., x_M,gM] ⟹
26           Y_1 : [y_1,1,..., y_1,q1],..., Y_n : [y_n,1,..., y_n,qn] ...Y_N : [y_N,1,..., y_N,qN]
27   guard γ_αh
28   do
29     y_1,1 := e_h_1,1;
30     ⋮
31     y_1,q1 := e_h_1,q1;
32     ⋮
33     y_N,qN := e_h_N,qN;
34   end
35 endactor
```

Listing 4.4: *Abstract example of a process in DDF.*

be executed in each firing. This generalization results in higher expressiveness and flexibility, however, may lead to non-deterministic behaviors e.g., a non-determinate merge [LePa95]. We use a variant of the DDF MoC that only supports concurrent and independent actions with specific restrictions. These restrictions demand that although multiple actions may fire in the same execution of a process, however, they must not give rise to read conflicts in buffers. Second, they must not lead to different output streams for the same input streams. It can be used to model well-behaved parallel nodes that exhibit deterministic behaviors e.g., the parallel OR (POR) node as illustrated in Figure 4.4 (discussed in the next few subsections).

The considered variant of DDF MoC offers a more flexible semantics where each process becomes enabled for an execution if only one of its inputs has required tokens and only one of its outputs has required space. The decision

on whether to consume/produce tokens and to execute each action of an enabled process is made dynamically at runtime when that particular process is triggered for an execution. A process once enabled, may fire, and once fired, it may trigger multiple actions for execution. A process can produce and consume different number of tokens in every firing. Thus, each process can have either static or dynamic consumption/production token rate per execution. The considered variant of DDF MoC can capture static, sequential and well-behaved parallel processes.

In the following, we present an abstract example of a well-behaved parallel process based on the DDF MoC and informally illustrate its semantics.

4.4.1. Syntax

The syntax of a well-behaved parallel process in DDF is illustrated with an abstract example as shown in Listing 4.4.

4.4.2. Semantics

A process $p \in \mathcal{P}$ in DDF consists of a set of actions $\mathsf{actions}(p) = \{\alpha_1, \ldots, \alpha_h\}$. The action labels need to be unique, i.e. the same label must not be used for more than one action (Lines 3, 14 and 25). For each action α_i, we define its guard γ_{α_i}. Different actions in $\mathsf{actions}(p)$ may specify completely different input and output buffers e.g., $\mathsf{inAct}(\alpha_1) \cap \mathsf{inAct}(\alpha_k) = \{\}$ and $\mathsf{outAct}(\alpha_1) \cap \mathsf{outAct}(\alpha_k) = \{\}$. This enables the modeling of processes with independent actions consisting of completely different inputs and outputs. Moreover, different actions in $\mathsf{actions}(p)$ may also specify common input and output buffers e.g., $\mathsf{inAct}(\alpha_1) \cap \mathsf{inAct}(\alpha_h) \neq \{\}$ and $\mathsf{outAct}(\alpha_k) \cap \mathsf{outAct}(\alpha_h) \neq \{\}$. The associated number of token variables (i.e., token consumption rates) of common input buffers can be different across actions. Multiple actions may fire in the same execution of a process. However, we demand that these firings must be consistent and do not give rise to non-deterministic behaviors. In particular, we demand that the guard conditions of actions with common input buffers are always mutually exclusive. Hence, in each firing of a process only a specific action from all actions having at least one common input buffer is executed whose guard is enabled.

This ensures that for each execution of a process, the actions with common input buffers will never compete for an execution for any set of tokens. Moreover, for actions with at least one common output buffer, we demand that each action upon firing produces the same sequence of tokens at the common output buffer. Hence, the firing of actions with common output buffers do not lead to different output streams. Altogether, these restrictions enable the execution of processes consisting of well-behaved parallel programs with dynamic consumption rates and dynamic production rates, mainly dependent on which guards are enabled on each execution. An example of a well-behaved parallel node exhibiting such a behavior is illustrated in Figure 4.4.

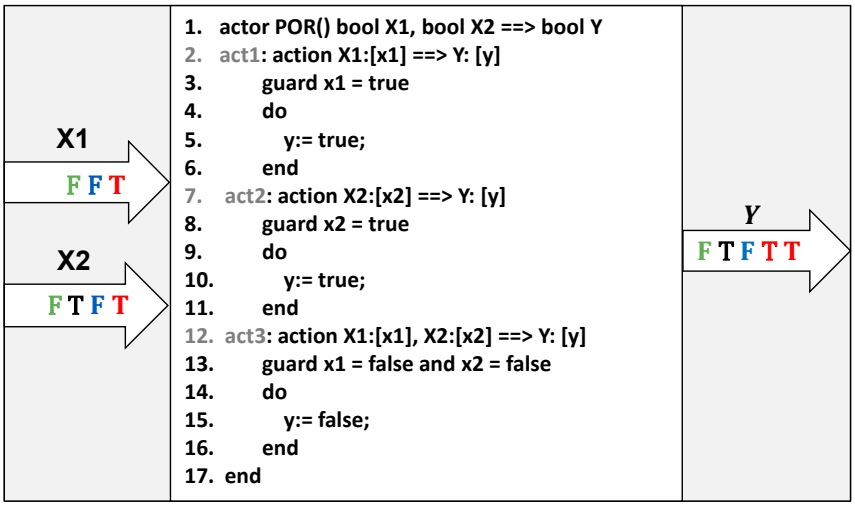

Figure 4.4.: *The parallel OR node: a simple example of a well-behaved parallel process in DDF. An example behavior is illustrated with a set of input values and the computed output values as shown inside arrows.*

Evaluation and Execution of Actions

Each action $\alpha_i \in \mathsf{actions}(p)$ of a process $p \in \mathcal{P}$ is evaluated for an execution dynamically when that particular process is triggered for an execution. Each action α_i fires for an execution iff: there are enough tokens available in $\mathsf{inAct}(\alpha_i)$, enough space available in $\mathsf{outAct}(\alpha_i)$, and the required values on the guarded inputs $\mathsf{inGrd}(\alpha_i)$ are available, i.e., the guard γ_i is true. When the action α_i fires, it consumes a finite number of tokens from $\mathsf{inAct}(\alpha_i)$ and produces a finite number of tokens to $\mathsf{outAct}(\alpha_i)$.

Since the actions are evaluated dynamically only after a particular process is triggered for an execution, there may be a case when for an action α_i, although the guard γ_i is true, however, either at least one of the input buffers in $\mathsf{inAct}(\alpha_i)$ does not have enough tokens, or at least one of the output buffers in $\mathsf{outAct}(\alpha_i)$ does not have enough space. In this case neither the input tokens are consumed from $\mathsf{inAct}(\alpha_i)$, nor the output tokens are produced to $\mathsf{outAct}(\alpha_i)$. Instead, the tokens are preserved in their respective input buffers.

Triggering Processes for Execution

As the decision to execute each action of a process and consume/produce data tokens is made dynamically at runtime, when that process is triggered for an execution. Therefore, each process $p \in \mathcal{P}$ is triggered for an execution if there exists at least one input buffer $X_j \in \mathsf{inAct}(\alpha_i)$ having enough input tokens and there exists at least one output buffer $Y_j \in \mathsf{inAct}(\alpha_i)$ having enough space. For instance, the process shown in Listing 4.4 is triggered for an execution when for a particular action (say α_1), any input buffer $X_j \in \mathsf{inAct}(\alpha_1)$ has at least p_j many tokens and any output buffer $Y_j \in \mathsf{outAct}(\alpha_1)$ has at least space for q_j many tokens.

DDF Process Example

A simple example of the parallel OR (POR) node is illustrated in Figure 4.4 that performs the logical OR operation on two Boolean inputs. The POR node consists of three actions *act1*, *act2* and *act3* that depending on the values of tokens in either or both of the inputs **X1** and **X2** produces tokens in the only output **Y**. The actions *act1* and *act3* share a common input **X1** (Lines 2 and 12). The actions *act2* and *act3* share a common input **X2** (Lines 7 and 12). All actions share the same output **Y** (Lines 2, 7 and 12). In each execution, depending on which guards are enabled, either one or both of the actions *act1*, *act2* can be fired. In case if both the actions are enabled, a token each is produced to the output **Y** by both actions (Lines 5 and 10). Since, they share a common output, hence, the same sequence of tokens is produced at the output **Y** by both actions. In contrast, if the guard is true for *act3*, the actions *act1*, *act2* become disabled where only the action *act3* is fired. Therefore, the actions with common inputs never compete for firing in any execution of the process. The POR node is only triggered for an execution if there is a token available in at least one of the inputs i.e., either **X1** or **X2**, and there is a space available for one token to be produced in the only output **Y**.

Chapter 5

Automatic Synthesis: The Tool Chain

Contents

Using our *Averest* tool, we model systems based on the synchronous reactive (SR) model of computation (MoC) and essentially transform them to the Cal actor language (CAL) DPN models as presented in Chapter 4. Second, a system can also be directly modeled in the proposed framework using the CAL DPN models. Regardless of which option is used, the CAL code consists of two parts: the CAL actors (processes) and the network description. The network description determines the topology of the network and is typically described using the XML based functional network language [BEJL11] as illustrated in Listing 2.2. Whereas, each CAL process based on a particular precise dataflow MoC models a behavior of a system's component as illustrated in the examples in Figures 4.2, 4.3 and 4.4. The proposed synthesis tool chain of the framework is provided with this CAL code.

In this chapter, we present in detail the synthesis tool chain. A brief overview of the tool chain is illustrated in Figure 5.1. It offers a comprehensive set of tools that work together to finally implement the CAL DPNs on the commercial off-the-shelf (COTS) target hardware. The tool chain is typically composed of two main parts: a special **back-end** comprising of specialized code generators for particular dataflow MoCs and the **runtime system**. Each code generator generates an OpenCL kernel for each process

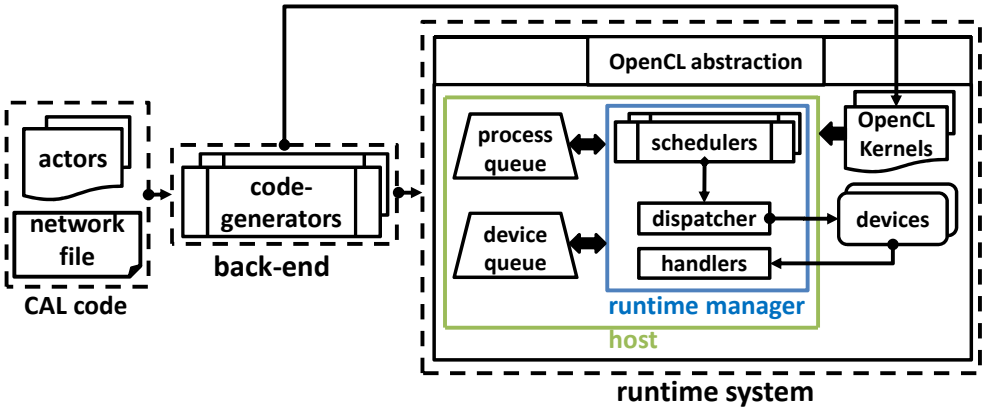

Figure 5.1.: *The proposed synthesis tool chain is composed of two main parts: the* ***back-end*** *features the specialized code generators for the supported dataflow MoCs. The* ***runtime system*** *organized in a host and kernels program model schedules and maps the modeled system on the target hardware.*

based on the underlying precise dataflow MoC. 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 deploy processes (generated kernels) on the target hardware.

5.1. Back-end

The back-end is designed to work in two different modes, namely the *manual* mode and the *auto* mode. In the manual mode, the back-end targets homogeneous implementations based on a specific user given dataflow MoC. In the auto mode, the back-end automatically classifies the processes into three categories mainly according to their kinds of behaviors that determine the precise dataflow MoCs. This classification of different kinds of behaviors involves: the static ones based on the static dataflow (SDF) MoC, the sequential ones based on the Kahn process network (KPN) MoC, and the parallel ones based on a variant of the dynamic dataflow (DDF) MoC. As a result, the back-end provides three different specialized code generators for particular dataflow MoCs: one for the SDF MoC, second for the KPN MoC, and finally for the used DDF MoC. Ideally, we identify the kinds of behaviors of processes during desynchronization based on a succinct formalization of input/output (I/O) firing rules of synchronous components [RBSY21; RBSY21a]. The identified dataflow MoC of each process is divulged to the synthesis phase through the network description file. However, the back-end also features a method that determines the kind of each process in the network by using the syntactical representation of each supported dataflow MoC as presented in Chapter 4. This method does not verify if a process exhibits a particular behavior, instead, it simply determines one of the three supported dataflow MoCs to which a particular process

belongs. This approach therefore assumes that the given process belongs to one of the supported dataflow MoCs. In this case, the identified dataflow MoC of each process is stored in the network description file at the back-end. Prior to presenting the specialized code generators, we first present the method designed to determine the underlying dataflow MoCs of processes based on their syntactical representations.

Algorithm 2: Pseudo code to determine the supported dataflow MoC of each process p in the network.

```
1  foreach process p in the network do
2      InFIFOsFirst ← input FIFOs of first action
3      OutFIFOsFirst ← output FIFOs of first action
4      DataflowMoC ← SDF                          ▷ assume the MoC is SDF
5      foreach action α in p do
6          if InFIFOsFirst ≠ inAct(α) ∨ OutFIFOsFirst ≠ outAct(α) then
7              DataflowMoC ← KPN      ▷ if not SDF, assume the MoC is KPN
8              break inner for loop
9          end
10     end
11     if DataflowMoC = KPN then
12         foreach action α in p do
13             CommonInFIFOs ← InFIFOsFirst ∩ inAct(α)
14             if IsEmpty(CommonInFIFOs) then
15                 DataflowMoC ← DDF          ▷ if not KPN, the MoC is DDF
16                 break inner for loop
17             end
18             else
19                 InFIFOsFirst ← CommonInFIFOs
20             end
21         end
22     end
23     store DataflowMoC for p in network description file
24 end
```

5.1.1. Dataflow MoC Identification

The method employed by the framework at the back-end to determine the precise dataflow model of each process in a network is illustrated by the pseudo code given in Algorithm 2. It follows a simple principle of validating the syntactical representations of processes starting from the most restrictive SDF MoC and then moving to the more generalized supported dataflow MoCs. The method works as follows:

It first assumes that the process exhibits a static behavior, or in other words, precisely belongs to the SDF MoC (Line 4). This assumption is then assessed by validating the process's syntax against the syntactical representation of the SDF MoC (Lines 5-10). To this end, the method iterates through the set of modeled actions and simply checks if each action exactly defines the same set of input and output FIFO buffers (Line 6). In case if one of the actions reveals a different set of input or output buffers, the assumption is considered

as invalid. The method then assumes that the process exhibits a sequential behavior (Line 7) and evaluates this assumption by validating the process's syntax against the syntactical representation of the KPN MoC (Lines 11-21). To this end, the method again iterates through the set of modeled actions and checks if the current action defines at least one common input buffer with the common input buffers of previous actions (Lines 13-14). In case if one of the actions does not have any input buffer common with the common input buffers of previous actions, the dataflow MoC of that process is simply validated as the most generalized DDF MoC (Line 15). The validated dataflow MoC of each process is finally stored in the network description file (Line 23).

In the following, we present the specialized code generators based on the supported dataflow MoCs.

5.1.2. Code Generators: Kernel Code Generation

A code generator each, is designed and developed based on the underlying semantics of each used dataflow MoC. Each code generator therefore generates an OpenCL kernel for each process strictly based on the underlying dataflow MoC. This section presents in detail the schemes employed for generating OpenCL kernel code based on all the supported dataflow MoCs. Moreover, we also illustrate the generated kernel code for each supported dataflow MoC based on the CAL models presented in Figures 4.2, 4.3 and 4.4.

Each code generator generates kernel code in two segments: First, the OpenCL specific code is targeted which involves the generation of the kernel header, the declaration of used inputs and outputs, and most importantly, the generation of generic kernel code that enables the host to dispatch multiple executions of the kernel on the device. This code segment is more or less same for all used dataflow MoCs. The second segment targets the code generation strictly based on the underlying semantics of the used dataflow MoC. We mainly present the schemes designed for generating code for the latter segment. The former segment is explained in detail with the generated kernel code examples.

SDF Code Generator

In SDF, it is statically determined that each firing of a process consumes/produces fixed number of tokens. A process in SDF is therefore simply scheduled by the host for execution if there is enough data available in all inputs and if there is enough space available in all outputs. The guards of actions are evaluated within kernels on the device side. The SDF code generation is relatively straightforward. The code generation based on the underlying semantics of the SDF MoC is illustrated by the pseudo code given in Algorithm 3.

For a static process, the proposed scheme works as follows: First, the code is generated to consume tokens from all input buffers of the process (Line 1). The algorithm then iterates through the set of modeled actions in the order of their definitions (Line 2) where for each action, it proceeds as follows: First, the code is generated to evaluate the guard (Line 3). Next, the code

is generated for the case if the guard is fulfilled (Lines 4-7). To this end, the code to perform the modeled computations is generated (Line 5), and then to produce the final output (Line 6).

Algorithm 3: Pseudo code for SDF kernel code generation of a process p.

```
1 ConsumeTokens(inBuf(p))          ▷ generate code to consume from all inputs
2 foreach action α in p do
3 |   EvaluateGuard(γ_α)           ▷ generate code to evaluate the guard of α
4 |   if GuardValid(γ_α) then                          ▷ generate code to fire α
5 |   |   PerformComputations(α)                  ▷ code to perform computations
6 |   |   ProduceTokens(outAct(α))             ▷ code to produce for all outputs
7 |   end
8 end
```

The generated kernel for a static process *ITE* as shown in Figure 4.2 is listed in Listing 5.1. The complete generated kernel code including all library functions are given in Appendix A.

OpenCL specific code segment. This segment (Lines 1-14) is mainly composed of the following parts: The generated kernel header uses the name of the process and defines the argument list (Line 1). The argument list mainly describes the OpenCL memory objects for the input and output FIFO buffers of the process. These memory objects are used by the host for data communication to and from the kernels (device side). These objects are defined with the *global* address space name that allocates them in the global memory shared between the host and devices. For better memory performance, the kernel instances (work-items) do not directly perform operations on the slower global memory. Instead, an array each is declared for the sequences of input/output tokens with the *private* address space name (Lines 4-11), which refers to a faster memory region only visible to individual instances. In each execution of a SDF process, only a particular action is executed, which depends on the enabled guard. To avoid unnecessary duplication, an array is declared for each input/output based on the statically determined consumption/production rate (Lines 4-7). Second, the individual input/output token variables are declared and pointed to their respective sequences i.e., arrays (Lines 8-11).

Moreover, the code generator generates generic kernel code (Lines 13-14) to enable the centralized host to dispatch multiple execution (instances) of kernels on the device at a time. The generic code involves two main components: First, the OpenCL function *get_global_id(0)* provides the unique global ID for the particular kernel instance or thread based on the number of instances dispatched by the host to execute the kernel. These dispatched instances are ideally executed in parallel on the device where each instance operates on data based on its own unique ID. We further introduced an additional parameter, namely *blockSize* (Lines 13-14) that allows us to manage the amount of workload associated with each instance. *blockSize* coalesces multiple instances in a single instance and executes them sequentially inside a loop (Line 14). Consequently, increasing *blockSize* implies increasing the workload per instance and

```
1 __kernel void ITE ( __global fifo_t* X1 , __global fifo_t* X2
      , __global fifo_t* X3, __global fifo_t* Y, int blockSize)
2 {
3   /*Generate Declarations for All Inputs and Outputs*/
4   __private int seq_X1[1];
5   __private int seq_X2[1];
6   __private int seq_X3[1];
7   __private int seq_Y[1];
8   int* x1_act1 = &seq_X1[0]; int* x1_act2 = &seq_X1[0];
9   int* x2_act1 = &seq_X2[0]; int* x2_act2 = &seq_X2[0];
10  int* x3_act1 = &seq_X3[0]; int* x3_act2 = &seq_X3[0];
11  int* y_act1 = &seq_Y[0]; int* y_act2 = &seq_Y[0];
12  /*Generate Generic Kernel Code*/
13  int gid = get_global_id(0) * blockSize;
14  for (int x = 0; x < blockSize; x++) {
15    /*Generate SDF Specific Kernel Code*/
16    fifoRead(X1, seq_X1, 1, gid, &cnt_X1);
17    fifoRead(X2, seq_X2, 1, gid, &cnt_X2);
18    fifoRead(X3, seq_X3, 1, gid, &cnt_X3);
19    if(*x1_act1 >= 0 ) {
20      *y_act1 = *x2_act1;
21      fifoWrite(Y, seq_Y, 1, gid, &cnt_Y);
22    }
23    else if(*x1_act2 < 0 ) {
24      *y_act2 = *x3_act2;
25      fifoWrite(Y, seq_Y, 1, gid, &cnt_Y);
26    }
27  }
28 }
```

Listing 5.1: *Generated kernel for ITE as illustrated in Figure 4.2.*

hence decreasing the total number of parallel instances. The combination of the unique global ID and *blockSize* can be effectively used by the host to fine tune the amount of data level parallelism according to the available resources, and most importantly, based on the kinds of behaviors of processes. For example, if the target device is a CPU offering only few cores, a larger *blockSize* usually achieves better performance [RaSc19]. Second, if a process exhibits a dynamic behavior, multiple instances can not be executed in parallel and therefore a *blockSize* equal to the number of instances can be used by the host.

SDF MoC specific code segment. This code segment (Lines 15-27) is strictly generated based on the scheme presented in Algorithm 3. First, the tokens are consumed from all inputs of the process (Line 16-18). Finally, a particular action (i.e., either $act1$ or $act2$) is executed based on the activation of guard. To this end, if the guard is true for $act1$ (Line 19), a token consumed from the input *X2* is written to the output Y (Lines 20-21). On the other hand, if the guard is true for $act2$ (Line 23), a token consumed from the input *X3* is written to the output Y (Lines 24-25).

KPN Code Generator

The KPN MoC supports static as well as sequential behaviors. Since, processes in KPNs are sequential, their firing rules (including guards) can be evaluated sequentially in a predefined order. In particular, a process is only triggered by the host for execution if it is already known that the guard of one of the actions is evaluated to true. In contrast to the SDF MoC, the guards are therefore evaluated at the time of scheduling on the host side. This essentially simplifies the kernel code generation, however, on the other hand, relatively complicates the scheduling. The code generation based on the underlying semantics of the KPN MoC is illustrated by the pseudo code given in Algorithm 4.

For a process in KPN, the proposed algorithm works as follows: It iterates through the set of modeled actions in the order of their definitions (Line 1) where for each action, it proceeds as follows: First, the code is generated that checks if the already evaluated guard is valid for the action (Line 2). Next, the code is generated for the case if the guard is valid (Lines 3-5). To this end, the code is generated to consume tokens from all inputs of the action (Line 3). Second, the generated code for the modeled computations is inserted (Line 4), prior to generating the code for writing the computed results on the outputs (Line 5). In contrast to the SDF MoC, where data is consumed from all inputs of the process in each execution, the KPN MoC only consumes data from the inputs of an enabled action.

Algorithm 4: Pseudo code for KPN kernel code generation of a process p.

```
1 foreach action α in p do
2     if EvaluatedGuardValid(α) then
3         ConsumeTokens(inAct(α))
4         PerformComputations(α)
5         ProduceTokens(outAct(α))
6     end
7 end
```

The generated kernel for a sequential process *split* as shown in Figure 4.3 is listed in Listing 5.2.

OpenCL specific code segment. As discussed, this code segment is largely same for all supported dataflow MoCs. In contrast to the SDF MoC, where the guards are evaluated within kernels at the device side, in the case of KPN, the guards are evaluated at the host side typically at the time of scheduling. The host provides the information regarding the evaluated guards to the kernel using a data structure *evaluatedGuard* through the argument list (Line 1). In particular, each element of *evaluatedGuard* holds the identifier for an action whose guard is valid for a particular instance. In each execution of a process, only a particular action is executed, which depends on the enabled guard. 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 $X2$ with the highest consumption rate of both actions

```
1 __kernel void split ( __global fifo_t* X1 , __global fifo_t*
      X2, __global fifo_t* Y1 , __global fifo_t* Y2, __global
      int* evaluatedGuard, int blockSize)
2 {
3   /*Generate Declarations for All Inputs and Outputs*/
4   __private int seq_X1[1];
5   __private int seq_X2[2];
6   __private int seq_Y1[1];
7   __private int seq_Y2[1];
8   int* x1_act1 = &seq_X1[0]; int* x1_act2 = &seq_X1[0];
9   int* x2_act1 = &seq_X2[0];
10  int* x2a_act2 = &seq_X2[0]; int* x2b_act2 = &seq_X2[1];
11  int* y1_act1 = &seq_Y1[0]; int* y1_act2 = &seq_Y1[0];
12  int* y2_act2 = &seq_Y2[0];
13  /*Generate Generic Kernel Code*/
14  int gid = get_global_id(0) * blockSize;
15  for (int x = 0; x < blockSize; x++) {
16    /*Generate KPN Specific Kernel Code*/
17    if(evaluatedGuard[x] == 0) {
18      fifoRead(X1, seq_X1, 1, gid, &cnt_X1);
19      fifoRead(X2, seq_X2, 1, gid, &cnt_X2);
20      *y1_act1 = *x2_act1;
21      fifoWrite(Y1, seq_Y1, 1, gid, &cnt_Y1);
22    }
23    else if(evaluatedGuard[x] == 1) {
24      fifoRead(X1, seq_X1, 1, gid, &cnt_X1);
25      fifoRead(X2, seq_X2, 2, gid, &cnt_X2);
26      *y1_act2 = *x2a_act2;
27      *y2_act2 = *x2b_act2;
28      fifoWrite(Y1, seq_Y1, 1, gid, &cnt_Y1);
29      fifoWrite(Y2, seq_Y2, 1, gid, &cnt_Y2);
30    }
31  }
32 }
```

Listing 5.2: *Generated kernel for split as illustrated in Figure 4.3.*

(Line 5). The remaining code of this segment is exactly the same as explained for the SDF MoC.

KPN MoC specific code segment. This code segment (Lines 16-31) is strictly generated based on the scheme presented in Algorithm 4. The generated code simply fires the enabled action whose guard is evaluated true at the time of scheduling. As discussed, *evaluatedGuard* provides the identifiers of actions whose guards are valid for particular iterations.

A particular action (i.e., either $act1$ or $act2$) is executed in each iteration based on the activated guard. To this end, if the guard is true for $act1$ (Line 17), a single token is consumed from $X1$ and a single token is consumed from $X2$ which is then written to the output $Y1$ (Lines 18-21). On the other hand, if the guard is true for $act2$, a token is consumed from $X1$, and two tokens are consumed from $X2$, where the first token of $X2$ is written to $Y1$, and the other to $Y2$ (Lines 24-29).

DDF Code Generator

The used variant of the DDF MoC supports sequential as well as parallel behaviors. In DDF, the decision on whether to consume tokens and to fire each action of a process is made dynamically at runtime when that particular process is triggered for execution. A process is simply scheduled by the host for execution if only one of its inputs has required tokens and only one of its outputs has required space. In contrast to SDF and KPN MoCs, the firing rules (including guards) of processes in DDF are completely evaluated within kernels at the device side. Each DDF kernel therefore must indicate to the host the number of tokens consumed/produced in each FIFO buffer for the dispatched execution instances. This fairly complicates the code generation of kernels. In particular, the code generator incorporates a number of DDF MoC specific library functions designed to enable the dynamic evaluation of actions within kernels. The code generation based on the underlying semantics of the used DDF MoC is illustrated by the pseudo code given in Algorithm 5.

Algorithm 5: Pseudo code for DDF kernel code generation of a process p.

```
1  PeekTokens(inBuf(p))                    ▷ code to peek from inputs of all actions
2  foreach action α in p do
3      if TokensAvailable(inGrd(α)) then
4          EvaluateGuard(γ_α)
5          if GuardValid(γ_α) && TokensAvailable(inAct(α)\inGrd(α)) &&
             SpaceAvailable(outAct(α)) then      ▷ code to check α is enabled
6              ConsumeTokens(inAct(α))
7              PerformComputations(α)
8              ProduceTokens(outAct(α))
9          end
10         else if GuardValid(γ_α) && !(TokensAvailable(inAct(α)\inGrd(α)) &&
             SpaceAvailable(outAct(α))) then
11             WriteBuffersStatus(inAct(α))    ▷ write status of tokens/space
12         end
13     end
14 end
```

First, the code is generated to peek tokens from all inputs for all actions of the process (Line 1). The algorithm then iterates through the modeled set of actions in the order of there definitions (Line 2). For each action, the algorithm proceeds as follows: First, the code is generated to check if enough tokens are available in the inputs used by the guard (Line 3). Second, code evaluating the guard is generated (Line 4). Next, the code is generated to fire an action (Lines 5-9). This involves code for checking if the guard is valid, the required number of tokens are available in all inputs and the required amount of space is available in all outputs (Line 5). The code is then generated for the enabled action (Lines 6-8) which involves code for consuming all inputs, performing modeled computations, and writing the computed results on the outputs. The code generator further generates code for the case if although the guard is true, however, either at least one of the inputs does not have enough tokens, or at

least one of the outputs does not have enough space (Lines 10-12). In this case, code is generated to ascertain the status of tokens and space in each input and output FIFO buffer of the action, respectively. The status of each buffer in this respect is written and conveyed to the host using a data structure (Line 11). This case is typically used to indicate which input buffers lack the required number of tokens and/or which output buffers lack the required space. This information can be utilized by the host to optimize the scheduling if required. In particular for sequential behaviors where only one of the actions fires in each execution, the scheduler can wait only for the required tokens/space in the inputs/outputs of the action whose guard was evaluated valid. This allows the host to avoid the overhead caused by repeatedly dispatching the kernels on the device unless tokens/space are available in the indicated buffers.
The generated kernel for a parallel process *POR* as shown in Figure 4.4 is listed in Listing 5.3. The complete generated code is given in Appendix A. For brevity, we illustrate here the generated code for the first two actions ($act1$ and $act2$).
OpenCL specific code segment. This code segment is exactly generated in the same way as generated in the case of the SDF MoC.
DDF MoC specific code segment. This code segment (Lines 13-49) is strictly generated based on the scheme presented in Algorithm 5. First, the tokens are peeked from all inputs for all actions of the process (Lines 14-17). The generated code then simply evaluates each action for firing. If there are enough tokens available in the inputs used for guard (Lines 21 and 35), the guard is evaluated (Lines 22 and 36). If the guard is valid and the required number of tokens and the required amount of space is available in all input and output FIFO buffers, respectively, the action is fired (Lines 26-30 and 40-44). Depending on the availability of tokens/space and the enabled guard, either one or both of the actions ($act1$ and $act2$) can be executed in each iteration. In particular, if both actions are enabled, a token each is consumed from both inputs *X1* and *X2* (Lines 27 and 41) and a token each is produced to the output *Y* (Lines 28-29 and 42-43) by both actions. In case if the guard is enabled for one of the actions (say $act1$), however, there is no space available in the output *Y*, the status of tokens and space in *X1* and *Y*, respectively, are determined and written using the DDF MoC specific library function *fifoWriteStatus* (Lines 32-33).

```
1  __kernel void POR ( __global fifo_t* X1 , __global fifo_t* X2,
       __global fifo_t* Y, int blockSize)
2  {
3    /*Generate Declarations for All Inputs and Outputs*/
4    __private bool seq_X1[1]; __private bool seq_X2[1];
5    __private bool seq_Y[1];
6    bool* x1_act1 = &seq_X1[0]; bool* x1_act3 = &seq_X1[0];
7    bool* x2_act2 = &seq_X2[0]; bool* x2_act3 = &seq_X2[0];
8    bool* y_act1 = &seq_Y[0]; bool* y_act2 = &seq_Y[0];
9    bool* y_act3 = &seq_Y[0];
10   /*Generate Generic Kernel Code*/
11   int gid = get_global_id(0) * blockSize;
12   for (int x = 0; x < blockSize; x++) {
13     /*Generate DDF Specific Kernel Code*/
14     bool ctrl_X1_act1 = fifoPeek(X1, seq_X1, 1, gid, &cnt_X1);
15     bool ctrl_X1_act3 = fifoPeek(X1, seq_X1, 1, gid, &cnt_X1);
16     bool ctrl_X2_act2 = fifoPeek(X2, seq_X2, 1, gid, &cnt_X2);
17     bool ctrl_X2_act3 = fifoPeek(X2, seq_X2, 1, gid, &cnt_X2);
18     bool ctrl_Y_act1 = fifoSpace(Y, 0, Y->tail+gid, &cnt_Y,1);
19     bool ctrl_Y_act2 = fifoSpace(Y, 0, Y->tail+gid, &cnt_Y,1);
20     bool ctrl_Y_act3 = fifoSpace(Y, 0, Y->tail+gid, &cnt_Y,1);
21     if(ctrl_X1_act1){
22       guard_act1 = (*x1_act1 == true );
23       eval_impl_act1 = evalImplication(guard_act1,
             ctrl_X1_act1 && ctrl_Y_act1);
24     }
25     else fifoWriteStatus(X1, 1, X1->head + gid, &cnt_X1, 1);
26     if(eval_impl_act1 == '1'){
27       fifoReadDDF(X1, 4, X1->head + gid, &cnt_X1, 1);
28       *y_act1 = true;
29       bytes = fifoWriteDDF(Y, seq_Y, 1, gid, &cnt_Y);
30     }
31     else if(eval_impl_act1 == '0'){
32       fifoWriteStatus(X1, 1, X1->head + gid, &cnt_X1, 1);
33       fifoWriteStatus(Y, 2, Y->tail + gid, &cnt_Y, 1);
34     }
35     if(ctrl_X2_act2){
36       guard_act2 = (*x2_act2 == true );
37       eval_impl_act2 = evalImplication(guard_act2,
             ctrl_X2_act2 && ctrl_Y_act2);
38     }
39     else fifoWriteStatus(X2, 1, X2->head + gid, &cnt_X2, 1);
40     if(eval_impl_act2 == '1'){
41        fifoReadDDF(X2, 4, X2->head + gid, &cnt_X2, 1);
42        *y_act2 = true;
43        bytes = fifoWriteDDF(Y, seq_Y, 1, gid, &cnt_Y);
44     }
45     else if(eval_impl_act2 == '0'){
46       fifoWriteStatus(X2, 1, X2->head + gid, &cnt_X2, 1);
47       fifoWriteStatus(Y, 2, Y->tail + gid, &cnt_Y, 1);
48     }
49   }
50 }
```

Listing 5.3: *Generated kernel for POR as illustrated in Figure 4.4.*

5.2. Runtime System

The runtime system systematically employs OpenCL in the composition of the synthesis components to finally map and execute models based on different dataflow MoCs on COTS target hardware. The runtime system is typically organized in a centralized host and kernels architecture, built under the OpenCL abstraction as shown in Figure 5.1. The host accommodates different essential components along with the *Runtime-Manager*. These components work together to implement low-level details such as the schedulers, the communication mechanism, resource allocation, kernels mapping and handling etc. This section covers all the runtime components and explains in detail the workflow of the composition of different components.

5.2.1. Process and Device Queues

The *Process-Queue* is generated for the host at the back-end. Each element of this queue provides a special object of a process. Each object provides the specific attributes of the process to the host. This includes: the process name, the identified dataflow MoC, the associated FIFO buffers, the type of each buffer, and the process type. The queue, once generated, is maintained and updated by the host. In particular, the host assigns each object the process's status (idle, running or blocked) and the associated kernel. Although the *Process-Queue* is generated at the back-end, however, it is a key component of the host and therefore we present the complete details in this section.

The back-end merely parses the network description file (*xdf*) and automatically generates the *Process-Queue*. As discussed and illustrated in Listing 2.2, *xdf* is mainly composed of two basic elements, namely *Instance* and *Connection* fields. Each *Instance* represents a process. The name of each process is described under the attribute *id* in *Instance*. Each *Connection* represents a FIFO buffer connecting two processes. The name of the destination process and the name of the source process of each FIFO buffer are described under the attributes *dst* and *src*, respectively, in *Connection*. *Instance* fields are further updated with the dataflow MoCs determined at the back-end. The underlying dataflow MoC of each process is described under the attribute *MoC* in *Instance*. The pseudo code for generating the *Process-Queue* using the network file *xdf* is given in Algorithm 6.

The algorithm constructs each object in the queue by extracting the required attributes of the process from *Instance* and *Connection* fields in *xdf*. First, the name and the identified dataflow MoC are assigned to each object by extracting the values of attributes *id* and *MoC*, respectively, from *Instance* fields (Lines 4-7). Second, the number of FIFO buffers used in each process is determined by calculating the number of *Connection* fields where a process appears as either a destination or a source (Lines 9-14). Each FIFO buffer in a process can either be an input or an output buffer. The type of each FIFO buffer in a process is determined by checking in *Connection* fields if the process is featured as a destination or a source. A connection where the process appears as a destination determines an input FIFO buffer of that

Algorithm 6: Pseudo code for *Process-Queue* generation

```
1  NumOfProcesses ← number of instances in xdf
2  ProcessQueue ← CreateProcessQueue(NumOfProcesses)
3  foreach process object p in ProcessQueue do
4  |   foreach instance i in xdf do          ▷ extract name and MoC of process
5  |   |   p.Name ← i.Attribute("id")
6  |   |   p.MoC ← i.Attribute("MoC")
7  |   end
8  |   NumOfFIFOs ← 0
9  |   foreach connection c in xdf do    ▷ extract number of FIFOs of process
10 |   |   if p.Name == c.Attribute("dst") || p.Name == c.Attribute("src") then
11 |   |   |   NumOfFIFOs ← NumOfFIFOs + 1
12 |   |   end
13 |   end
14 |   p.NumOfFIFOs ← NumOfFIFOs
15 end
16 foreach process object p in ProcessQueue do
17 |   Count ← 0
18 |   FlagProducer ← false
19 |   FlagConsumer ← false
20 |   while Count <p.NumOfFIFOs do   ▷ extract type of FIFO and process
21 |   |   foreach connection c in xdf do
22 |   |   |   if p.Name == c.Attribute("dst") then
23 |   |   |   |   p.FIFO[Count].Type ← "input"
24 |   |   |   |   FlagConsumer ← true
25 |   |   |   end
26 |   |   |   else if p.Name == c.Attribute("src") then
27 |   |   |   |   p.FIFO[Count].Type ← "output"
28 |   |   |   |   FlagProducer ← true
29 |   |   |   end
30 |   |   |   p.FIFO[Count].UniqueID ← connection count ▷ assign unique ID
31 |   |   |   Count ← Count + 1
32 |   |   end
33 |   end
34 |   if FlagProducer && FlagConsumer then
35 |   |   p.type ← "Worker"
36 |   end
37 |   else if FlagConsumer then
38 |   |   p.type ← "Consumer"
39 |   end
40 |   else if FlagProducer then
41 |   |   p.type ← "Producer"
42 |   end
43 |   FlagProducer ← false
44 |   FlagConsumer ← false
45 end
```

process (Lines 22-25). Similarly, a connection where the process appears as a source determines an output FIFO buffer (Lines 26-29). Each FIFO buffer in the network is assigned with a unique identifier provided by the number of the connection field. These unique identifiers are stored in the process objects to identify the associated FIFO buffers of each process (Line 30). Finally, the type of each process is determined by examining the types of buffers used in the process (Lines 34-42). If a process features both input and output buffers, it is recognized as a *Worker* process. In case if a process only involves input buffers, it is termed as a *Consumer*. On the contrary, it is termed as a *Producer* if only output buffers are used. The process type is effectively used by the schedulers to only look for the used types of buffers.

Apart from the *Process-Queue* that is provided by the back-end, the host also generates a queue, namely the *Device-Queue*, using the OpenCL specification as depicted in Figure 5.1. The *Device-Queue* lists all the available devices of the target hardware. Each element of this queue provides a special object of a device. The device object provides an interface that is used by the host to access and to use the device for systematically executing kernels. In particular, each object features a command queue of a device, where the processes (kernels) can be mapped for execution. Each command queue can represent a complete device (e.g., a CPU) or even a compute unit of that device (e.g., a CPU-core). Specifically for CPUs, OpenCL features the concept of *device fission*[1] that allows one to subdivide a device into one or more sub-devices. To this end, a sub-device can be created for each CPU core and for each sub-device a dedicated command queue can be assigned. In contrast to CPUs, OpenCL does not support GPUs for device fission. OpenCL only grants access up to the device level for creating command queues for GPUs. A command queue can therefore only be generated for the complete device for GPUs.

The device object also provides the information about the processes currently running on the device. Moreover, it also contains the OpenCL event object and the corresponding callback handler pointer associated with the command queue of the device. The event object and the handler pointer are typically used by the dispatcher and the handler to realize a smooth workflow of the complete system from host to device and back to host.

5.2.2. The Runtime-Manager

The heart of the runtime system around which the overall implementation circulates is the *Runtime-Manager*, as shown in Figure 5.1. The Runtime-Manager is a part of the host that uses the *Process-Queue* and the *Device-Queue*, and provides: the schedulers for scheduling processes based on the supported dataflow MoCs, the communication mechanism between the host and kernels, a dispatcher for mapping kernels to devices, and a handler mechanism for kernels using specialized callbacks.

[1] `https://www.khronos.org/registry/OpenCL/extensions/ext/`

Schedulers

The schedulers are designed to schedule processes in the network based on the underlying precise dataflow MoC of each process. To support the execution of heterogeneous dataflow models characterized by different kinds of behaviors, the *Runtime-Manager* provides a two-level hierarchical scheduling scheme. At the first level, a baseline global scheduler is used that works in a dynamic round robin scheme. At the second level, specialized local schedulers are used, where each local scheduler is designed for a particular dataflow MoC. Altogether, the baseline global scheduler iterates through the *Process-Queue* in a round-robin fashion, and invokes the corresponding local scheduler for each process based on its underlying dataflow MoC. All the local schedulers are designed based on the dynamic data driven scheduling schemes mainly because of the following reasons: The target DPNs involve heterogeneous dataflow models consisting of static as well as dynamic behaviors. A common consistent dynamic environment is therefore needed to systematically schedule and execute heterogeneous models. Second, as the proposed synthesis method targets parallel architectures like GPUs, the dynamic scheduling scheme may potentially benefit from the data level parallelism (DLP) by mapping multiple executions in parallel based on the availability of data at runtime. Nevertheless, for a fully SDF network only consisting of static processes, one can generate a static scheduler at compile time.

Algorithm 7: Pseudo code for the baseline global scheduler.

```
1  while true do
2      foreach process p in the network do
3          if ProcessStatusIdle(p) then
4              if DataflowMoC(p) == "SDF" then
5                  CallSDFScheduler(p)
6              end
7              else if DataflowMoC(p) == "KPN" then
8                  CallKPNScheduler(p)
9              end
10             else if DataflowMoC(p) == "DDF" then
11                 CallDDFScheduler(p)
12             end
13         end
14     end
15 end
```

Baseline Global Scheduler The *Runtime-Manager* employs a simple global scheduler typically designed to handle the invocation of the specialized local schedulers for scheduling heterogeneous dataflow models. The global scheduling scheme is depicted in Algorithm 7.

As DPNs do not generally enforce any termination criteria, the global scheduler runs forever (Line 1). It works in a round-robin fashion and selects the next process in the *Process-Queue* that is not currently running, in particular, has no pending dispatched executions (Lines 2-3). In case if a process

has a pending call i.e., all dispatched kernel instances are not completely executed, the next invocation of this process is delayed until the process's status is idle again. The scheduler simply invokes the local scheduler of the selected process according to the identified underlying dataflow MoC provided by the *Process-Queue*.

In the following, we present the specialized local schedulers based on the supported dataflow MoCs.

SDF Scheduler The scheduling scheme based on the SDF MoC is depicted in Algorithm 8. A process based on the SDF MoC always consumes and produces fixed number of tokens in each execution. This greatly simplifies the scheduling, in particular, a process is simply scheduled for execution if there is enough data available in all input buffers (Line 2) and if there is enough space available in all output buffers (Line 3).

Algorithm 8: Pseudo code for the SDF scheduling scheme.

```
1  for process p with SDF MoC in the network do
2      if TokensAvailable(inBuf(p)) then          ▷ in all input buffers of p
3          if SpaceAvailable(outBuf(p)) then      ▷ in all output buffers of p
4              ScheduleForExecution(p)
5          end
6      end
7  end
```

Algorithm 9: Pseudo code to determine the number of parallel instances using the SDF scheduling scheme.

```
1  for process p with SDF MoC in the network do
2      TokensAvailable ← MaxBufferSize
3      SpaceAvailable ← MaxBufferSize
4      foreach input FIFO inF in inBuf(p) do
5          TokensAvailable ← min(TokensAvailable,
             TokensAvailable(inF)/ConsumptionRate(inF))
6      end
7      foreach output FIFO outF in outBuf(p) do
8          SpaceAvailable ← min(SpaceAvailable,
             SpaceAvailable(outF)/ProductionRate(outF))
9      end
10     NumOfInstances = min(TokensAvailable, SpaceAvailable);
11 end
```

Moreover, the OpenCL abstraction offers the opportunity to exploit data level parallelism (DLP) by performing the same operation on different data in parallel. Depending on the availability of data tokens, multiple instances of a process (i.e., its kernel) can be dispatched and executed in parallel on the target device. However, this can be achieved only if each dispatched instance knows exactly where to read/write its input/output data. Hence, DLP is only possible for SDF processes exhibiting static behaviors. The SDF scheduling scheme therefore can target DLP by calculating the number of

parallel instances of a process based on the availability of data at runtime. The scheme used by the SDF scheduler to determine the number of instances is illustrated by the pseudo code given in Algorithm 9.

The algorithm first calculates the minimum available tokens across all inputs of a process by dividing the available number of tokens in each input FIFO buffer with the number of tokens consumed per instance on that buffer (Line 5). Second, it calculates the minimum available space across all outputs by dividing the available free space in each output FIFO buffer with the number of tokens produced per instance on that buffer (Line 8). Finally, the number of parallel instances is calculated as the minimum value of the minimum available tokens and the minimum available space (Line 10). The scheduled process is then forwarded to the dispatcher along with the calculated number of instances.

KPN Scheduler As discussed, the KPN MoC supports static as well as sequential behaviors. A process is only scheduled for execution if it is already known that the firing rules (including guard) of one of the actions are valid. The firing rules in a process are therefore evaluated at the time of scheduling. This relatively complicates the scheduling. The scheduling scheme based on the KPN MoC is illustrated by the pseudo code given in Algorithm 10.

Algorithm 10: Pseudo code for the KPN scheduling scheme.

```
1  for process p with KPN MoC in the network do
2      foreach action α in p do
3          if TokensAvailable(inGrd(α)) then    ▷ input buffers used by guard
4              EvaluateGuard(γ_α)
5              if GuardValid(γ_α)) then
6                  if TokensAvailable(inAct(α)\inGrd(α)) then
7                      if SpaceAvailable(outAct(α)) then
8                          ScheduleForExecution(p)
9                      end
10                     else
11                         BlockUntilSpaceAvailable(p)
12                         break foreach loop
13                     end
14                 end
15                 else
16                     BlockUntilTokensAvailable(p)      ▷ blocking read of KPN
17                     break foreach loop
18                 end
19             end
20         end
21     end
22 end
```

Since the KPN MoC supports processes having sequential behaviors, it schedules a process for execution by evaluating the firing rules (including guards) sequentially in a predefined order. The KPN scheduler iterates through the set of modeled actions in the order of their definitions where

for each action, it works as follows: First, the scheduler checks if enough tokens are available in the input buffers used by the guard (Line 3). If enough tokens are available and if the guard is valid (Line 5), the remaining (non-guarded) input buffers of the action are checked for enough tokens (Line 6). Only if there are enough tokens available, the output buffers are checked for space (Line 7). If enough space is available, the action is finally scheduled for execution (Line 8). In case if one of the conditions does not meet, the process is blocked until that condition is fulfilled (Lines 11 and 16).

Since the host and generated kernels are independent components, the evaluation sequence or order needs to be extracted from modeled processes at compile time. The extracted sequence can be used by the KPN scheduler at runtime to schedule processes for execution. To this end, we propose a systematic way of extracting the evaluation sequence by introducing the *input-output tree wrapper* (IOT-wrapper).

Introducing IOT-wrapper: The IOT-wrapper wraps the exact information of inputs/outputs required to schedule a process in a standard tree structure, while taking into account the underlying semantics of the KPN MoC. For each process, a wrapper is generated at compile time from the modeled behavior. The IOT-wrapper generation based on the underlying KPN semantics is illustrated by the pseudo code given in Algorithm 11. For each process with KPN MoC, 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 2-16). The *StepFunction* of the root node is generated with the code specifically related to the guards (Lines 4-16) 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 has enough tokens (Line 7), and second to evaluate the guard (Line 8). For each action, if the guard is true (Line 9), the *StepFunction* returns the action number $num \in \mathbb{Z}$ (Line 10) that corresponds to a specific branch in the tree originating from the root node. The algorithm then iterates through the modeled set of actions in the order of there definitions (Line 18). For each action, the algorithm first adds nodes for all the remaining (non-guarded) inputs (Lines 20-38). For each non-guarded input, 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 22), which is incremented by one for each action (Line 58). 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 34). Second, for a non-guarded input node, the *StepFunction* is generated with the code to check if that input has enough tokens (Lines 23-30). The algorithm then adds nodes for all the outputs (Lines 39-57). In contrast to an input node, the *StepFunction* of an output node is generated with the code to check if that output has enough space (Line 42-49).

The IOT-wrapper generated for a sequential process *split* as illustrated in Figure 4.3, is shown in Figure 5.2. The root node only involves the input

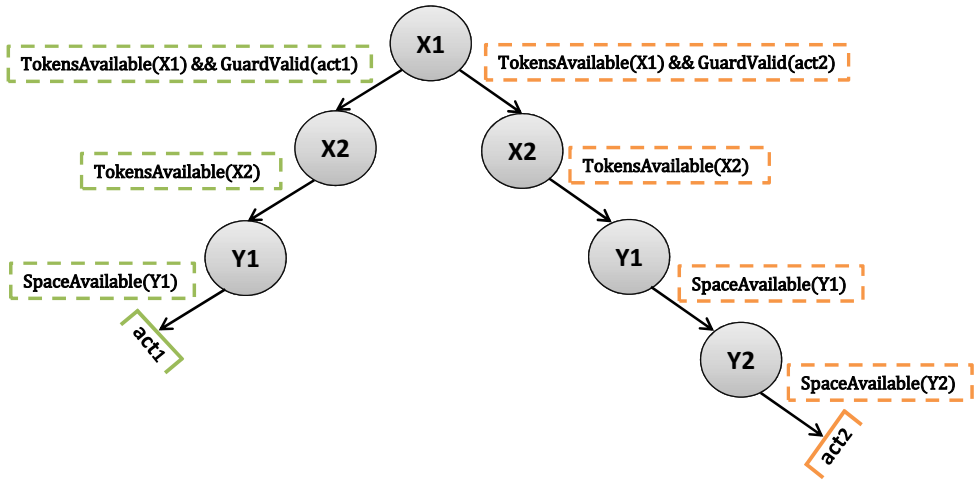

Figure 5.2.: *Generated IOT-wrapper for split process as shown in Figure 4.3: it consists of two branches where each branch corresponds to a particular action i.e., act1 or act2 of the process. The step functions of all nodes are illustrated in dashed boxes.*

$X1$ as it is the only input used for guard by the process. The *StepFunction* generated and assigned to each node is shown 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, $act2$ is represented by branches originating from the root node ($X1$) and extending up to the leaf node ($Y2$).

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 the depth-first search (DFS) method [Tarj72] 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 used to select a specific branch originating from the root node that directs to a specific action whose 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*.

DDF Scheduler The DDF MoC evaluates the firing rules (including guard) of each action in a process as a part of the kernel at the device. This greatly simplifies the scheduling at the host and in fact the scheduler based on the DDF MoC is the simplest of all supported dataflow MoCs. The scheduling scheme based on the DDF MoC is illustrated by the pseudo code given in Algorithm 12. It follows an optimistic scheduling strategy that expects the firing of actions even if there is data available in only one input buffer and if there is space available in only one output buffer of the process. A process is

Algorithm 11: Pseudo code for generating IOT-wrappers for all KPN processes in the network.

```
1  foreach process p with KPN MoC do
2      Initialize IOT-wrapper{
3      add root (guard) node
4      for root node, generate StepFunction{
5      num ← 0
6      foreach action α in p do
7          if TokensAvailable(inGrd(α)) then
8              EvaluateGuard(γ_α)
9              if GuardValid(γ_α) then
10                 "return num;"
11             end
12         end
13         num ← num + 1
14     end
15     "return -1;"                    ▷ in the case if none of the actions is enabled
16     }}
17     num ← 0
18     foreach action α in p do
19         current node = root node
20         foreach input FIFO inF in inAct(α)\inGrd(α) do   ▷ all non-guarded inputs
21             if current node == root node then
22                 new node = add child node to current node at branch num
23                 for new node, generate StepFunction{
24                 if TokensAvailable(inF) then
25                     "return true;"
26                 end
27                 else
28                     "return false;"
29                 end
30                 }
31                 current node = new node
32             end
33             else
34                 new node = add child node to current node at branch 0
35                 for new node, generate StepFunction{ same as Lines 24-29 }
36                 current node = new node
37             end
38         end
39         foreach output FIFO outF in outAct(α) do              ▷ for all outputs
40             if current node == root node then
41                 new node = add child node to current node at branch num
42                 for new node, generate StepFunction{
43                 if SpaceAvailable(outF) then
44                     "return true;"
45                 end
46                 else
47                     "return false;"
48                 end
49                 }
50                 current node = new node
51             end
52             else
53                 new node = add child node to current node at branch 0
54                 for new node, generate StepFunction{ same as Lines 43-48 }
55                 current node = new node
56             end
57         end
58         num ← num + 1
59     end
60 end
```

Algorithm 12: Pseudo code for the DDF scheduling scheme.

```
1 for process p with DDF MoC in the network do
2     if TokensAvailableAnyInputFIFO(p) then
3         if SpaceAvailableAnyOutputFIFO(p) then
4             ScheduleForExecution(p);
5         end
6     end
7 end
```

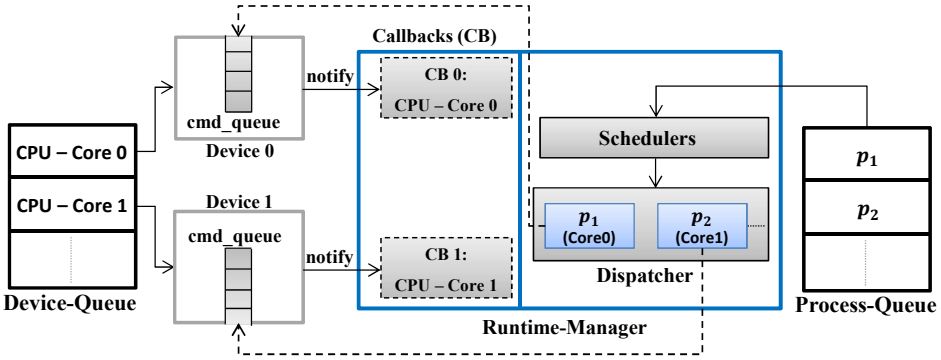

Figure 5.3.: *The dispatching and handling mechanism of the Runtime-Manager. The dispatcher provided with the scheduled process maps the associated kernel on the command queue of the target device. The handler is mainly responsible for updating the runtime components at the host after the dispatched executions are performed.*

therefore simply scheduled by the host for execution if there is enough data available in at least one of the input buffers (Line 2) and if there is enough space available in at least one of the output buffers (Line 3).

With that, we illustrated all the specialized schedulers based on the supported dataflow MoCs. In the following, we present two of the main components of the *Runtime-Manager*, namely the *dispatcher* and the *handler*.

Dispatcher

In general, a dispatcher is a special program which comes into play after the scheduler. When the scheduler completes its job of selecting a process, it is the dispatcher that gives a process control over the target device. The runtime system of the proposed framework also provides a special dispatcher built under the OpenCL abstraction. The global scheduler evokes one of the local schedulers based on the dataflow MoC of a process. The evoked scheduler fetches a ready process from the *Process-Queue* and provides it to the dispatcher as depicted in Figure 5.3. The dispatcher acquires the device object from the *Device-Queue* and finally maps the fetched process on the command queue of the target device. The generated kernel of the dispatched process is then executed based on the used dataflow MoC. Moreover, the dispatcher also receives the number of instances of the process from the schedulers. The dispatcher

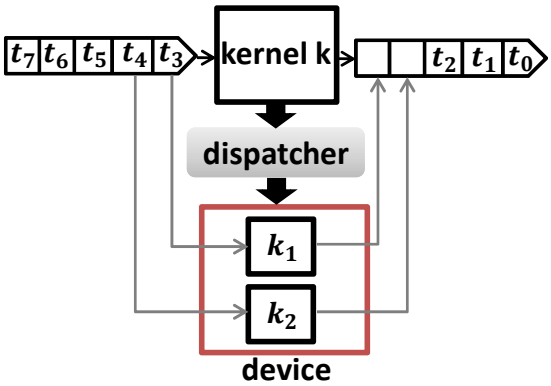

Figure 5.4.: *Exploiting data level parallelism in SDF. A simple example of data level parallelism exploited in a kernel based on the SDF scheme presented in Algorithm 9 is illustrated in this figure.*

therefore maps multiple instances of the corresponding OpenCL kernel on the device that depending on the underlying dataflow MoC either execute sequentially or in parallel. In particular for the SDF processes, multiple instances (work-items) of the corresponding kernels can be mapped in parallel by the dispatcher as illustrated in Algorithm 9. To exemplify, a simple example is illustrated in Figure 5.4 that considers a kernel of the SDF process having a single input buffer and a single output buffer. The consumption rate of the input buffer as well as the production rate of the output buffer is considered to be equal to one. A total of five tokens are available on the input buffer and a space for two tokens is available on the output buffer. The dispatcher maps two instances of the kernel on the target device based on the SDF scheme presented in Algorithm 9.

For the dynamic dataflow MoCs, however, the dispatcher maps sequential executions of the kernels on the target device that are always executed sequentially.

Communication Mechanism and Handlers

The proposed framework implements the FIFO buffers as bounded circular ring buffers as shown in Figure 5.5. In general, this design enables the buffers to work as if the memory is contiguous and circular in nature. As data is produced and consumed, it does not need to be restructured, instead, the head/-tail pointers are updated. When data is produced, the tail pointer increments, whereas, when data is consumed, the head pointer advances. Reaching the end of the buffer, the pointers simply turn around to the start. The employed FIFO structure serves two purposes, namely data transfer and, implicitly, storing the status of tokens and space in buffers at runtime. For each data element in the buffer, a corresponding control bit is provided that determines the presence (denoted by '1') or the absence (denoted by '0') of a data value. This design provides a convenient structure for dynamically communicating the status of buffers between a kernel and the host.

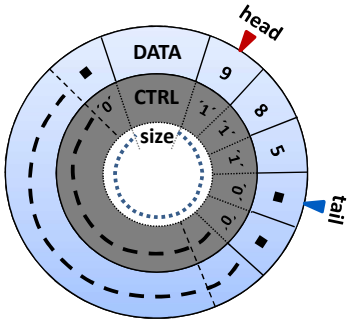

Figure 5.5.: *The FIFO buffers are structured as bounded circular ring buffers. The structure supports the transfer of data as well as the communication of the status of buffers between the host and device.*

The communication between the host and kernels is realized using OpenCL memory objects (buffers). For each bounded FIFO buffer, an OpenCL buffer is created with the same design and size of the FIFO buffer as shown in Figure 5.5. Each process object provided by the *Process-Queue* stores and links the address of each FIFO buffer with the associated OpenCL buffer. During the execution of a process, i.e., when the kernel of a process is being executed, data is read/written from/to the associated OpenCL buffers. When all the instances of the kernel are executed, i.e., the dispatched process executions are completed, the *Runtime-Manager* is then automatically notified to update the components. For that purpose, a handler mechanism is developed using callbacks as shown on the left part of Figure 5.3. The *Runtime-Manager* generates a callback interface each, for every existing device in the *Device-Queue* during the initialization of the queue. Based on the used dataflow MoC, the *Runtime-Manager* provides the MoC specific implementations for each generated callback interface. As a result, the dataflow MoC specific handler mechanism is invoked. The dispatcher sets up a callback event for each fetched process and links it with the callback handler of the device where it is dispatched. Hence, the completion of the kernel of the dispatched process automatically invokes the callback handler of the used device. The callback handler performs a set of general tasks including: retrieving data from the kernel (OpenCL buffers), updating all the FIFO buffers of the process, updating the status of the process, updating the *Process-Queue*, updating the device's load, and finally updating the OpenCL buffers. However, updating the FIFO buffers is a dataflow MoC specific task, and is therefore managed differently for the supported dataflow MoCs. Based on the SDF MoC, the data rate of a process remains fixed in each execution, and therefore each FIFO buffer is simply updated based on the specified static data rate. Based on the KPN MoC, since the processes are only scheduled if there exists one enabled action, each FIFO buffer is simply updated based on the enabled actions of dispatched instances. On the contrary, based on the DPN MoC, the processes are evaluated for their firing rules within kernels. Therefore, the amount of data consumed and produced is first measured at the host (handler) using

the status of buffers, and finally each FIFO buffer of the process is updated accordingly. The presented workflow of the *Runtime-Manager* is summarized in Figure 5.6.

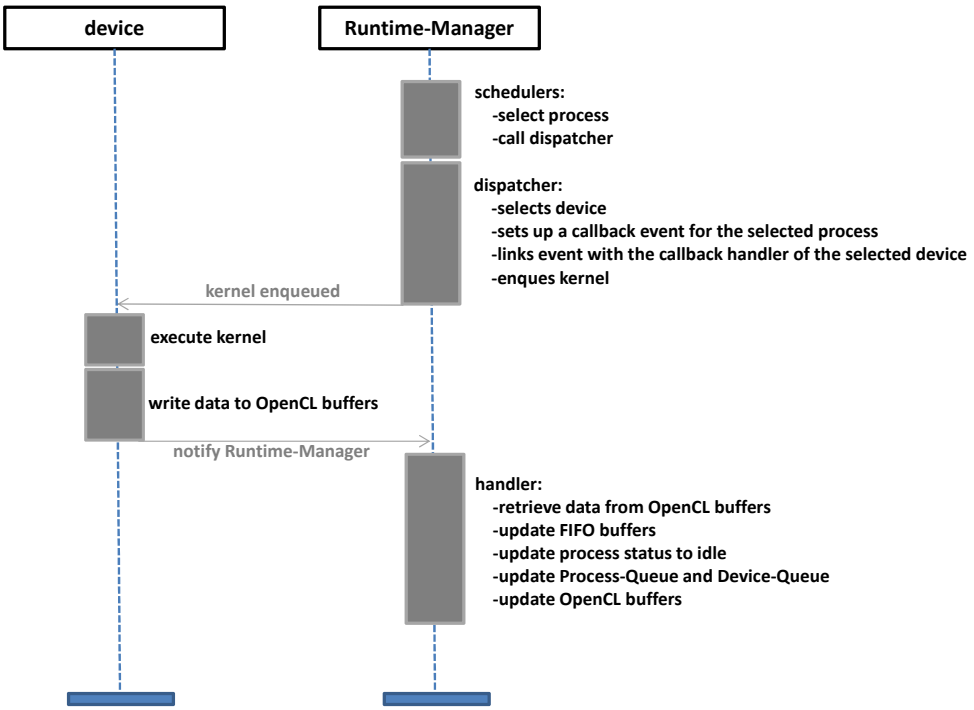

Figure 5.6.: *The complete workflow starting from the Runtime-Manager to the device and back to the Runtime-Manager is illustrated in this figure. The tasks of the workflow are demonstrated from top to bottom.*

Chapter 6

Evaluation

Contents

6.1. Overview

The proposed model-based synthesis design flow has been successfully used as a part of our long term project *Averest* [ScBr17] as illustrated in Figure 1.2. Furthermore, it has been implemented in a standalone framework. The former models the applications based on the synchronous reactive (SR) model of computation (MoC) and essentially transforms them to dataflow process networks (DPNs) which are then directly used by the synthesis framework to automatically generate implementations [BaRS21]. In contrast, the latter directly models the applications using the proposed models of the supported dataflow MoCs and automatically synthesizes them to corresponding implementations. In either case, the synthesis tool chain generally generates implementations based on the underlying precise dataflow MoC of each process in the network

and deploys them on any open specification language (OpenCL) abstracted target hardware.

We therefore organized our experimental evaluations into two parts. In the first part, the applications are modeled, synthesized and evaluated directly by using the proposed synthesis framework. This part features two types of applications. The first type involves simple standalone benchmarks that perform simple operations and are especially designed to evaluate and compare the homogeneous versions generated by all individual supported dataflow MoCs for each benchmark (when possible). The second type presents a particular case study of the ConceptCar [RaGS13; RaGS13c] where different configurations of the ConceptCar's architecture are used to model DPNs and automatically generate implementations based on the individual dataflow MoCs as well as based on their heterogeneous combinations.

In the second part, a particular case study of a building automation system (BAS) is presented. A number of different versions of BAS with different configurations are first modeled based on the SR MoC and then desynchronized to DPNs using *Averest*. The synthesis framework automatically generated implementations from these desynchronized models (DPNs) either based on a particular dataflow MoC or precisely based on the kinds of behaviors of the processes in the network.

6.2. Experimental Setup

A variety of OpenCL supported devices have been employed for evaluation as listed in Figure 6.1. The list involves five devices featuring three different device types from three different vendors. In particular, two different CPUs (**CPU1** and **CPU2**), one integrated GPU (**GPU1**) and two dedicated GPUs (**GPU2** and **GPU3**) featuring *Intel*, *AMD* and *NVIDIA* have been employed. The integrated GPU (**GPU1**) is built into the processor, and uses the system memory that is shared with the CPU (**CPU2**). In contrast, the dedicated GPUs (**GPU2** and **GPU3**) from *AMD* and *NVIDIA* feature their own processors and their own source of memory. Different versions generated for different benchmarks and case studies have been executed and evaluated on different OpenCL supported target devices.

For different target devices, different vendor specific OpenCL implementations have been installed. This involves the software development kits (SDKs) from device vendors and the appropriate device drivers supporting OpenCL runtimes. The complete software environment used for executing generated versions on the target devices is summarized in Figure 6.2.

6.3. Standalone Benchmarks

We designed a set of simple benchmarks consisting of processes having static, sequential or parallel functions. These benchmarks are therefore typically designed to offer a variety of processes having different kinds of behaviors that enable the evaluation and comparison of implementations based on all three

CPU1: Intel i5-8640U	
# physical cores	4
# logical cores	4
cache	6 MB Intel® Smart Cache
base frequency	3.20 GHz
RAM	8 GB
CPU2: Intel i7-8650U	
# physical cores	4
# logical cores	8
cache	8 MB Intel® Smart Cache
base frequency	1.90 GHz
RAM	8 GB
GPU1: Intel UHD Graphics 620	
# shaders/cuda cores	192
# compute units	24
cache	-
base clock	300 MHz
RAM	-
GPU2: AMD Radeon HD 5450 GPU	
# shaders/cuda cores	80
# compute units	2
cache	L1/L2: 8 KB/128 KB
base clock	650 MHz
RAM	512 MB GDDR2
GPU3: NVIDIA GeForce GTX 1050	
# shaders/cuda cores	640
# compute units	5
cache	L1/L2: 48 KB/1024 KB
base clock	1354 MHz
RAM	2 GB GDDR5

Figure 6.1.: *The experimental setup: list of target devices employed to evaluate the proposed synthesis design flow. The specification of each target device is shown in the figure.*

Device	Software Toolkit	Drivers
CPU1	Intel OpenCL SDK Version 6.3.0.1904	Intel processor driver version 10.0.1.9041.546
CPU2	Intel OpenCL SDK Version 6.3.0.1904	Intel processor driver version 10.0.1.9041.546
GPU1	Intel OpenCL SDK Version 6.3.0.1904	Intel graphics driver version 26.20.100.7639
GPU2	AMD OpenCL SDK version 3.0.130.135	AMD Radeon 5450 Driver Version 15.201.1151.1008
GPU3	CUDA Toolkit 10.1.243_426.00	NVIDIA Graphics Driver 451.67
Operating System		
Windows 10 Pro Version 1903 Build 18362.720		

Figure 6.2.: *The software environment: list of software toolkits and drivers installed to enable the OpenCL implementations.*

		Generated Network		
Benchmark	**Function Type**	**SDF**	**KPN**	**DDF**
SeqDySwitch	sequential	✗	✓	✓
SeqDyWorker	sequential	✗	✓	✓
SeqDySelect	sequential	✗	✓	✓
StITE	static	✓	✓	✓
StOR	static	✓	✓	✓
SeqDyMerge	sequential	✗	✓	✓
SeqDySplit	sequential	✗	✓	✓
ParDyOR	parallel	✗	✗	✓

Figure 6.3.: *The designed standalone benchmarks. Each benchmark offers processes based on a particular kind of behavior as depicted by the function type. The right hand side indicates the supported dataflow MoCs that were able to generate implementations for the particular benchmarks.*

different dataflow MoCs of the framework. Each benchmark only features processes based on a particular dataflow MoC. 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 provides the main functionality of the benchmark and therefore performs the main operation, e.g., the POR node illustrated in Figure 4.4. The designed standalone benchmarks along with their function types are listed in Figure 6.3. A brief description of each benchmark is given as follows:

The sequential dynamic switch (*SeqDySwitch*) benchmark is designed to switch the only data input channel to any one of a number of individual output channels by the application of a control input. In contrast, the sequential dynamic worker (*SeqDyWorker*) benchmark performs the operation by taking one single input channel and copying its data to the only output channel based on the value of data of the only input channel. The sequential dynamic select (*SeqDySelect*) benchmark is a multiplexer that processes the information from multiple input channels into a single output channel by the application of a control input. It can simply be understood as 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 static if-then-else (*StITE*) benchmark is a static version of the if-then-else operation that always consumes data from all input channels in each execution. The sequential

dynamic merge (*SeqDyMerge*) benchmark is designed to merge several input channels to a single common output channel by the application of a control input. In contrast, the sequential dynamic split (*SeqDySplit*) benchmark is designed to split a single input channel to a number of individual output channels by the application of a control input. The parallel dynamic OR (*ParDyOR*) benchmark performs the logical OR operation on two Boolean input channels and produces the result on the only output channel. It is a parallel version of the logical OR operation that can consume and produce tokens in parallel based on the availability of data on each input channel. In contrast, the static OR (*StOR*) benchmark is a synchronous version of the logical OR operation that always consumes tokens from both inputs in each execution. Apart from *ParDyOR* that incorporates the parallel function, all other benchmarks employ either static or sequential functions. In particular, apart from *StITE* and *StOR* which involve only static processes, all sequential benchmarks exhibit dynamic behaviors.

Each benchmark is modeled and automatically synthesized (when possible) based on all three supported dataflow MoCs of the framework. Thereby, three different implementations are automatically generated, namely the SDF MoC version, the KPN MoC version and the DDF MoC version. Since the SDF MoC only supports static behaviors, it could model and synthesize the *StITE* and *StOR* benchmarks. The KPN MoC supports both static and sequential functions and therefore able to model and synthesize all static and sequential benchmarks. However, it could not model and synthesize the only benchmark with the parallel function, namely the *ParDyOR* benchmark. The DDF MoC being the most generalized dataflow MoC of the lot supports static, sequential as well as parallel functions and therefore generated implementations for all designed benchmarks. The information regarding the implementations generated by the supported dataflow MoCs for the designed benchmarks is depicted in Figure 6.3.

The generated versions for each benchmark based on different dataflow MoCs are evaluated based on their code size and the end-to-end performance. 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. 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 ten thousand samples per input and the average of 50 repetitions is taken for each version.

6.3.1. Evaluation: The Generated Code Size

The modeled behaviors of particular benchmarks (mainly worker nodes) are given in Appendix A.1. The SDF MoC only supports static behaviors and therefore triggers a process when the data/space is available for all inputs/outputs. In each execution it consumes and produces statically determined fixed number of tokens from all inputs and outputs, respectively. This simplifies the code generation and in particular generates very succinct kernel

	Lines of Generated Code		
Benchmark	SDF	KPN	DDF
SeqDySwitch	-	84	147
SeqDyWorker	-	63	123
SeqDySelect	-	79	140
StITE	81	83	148
StOR	70	72	133
SeqDyMerge	-	83	146
SeqDySplit	-	88	153
ParDyOR	-	-	159

Figure 6.4.: *The generated code size for standalone benchmarks based on all three supported dataflow MoCs.*

code for static processes. The KPN MoC supports static as well as sequential behaviors. Since, processes in KPNs are sequential, their firing rules can be evaluated in a predefined order. It only triggers a process for execution when the exact information on inputs/outputs required to fire an action is available. This essentially simplifies the kernel code generation and therefore generates succinct kernel code for sequential processes. In contrast, the DDF MoC supports sequential as well parallel functions, and therefore 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. The KPN MoC therefore generates more concise kernel code for sequential processes than the DDF MoC. The generated code size of each benchmark for the complete network based on all three dataflow MoCs is depicted in Figure 6.4.

The additional kernel code overhead associated with the DDF MoC can therefore be observed from the number of lines of the generated code for each benchmark. For static benchmarks *StITE* and *StOR* that consist of only static processes, the SDF MoC generated the most succinct code of all generated versions. In particular, the generated code size based on the SDF MoC for *StITE* is about 83% and 3% lesser than the DDF and KPN versions, respectively. Similarly for *StOR*, the SDF version demonstrated a code reduction of about 90% and 3% in comparison to the DDF and KPN versions, respectively.

For all benchmarks consisting of sequential processes, the KPN MoC generated the most succinct code of all generated versions. For all sequential benchmarks, the KPN MoC generated at least 74% less lines of code than the DDF MoC. In particular, the biggest difference is recorded in *SeqDyWorker*

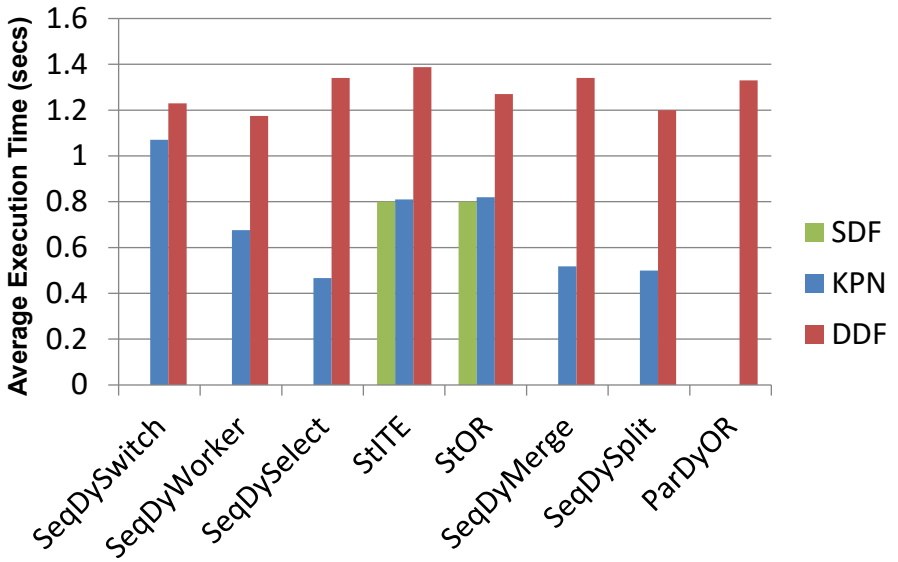

Figure 6.5.: *End-to-end performance on **GPU2** for the generated versions of the standalone benchmarks. The results depicted in the figure are measured for 10K samples per input.*

where the generated code size based on the KPN MoC is 95% lesser than the DDF version. The *ParDyOR* benchmark features a parallel function and could only be modeled and synthesized based on the DDF MoC. The DDF MoC therefore offers a more generalized dataflow MoC that supports both sequential as well as parallel behaviors but at the cost of the additionally generated lines of kernel code.

6.3.2. Evaluation: The End-to-end Performance

Each generated version of a benchmark is either executed on **CPU1** (Intel) or **GPU2** (AMD) at a time to evaluate and compare the end-to-end performance of all used dataflow MoCs. On each target hardware, i.e., **CPU1** and **GPU2**, the average execution time of each generated version of a benchmark is measured against the number of data samples. The complete results for all benchmarks based on different number of input samples are given in Appendix A.2. For brevity, in this section, we only show the results for the maximum number of samples (i.e., ten thousand samples per input) where the biggest differences in execution times have been recorded as shown in Figure 6.5 and Figure 6.6. All input samples are not processed at once, instead, depending on the fixed buffer sizes, they are processed in different cycles of execution.

Results: GPU2

The end-to-end performance of all generated versions of benchmarks for ten thousand input samples on **GPU2** is shown in Figure 6.5. As discussed for

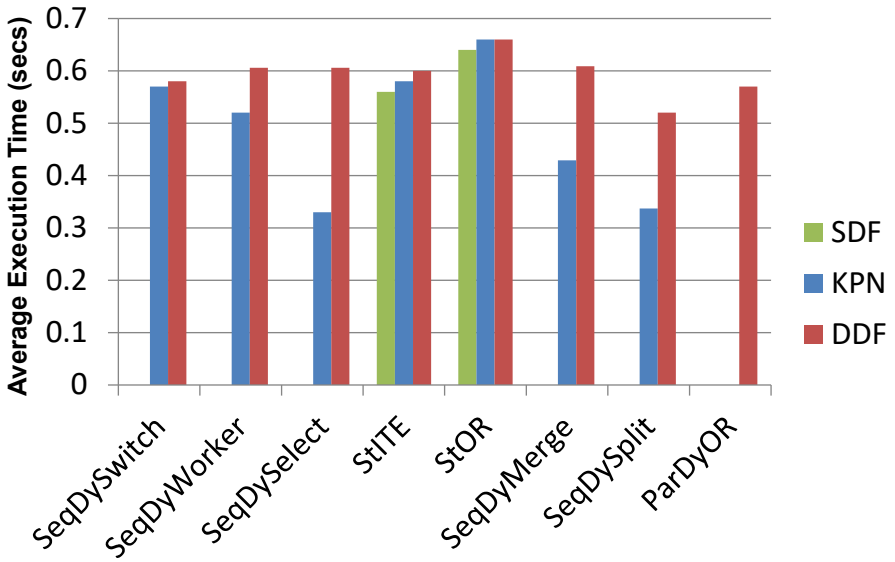

Figure 6.6.: *End-to-end performance on **CPU1** for the generated versions of the standalone benchmarks. The results depicted in the figure are measured for 10K samples per input.*

static benchmarks, the SDF MoC generated the most succinct kernel code. Considering the fact that the designed benchmarks only involve simple operations, the SDF MoC demonstrated the best end-to-end performance for *StITE* and *StOR*. In particular, the SDF version of *StITE* executed 1.74× faster than the DDF version and performed only slightly better than the KPN version. Similarly for *StOR*, the SDF version executed 1.59× faster than the DDF version and executed about 3% faster than the KPN version. Similarly, for all sequential benchmarks, the additional runtime overhead associated with the DDF MoC is propagated to the total execution time of the network resulting in elevated execution times. As a result, the KPN versions performed substantially better than the DDF versions. For all sequential benchmarks, the KPN versions executed at least 1.15× faster than the DDF versions. In particular, the biggest difference is observed in the case of *SeqDySelect* where the KPN version executed 2.87× faster than the DDF version.

Results: CPU1

In comparison to **GPU2**, the average execution time of each benchmark version is substantially reduced on **CPU1** as shown in Figure 6.6. In contrast to the OpenCL CPU where the host and the kernels reside on the same device, in the case of GPU, the data has to be transferred to the GPU and back to the main memory (host). This overhead therefore contributes in elevating the total execution time. On average, the generated versions on **CPU1** executed 1.75× faster than on **GPU1**. Similar to **GPU2**, the same trend of end-to-end performance has been observed on **CPU1**. The SDF MoC demonstrated the best end-to-end performance for static benchmarks *StITE* and *StOR*. The SDF ver-

sion of *StITE* executed 7% and 3.5% faster than the DDF and KPN versions, respectively. Similarly for *StOR*, the SDF version performed slightly better than the DDF and KPN versions. For all sequential benchmarks, the KPN versions performed substantially better than the DDF versions. To this end, the KPN versions executed on average 1.4× faster than the DDF versions. In particular, the biggest difference is observed in the case of *SeqDySelect* where the KPN version executed 1.83× faster than the DDF version.

Results: Summary

The SDF MoC generated the most succinct kernel code and demonstrated the best end-to-end performance for simple static benchmarks. The KPN MoC performed significantly faster than the DDF MoC for all sequential benchmarks. The DDF MoC offers the most expressive semantics of all supported dataflow MoCs and therefore was able to generate implementations for all designed benchmarks. The DDF MoC enables one to model static, sequential as well as parallel behaviors but at the cost of additional runtime overhead. Thus, even for the simplest of the benchmarks, we observed that generating implementations based on the kind of behavior or the underlying precise dataflow MoC of each process results in substantially improved end-to-end performance. In other words, using a more generalized dataflow MoC for scheduling and executing rather restricted dataflow behaviors could result in inefficient system implementations.

6.4. Case Study I: The ConceptCar's Dataflow Emulation

The *ConceptCar* [RaGS13; RaGS13c] (designed and developed by our group) is an experimental embedded system with the objective of testing and verifying car features by deploying different classes of applications. The ConceptCar, although not as big as a conventional car, has been built and engineered as close to a modern car as possible.

Hardware Design

The ConceptCar is a research platform remotely operated via a standard 2-channel (throttle and steering) 27MHz radio transmitter system. It incorporates a set of sensors (wheel speed, gyro/accelerometer, distance etc.) for interacting with the environment and surroundings. It uses an air-cooled sensorless brushless electrical motor for driving, and a servo motor for steering. The power train of the ConceptCar features two independent power sources: one for the heavy load electric system (motors/actuators), and another one for powering up all the electronic control units (ECUs).

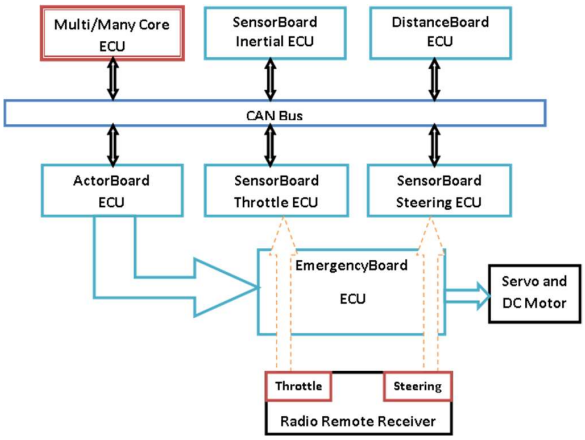

Figure 6.7.: *Architecture of the ConceptCar.*

Computational Architecture

Although not incorporated with as many ECUs as a modern car can carry, the ConceptCar still features 7 different ECUs, as shown in Figure 6.7. These ECUs are organized in three processing units. The *SensorBoard* ECUs, as incorporated with different sensors, form the input processing unit which is responsible for interacting with the environment. The *multicore* ECU also known as *DataBoard* is used as a data processing unit and only comes into play when complex mathematical computations are required. The *ActorBoard* ECU forms the output processing unit and is responsible for creating the PWM signals to drive the actuators (dc motor and servo). The selector switch on *ActorBoard* chooses the source of data, either receiving processed data from *DataBoard* or normalized data from *SensorBoards*. Similar to a modern car, all ECUs interact with each other via a centralized CAN bus architecture. Since the powertrain of the ConceptCar features two independent power sources, a special ECU called *EmergencyBoard* is integrated which separates the actuators from the other ECUs by galvanic isolation. *EmergencyBoard* therefore isolates functional sections of electrical systems to prevent current flow. This ECU only accepts the input from the radio receiver and *ActorBoard*, and bypasses it to *SensorBoards* and the actuators, respectively.

Dataflow Emulation

A dataflow emulation of the ConceptCar is devised where the operations of all the ECUs are emulated in a network of processes. The computations performed by each ECU are therefore modeled in a DPN process. This dataflow emulation allows us to produce two different test cases: The first test case as shown in Figure 6.8 emulates the initial design of the ConceptCar without galvanic isolation i.e., the actuators are directly fed by *ActorBoard*. The second test case as shown in Figure 6.13 considers the design with galvanic isolation

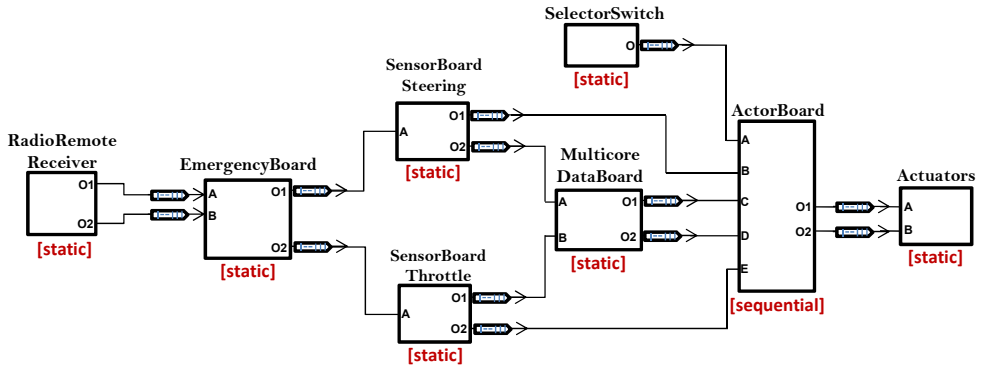

Figure 6.8.: *The original dataflow network of the ConceptCar based on the open-loop configuration. As shown in the figure, the actuators are directly fed by ActorBoard. The behavior type of each process is described at the bottom of each node in square brackets.*

provided by *EmergencyBoard*. For both test cases, the input data provided by the process *RadioRemoteReceiver* is collected from the centralized CAN bus and the results are validated against the logged outputs of the ConceptCar.

A number of implementations (versions) are automatically generated by the synthesis framework for both test cases where each test case is focused to evaluate particular aspects of synthesis. Each generated version is executed on three different devices, namely **CPU2** from Intel, **GPU1** from Intel and **GPU3** from NVIDIA as listed in Figure 6.1. The target hardware for this case study therefore features three different types of devices involving a CPU, an integrated GPU and a dedicated GPU. Each generated version is evaluated for the resulting code size, the total network build time, and the end-to-end performance. The code size of each generated version is described as the sum of lines of code of all generated kernels. The network build time is defined as the total time taken by the OpenCL just-in-time (JIT) compiler to build all the kernels in the network and the MoC specific API functions of the used dataflow MoC(s). The build time is measured only by using **CPU2**. The end-to-end performance is defined as the total execution time of the network to process the complete input data set including initialization and termination of the program. The data set used has a maximum of five thousand samples per input and the average of 50 repetitions is taken for each version.

6.4.1. Test Case I: Open-loop Configuration

The dataflow emulation of the design where *ActorBoard* directly feeds the actuators resulted in the open-loop configuration of the ConceptCar, as shown in Figure 6.8. The original network features a heterogeneous DPN of different kinds of processes exhibiting static as well as sequential behaviors. The function or behavior type of each process is described at the bottom of each node as shown in Figure 6.8. The main focus of this test case is to generate homogeneous implementations based on the individual dataflow MoCs of the framework and evaluate them for their resulting code size, the total network

	Generated networks (open-loop)			
		SDF	DDF	KPN
Process size (lines of code)	Processes			
	RadioRemote	59	83	60
	SelectorSwitch	28	48	29
	EmergencyBoard	59	83	60
	SensorBoardSteering	102	171	105
	SensorBoardThrottle	102	171	105
	Databoard	62	87	64
	ActorBoard (static)	206	-	-
	ActorBoard (dynamic)	-	340	172
	Actuators	29	49	29
total		647	1032	624
Network build time (msecs.)		249	1205	245

Figure 6.9.: *ConceptCar's open-loop setting: comparison of generated code size and network build time of all supported dataflow MoCs.*

build time, and the end-to-end performance. The open-loop configuration is therefore modeled and automatically synthesized thrice, once for each individual dataflow MoC. Hence, three different versions are automatically generated by the synthesis framework, i.e., first based on the SDF MoC, second using the KPN MoC, and finally based on the DDF MoC. The original dataflow model of *ActorBoard* exhibits a sequential behavior as it utilizes data either from *DataBoard* or *SensorBoards* based on the input provided by *SelectorSwitch*. A static version of *ActorBoard* is also modeled that consumes data from all inputs in each execution. This allowed us to design a fully static network and to generate an implementation of the open-loop network based on the SDF MoC.

Generated Code Size and Network Build Time

The generated versions of the open-loop network based on all three dataflow MoCs are illustrated in Figure 6.9. In particular, the generated kernel code size of each DPN process and the total build time for the complete network are given for each version.

The SDF MoC generated the most succinct kernel code for static processes. On the other hand, the KPN MoC generated the most concise code for sequential processes. In contrast to SDF and KPN MoCs, the DDF MoC offers a more flexible semantics where the decision on whether to consume/produce data in each execution is taken dynamically at runtime in the kernel code. Consequently, the generated DDF version accommodates additional kernel code for enabling the dynamic evaluation of actions at runtime when the pro-

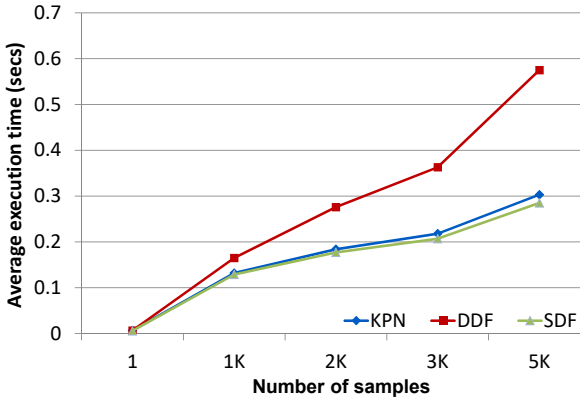

Figure 6.10.: *Open-loop setting: comparison of end-to-end performance of all supported dataflow MoCs on* ***CPU2****.*

cess is triggered for execution. This overhead can therefore be observed from the number of lines of the generated code for each process and the total network build time. The generated code size of the DDF version for the complete network is 65% and 59% greater than the KPN and SDF versions, respectively. This results in an additional build time overhead of 391% and 384% in comparison to the build times of KPN and SDF versions, respectively. The overhead also reflects the additional time taken to build the DDF MoC specific API functions used for dynamic execution within kernels.

Finally, we also observed that the KPN version has slightly less code size than the SDF version for the complete open-loop network. This is mainly because the static version of *ActorBoard* consumes data from all inputs in each execution and therefore the corresponding generated kernel accommodated more lines of code. Precisely, the generated code for the static version of *ActorBoard* based on the SDF MoC is about 20% more than the dynamic version generated based on the KPN MoC.

End-to-end performance

Each generated version of the open-loop network is executed on each target hardware at a time to evaluate and compare the end-to-end performance. On each target hardware. i.e., **CPU2**, **GPU1** and **GPU3**, the average execution time (in seconds) of each version is measured against the number of data samples as shown in Figure 6.10, Figure 6.11 and Figure 6.12, respectively.

Regardless of which target hardware is used, the SDF version demonstrated the best end-to-end performance of all generated versions. Apart from *ActorBoard*, all processes in the original open-loop network are static. As already observed in the previous section, the SDF MoC generated the most succinct kernel code for static processes. Second, since the SDF MoC also simplifies the scheduling of processes, this further contributes in the improved end-to-end performance for the SDF version. The KPN version although offered a slightly less code size, however, induced a slight overhead in scheduling static processes in comparison to the SDF version. The SDF version therefore per-

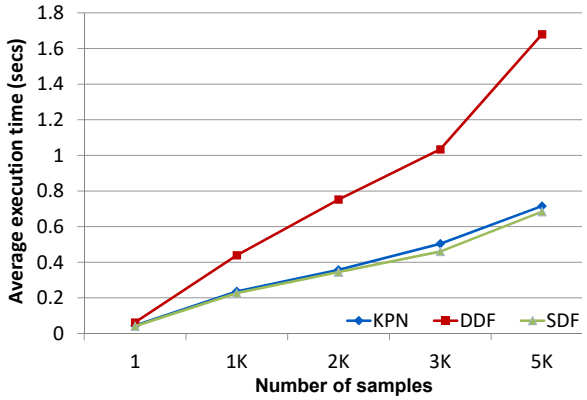

Figure 6.11.: *Open-loop setting: comparison of end-to-end performance of all supported dataflow MoCs on **GPU1**.*

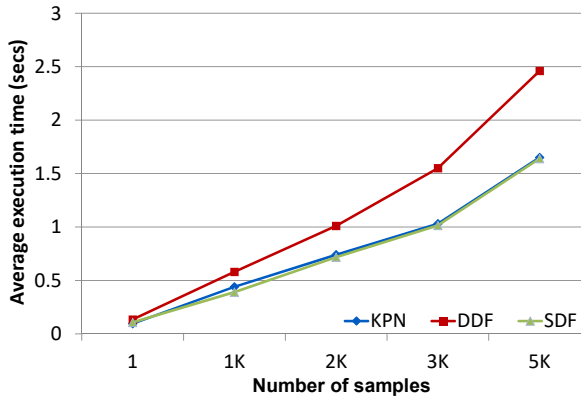

Figure 6.12.: *Open-loop setting: comparison of end-to-end performance of all supported dataflow MoCs on **GPU3**.*

formed only slightly better than the KPN version, in particular, executed only 7% and 4.5% faster on **CPU2** and **GPU1**, respectively. On **GPU3** however the difference in performance is negligible.

In contrast to SDF and KPN MoCs, the additional runtime overhead associated with the DDF MoC is propagated to the end-to-end performance resulting in elevated execution times. Based on the results, as the number of samples increases, this effect induced by the overhead can be clearly observed. On **CPU2**, the DDF version took twice as much time as taken by the SDF version and took about 90% more time than the KPN version to execute the complete network for five thousand samples. On **GPU1**, the DDF version yielded about 145% and 135% more execution time than the SDF and KPN versions, respectively. Finally, on **GPU3**, the DDF version required 50% more time to process five thousand samples in comparison to the SDF and KPN versions. The DDF MoC although offers semantics to model sequential as well as parallel behaviors, but at the cost of the additional runtime overhead. Therefore, it exhibits a trade-off between expressiveness and overall performance.

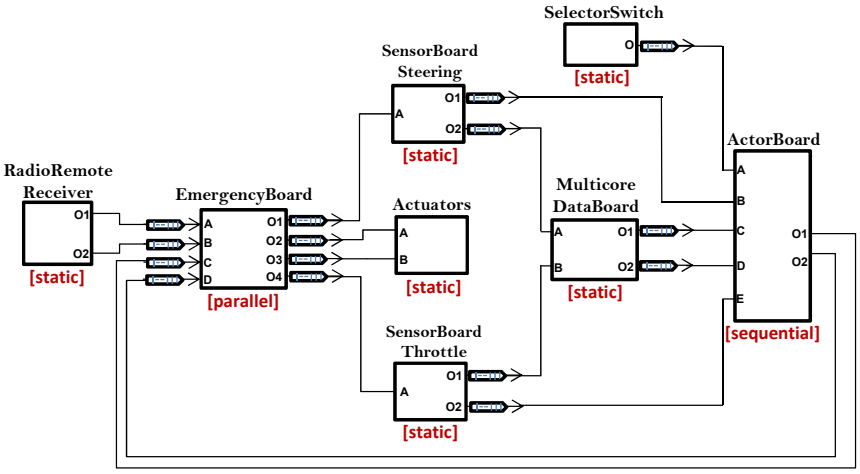

Figure 6.13.: *The original dataflow network of the ConceptCar based on the closed-loop configuration. As shown in the figure, a feedback loop is introduced into the network from ActorBoard into EmergencyBoard. The behavior type of each process is described at the bottom of each node in square brackets.*

All generated versions executed substantially faster on the CPU than on the used GPUs mainly because of the communication overhead associated with the OpenCL GPU. For instance, the SDF version on **CPU2** executed 4.75× and 1.4× faster than on **GPU3** and **GPU1**, respectively. Since, the integrated GPU (**GPU1**) share the same memory of the host, the versions executed substantially faster on **GPU1** than on the dedicated GPU (**GPU3**). For example, the SDF version on **GPU1** performed 1.39× faster than on **GPU3**.

6.4.2. Test Case II: Closed-loop Configuration

The dataflow emulation of the design where the *EmergencyBoard* ECU separates the actuators from the rest of the ECUs by galvanic isolation resulted in the closed-loop configuration of the ConceptCar. The closed-loop setting therefore introduces a feedback loop in the network from *ActorBoard* into *EmergencyBoard* as shown in Figure 6.13. The original network features a heterogeneous DPN of different kinds of processes exhibiting static, sequential and parallel behaviors. In particular, *EmergencyBoard* exhibits a parallel behavior as it features independent actions operating on the independent sets of inputs from *RadioRemoteReceiver* and *ActorBoard* and producing data to the independent sets of outputs. This test case focuses on observing how the feedback loop in the network affects the performances of the individual homogeneous implementations of all supported dataflow MoCs. Second, and most importantly, it also demonstrates how the proposed synthesis method effectively exploits the heterogeneity by generating implementations based on the underlying precise dataflow MoC of each process to further improve the end-to-end performance.

	Generated networks (closed-loop)				
		SDF	DDF	KPN	Hetero
Process size (lines of code)	Processes				
	RadioRemote	59	83	60	59
	SelectorSwitch	28	48	29	28
	EmergencyBoard (static)	86	-	87	-
	EmergencyBoard (parallel)	-	139	-	139
	SensorBoardSteering	102	171	105	102
	SensorBoardThrottle	102	171	105	102
	Databoard	62	87	64	62
	ActorBoard (static)	206	-	-	-
	ActorBoard (dynamic)	-	340	172	172
	Actuators	29	46	29	29
total		674	1085	651	693
Network build time (msecs.)		252	1208	249	348

Figure 6.14.: *Closed-loop setting: comparison of generated code size and network build time of all supported dataflow MoCs including their heterogeneous combination.*

The closed-loop configuration is modeled and automatically synthesized four times, once for each individual dataflow MoC and once based on the heterogeneous combination of all dataflow MoCs. Hence, four different versions are automatically generated by the synthesis framework, i.e., first based on the SDF MoC, second using the KPN MoC, third based on the DDF MoC, and finally the heterogeneous version based on the combination of all used dataflow MoCs. Since, the original dataflow model of *EmergencyBoard* exhibits a parallel behavior, a static version is also designed that consumes data from all inputs in each execution. This allowed us to generate implementations of the closed-loop network based on the SDF and KPN MoCs.

Generated Code Size and Network Build Time

The generated versions of the closed-loop network are illustrated in Figure 6.14. The generated homogeneous versions based on the individual dataflow MoCs demonstrated the same pattern in code size as observed in the case of the open-loop network. The generated code size of the DDF version for the complete network is 67% and 61% greater than the KPN and SDF versions, respectively. This resulted in an additional build time overhead of 385% and 379% in comparison to the build times of KPN and SDF versions, respectively.

The heterogeneous version is automatically generated based on the kind of behavior or the underlying precise dataflow MoC of each process in the network. The generated code size of the heterogeneous version is only slightly

greater than the KPN and SDF versions. Since the heterogeneous version employed the MoC specific API functions of all the used dataflow MoCs, this resulted in the build time overhead of about 40% and 38% in comparison to the build times of KPN and SDF versions, respectively. However, the code size of the DDF version is 56% greater than the heterogeneous version and therefore required 247% more time to build the complete network.

End-to-end Performance

Each generated version of the closed-loop network is executed on each target hardware at a time to evaluate and compare the end-to-end performance. On each target hardware. i.e., **CPU2**, **GPU1** and **GPU3**, the average execution time (in seconds) of each version is measured against the number of data samples as shown in Figure 6.15, Figure 6.16 and Figure 6.17, respectively.

Regardless of which target hardware is used, it can be observed that the introduction of the feedback loop in the network elevates the execution times of the SDF and KPN versions to an unacceptable level. As discussed, the original dataflow model of *EmergencyBoard* exhibits a parallel behavior. Since, the SDF MoC only supports static behaviors, a static version of *EmergencyBoard* is designed to generate the SDF and KPN versions. The static version of *EmergencyBoard* therefore requires data in all inputs before it can be scheduled for execution. With a feedback loop introduced into *EmergencyBoard*, the SDF and KPN versions only schedule and communicate a single execution at a time for all processes (except *RadioRemoteReceiver*) at the device. Consequently, this induces a lot of scheduling and communication overhead between the host and device, and therefore resulted in excessively elevated execution times. The DDF version on the other hand employs a parallel version of *EmergencyBoard* and therefore attempts to schedule and communicate as many executions at a time as possible based on the availability of data on independent sets of inputs. Thus, even with the associated runtime overhead, the DDF version outperformed the SDF and KPN versions.

On **CPU2**, the DDF version executed 31× and 41× faster than the SDF and KPN versions, respectively, in executing the complete network for five thousand samples. Similarly, the end-to-end performance of SDF and KPN versions on both the used GPUs reached to an unacceptable level. On **GPU1**, the SDF executed 66× and the KPN version executed 149× slower in comparison to the DDF version. On **GPU3**, the SDF and KPN versions are 51× and 45×, respectively, slower than the DDF version. The difference in execution times is bigger on the GPUs than on the CPU mainly because of the communication overhead associated with OpenCL GPUs. The SDF version performed substantially better than the KPN version on the CPU and the integrated GPU, in particular, executed about 1.33× and 2.2× faster, respectively. This is mainly because the KPN version induced an overhead in scheduling static processes for a large number of single executions. Interestingly, the KPN version executed about 1.13× faster than the SDF version on the dedicated GPU. The KPN version only dispatches a process for execution on the device if there exists one of the actions whose firing rules are satisfied. It therefore evaluates

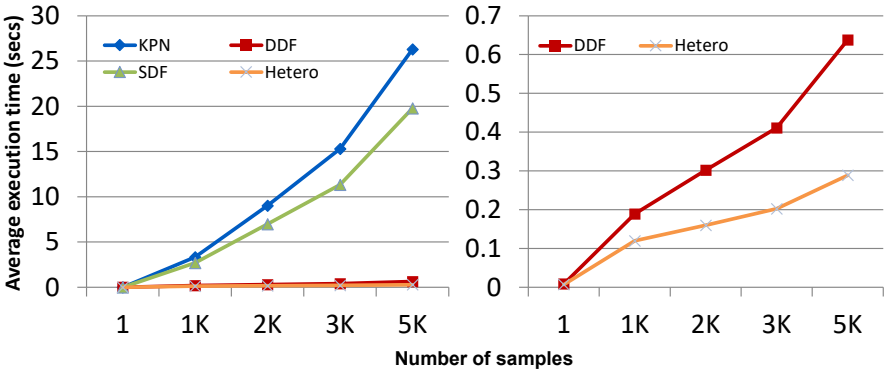

Figure 6.15.: *Closed-loop setting: comparison of end-to-end performance of all supported dataflow MoCs on **CPU2**. The right hand side graph particularly shows the performance comparison between the DDF version and the heterogeneous version.*

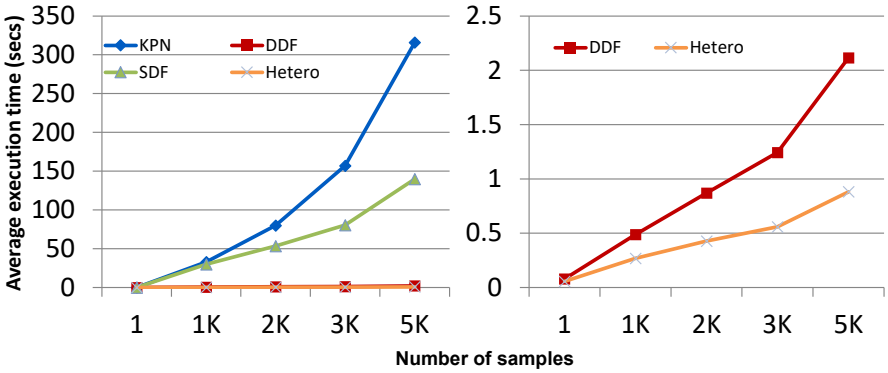

Figure 6.16.: *Closed-loop setting: comparison of end-to-end performance of all supported dataflow MoCs on **GPU1**. The right hand side graph particularly shows the performance comparison between the DDF version and the heterogeneous version.*

the firing rules at the scheduling time on the host side. This relaxes the computation on the device side. Since the dedicated GPU uses its own CPU and memory, the execution of simpler operations on less powerful processing cores contributed in the improved end-to-end performance for the KPN version.

The heterogeneous version. The main highlight of this test case is to evaluate the ability of the proposed synthesis method to efficiently exploit the heterogeneity of different kinds of behaviors of processes in the network. The heterogeneous version is therefore automatically generated based on the underlying precise dataflow MoC of each process in the network. Regardless of which device is used for execution, the heterogeneous version demonstrated a substantial improvement in performance in comparison to the most efficient homogeneous version of the closed-loop network. The performance comparison between the heterogeneous version and the DDF version is especially shown on the right hand side graphs of Figures 6.15, 6.16 and 6.17. In particular, the heterogeneous version executed 2.2×, 2.4× and 1.86× faster than the DDF ver-

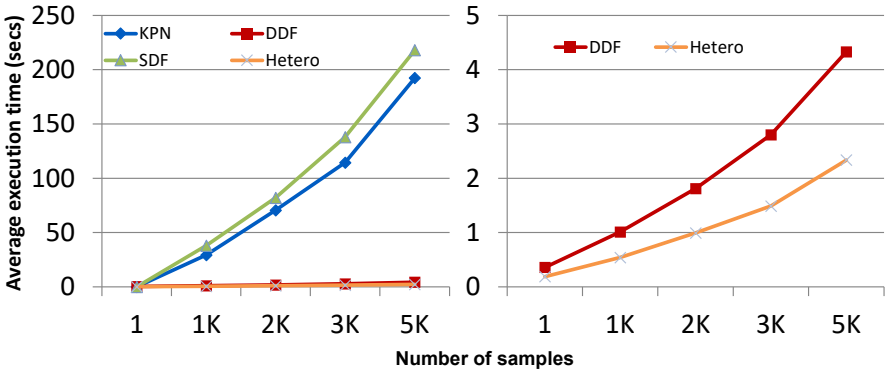

Figure 6.17.: *Closed-loop setting: comparison of end-to-end performance of all supported dataflow MoCs on **GPU3**. The right hand side graph particularly shows the performance comparison between the DDF version and the heterogeneous version.*

sion on **CPU2**, **GPU1** and **GPU3**, respectively. The heterogeneous version therefore significantly improved the performance by eliminating the overhead induced by the feedback loop in the SDF and KPN versions, and by avoiding the additional runtime overhead of the DDF version.

Hence, it can be concluded from this particular test case that the ability to exploit the heterogeneity of different kinds of behaviors of processes in DPNs contributes in efficient system implementations with substantially improved end-to-end performance.

6.4.3. Summary

The first test case featured an open-loop network which is mainly designed to generate and evaluate homogeneous implementations based on all individual supported dataflow MoCs. Considering the fact that most of the processes in the network exhibit static behaviors, the SDF version demonstrated the best end-to-end performance of all generation versions. The DDF version induced the runtime overhead of evaluating actions within kernels and performed the slowest of all generated versions. In particular, the DDF version took about 145% and 135% more execution time than the SDF and KPN versions, respectively.

The second test case introduced a feedback loop from *ActorBoard* to *EmergencyBoard* that resulted in a closed-loop network. We observed that the introduction of the feedback loop in the network greatly degraded the end-to-end performance of SDF and KPN versions. In particular, the DDF version even with the associated runtime overhead outperformed the SDF and KPN versions. However, the heterogeneous version that exploited the heterogeneity of different kinds of processes significantly improved the performance, in particular, executed up to 2.4× faster than the fastest homogeneous DDF version. Hence, the ability of the proposed synthesis method to exploit heterogeneity in a DPN effectively contributed in achieving the best end-to-end performance.

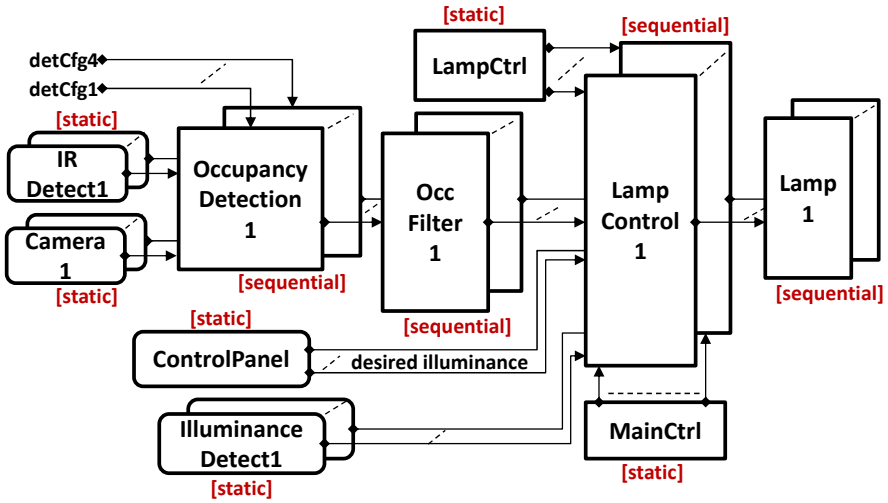

Figure 6.18.: *The main architecture of the building automation system (BAS). The architecture can be easily expanded by configuring different number of rooms.*

6.5. Case Study II: Building Automation System (BAS)

As discussed, the proposed synthesis design flow has been mainly used as a part of our long term project *Averest*. This case study also demonstrates the successful usage of the proposed synthesis method in conjunction with our *Averest* tool. We designed a smart building automation system (BAS) in the spirit of [Trap05]. We modeled a number of different versions of BAS with different configurations based on the SR MoC using the synchronous language *Quartz* [ScBr17] of *Averest*. The modeled synchronous versions of BAS are then verified for desynchronization and automatically transformed to DPNs using *Averest*. These desynchronized models (DPNs) are then used by the proposed synthesis framework to generate various versions involving homogeneous as well as heterogeneous implementations.

Computational Architecture

The main purpose of BAS is to control the lamps in the rooms of a building and notify the security based on occupancy detection. The basic building block diagram of BAS is shown in Figure 6.18. The two main components in the system are *OccupancyDetection* and *LampControl*. The occupancy is detected by *OccupancyDetection* based on multiple resources. First, it is possible to identify the occupancy by detecting motion, which is provided by the infrared motion detectors (*IRDetect*). Second, the occupancy can be determined using a camera image, which is provided by the smart cameras (*Camera*). Altogether, *OccupancyDetection* takes as input the movements detected by a set of infrared motion detectors and smart cameras and reports the occupancy status of the rooms. Apart from the occupancy, *LampControl* requires the current and the desired illuminance, which are provided by

the illuminance sensors (*IlluminanceDetect*) and the control panel (*ControlPanel*), respectively. The lamp configurations of the rooms are provided by the component *LampCtrl*. *LampControl* accepts the lamp configurations, the occupancy status, and the current and the desired illumination levels to control the switching of the lamps (*Lamp*). A lamp can be switched either automatically based on the occupancy or manually using a switch provided by the component *MainCtrl*.

BAS Configurations

This case study is organized into two test cases, where each test case is dedicated to analyse and evaluate particular aspects of synthesis based on the BAS architecture. BAS offers an architecture that can be easily expanded with a number of rooms as shown in Figure 6.18. The first test case realizes various versions of BAS mainly by configuring different number of rooms. Whereas, the second test case focuses on various versions of a particular BAS configuration of four rooms. For each test case, different synchronous networks are first modeled and then automatically desynchronized to the corresponding DPNs using *Averest*.

A number of implementations (versions) are automatically generated by the synthesis framework from the desynchronized models (DPNs) of both test cases. Each generated version is executed on all five target devices as listed in Figure 6.1. The target hardware for this case study therefore features two CPUs (**CPU1** and **CPU2**), one integrated GPU (**GPU1**) and two dedicated GPUs (**GPU2** and **GPU3**). Each generated version of the first test case is evaluated for its network build time and the average execution time. The generated versions of the second test case based on a fixed four rooms configuration are thoroughly evaluated for end-to-end performance. The data set used offered a maximum of ten thousand samples per input and the average of fifty repetitions is taken for each version.

6.5.1. Test Case I: Network Complexity

This test case models various synchronous versions of BAS mainly by configuring different number of rooms. In particular, five different synchronous networks of BAS are designed based on one, two, four, six and eight room configurations. These synchronous networks are then automatically desynchronized to the corresponding DPNs using *Averest*. All the desynchronized networks based on different configurations feature heterogeneous DPNs of different kinds of processes involving static as well as sequential behaviors, as labelled in Figure 6.18. The resulting complexity of each desynchronized network is summarized in Figure 6.19. The simplest of the network consisting of a single room only features seventeen FIFO buffers and seven processes having twenty-four actions. The complexity increases with the number of rooms where the most complex network in terms of number of processes and actions is resulted in the case of eight-rooms configuration.

The main focus of this test case is to evaluate how the resulting complexity

	# rooms	# processes	# actions	# FIFO buffers
network 1	1	7	24	17
network 2	2	16	48	36
network 3	4	30	96	70
network 4	6	44	144	104
network 5	8	58	192	138

Figure 6.19.: *Test case I: network complexity of generated dataflow networks based on different BAS configurations.*

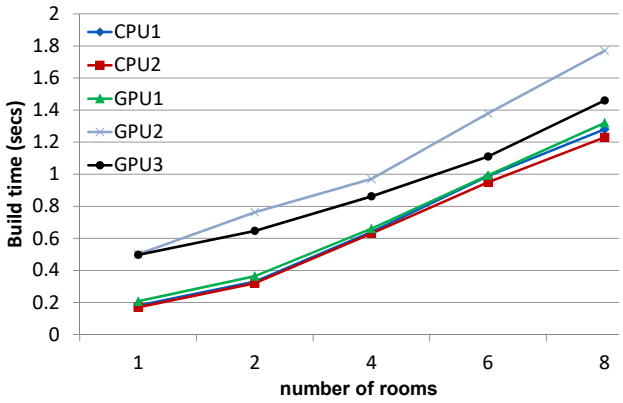

Figure 6.20.: *Test case I: the affect of increasing the network complexity on the build time on all five target devices.*

affects the total build time and the average execution time of each network. Since, the KPN MoC supports both static as well as sequential behaviors, a homogeneous implementation (version) based on the KPN MoC is automatically generated for each desynchronized network by the synthesis framework. Furthermore, we also evaluate the affect of increasing the number of samples on the execution time for the most complex network with eight rooms.

Network Complexity vs Build Time

The build time of each generated homogeneous version is measured on all target devices as shown in Figure 6.20. Regardless of which target device is used, the build time increases linearly with the increasing network complexity. On average, the build time shows an increase by about 5.5× on all devices from the simplest one-room version to the most complex eight-rooms version. Both CPUs (**CPU1** and **CPU2**) demonstrated the best build time performance of all used devices. The integrated GPU (**GPU1**) demonstrated the build time performance close to **CPU1** and **CPU2**. Moreover, **GPU1** acquired better build time performance in comparison to the dedicated GPUs (**GPU2** and

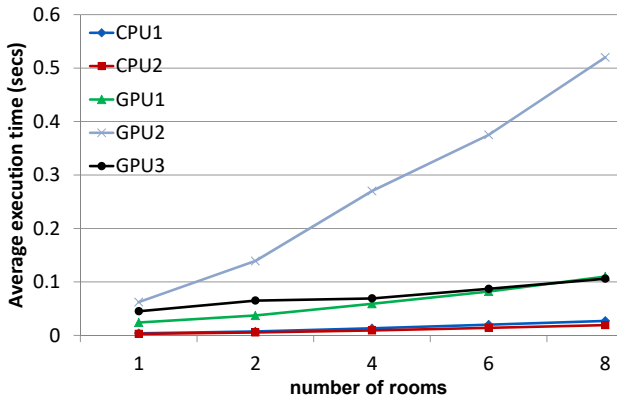

Figure 6.21.: *Test case I: the affect of increasing the network complexity on the average execution time on all five target devices.*

GPU3). **GPU2** features the least processing power of the used dedicated GPUs and therefore produced the longest build time of all used devices. The OpenCL JIT compiler generally does not target building independent parts of programs in parallel and therefore typically builds programs faster on the CPUs than on the GPUs.

Network Complexity vs Average Execution Time

Each generated version is executed on each target hardware at a time for one thousand samples to measure the execution time. The affect of increasing the network complexity (number of rooms) on the average execution time on each target device is depicted in Figure 6.21. Irrespective of which target device is used, the average execution time increases with the increasing complexity. On both CPUs (**CPU1** and **CPU2**), the generated version of eight-rooms configuration showed an increase in execution time by 7× in comparison to the one-room version. Similarly, on **GPU1**, **GPU2** and **GPU3**, the execution time is increased by 4.6×, 8.4× and 2.35×, respectively. All the generated versions executed faster on the CPUs than on the GPUs. This is mainly because of the following reason: The generated implementations do not yet exploit data level parallelism and therefore the additional OpenCL communication overhead on GPUs contributes in elevated execution times. The integrated GPU (**GPU1**) uses the memory which is shared with **CPU2** and generally performed better than the dedicated GPUs. **GPU2** offering the least processing power of the used dedicated GPUs demonstrated the longest average execution times for all versions.

Sample Size vs Execution Time

We also measured the affect of increasing the number of samples on the average execution time of the most complex eight-rooms network version as shown in Figure 6.22. On each target device, the average execution time increases linearly with the increasing number of samples. On both CPUs (**CPU1** and

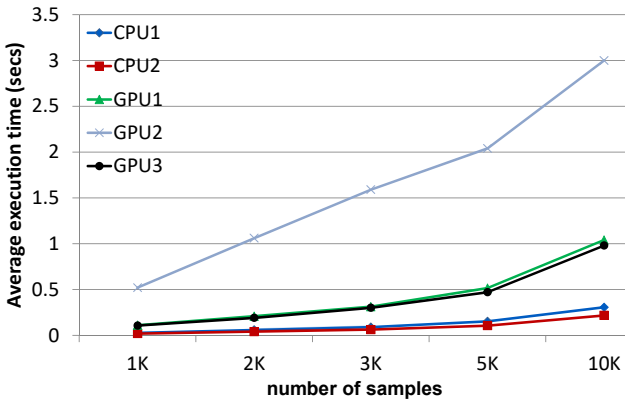

Figure 6.22.: *Test case I: the affect of increasing the number of samples on the average execution time on all five target devices. The eight-room configuration is particularly used for this part of the test case.*

Synchronous networks	Synthesized versions	Implementation type
original network	• KPN (reference) • Hetero-Original • Hetero-Original-PAR	• homogeneous • heterogeneous • heterogeneous
transformed network	• Hetero-Transform • Hetero-Transform-PAR	• heterogeneous • heterogeneous
fully static network	• SDF • SDF-PAR	• homogeneous • homogeneous

Figure 6.23.: *Test case II: the synchronous networks and the finally synthesized versions. The generated implementation type is also depicted in the figure.*

CPU2), the average execution time is increased by more than 11× by increasing the number of samples by ten times. Similarly, on **GPU1**, **GPU2** and **GPU3**, the execution time is increased by 9.4×, 5.8× and 9.2×, respectively. In terms of the used target devices, the results showed almost the same pattern as observed in the case of increasing complexity. Interestingly, the dedicated GPU (**GPU3**) demonstrated slightly lower execution times than the integrated GPU (**GPU1**) with the increase in the number of samples.

6.5.2. Test Case II: Exploiting Network Heterogeneity

This test case models various synchronous networks based on the particular four-rooms BAS configuration that resulted in a system of thirty components (processes) as illustrated in Figure 6.19. First, the original synchronous network of four-rooms from the previous test case is used as illustrated in Figure 6.18. The generated homogeneous dataflow version of four-rooms based on the KPN MoC from the previous test case is used as the **reference** version for this test case. We further generated two heterogeneous versions labelled by **Hetero-Original** and **Hetero-Original-PAR** based on the original synchronous network, where each process is synthesized based on the underlying

precise dataflow MoC. The term 'PAR' is used to indicate the version that further exploits the data level parallelism (DLP) whenever possible in fully static processes. We further transformed the original synchronous network and worked out two different synchronous networks. Firstly, we transformed the original synchronous network by converting all dynamic components into static ones except the *OccupancyDetection* component, which is dynamic. Two different heterogeneous versions labelled by **Hetero-Transform** and **Hetero-Transform-PAR** are generated by the synthesis framework, where each process is synthesized based on the underlying dataflow MoC. Secondly, the original synchronous network is transformed by accommodating dummy inputs in all dynamic components to make them static and thereby yielding a fully static network consisting of only static components. Two different homogeneous dataflow versions based on the SDF MoC are generated from the completely static network. These versions are labelled by **SDF** and **SDF-PAR**. The modeled synchronous networks and the finally synthesized dataflow versions are summarized in Figure 6.23.

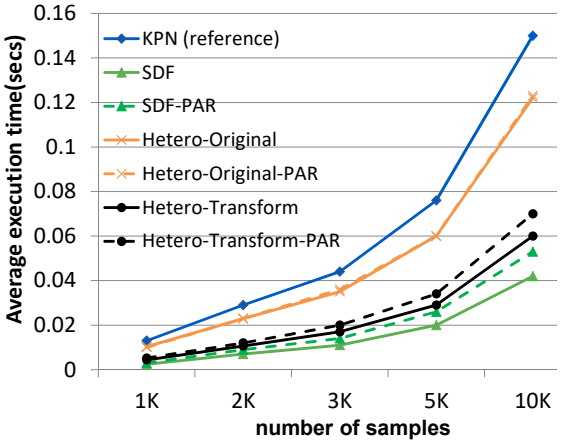

Figure 6.24.: *Test case II: the performance comparison of all generated versions on **CPU1**. The results of the parallel versions that exploit data level parallelism are represented by dashed lines.*

The main focus of this test case is to validate and evaluate the ability of the proposed synthesis method to generate efficient system implementations based on the underlying precise dataflow MoC of each process in the network. This test case therefore employs the synthesis framework for generating homogeneous as well as heterogeneous implementations. All the generated versions are evaluated for end-to-end performance in comparison to the homogeneous **reference** version that is generated without targeting heterogeneity of different kinds of processes.

Each generated version is executed on each target device at a time to evaluate and compare the end-to-end performance of all generated versions. On each target hardware, the average execution time of each generated version is measured against the number of data samples as shown in Figures 6.24, 6.25, 6.26, 6.27 and 6.28.

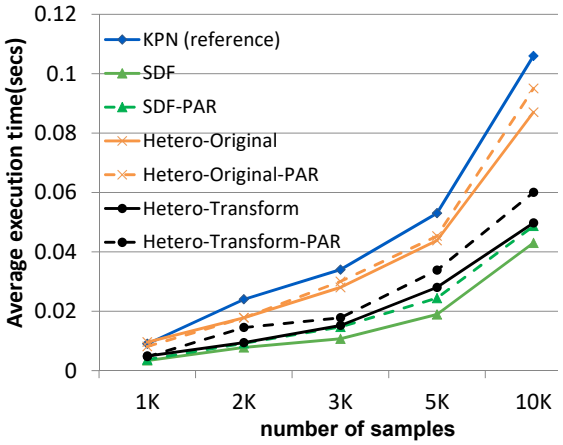

Figure 6.25.: *Test case II: the performance comparison of all generated versions on **CPU2**. The results of the parallel versions that exploit data level parallelism are represented by dashed lines.*

End-to-end performance: CPUs

The end-to-end performance of all generated versions on **CPU1** and **CPU2** is depicted in Figure 6.24 and Figure 6.25, respectively. On both CPUs, the results clearly demonstrate the benefit of synthesizing components based on the underlying precise dataflow MoC of each process. All the versions that are generated based on the kinds of behaviors of the processes performed significantly better than the **reference** version. The completely static versions **SDF** and **SDF-PAR** performed the fastest of all generated versions, in particular, executed 3.57× and 2.83× faster than the **reference** version. The **SDF** version performed about 20% faster than the **SDF-PAR** version that exploits DLP. This is because the ability of a CPU to handle parallel instances is just limited to the number of few available cores. Therefore, dispatching a large number of parallel instances on few available cores results in OpenCL implementation overhead on CPUs and reduces the overall performance. Similarly, the heterogeneous versions **Hetero-Transform** and **Hetero-Original** that do not exploit DLP performed substantially better than the **reference** version, in particular, executed 2.5× and 1.22× faster, respectively.

End-to-end performance: Integrated GPU

The end-to-end performance of all generated versions on the integrated GPU is depicted in Figure 6.26. In general, an integrated GPU doesn't have its own private memory, instead, it uses the memory that is shared with the system. To this end, the target integrated GPU demonstrated largely the same trend of end-to-end performance of generated versions as was observed in the case of the target CPUs. The results again show the effectiveness of synthesizing components based on the kinds of behaviors. However, since the integrated GPU offers a relatively large number of cores than the used CPUs, the parallel versions that exploit DLP performed significantly better than the other generated

versions. In particular, the generated versions **SDF**, **Hetero-Original** and **Hetero-Transform** that do not exploit DLP still executed 1.23×, 1.06× and 1.22× faster than the **reference** version, respectively. Their parallel versions **SDF-PAR**, **Hetero-Original-PAR** and **Hetero-Transform-PAR** that exploit DLP further improved their performance by 1.61×, 1.04×, and 1.45×, respectively. Overall, the **SDF-PAR** version stands the fastest of all generated versions on the integrated GPU.

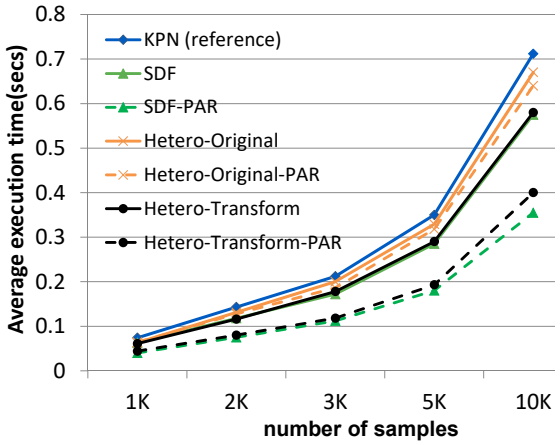

Figure 6.26.: *Test case II: the performance comparison of all generated versions on **GPU1**. The results of the parallel versions that exploit data level parallelism are represented by dashed lines.*

End-to-end performance: Dedicated GPUs

The end-to-end performance of all generated versions on the dedicated GPUs (**GPU2** and **GPU3)** is depicted in Figure 6.27 and Figure 6.28, respectively. **GPU2** offers a relatively large number of less powerful processing cores than the target CPUs. Expectedly, the generated versions that do not exploit DLP including the **reference** version performed the slowest of all versions. This is mainly because with the sequential execution of instances on the GPU, the large number of available processing units are not completely utilized. Consequently, this substantially degrades the performance of all versions that do not exploit DLP. The parallel versions on **GPU2** simply outperformed the other generated versions. In particular, the parallel versions **SDF-PAR**, **Hetero-Transform-PAR** and **Hetero-Original-PAR** executed about 25×, 5× and 1.3× faster than their sequential versions **SDF**, **Hetero-Transform** and **Hetero-Original**, respectively. Moreover, the parallel versions **SDF-PAR**, **Hetero-Transform-PAR** and **Hetero-Original-PAR** executed 18.5×, 3.65× and 1.06× faster than the **reference** version, respectively.

GPU3 is the most powerful device of all the employed GPUs as shown in Figure 6.1. It offers a relatively large number of processing cores as well as the highest clock frequency of both the used integrated and dedicated GPUs.

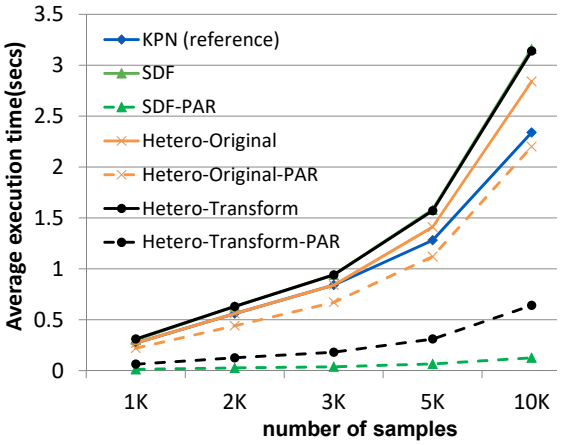

Figure 6.27.: *Test case II: the performance comparison of all generated versions on **GPU2**. The results of the parallel versions that exploit data level parallelism are represented by dashed lines.*

As expected, the parallel versions generally displayed better end-to-end performance of all generated versions. However, **GPU3** also demonstrated some very intriguing evaluation results. The versions having more static processes showed reduced end-to-end performance on **GPU3** in comparison to all other target devices. The parallel version **SDF-PAR** that clearly dominated on all the used GPUs performed slower than the **reference** version. In particular, the **reference** version executed 1.28× faster than the **SDF-PAR** version. Similarly, the **Hetero-Original-PAR** version that incorporates relatively more dynamic processes executed 1.62× and 1.49× faster than the **SDF-PAR** and **Hetero-Transform-PAR** versions, respectively. This is mainly because of the following reasons: First, the original synchronous network is transformed by accommodating dummy inputs in the dynamic components to make them static. Static processes are scheduled for execution only if data is available in all inputs and their guards are evaluated on the target device. This introduces additional communication of data in the form of dummy inputs. In contrast, the dynamic processes are scheduled only if there exists one of the actions whose firing rules are satisfied. Their firing rules are therefore evaluated at the scheduling time on the host side. This relaxes the computation on the device side. Since **GPU3** uses its own CPU and memory as well as it offers the largest number of cores, the execution of simpler operations on less powerful processing cores contributed in the improved end-to-end performance for the versions having more dynamic processes.

Summary

We modeled the original synchronous network and further transformed it to exploit the design space and to leverage the balance between communication and scheduling. We synthesized from both the original and the transformed networks different dataflow versions based on the kinds of behaviors of the

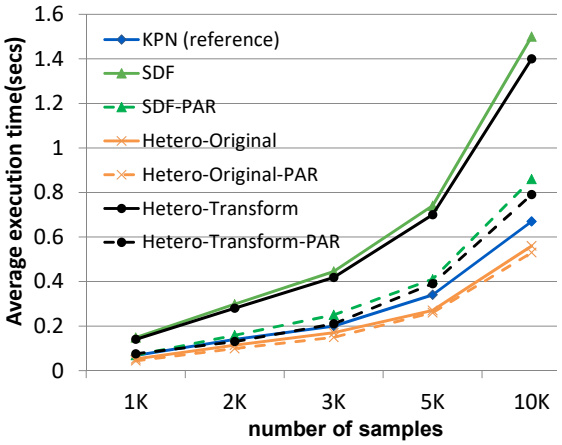

Figure 6.28.: *Test case II: the performance comparison of all generated versions on **GPU3**. The results of the parallel versions that exploit data level parallelism are represented by dashed lines.*

processes. Regardless of which target device is used, the generated versions based on the proposed synthesis method substantially increased the performance. The completely static version **SDF** without DLP demonstrated the best end-to-end performance of all generated versions on the target CPUs. As we moved to the GPUs that offered a large number of cores, the parallel versions that exploited DLP mostly displayed the best performance. The integrated GPU (**GPU1**) shares the memory with **CPU2** and demonstrated largely the same trend of end-to-end performance as was observed in the case of **CPU2**. On the dedicated GPUs, however, the parallel versions clearly dominated the charts in terms of end-to-end performance.

On **GPU2**, the **SDF-PAR** version executed up to 18.5× faster than the **reference** version. Similarly, the parallel version **Hetero-Transform-PAR** generated from the transformed synchronous network displayed a significant increase in performance, in particular, increased the performance by 3.65× in comparison to the **reference** version. On **GPU3** that offered the highest processing power, demonstrated best results for parallel versions accommodating relatively more dynamic processes.

Finally, all the generated versions generally executed substantially faster on the used CPUs than on the used GPUs. As an example, the **Hetero-Transform-PAR** version on **CPU1** executed 9× faster than on **GPU2**. This is mainly because the communication overhead of OpenCL on GPU is higher than the performance gain of executing parallel instances on many cores. In contrast to the OpenCL CPU where the host and the kernels reside on the same device, in the case of GPU, the data has to be transferred to the GPU and back to the main memory (host).

Chapter 7

Conclusions

7.1. Conclusions of the Thesis

Dataflow process networks (DPNs) are intrinsically data-driven and therefore a suitable model of computation (MoC) for implementing asynchronous and distributed systems. However, the verification of the major correctness properties like the absence of deadlocks and buffer overflows in DPNs is in general not decidable. For this reason, desynchronization techniques are used to convert clock-driven models into data-driven ones in order to more efficiently support distributed implementations. These techniques preserve the functional specification of the synchronous models and moreover preserve properties like deadlock-freedom and bounded memory usage that are otherwise difficult to ensure in DPNs. These desynchronized models (DPNs) are the starting point of this thesis.

A DPN can be heterogeneous in the sense that different processes may exhibit different kinds of behaviors that determine the precise dataflow MoCs. To efficiently implement systems based on DPNs, this heterogeneity can be exploited by scheduling and executing the processes based on their kinds of behaviors. The existing design tools for software synthesis are usually dedicated to provide homogeneous implementations based on a particular dataflow MoC. Each tool provides a specialized tool chain including a code generator and the runtime system targeting only a specific dataflow MoC. Second, the final result of most of these design tools is more or less a set of automatically generated C programs. These programs are then mapped on a specific target hardware in an additional deployment step. These tools essentially enable the mapping and execution of generated C programs based on a multithreading concept at the abstraction of an operating system (OS). Thus, they offer parallelism at the coarse-grained task level, which is usually restricted to devices like multi-core CPUs.

This thesis considered the automatic software synthesis of systems based on three different well-defined and precise dataflow MoCs including their heterogeneous combinations. These involve the static dataflow (SDF) MoC, the Kahn process networks (KPN) MoC and a deterministic variant of the dy-

namic dataflow (DDF) MoC. We proposed and presented a synthesis design flow for automatic software synthesis of systems from DPN models. The design flow is implemented in an extendable model-based design framework that offers a comprehensive tool chain including specialized code generators and schedulers for the supported dataflow MoCs. First, this allowed us to meet the objective of validating, evaluating and comparing the artifacts exhibited by different dataflow MoCs at the implementation level under the shed of a common design tool. Second, the proposed framework employed an efficient and smarter synthesis method that targets and exploits heterogeneity in dataflow networks by generating implementations based on the kinds of behaviors of the processes. To this end, the synthesis of heterogeneous DPNs is realized by using the combination of different specialized code generators and the runtime components based on the supported dataflow MoCs. In particular, the proposed synthesis tool chain is designed under a common dynamic environment to enable the dynamic scheduling and execution of heterogeneous composition of different kinds of processes.

Finally, this thesis also tackled the challenge of systematically handling the portability of systems on cross-vendor commercial off-the-shelf (COTS) heterogeneous platforms. To this end, the framework systematically employed the open specification language (OpenCL) as a standard hardware abstraction in the composition of the synthesis tool chain. The systematic integration of OpenCL as a foundational part of the tool chain not only allowed us to target vendor-neutral hardware, but also provided the opportunity to target task-level as well as data-level parallelism. Altogether, this enabled a more generalized synthesis that automatically maps the dataflow models on cross-vendor target hardware involving multi-core CPUs as well as GPUs, without the need of the user to manually identify the potential parallelism in the model.

We designed a number of standalone benchmarks to evaluate and compare the artifacts exhibited by the individual supported dataflow MoCs. Moreover, we also worked out two interesting case studies that featured heterogeneous DPNs consisting of different kinds of processes. The case studies are mainly designed with the aim to evaluate and compare the implementations generated precisely based on the kinds of behaviors of the processes. A variety of OpenCL supported devices have been employed involving multi-core CPUs as well as GPUs from different vendors.

Based on our detailed evaluations, we observed that even for the simplest of the benchmarks, the generated versions based on the kinds of behaviors of the processes demonstrated the best end-to-end performance. In particular, using a more generalized dataflow MoC for scheduling and executing rather restricted dataflow behaviors resulted in inefficient system implementations. For instance, for benchmarks involving static behaviors, the SDF MoC performed up to 1.79× faster than the DDF MoC in terms of the end-to-end performance. Similarly, for sequential behaviors, the KPN MoC performed up to 2.87× faster than the DDF MoC.

The heterogeneous versions generated from desynchronized or modeled DPNs by the proposed synthesis method in both case studies demonstrated

a substantial improvement in performance in comparison to their generated homogeneous versions. For the first case study, the heterogeneous version performed up to 2.4× faster than the most efficient generated homogeneous version. The heterogeneous versions generated for the second case study showed up to 3.65× improvement in performance in comparison to the reference homogeneous version. Based on the detailed evaluations, it can be concluded that the ability to exploit the heterogeneity of different kinds of behaviors of processes in DPNs contributes in efficient system implementations with substantially improved end-to-end performance.

7.2. Future Prospects

This thesis presented an extendable model-based design framework that offers a common platform for automatic software synthesis of systems from different types of dataflow models. In this thesis, we focused mainly on validating and evaluating the artifacts exhibited by different dataflow models, especially by their heterogeneous combinations, at the implementation level of systems. However, the presented framework also offers some interesting prospects for future work:

7.2.1. Design Decisions

So far, we modeled the systems either based on the synchronous reactive (SR) MoC or directly using the supported dataflow models. Different versions of the systems are designed to explore the design space by deliberately modeling behaviors based on particular kinds of processes. A design decision-making method can be integrated ideally at the synchronous level to analyse and automatically explore the design space. In particular, such a method can be used to leverage the balance between different kinds of processes in DPNs.

7.2.2. Optimized Scheduling

In this thesis a baseline global dynamic scheduler is used to enable the scheduling of heterogeneous dataflow models based on their specialized dynamic local schedulers. Static scheduling is generally only possible for full static networks consisting of only static processes. Since, this thesis targets DPNs that consist of static as well as dynamic nodes, each local scheduler based on a particular dataflow MoC therefore follows a dynamic scheme and schedules a process based on the underlying semantics of that dataflow MoC. In a DPN consisting of a heterogeneous combination of static and dynamic processes, the overhead of dynamically scheduling processes at runtime can be reduced by optimizing the scheduling of static parts in the network. In particular, the clusters of static processes in a DPN can be replaced by composite processes. Each composite process may contain a local scheduler that executes sequences of statically scheduled process firings. This optimization can potentially provide performance gains by targeting groups of processes instead of individual pro-

cesses. A lot work [PiLe95; TBGR13; FSGT15] has been done in this area and can be considered for the presented framework in the future.

7.2.3. Dynamic Load Balancing and Mapping

The proposed framework provides an interface in the form of the *Device-Queue* to dynamically take into account the current load of each of the computing units in the system. A dynamic load distribution scheme can be designed and integrated in the framework to further optimize the mapping scheme and potentially improve the performance. Using such an approach, processes can be migrated dynamically from one compute unit to another based on more advanced load balancing algorithms. Some work [ChMi15; ZhGW09; WiRe93] has been presented in this area and can be considered for the proposed framework in the future.

Bibliography

[BSBK14] Y. Bai, K. Schneider, N. Bhardwaj, B. Katti, and T. Shazadi. "From Clock-Driven to Data-Driven Models". In: *Formal Methods and Models for Codesign (MEMOCODE)*. Lausanne, Switzerland: IEEE Computer Society, 2014, pp. 32–41.

[Bai16] Y. Bai. "Model-based Design of Embedded Systems by Desynchronization". PhD. PhD thesis. Department of Computer Science, University of Kaiserslautern, Germany, 2016.

[BeCG99] A. Benveniste, B. Caillaud, and P. Le Guernic. "From Synchrony to Asynchrony". In: *Concurrency Theory (CONCUR)*. Ed. by J.C.M. Baeten and S. Mauw. Vol. 1664. LNCS. Eindhoven, The Netherlands: Springer, 1999, pp. 162–177.

[BCEH03] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. "The Synchronous Languages Twelve Years Later". In: *Proceedings of the IEEE* 91.1 (2003), pp. 64–83.

[BTRM14] E. Bezati, R. Thavot, G. Roquier, and M. Mattavelli. "High-level dataflow design of signal processing systems for reconfigurable and multicore heterogeneous platforms". In: *Journal of Real-Time Image Processing* 9.1 (2014), pp. 251–262.

[BoGh15] J. Boutellier and A. Ghazi. "Multicore execution of dynamic dataflow programs on the distributed application layer". In: *Global Conference on Signal and Information Processing (GlobalSIP)*. Orlando, USA: IEEE Computer Society, 2015, pp. 893–897.

[BeGo92] G. Berry and G. Gonthier. "The Esterel Synchronous Programming Language: Design, Semantics, Implementation". In: *Science of Computer Programming* 19.2 (1992), pp. 87–152.

[BoHa16] J. Boutellier and I. Hautala. "Executing Dynamic Data Rate Actor Networks on OpenCL Platforms". In: *Signal Processing Systems*. TX, USA: IEEE Computer Society, 2016, pp. 98–103.

[BBJE09] S.S. Bhattacharyya, G. Brebner, J.W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet. "OpenDF-A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems". In:

ACM SIGARCH Computer Architecture News 36.5 (Dec. 2009), pp. 29–35.

[BEJL11] S.S. Bhattacharyya, J. Eker, J.W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet. "Overview of the MPEG Reconfigurable Video Coding Framework". In: *Journal of Signal Processing Systems* 63.2 (May 2011), pp. 251–263.

[BrLT10] C.X. Brooks, E.A. Lee, and S. Tripakis. "Exploring models of computation with ptolemy II". In: *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Ed. by T. Givargis and A. Donlin. Arizona, USA: ACM, 2010, pp. 331–332.

[BoNy15] J. Boutellier and T. Nylanden. "Programming graphics processing units in the RVC-CAL dataflow language". In: *Signal Processing Systems (SiPS)*. Hangzhou, China: IEEE Computer Society, 2015, pp. 1–6.

[BWHB18] J. Boutellier, J. Wu, H. Huttunen, and S.S. Bhattacharyya. "PRUNE: Dynamic and Decidable Dataflow for Signal Processing on Heterogeneous Platforms". In: *IEEE Transactions on Signal Processing* 66.3 (Feb. 2018), pp. 654–665.

[BaRS21] Y. Bai, O. Rafique, and K. Schneider. "A Model-based Design Flow for Asynchronous Implementations from Synchronous Specifications". In: *Design, Automation and Test in Europe (DATE)*. Grenoble, France: IEEE Computer Society, 2021.

[BrSB14] J. Brandt, K. Schneider, and Y. Bai. "Passive Code in Synchronous Programs". In: *Transactions on Embedded Computing Systems (TECS)* 13.2 (Jan. 2014), p. 67.

[Buck93] J.T. Buck. "Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model". PhD. PhD thesis. USA: University of California, 1993.

[ChMi15] E. Chovancova and J. Mihal'ov. "Load balancing strategy for multicore systems". In: *International Conference on Emerging eLearning Technologies and Applications (ICETA)*. Stary Smokovec, Slovakia: IEEE, 2015, pp. 1–6.

[Denn74] J.B. Dennis. "First Version of a Data-Flow Procedure Language". In: *Programming Symposium*. Ed. by B. Robinet. Vol. 19. LNCS. France: Springer, 1974, pp. 362–376.

[DPIC19] K. Didier, D. Potop-Butucaru, G. Iooss, A. Cohen, J. Souyris, P. Baufreton, and A. Graillat. "Correct-by-Construction Parallelization of Hard Real-Time Avionics Applications on Off-the-Shelf Predictable Hardware". In: *ACM Trans. Archit. Code Optim.* 16.3 (2019), 24:1–24:27.

[EkJa03] J. Eker and J.W. Janneck. *CAL Language Report*. ERL Technical Memo UCB/ERL M03/48. Berkeley, California, USA: EECS Department, University of California at Berkeley, Dec. 2003.

[EJLL03] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. "Taming Heterogeneity – the Ptolemy Approach". In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 127–144.

[EBLP95] M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete. "Cyclo-static dataflow". In: *International Conference on Acoustics, Speech and Signal Processing*. USA: IEEE Computer Society, 1995, pp. 3255–3258.

[FSGT15] J. Falk, T. Schwarzer, M. Glass, J. Teich, C. Zebelein, and C. Haubelt. "Quasi-static scheduling of data flow graphs in the presence of limited channel capacities". In: *IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*. Amsterdam, Netherlands: IEEE, 2015, pp. 1–10.

[FHTZ17] J. Falk, C. Haubelt, J. Teich, and C. Zebelein. "SysteMoC: A Data-Flow Programming Language for Codesign". In: *Handbook of Hardware/Software Codesign*. Springer Netherlands, 2017, pp. 59–97.

[Faus82] A.A. Faustini. "An operational semantics for pure data flow". In: *International Colloquium on Automata, Languages and Programming (ICALP)*. Ed. by M. Nielsen and E.M. Schmidt. Vol. 140. LNCS. Aarhus, Denmark: Springer, 1982, pp. 212–224.

[GeBa03] M. Geilen and T. Basten. "Requirements on the Execution of Kahn Process Networks". In: *European Symposium on Programming (ESOP)*. Ed. by P. Degano. Vol. 2618. LNCS. Warsaw, Poland: Springer, 2003, pp. 319–334.

[Gira05a] A. Girault. "A Survey of Automatic Distribution Method for Synchronous Programs". In: *Synchronous Languages, Applications, and Programming (SLAP)*. unpublished workshop proceedings. Edinburgh, Scotland, UK, 2005, pp. 1–20.

[HFKS07] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich. "A SystemC-based Design Methodology for Digital Signal Processing Systems". In: *EURASIP Journal on Embedded Systems* 2007.1 (Jan. 2007), pp. 15–15.

[HSKM08] C. Haubelt, T. Schlichter, J. Keinert, and M. Meredith. "SystemCoDesigner: automatic design space exploration and rapid prototyping from behavioral models". In: *Design Automation Conference (DAC)*. Ed. by L. Fix. Anaheim, California, USA: ACM, 2008, pp. 580–585.

[JMPR08] J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, and M. Raulet. “Synthesizing Hardware from Dataflow Programs: An MPEG-4 Simple Profile Decoder Case Study”. In: *Signal Processing Systems (SiPS)*. Washington, USA: IEEE Computer Society, 2008, pp. 287–292.

[Kahn74] G. Kahn. “The Semantics of a Simple Language for Parallel Programming”. In: *Information Processing*. Ed. by J.L. Rosenfeld. Sweden: North-Holland, 1974, pp. 471–475.

[KaMi66] R.M. Karp and R.E. Miller. “Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing”. In: *SIAM Journal on Applied Mathematics (SIAP)* 14.6 (Nov. 1966), pp. 1390–1411.

[KaMa77] G. Kahn and D.B. MacQueen. “Coroutines and networks of parallel processes”. In: *Information Processing*. Ed. by B. Gilchrist. North-Holland, 1977, pp. 993–998.

[Kosi78] P. Kosinski. “A straightforward denotational semantics for nondeterminate dataflow programs”. In: *Principles of Programming Languages (POPL)*. ACM, 1978, pp. 214–219.

[KFBG13] T. Kuhn, T. Forster, T. Braun, and R. Gotzhein. “FERAL - Framework for simulator coupling on requirements and architecture level”. In: *Formal Methods and Models for Codesign*. USA: IEEE Computer Society, 2013, pp. 11–22.

[KTPB18] V.K. Kukkala, J. Tunnell, S. Pasricha, and T. Bradley. “Advanced Driver-Assistance Systems: A Path Toward Autonomous Vehicles”. In: *IEEE Consumer Electronics Magazine* 7 (Sept. 2018), pp. 18–25.

[LNKP15] J.H. Lee, N. Nigania, H. Kim, K. Patel, and H. Kim. “OpenCL Performance Evaluation on Modern Multicore CPUs”. In: *Scientific Programming* (Jan. 2015), 859491:1–859491:20.

[LeMe87a] E.A. Lee and D.G. Messerschmitt. “Synchronous Data Flow”. In: *Proceedings of the IEEE* 75.9 (Sept. 1987), pp. 1235–1245.

[LePa95] E.A. Lee and T. Parks. “Dataflow Process Networks”. In: *Proceedings of the IEEE* 83.5 (May 1995), pp. 773–801.

[LKET15] W. Lund, S. Kanur, J. Ersfolk, L. Tsiopoulos, J. Lilius, J. Haldin, and U. Falk. “Execution of Dataflow Process Networks on OpenCL Platforms”. In: *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Finland: IEEE Computer Society, 2015, pp. 618–625.

[Park95] T. Parks. “Bounded Scheduling of Process Networks”. PhD. PhD thesis. Department of Electrical Engineering and Computer Sciences, University of California, Dec. 1995.

[PoCB06] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. "Concurrency in Synchronous Systems". In: *Formal Methods in System Design (FMSD)* 28.2 (2006), pp. 111–130.

[PiLe95] J.L. Pino and E.A. Lee. "Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors". In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE Computer Society, 1995, pp. 2643–2646.

[PSST11a] D. Potop-Butucaru, Y. Sorel, R. de Simone, and J.-P. Talpin. "From Concurrent Multi-clock Programs to Deterministic Asynchronous Implementations". In: *Fundamentae Informaticae* 107 (2011), pp. 1–28.

[PaPL95] T.M. Parks, J.L. Pino, and E.A. Lee. "A comparison of synchronous and cyclo-static dataflow". In: *Asilomar Conference on Signals, Systems and Computers*. Washington, USA: IEEE Computer Society, 1995, pp. 204–210.

[RBSY21] O. Rafique, Y. Bai, K. Schneider, and G. Yan. "Efficient Implementation of Heterogeneous Dataflow Models using Synchronous IO Patterns". In: *Euromicro Conference on Digital System Design (DSD)*. Palermo, Sicily, Italy: IEEE Computer Society, 2021, pp. 82–89.

[RBSY21a] O. Rafique, Y. Bai, K. Schneider, and G. Yan. "Synthesis of Heterogeneous Dataflow Models from Synchronous Specifications". In: *Computers, Software, and Applications Conference (COMPSAC)*. Virtual Conference: IEEE Computer Society, 2021.

[RaGS13] O. Rafique, M. Gesell, and K. Schneider. "Generating hardware specific code at different abstraction levels using Averest". In: *Int. Workshop on Software and Compilers for Embedded Systems*. Sankt Goar, Germany: ACM, 2013, pp. 90–92.

[RaGS13c] O. Rafique, M. Gesell, and K. Schneider. "Learning Various Aspects of a Distributed Real-Time Automotive Embedded System". In: *Workshop on Embedded and Cyber-Physical Systems Education (WESE)*. Ed. by P. Marwedel, J. Jackson, and K. Ricks. Canada: ACM, 2013.

[RaKS19] O. Rafique, F. Krebs, and K. Schneider. "Generating Efficient Parallel Code from the RVC-CAL Dataflow Language". In: *Euromicro Conference on Digital System Design (DSD)*. Kallithea, Chalkidiki, Greece: IEEE Computer Society, 2019.

[RWRJ08] G. Roquier, M. Wipliez, M. Raulet, J.W. Janneck, I.D. Miller, and D.B. Parlour. "Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study". In: *Signal Processing Systems (SiPS)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 281–286.

[RaSc18] O. Rafique and K. Schneider. “A Model-based Synthesis Framework for the Execution of Dynamic Dataflow Actors”. In: *Intern. Conference on Internet of Things Embedded Systems and Communications*. Hammamet, Tunisia: IEEE Computer Society, 2018.

[RaSc19] O. Rafique and K. Schneider. “Evaluating OpenCL as a Standard Hardware Abstraction for a Model-based Synthesis Framework: A Case Study”. In: *International Conference on Model Driven Engineering and Software Development (MODELSWARD)*. Prague, Czech Republic, 2019.

[RaSc19a] O. Rafique and K. Schneider. “Automatic Software Synthesis of Static and Dynamic Dataflow Process Networks”. In: *International Workshop on Interplay of Model-Driven and Component-Based Software Engineering (ModComp)*. Munich, Germany, 2019.

[RaSc20] O. Rafique and K. Schneider. “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, 2020.

[RaSc20b] O. Rafique and K. Schneider. “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, 2020.

[RaSc21] O. Rafique and K. Schneider. “Integrating Kahn Process Networks as a Model of Computation in an Extendable Model-based Design Framework”. In: *International Conference on Model Driven Engineering and Software Development (MODELSWARD)*. SCITEPRESS, 2021.

[ScBr17] Klaus Schneider and Jens Brandt. “Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems”. In: *Handbook of Hardware/Software Codesign*. Ed. by Soonhoi Ha and Jürgen Teich. Dordrecht: Springer Netherlands, 2017, pp. 29–58.

[SBRY12] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele. “Scenario-based design flow for mapping streaming applications onto on-chip many-core systems”. In: *Compilers, Architecture, and Synthesis for Embedded Systems*. Finland: ACM, 2012, pp. 71–80.

[STST13] L. Schor, A. Tretter, T. Scherer, and L. Thiele. “Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of OpenCL”. In: *IEEE Symposium on Embedded Systems for Real-time Multimedia*. IEEE Computer Society, 2013, pp. 41–50.

[Schn09] K. Schneider. *The Synchronous Programming Language Quartz.* Internal Report 375. Kaiserslautern, Germany: Department of Computer Science, University of Kaiserslautern, Dec. 2009.

[StGB06a] S. Stuijk, M. Geilen, and T. Basten. "SDF3: SDF For Free". In: *Application of Concurrency to System Design (ACSD).* Turku, Finland: IEEE Computer Society, 2006, pp. 276–278.

[StGS10] J.E. Stone, D. Gohara, and G. Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science and Engineering* 12.3 (May 2010), pp. 66–73.

[SFSV13] J. Shen, J. Fang, H. Sips, and A.L. Varbanescu. "An Application-centric Evaluation of OpenCL on Multi-core CPUs". In: *Parallel Computing* 39.12 (Dec. 2013), pp. 834–850.

[SaJA17] I. Sander, A. Jantsch, and S.-H. Attarzadeh-Niaki. "ForSyDe: System Design Using a Functional Language and Models of Computation". In: *Handbook of Hardware/Software Codesign.* Ed. by S. Ha and J. Teich. Springer, 2017. Chap. 4.

[Star87] E.W. Stark. "Concurrent transition system semantics of process networks". In: *Principles of Programming Languages (POPL).* ACM, 1987, pp. 199–210.

[SZTK04] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E.F. Deprettere. "System Design Using Kahn Process Networks: The Compaan/Laura Approach". In: *Design, Automation and Test in Europe (DATE).* Paris, France: IEEE Computer Society, 2004, pp. 340–345.

[SGTB11] S. Stuijk, M. Geilen, B.D. Theelen, and T. Basten. "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications". In: *Intern. Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation.* Samos, Greece: IEEE Computer Society, 2011, pp. 404–411.

[TOGB12] J.-P. Talpin, J. Ouy, T. Gautier, L. Besnard, and P. Le Guernic. "Compositional design of isochronous systems". In: *Science of Computer Programming* 77.2 (Feb. 2012), pp. 113–128.

[Tarj72] Robert E. Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM J. Comput.* 1 (1972), pp. 146–160.

[Trap05] M. Trapp. "Modeling the Adaptation Behavior of Adaptive Embedded Systems". PhD thesis. University of Kaiserslautern, Germany, 2005.

[TBGR13] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E.A. Lee. "Compositionality in Synchronous Data Flow: Modular Code Generation from Hierarchical SDF Graphs". In: *Transactions on Embedded Computing Systems (TECS)* 12.3 (Mar. 2013), 83:1–83:26.

[WiRe93] M. H. Willebeek-LeMair and A. P. Reeves. “Strategies for dynamic load balancing on highly parallel computers”. In: *IEEE Transactions on Parallel and Distributed Systems* 4.9 (Mar. 1993), pp. 979–993.

[YLJC13a] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet. “ORCC: multimedia development made easy”. In: *Intern. conference on Multimedia*. Barcelona, Spain: ACM, 2013, pp. 863–866.

[ZhGW09] J. Guo Y. Zhu and Y. Wang. “Study on Dynamic Load Balancing Algorithm Based on MPICH”. In: *WRI World Congress on Software Engineering*. Xiamen, China: IEEE, 2009, pp. 103–107.

Appendix A

Standalone Benchmarks

A.1. Benchmarks

The standalone benchmarks as presented in Section 6.3 are especially designed to offer different kinds of behaviors of the processes. We present here the model and the generated code of particular benchmarks covering all three supported dataflow MoCs of the framework. We thereby consider showing mainly the worker process model and the corresponding generated kernel code of the following benchmarks: the *StITE* benchmark consisting of static processes, the *SeqDySplit* benchmark offering sequential processes, and the *ParDyOR* benchmark involving processes with parallel behaviors.

A.1.1. StITE Benchmark

```
1 actor ite() int X1, int X2, int X3 ==> int Y :
2
3 act1: action X1:[x1], X2:[x2], X3:[x3] ==> Y:[y]
4   guard x1 >= 0
5   do
6     y := x2;
7   end
8 act2: action X1:[x1], X2:[x2], X3:[x3] ==> Y:[y]
9   guard x1 < 0
10  do
11    y := x3;
12  end
13 end
```

Listing A.1: *Worker process of StITE Benchmark*

Generated Kernel Code

SDF Kernel

```
1  __kernel void ite ( __global fifo_t* X1 , __global fifo_t* X2
       , __global fifo_t* X3, __global fifo_t* Y, int blockSize)
2  {
3    /*Generate Declarations for All Inputs*/
4    __private int seq_X1[1];
5    __private int seq_X2[1];
6    __private int seq_X3[1];
7    int* x1_act1 = &seq_X1[0];
8    int* x1_act2 = &seq_X1[0];
9    int* x2_act1 = &seq_X2[0];
10   int* x2_act2 = &seq_X2[0];
11   int* x3_act1 = &seq_X3[0];
12   int* x3_act2 = &seq_X3[0];
13   /*Generate Declarations for All Outputs*/
14   __private int seq_Y[1];
15   int* y_act1 = &seq_Y[0];
16   int* y_act2 = &seq_Y[0];
17   /*Generate Generic Kernel Code*/
18   __private dynamicCount cnt_X1  = { .instance_count = 0,.
         current_count = 0 };
19   X1->stat_updated = 0;
20   __private dynamicCount cnt_X2  = { .instance_count = 0,.
         current_count = 0 };
21   X2->stat_updated = 0;
22   __private dynamicCount cnt_X3  = { .instance_count = 0,.
         current_count = 0 };
23   X3->stat_updated = 0;
24   __private dynamicCount cnt_Y  = { .instance_count = 0,.
         current_count = 0 };
25   Y->stat_updated = 0;
26   int bytes;
27   int gid = get_global_id(0) * blockSize;
28   for (int x = 0; x < blockSize; x++) {
29     /*Generate SDF Specific Kernel Code*/
30     fifoRead(X1, seq_X1, 1, gid, &cnt_X1);
31     fifoRead(X2, seq_X2, 1, gid, &cnt_X2);
32     fifoRead(X3, seq_X3, 1, gid, &cnt_X3);
33     if(*x1_act1 >= 0 ) {
34       *y_act1 = *x2_act1;
35       fifoWrite(Y, seq_Y, 1, gid, &cnt_Y);
36     }
37     else if(*x1_act2 < 0 ) {
38       *y_act2 = *x3_act2;
39       fifoWrite(Y, seq_Y, 1, gid, &cnt_Y);
40     }
41     updateDynamicCount(&cnt_X1);
42     updateDynamicCount(&cnt_X2);
43     updateDynamicCount(&cnt_X3);
44     updateDynamicCount(&cnt_Y);
45   }
46 }
```

Listing A.2: *Generated SDF kernel for StITE Benchmark*

KPN Kernel

```
1 __kernel void ite ( __global fifo_t* X1 , __global fifo_t* X2
      , __global fifo_t* X3, __global fifo_t* Y, __global int*
      evaluatedGuard, int blockSize)
2 {
3   /*Generate Declarations for All Inputs*/
4   __private int seq_X1[1];
5   int* x1_act1 = &seq_X1[0];
6   int* x1_act2 = &seq_X1[0];
7   __private int seq_X2[1];
8   __private int seq_X3[1];
9   int* x2_act1 = &seq_X2[0];
10  int* x2_act2 = &seq_X2[0];
11  int* x3_act1 = &seq_X3[0];
12  int* x3_act2 = &seq_X3[0];
13  /*Generate Declarations for All Outputs*/
14  __private int seq_Y[1];
15  int* y_act1 = &seq_Y[0];
16  int* y_act2 = &seq_Y[0];
17  /*Generate Generic Kernel Code*/
18  __private dynamicCount cnt_X1  = { .instance_count = 0,.
        current_count = 0 };
19  X1->stat_updated = 0;
20  __private dynamicCount cnt_X2  = { .instance_count = 0,.
        current_count = 0 };
21  X2->stat_updated = 0;
22  __private dynamicCount cnt_X3  = { .instance_count = 0,.
        current_count = 0 };
23  X3->stat_updated = 0;
24  __private dynamicCount cnt_Y  = { .instance_count = 0,.
        current_count = 0 };
25  Y->stat_updated = 0;
26  int gid = get_global_id(0) * blockSize;
27  for (int x = 0; x < blockSize; x++) {
28    /*Generate KPN Specific Kernel Code*/
29    if(evaluatedGuard[x] == 0) {
30      fifoRead(X1, seq_X1, 1, gid, &cnt_X1);
31      fifoRead(X2, seq_X2, 1, gid, &cnt_X2);
32      fifoRead(X3, seq_X3, 1, gid, &cnt_X3);
33      *y_act1 = *x2_act1;
34      fifoWrite(Y, seq_Y, 1, gid, &cnt_Y);
35    }
36    else if(evaluatedGuard[x] == 1) {
37      fifoRead(X1, seq_X1, 1, gid, &cnt_X1);
38      fifoRead(X2, seq_X2, 1, gid, &cnt_X2);
39      fifoRead(X3, seq_X3, 1, gid, &cnt_X3);
40      *y_act2 = *x3_act2;
41      fifoWrite(Y, seq_Y, 1, gid, &cnt_Y);
42    }
43    updateDynamicCount(&cnt_X1);
44    updateDynamicCount(&cnt_X2);
45    updateDynamicCount(&cnt_X3);
46    updateDynamicCount(&cnt_Y);
47  }}
```

Listing A.3: *Generated KPN kernel for StITE Benchmark*

DDF Kernel

```
1  __kernel void ite ( __global fifo_t* X1 , __global fifo_t* X2
       , __global fifo_t* X3, __global fifo_t* Y, int blockSize)
2  {
3    /*Generate Declarations for All Inputs*/
4    __private int seq_X1[1];
5    int* x1_act1 = &seq_X1[0];
6    int* x1_act2 = &seq_X1[0];
7    __private int seq_X2[1];
8    __private int seq_X3[1];
9    int* x2_act1 = &seq_X2[0];
10   int* x2_act2 = &seq_X2[0];
11   int* x3_act1 = &seq_X3[0];
12   int* x3_act2 = &seq_X3[0];
13   /*Generate Declarations for All Outputs*/
14   __private int seq_Y[1];
15   int* y_act1 = &seq_Y[0];
16   int* y_act2 = &seq_Y[0];
17   bool guard_act1;
18   bool guard_act2;
19   char eval_impl_act1;
20   char eval_impl_act2;
21   bool result = false;
22   /*Generate Generic Kernel Code*/
23   __private dynamicCount cnt_X1  = { .instance_count = 0,.
         current_count = 0 };
24   X1->stat_updated = 0;
25   __private dynamicCount cnt_X2  = { .instance_count = 0,.
         current_count = 0 };
26   X2->stat_updated = 0;
27   __private dynamicCount cnt_X3  = { .instance_count = 0,.
         current_count = 0 };
28   X3->stat_updated = 0;
29   __private dynamicCount cnt_Y  = { .instance_count = 0,.
         current_count = 0 };
30   Y->stat_updated = 0;
31   int bytes;
32   int gid = get_global_id(0) * blockSize;
33   for (int x = 0; x < blockSize; x++) {
34     /*Generate DDF Specific Kernel Code */
35     bool ctrl_X1_act1 = fifoPeek(X1, seq_X1, 1, gid, &cnt_X1);
36     bool ctrl_X1_act2 = fifoPeek(X1, seq_X1, 1, gid, &cnt_X1);
37     bool ctrl_X2_act1 = fifoPeek(X2, seq_X2, 1, gid, &cnt_X2);
38     bool ctrl_X2_act2 = fifoPeek(X2, seq_X2, 1, gid, &cnt_X2);
39     bool ctrl_X3_act1 = fifoPeek(X3, seq_X3, 1, gid, &cnt_X3);
40     bool ctrl_X3_act2 = fifoPeek(X3, seq_X3, 1, gid, &cnt_X3);
41     bool ctrl_Y_act1 = fifoSpace(Y, 0, Y->tail+gid, &cnt_Y,1);
42     bool ctrl_Y_act2 = fifoSpace(Y, 0, Y->tail+gid, &cnt_Y,1);
43     if(ctrl_X1_act1){
44       guard_act1 = (*x1_act1 >= 0 );
45       eval_impl_act1 = evalImplication(guard_act1,
           ctrl_X1_act1 & ctrl_X2_act1 & ctrl_X3_act1 &
           ctrl_Y_act1);
46     }
47     else{
```

```
48        fifoWriteStatus(X1, 1, X1->head + gid, &cnt_X1, 1);
49      }
50      if(eval_impl_act1 == '1'){
51        fifoReadDDF(X1, 4, X1->head + gid, &cnt_X1, 1);
52        fifoReadDDF(X2, 4, X2->head + gid, &cnt_X2, 1);
53        fifoReadDDF(X3, 4, X3->head + gid, &cnt_X3, 1);
54        *y_act1 = *x2_act1;
55        bytes = fifoWriteDDF(Y, seq_Y, 1, gid, &cnt_Y);
56      }
57      else if(eval_impl_act1 == '0'){
58        fifoWriteStatus(X1, 1, X1->head + gid, &cnt_X1, 1);
59        fifoWriteStatus(X2, 1, X2->head + gid, &cnt_X2, 1);
60        fifoWriteStatus(X3, 1, X3->head + gid, &cnt_X3, 1);
61        fifoWriteStatus(Y, 2, Y->tail + gid, &cnt_Y, 1);
62      }
63      if(ctrl_X1_act2){
64        guard_act2 = (*x1_act2 < 0 );
65        eval_impl_act2 = evalImplication(guard_act2,
              ctrl_X1_act2 & ctrl_X2_act2 & ctrl_X3_act2 &
              ctrl_Y_act2);
66      }
67      else{
68        fifoWriteStatus(X1, 1, X1->head + gid, &cnt_X1, 1);
69      }
70      if(eval_impl_act2 == '1'){
71        fifoReadDDF(X1, 4, X1->head + gid, &cnt_X1, 1);
72        fifoReadDDF(X2, 4, X2->head + gid, &cnt_X2, 1);
73        fifoReadDDF(X3, 4, X3->head + gid, &cnt_X3, 1);
74        *y_act2 = *x3_act2;
75        bytes = fifoWriteDDF(Y, seq_Y, 1, gid, &cnt_Y);

76      }
77      else if(eval_impl_act2 == '0'){
78        fifoWriteStatus(X1, 1, X1->head + gid, &cnt_X1, 1);
79        fifoWriteStatus(X2, 1, X2->head + gid, &cnt_X2, 1);
80        fifoWriteStatus(X3, 1, X3->head + gid, &cnt_X3, 1);
81        fifoWriteStatus(Y, 2, Y->tail + gid, &cnt_Y, 1);
82      }
83      updateDynamicCount(&cnt_X1);
84      updateDynamicCount(&cnt_X2);
85      updateDynamicCount(&cnt_X3);
86      updateDynamicCount(&cnt_Y);
87      eval_impl_act1 = 'u';
88      eval_impl_act2 = 'u';
89    }
90  }
```

Listing A.4: *Generated DDF kernel for StITE Benchmark*

A.1.2. SeqDySplit Benchmark

```
1 actor split() int X1, int X2 ==> int Y1, int Y2 :
2
3 act1: action X1:[x1], X2:[x2] ==> Y1:[y1]
4   guard x1 = 1
5   do
6     y1 := x2;
7   end
8 act2: action X1:[x1], X2:[x2a, x2b] ==> Y1:[y1], Y2:[y2]
9   guard x1 = 2
10  do
11    y1 := x2a;
12    y2 := x2b;
13  end
14 end
```

Listing A.5: *Worker process of SeqDySplit Benchmark*

Generated Kernel Code

KPN Kernel

```
1 __kernel void split ( __global fifo_t* X1 , __global fifo_t*
      X2, __global fifo_t* Y1 , __global fifo_t* Y2, __global
      int* evaluatedGuard, int blockSize)
2 {
3   /*Generate Declarations for All Inputs*/
4   __private int seq_X1[1];
5   int* x1_act1 = &seq_X1[0];
6   int* x1_act2 = &seq_X1[0];
7   __private int seq_X2[2];
8   int* x2_act1 = &seq_X2[0];
9   int* x2a_act2 = &seq_X2[0];
10  int* x2b_act2 = &seq_X2[1];
11  /*Generate Declarations for All Outputs*/
12  __private int seq_Y1[1];
13  __private int seq_Y2[1];
14  int* y1_act1 = &seq_Y1[0];
15  int* y1_act2 = &seq_Y1[0];
16  int* y2_act2 = &seq_Y2[0];
17  /*Generate Generic Kernel Code*/
18  __private dynamicCount cnt_X1  = { .instance_count = 0,.
        current_count = 0 };
19  X1->stat_updated = 0;
20  __private dynamicCount cnt_X2  = { .instance_count = 0,.
        current_count = 0 };
21  X2->stat_updated = 0;
22  __private dynamicCount cnt_Y1  = { .instance_count = 0,.
        current_count = 0 };
23  Y1->stat_updated = 0;
24  __private dynamicCount cnt_Y2  = { .instance_count = 0,.
        current_count = 0 };
25  Y2->stat_updated = 0;
26  int bytes;
```

```
27    int gid = get_global_id(0) * blockSize;
28    for (int x = 0; x < blockSize; x++) {
29      /*Generate KPN Specific Kernel Code */
30      if(evaluatedGuard[x] == 0) {
31        fifoRead(X1, seq_X1, 1, gid, &cnt_X1);
32        fifoRead(X2, seq_X2, 1, gid, &cnt_X2);
33        *y1_act1 = *x2_act1;
34        fifoWrite(Y1, seq_Y1, 1, gid, &cnt_Y1);
35      }
36      else if(evaluatedGuard[x] == 1) {
37        fifoRead(X1, seq_X1, 1, gid, &cnt_X1);
38        fifoRead(X2, seq_X2, 2, gid, &cnt_X2);
39        *y1_act2 = *x2a_act2;
40        *y2_act2 = *x2b_act2;
41        fifoWrite(Y1, seq_Y1, 1, gid, &cnt_Y1);
42        fifoWrite(Y2, seq_Y2, 1, gid, &cnt_Y2);
43      }
44      updateDynamicCount(&cnt_X1);
45      updateDynamicCount(&cnt_X2);
46      updateDynamicCount(&cnt_Y1);
47      updateDynamicCount(&cnt_Y2);
48    }
49  }
```

Listing A.6: *Generated KPN kernel for SeqDySplit Benchmark*

DDF Kernel

```
1  __kernel void split ( __global fifo_t* X1 , __global fifo_t*
       X2, __global fifo_t* Y1 , __global fifo_t* Y2, int
       blockSize)
2  {
3    /*Generate Declarations for All Inputs*/
4    __private int seq_X1[1];
5    int* x1_act1 = &seq_X1[0];
6    int* x1_act2 = &seq_X1[0];
7    __private int seq_X2[2];
8    int* x2_act1 = &seq_X2[0];
9    int* x2a_act2 = &seq_X2[0];
10   int* x2b_act2 = &seq_X2[1];
11   /*Generate Declarations for All Outputs*/
12   __private int seq_Y1[1];
13   __private int seq_Y2[1];
14   int* y1_act1 = &seq_Y1[0];
15   int* y1_act2 = &seq_Y1[0];
16   int* y2_act2 = &seq_Y2[0];
17   bool guard_act1;
18   bool guard_act2;
19   char eval_impl_act1;
20   char eval_impl_act2;
21   bool result = false;
22   /*Generate Generic Kernel Code*/
23   __private dynamicCount cnt_X1  = { .instance_count = 0,.
         current_count = 0 };
24   X1->stat_updated = 0;
25   __private dynamicCount cnt_X2  = { .instance_count = 0,.
         current_count = 0 };
26   X2->stat_updated = 0;
27   __private dynamicCount cnt_Y1  = { .instance_count = 0,.
         current_count = 0 };
28   Y1->stat_updated = 0;
29   __private dynamicCount cnt_Y2  = { .instance_count = 0,.
         current_count = 0 };
30   Y2->stat_updated = 0;
31   int bytes;
32   int gid = get_global_id(0) * blockSize;
33   for (int x = 0; x < blockSize; x++) {
34     /*Generate DDF Specific Kernel Code */
35     bool ctrl_X1_act1 = fifoPeek(X1, seq_X1, 1, gid, &cnt_X1);
36     bool ctrl_X1_act2 = fifoPeek(X1, seq_X1, 1, gid, &cnt_X1);
37     bool ctrl_X2_act1 = fifoPeek(X2, seq_X2, 1, gid, &cnt_X2);
38     bool ctrl_X2_act2 = fifoPeek(X2, seq_X2, 2, gid, &cnt_X2);
39     bool ctrl_Y1_act1= fifoSpace(Y1,0,Y1->tail+gid,&cnt_Y1,1);
40     bool ctrl_Y1_act2= fifoSpace(Y1,0,Y1->tail+gid,&cnt_Y1,1);
41     bool ctrl_Y2_act2= fifoSpace(Y2,0,Y2->tail+gid,&cnt_Y2,1);
42     if(ctrl_X1_act1){
43       guard_act1 = (*x1_act1 == 1 );
44       eval_impl_act1 = evalImplication(guard_act1,
           ctrl_X1_act1 & ctrl_X2_act1 & ctrl_Y1_act1);
45     }
46     else{
47       fifoWriteStatus(X1, 1, X1->head + gid, &cnt_X1, 1);
```

```
48      }
49      if(eval_impl_act1 == '1'){
50        fifoReadDDF(X1, 4, X1->head + gid, &cnt_X1, 1);
51        fifoReadDDF(X2, 4, X2->head + gid, &cnt_X2, 1);
52        *y1_act1 = *x2_act1;
53        bytes = fifoWriteDDF(Y1, seq_Y1, 1, gid, &cnt_Y1);

54      }
55      else if(eval_impl_act1 == '0'){
56        fifoWriteStatus(X1, 1, X1->head + gid, &cnt_X1, 1);
57        fifoWriteStatus(X2, 1, X2->head + gid, &cnt_X2, 1);
58        fifoWriteStatus(Y1, 2, Y1->tail + gid, &cnt_Y1, 1);
59      }
60      if(ctrl_X1_act2){
61        guard_act2 = (*x1_act2 == 2 );
62        eval_impl_act2 = evalImplication(guard_act2,
              ctrl_X1_act2 & ctrl_X2_act2 & ctrl_Y1_act2 &
              ctrl_Y2_act2);
63      }
64      else{
65        fifoWriteStatus(X1, 1, X1->head + gid, &cnt_X1, 1);
66      }
67      if(eval_impl_act2 == '1'){
68        fifoReadDDF(X1, 4, X1->head + gid, &cnt_X1, 1);
69        fifoReadDDF(X2, 4, X2->head + gid, &cnt_X2, 2);
70        *y1_act2 = *x2a_act2;
71        *y2_act2 = *x2b_act2;
72        bytes = fifoWriteDDF(Y1, seq_Y1, 1, gid, &cnt_Y1);
73        bytes = fifoWriteDDF(Y2, seq_Y2, 1, gid, &cnt_Y2);

74      }
75      else if(eval_impl_act2 == '0'){
76        fifoWriteStatus(X1, 1, X1->head + gid, &cnt_X1, 1);
77        fifoWriteStatus(X2, 1, X2->head + gid, &cnt_X2, 2);
78        fifoWriteStatus(Y1, 2, Y1->tail + gid, &cnt_Y1, 1);
79        fifoWriteStatus(Y2, 2, Y2->tail + gid, &cnt_Y2, 1);
80      }
81      updateDynamicCount(&cnt_X1);
82      updateDynamicCount(&cnt_X2);
83      updateDynamicCount(&cnt_Y1);
84      updateDynamicCount(&cnt_Y2);
85      eval_impl_act1 = 'u';
86      eval_impl_act2 = 'u';
87    }
88  }
```

Listing A.7: *Generated DDF kernel for SeqDySplit Benchmark*

A.1.3. ParDyOR Benchmark

```
1 actor POR() bool X1, bool X2 ==> bool Y :
2
3 act1: action X1:[x1] ==> Y:[y]
4   guard x1 = true
5   do
6     y := true;
7   end
8 act2: action X2:[x2] ==> Y:[y]
9   guard x2 = true
10  do
11    y := true;
12  end
13 act3: action X1:[x1], X2:[x2] ==> Y:[y]
14  guard x1 = false and x2 = false
15  do
16    y := false;
17  end
18 end
```

Listing A.8: *Worker process of ParDyOR Benchmark*

Generated Kernel Code

DDF Kernel

```
1 __kernel void POR ( __global fifo_t* X1 , __global fifo_t* X2,
      __global fifo_t* Y, int blockSize)
2 {
3   /*Generate Declarations for All Inputs*/
4   __private bool seq_X1[1];
5   __private bool seq_X2[1];
6   bool* x1_act1 = &seq_X1[0];
7   bool* x1_act3 = &seq_X1[0];
8   bool* x2_act2 = &seq_X2[0];
9   bool* x2_act3 = &seq_X2[0];
10  /*Generate Declarations for All Outputs*/
11  __private bool seq_Y[1];
12  bool* y_act1 = &seq_Y[0];
13  bool* y_act2 = &seq_Y[0];
14  bool* y_act3 = &seq_Y[0];
15  bool guard_act1;
16  bool guard_act2;
17  bool guard_act3;
18  char eval_impl_act1;
19  char eval_impl_act2;
20  char eval_impl_act3;
21  bool result = false;
22  /*Generate Generic Kernel Code*/
23  __private dynamicCount cnt_X1  = { .instance_count = 0,.
      current_count = 0 };
24  X1->stat_updated = 0;
25  __private dynamicCount cnt_X2  = { .instance_count = 0,.
      current_count = 0 };
```

```
26   X2->stat_updated = 0;
27   __private dynamicCount cnt_Y  = { .instance_count = 0,.
         current_count = 0 };
28   Y->stat_updated = 0;
29   int bytes;
30   int gid = get_global_id(0) * blockSize;
31   for (int x = 0; x < blockSize; x++) {
32     /*Generate DDF Specific Kernel Code */
33     bool ctrl_X1_act1 = fifoPeek(X1, seq_X1, 1, gid, &cnt_X1);
34     bool ctrl_X1_act3 = fifoPeek(X1, seq_X1, 1, gid, &cnt_X1);
35     bool ctrl_X2_act2 = fifoPeek(X2, seq_X2, 1, gid, &cnt_X2);
36     bool ctrl_X2_act3 = fifoPeek(X2, seq_X2, 1, gid, &cnt_X2);
37     bool ctrl_Y_act1 = fifoSpace(Y, 0, Y->tail+gid, &cnt_Y,1);
38     bool ctrl_Y_act2 = fifoSpace(Y, 0, Y->tail+gid, &cnt_Y,1);
39     bool ctrl_Y_act3 = fifoSpace(Y, 0, Y->tail+gid, &cnt_Y,1);
40     if(ctrl_X1_act1){
41       guard_act1 = (*x1_act1 == true );
42       eval_impl_act1 = evalImplication(guard_act1,
             ctrl_X1_act1 & ctrl_Y_act1);
43     }
44     else{
45       fifoWriteStatus(X1, 1, X1->head + gid, &cnt_X1, 1);
46     }
47     if(eval_impl_act1 == '1'){
48       fifoReadDDF(X1, 4, X1->head + gid, &cnt_X1, 1);
49       *y_act1 = true;
50       bytes = fifoWriteDDF(Y, seq_Y, 1, gid, &cnt_Y);

51     }
52     else if(eval_impl_act1 == '0'){
53       fifoWriteStatus(X1, 1, X1->head + gid, &cnt_X1, 1);
54       fifoWriteStatus(Y, 2, Y->tail + gid, &cnt_Y, 1);
55     }
56     if(ctrl_X2_act2){
57       guard_act2 = (*x2_act2 == true );
58       eval_impl_act2 = evalImplication(guard_act2,
             ctrl_X2_act2 & ctrl_Y_act2);
59     }
60     else{
61       fifoWriteStatus(X2, 1, X2->head + gid, &cnt_X2, 1);
62     }
63     if(eval_impl_act2 == '1'){
64       fifoReadDDF(X2, 4, X2->head + gid, &cnt_X2, 1);
65       *y_act2 = true;
66       bytes = fifoWriteDDF(Y, seq_Y, 1, gid, &cnt_Y);

67     }
68     else if(eval_impl_act2 == '0'){
69       fifoWriteStatus(X2, 1, X2->head + gid, &cnt_X2, 1);
70       fifoWriteStatus(Y, 2, Y->tail + gid, &cnt_Y, 1);
71     }
72     if(ctrl_X1_act3 & ctrl_X2_act3){
73       guard_act3 = (*x1_act3 == false & *x2_act3 == false );
74       eval_impl_act3 = evalImplication(guard_act3,
             ctrl_X1_act3 & ctrl_X2_act3 & ctrl_Y_act3);
75     }
76     else{
```

```
77        fifoWriteStatus(X1, 1, X1->head + gid, &cnt_X1, 1);
78        fifoWriteStatus(X2, 1, X2->head + gid, &cnt_X2, 1);
79      }
80      if(eval_impl_act3 == '1'){
81        fifoReadDDF(X1, 4, X1->head + gid, &cnt_X1, 1);
82        fifoReadDDF(X2, 4, X2->head + gid, &cnt_X2, 1);
83        *y_act3 = false;
84        bytes = fifoWriteDDF(Y, seq_Y, 1, gid, &cnt_Y);
85      }
86      else if(eval_impl_act3 == '0'){
87        fifoWriteStatus(X1, 1, X1->head + gid, &cnt_X1, 1);
88        fifoWriteStatus(X2, 1, X2->head + gid, &cnt_X2, 1);
89        fifoWriteStatus(Y, 2, Y->tail + gid, &cnt_Y, 1);
90      }
91      updateDynamicCount(&cnt_X1);
92      updateDynamicCount(&cnt_X2);
93      updateDynamicCount(&cnt_Y);
94      eval_impl_act1 = 'u';
95      eval_impl_act2 = 'u';
96      eval_impl_act3 = 'u';
97    }
98 }
```

Listing A.9: *Generated DDF kernel for ParDyOR Benchmark*

A.2. Results: All Benchmarks

A.2.1. StOR

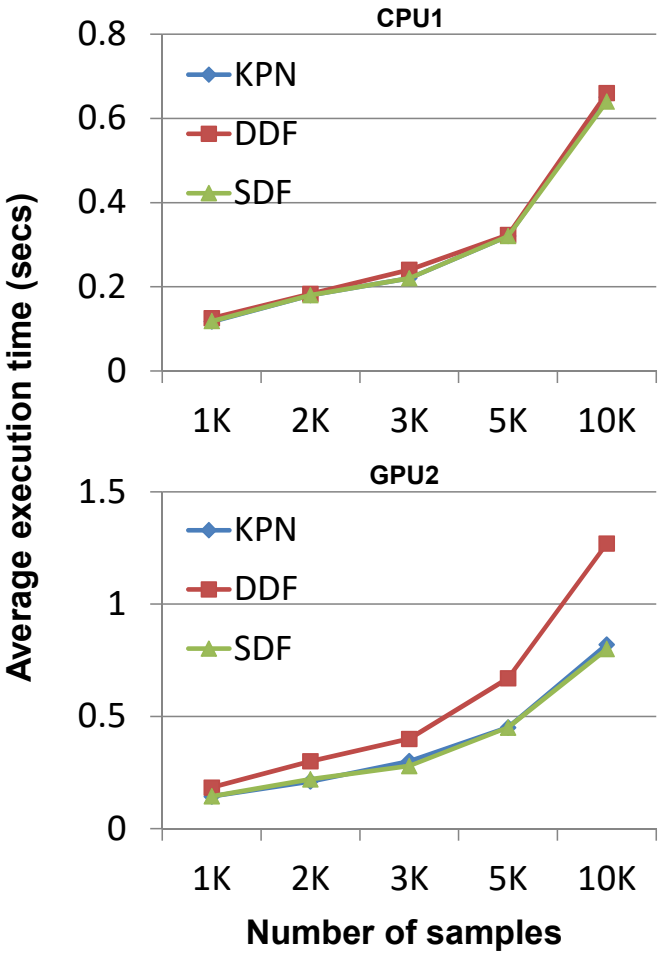

Figure A.1.: *StOR: number of samples vs end-to-end performance.*

A.2.2. StITE

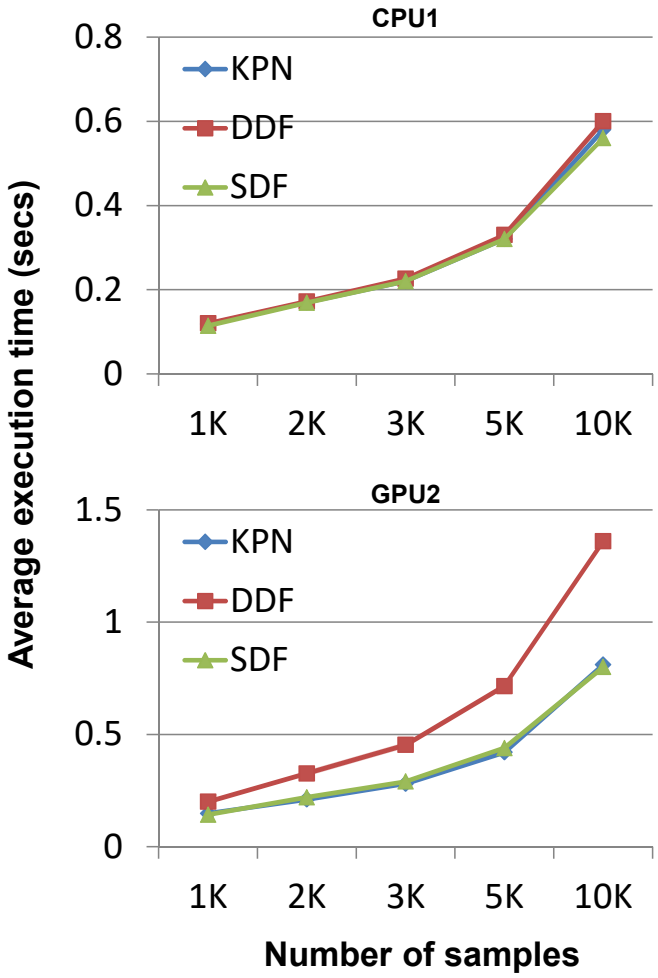

Figure A.2.: *StITE: number of samples vs end-to-end performance.*

A.2.3. SeqDySwitch

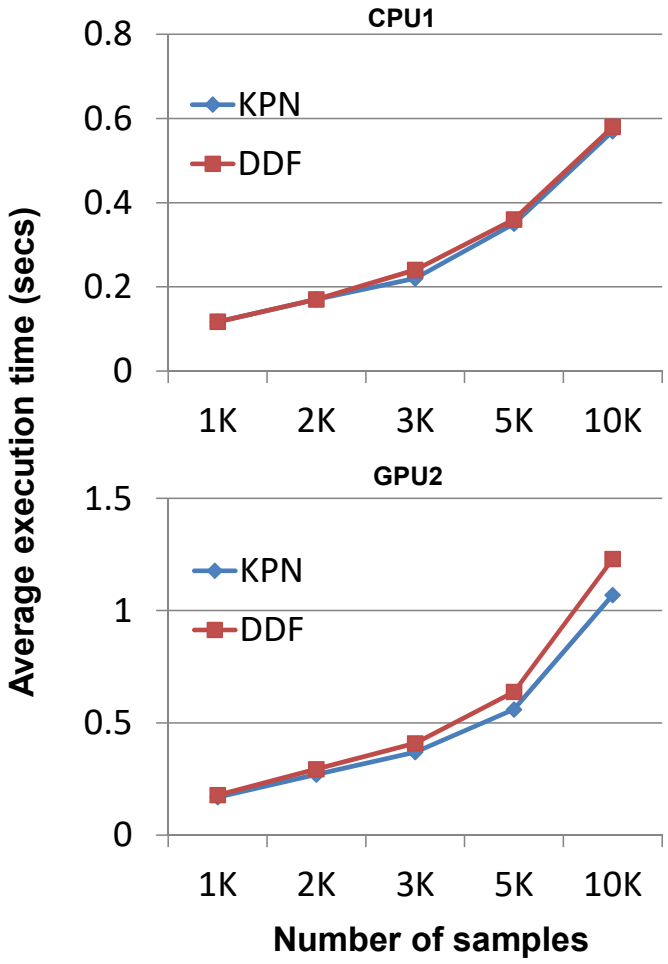

Figure A.3.: *SeqDySwitch: number of samples vs end-to-end performance.*

A.2.4. SeqDySelect

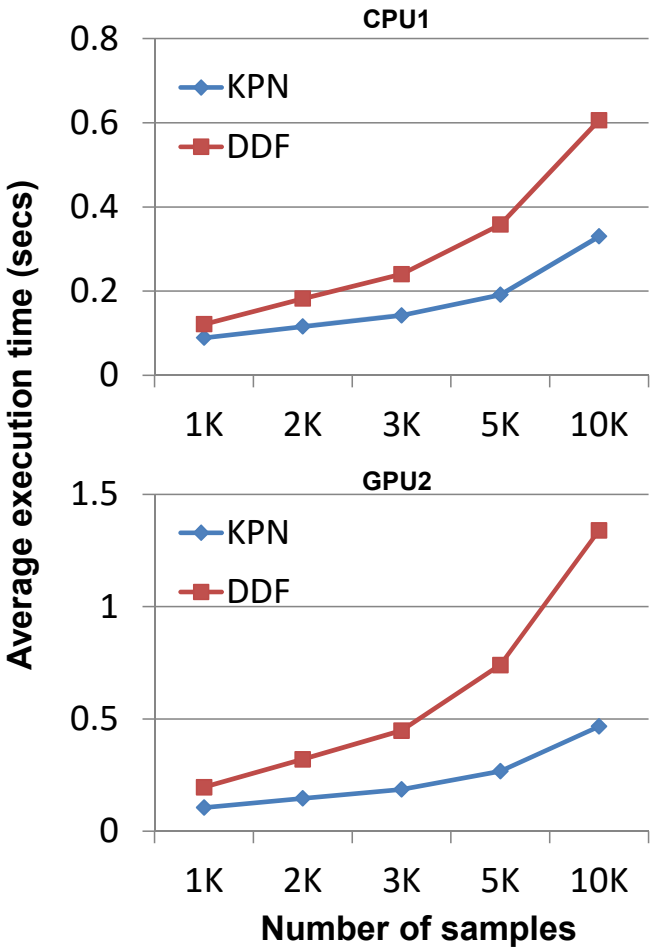

Figure A.4.: *SeqDySelect: number of samples vs end-to-end performance.*

A.2.5. SeqDyWorker

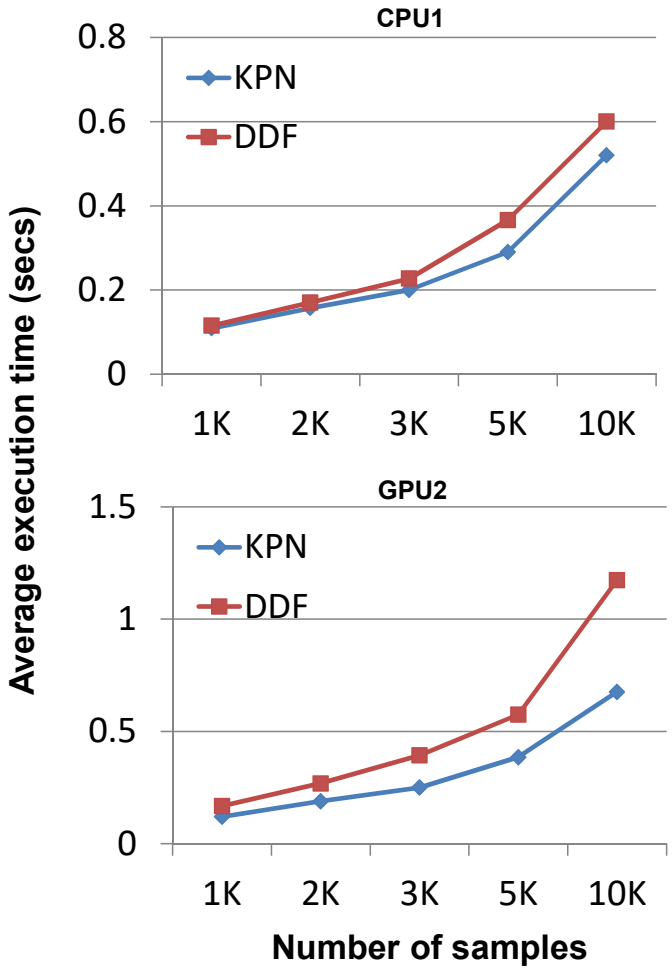

Figure A.5.: *SeqDyWorker: number of samples vs end-to-end performance.*

A.2.6. SeqDySplit

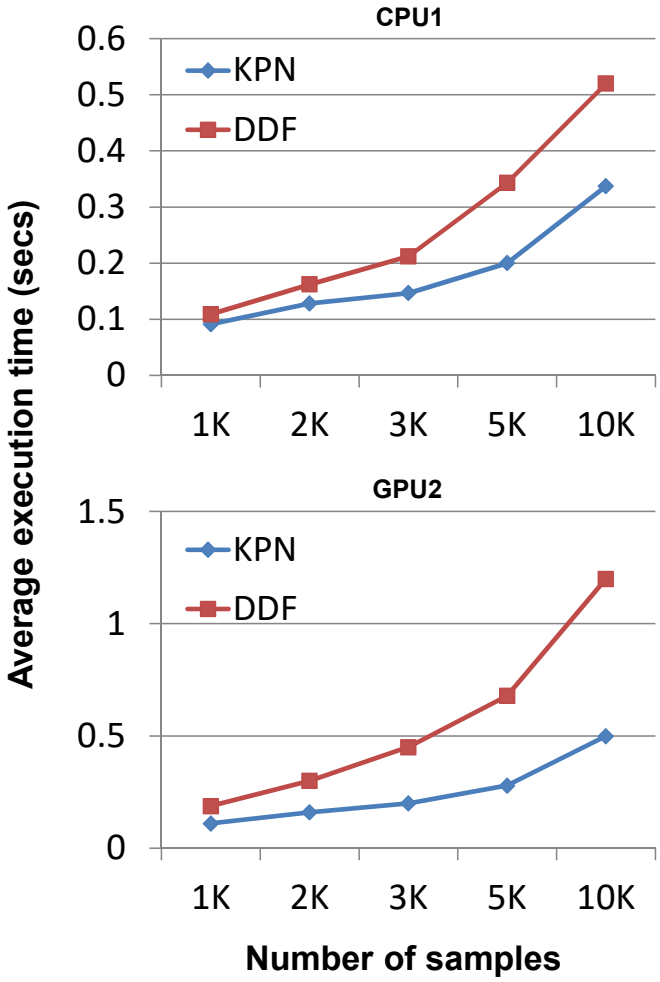

Figure A.6.: *SeqDySplit: number of samples vs end-to-end performance.*

A.2.7. SeqDyMerge

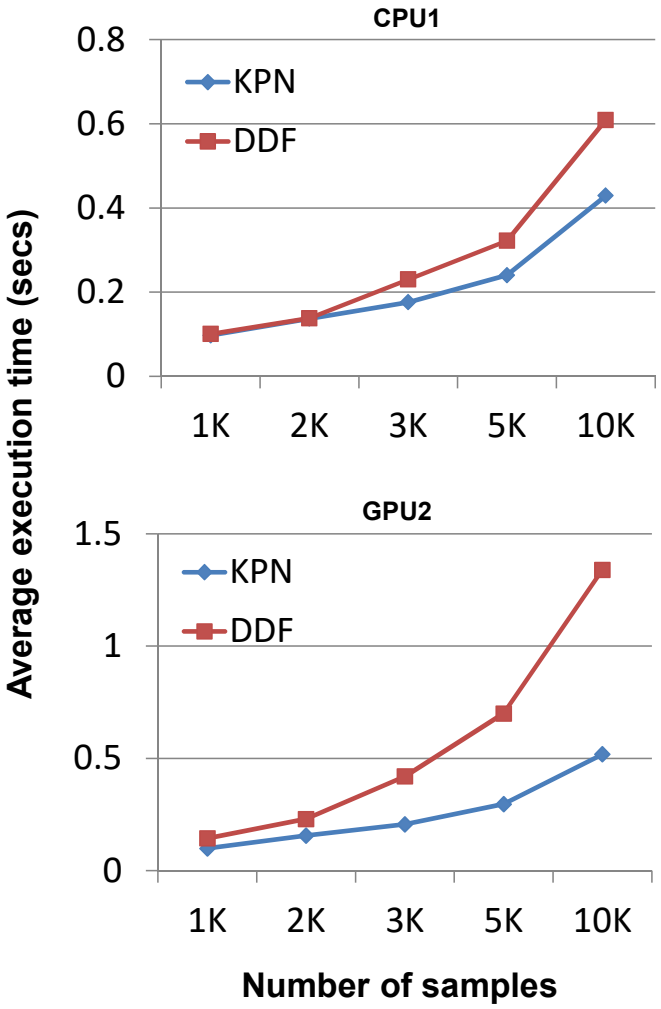

Figure A.7.: *SeqDyMerge: number of samples vs end-to-end performance.*

A.2.8. ParDyOR

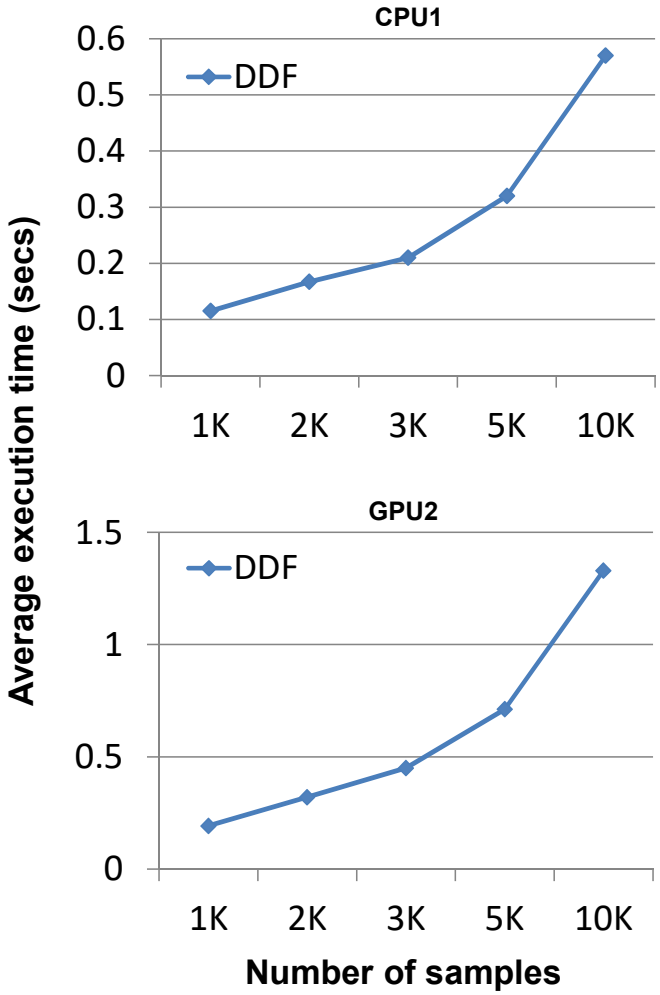

Figure A.8.: *ParDyOR: number of samples vs end-to-end performance.*

Appendix B

Curriculum Vitae

Berufserfahrung

2013–2015 **Embedded Software Engineer** Daimler AG / EDAG Engineering
Daimler Trucks Product Engineering

2011–2012 **Internee** John Deere Innovation Center Kaiserslautern
Advanced Engineering Group

2009–2010 **Offshore Electronics Engineer** Courseware Solutions International, Inc
Embedded Systems Development

Akademische Ausbildung

2010–2013 **MSc in Electrical and Computer Engineering 1,2**
TU Kaiserslautern
Masterarbeit: *Design, Development and Integration of a Wireless Communication Unit in the ConceptCar*

2005–2008 **Bachelors of Engineering 78%**
IIEE (NED) Karachi
Bachelorarbeit: *Industrial Building Automation System.*

Schulausbildung

1993-2002 **High School** Happy Home, Karachi
Majors: Maths, Physics, Chemistry.