Degree Project in Technology

Second cycle, 30 credits

Automated Design Space Exploration for Hardware and Software Implementations on Heterogeneous MPSoCs

LUDVIG LARSSON

Automated Design Space Exploration for Hardware and Software Implementations on Heterogeneous MPSoCs

LUDVIG LARSSON

Master's Programme, Embedded Systems, 120 credits
Date: July 9, 2024

Supervisors: Fahimeh Bahrami, Ludwig Karlsson
Examiner: Ingo Sander
School of Electrical Engineering and Computer Science
Host company: Saab AB
Swedish title: Automatiserad Designrymdsutforskning för Hårdvaru- och Mjukvarumimplementationer på Heterogena Multiprocessorsystemschip

Abstract

Identifying embedded system design solutions that utilize platform resources effectively is often difficult. Due to the demands of modern applications and the complexity of heterogeneous platforms, tailoring the system design to maximize throughput, minimize power consumption, or meet other design goals may pose a highly demanding task for system designers. Design Space Exploration (DSE) is a technique commonly applied within the embedded system design process that utilizes a systematic search to find optimal design solutions. These solutions include where applications should be executed, how applications should be scheduled, and where application data should be stored. The design space is the core of DSE, defining the possible design alternatives based on platforms and application specifications.

This thesis explores how DSE can evaluate design spaces where applications are mapped to hardware- or software-programmable processing units. In this work, *IDeSyDe* is the targeted DSE tool, that originally only supported DSE of software-programmable platforms, and *ForSyDe IO* is used for creating the system specifications. These tools were adapted to support hardware-programmable platforms and hardware implementations, wherein analysis was necessary to decide relevant parameters. A quality assessment of these novel DSE capabilities was performed on a model of the heterogeneous Zynq UltraScale+ XCZU9EG Multi-processor System on Chip through test cases and a realistic video streaming application.

An FPGA is a hardware-programmable processing unit providing hardware resources to implement functionality. Extensions to the tools mainly focus on providing support for specifying FPGA components and applications implemented on an FPGA. The achieved DSE support covers FPGA implementations through the specification of execution latency, required block RAM, and required logic blocks, relevant for FPGA modeling due to its characterizing resource limitations. These parameters are suitable as they are commonly found in libraries for reusable FPGA implementations. Likewise, the FPGA component is modeled with the availability of block RAM and logic blocks.

The project's goals were met and can thus be considered successful. Assessment of the test cases shows that the novel extensions are well incorporated into *IDeSyDe*. However, *IDeSyDe* did not find a solution for the realistic application within a reasonable time due to high computational demands, and the modeling did not adequately account for certain real-world aspects of the platform. Also, although the platform model assumed

rather pessimistic performance, the lack of performance guarantees between modeling and reality results poses a significant limitation and is left for future work.

Keywords

Design Space Exploration, Embedded System Design, System Modeling, Multi-Processor System on Chip, Field Programmable Gate Array

Sammanfattning

På senare tid har designprocessen för inbyggda system ökat markant i komplexitet vilket leder svårigheter i att hitta resurseffektiva och optimala lösningar. Detta är en direkt följd av de krav som ställs av moderna tillämpningar som vill verka i teknikens framkant och att inbyggda plattformar besitter alltmer heterogena resurser för ökad beräkningskapacitet. Designprocessen för resurs- och prestandakrävande applikationer kan hanteras med hjälp av resignrymdsutforskning genom skapandet av en designrymd utifrån systemparametrar för applikationer och den underliggande plattformen. Denna designrymd kan därefter systematiskt genomsökas för att hitta de bästa designlösningarna givet optimeringmål och krav på systemet. Dessa designlösningar kan beskriva hur applikationsexekvering ska distribueras över plattformens beräkningsenheter och hur applikationer ska schemaläggas samt vilka fysiska minnen på plattformen som ska användas för applikationsdata.

Detta examensarbete syftar till att möjliggöra designrymdsutforsking med en designrymd som utgörs av applikationer som kan exekvera på en plattform med både mjukvaru- och hårdvaruprogrammerbara beräkningsenheter. För att åstadkomma detta tillämpades designrymdsutforskingverktyget *IDeSyDe* som behövde utökas från att bara ta hänsyn till implementationer på mjukvaruprogrammerbara beräkningsenheter. Detta krävde analys av hur hårdvarubaserade beräkningsenheter och dess applikationsimplementationer bäst kan modelleras genom parametrisering, vilket utgjorde grunden för nödvändiga utökningar till verktyget *ForSyDe IO* som används för generell systemmodellering. En kvalitetsutvärdering av de utökade modelleringsmöjligheterna utfördes på en modell av det heterogena multiprocessorsystemschippet Zynq UltraScale+ XCZU9EG med testapplikationer och en verklighetsanpassad videoapplikation.

Den huvudsakliga utvecklingen av designverktygen tillägnades på-platsprogrammerbara grindmatriser (eng. FPGA)—en hårdvaruprogrammerbar beräkningsenhet, och tillhörande applikationsimplementationer för grindmatriser. En grindmatris karaktäriseras främst av dess begränsade resurser som avgör mängden funktionalitet som får plats. Utökningarna tillåter specificering av grindmatrisens resursbegränsningar för logiska block och block-RAM samt resursbehov för hårdvaruimplementationer och tillhörande exekveringstid i hårdvara. Behjälpt av att det finns bibliotek som tillhandahåller återanvändbara gridmatrisimplementationer som specificerar dessa parametrar anses modelleringen vara välanpassad.

Projektets huvudmål uppfylldes och kan därmed anses vara lyckat. Trots att

den skapade plattformsmodellen har pessimistisk prestanda är avsaknaden av garantier på hur designlösningar förhåller sig till verkligheten en kritisk aspekt. Utvärderingen av specifikt den utökade designrymdsutforskningen visade på goda resultat. Däremot kunde inte *IDeSyDe* hantera videoapplikationen då beräkningskomplexiteten blev för stor och i allmänhet visade designlösningarna på vissa svagheter i att respektera realistiska aspekter gällande plattformen.

Nyckelord

Designrymdsutforskning, Design av Inbyggda System, Systemmodellering, Multiprocessorsystemschip, På-plats-programmerbar Grindmatris

Acknowledgments

It feels almost surreal to think that five years of studies have passed and the fascinating knowledge I have acquired from not knowing how to write a single line of code on day one. Although this graduation is solely my achievement on paper, there are people involved I wish to credit for making the continuation of my journey always apparent.

I want to thank Saab and my industrial supervisor Ludwig Karlsson for the opportunity to conduct my master's thesis in an industrial R&D-like setting and emphasize Ludwig's continuous will to discuss, give feedback, and guide my work. Over at Saab, I am also grateful for the frequent discussions with Karl Lundén and Joakim Lindén that provided practical insights and perspectives around my project.

A special thanks to the ForSyDe group at KTH Kista, led by Professor Ingo Sander, for enabling the collaboration between research and industry via interesting ideas and Ingo's genuine passion and encouragement towards my work. I would also like to credit my KTH supervisor, Fahimeh Bahrami, for good discussions, proofreading, and her ForSyDe-centered viewpoint on my project. Last but certainly not least, I want to express my immense gratitude to PhD Rodolfo Jordão for his involvement. His expertise in collaborating on the tool extensions, his endless support and encouragement, and his ability to ensure meetings are enjoyable while most importantly being rewarding have been invaluable to me. Además, a el quiero agradecerle por algunas pequeñas lecciónes de Español que he disfrutado mucho.

Broadening the perspective from this thesis work, I would like to express love toward my parents who have always believed in me and supported (/curled) me in all ways possible. I would also like to thank my lovely girlfriend Saga who has always expressed pride in my achievements and supported me during intense periods to help me pull through while also ensuring that I do not overwork myself. And of course, my dear friend William, for making this journey feel so much more enjoyable by being someone to share the true highs and lows with, and showing me that KTH is more than just academics.

Stockholm, July 2024
Ludvig Larsson

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

AGC	Apollo Guidance Computer
API	Application Programming Interface
APU	Application Programming Unit
ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory
CNN	Convolutional Neural Network
CP	Constraint Programming
DFG	Data-flow Graph
DMA	Direct Memory Access
DPU	Deep Learning Processor Unit
DRAM	Distributed Random Access Memory
DSE	Design Space Exploration
DSI	Design Space Identification
DSP	Digital Signal Processor
FIFO	First-in First-out
FLOP	Floating Point Operation
ForSyDe	Formal System Design
FPD	Full Power Domain
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
IP	Intellectual Property
LPD	Low Power Domain
MILP	Mixed-Integer Linear Programming
MJPEG	Motion JPEG
MPSoC	Multi-processor System on Chip

NN	Neural Network
OCM	On-chip Memory
RAM	Random Access Memory
RPU	Real-time Programming Unit
SDF	Synchronous Data Flow
SoC	System on Chip
TCM	Tightly Coupled Memory

Chapter 1

Introduction

The Apollo Guidance Computer (AGC) is often regarded as the first embedded system developed for commercial use and has inspired many embedded systems up to the present day. Comparison of the AGC specification of 1024 16-bit words of memory and instructions at the microsecond level [1] to modern chips with gigabytes of memory and nanosecond instructions makes the development achievements apparent. The price to pay for these increased system capabilities is the challenging task of utilizing the full potential of these complex systems in commercial applications.

Automation is a key practice in today's industry, driving product development forward at immense speeds thanks to innovation that reduces time-consuming manual efforts and the need for widespread domain expertise among developers. Automation is beneficial for saving time and also for managing complexity with a systematic approach to the given problem. This has become increasingly important in the up-rise of embedded systems, where system resources continuously increase in quantity and complexity caused by heterogeneous computing power. In today's society, embedded systems support a range of applications, including the healthcare sector, artificial intelligence, and the automotive industry [2, 3, 4]. No matter the domain, there are unavoidable design challenges that must be handled with adequate methods, for which system modeling has become a prominent practice.

One interesting aspect in the embedded system design process is Design Space Exploration (DSE), which is used to find optimal solutions to the design problem that can guide the final implementation. DSE is already supported for homogeneous embedded platforms, for example, CPU architectures [5] with memory and component interconnections. However, efficient and automated DSE for heterogeneous platforms, such as Multi-processor System on Chips

(MPSoCs), does have missing gaps in research which translates into a field of complex platforms lacking DSE support. While there are multitudes of heterogeneous processing units to add support for, the focus is on hardware-programmable processing units, specifically the Field Programmable Gate Array (FPGA).

1.1 Background

Finding optimal solutions to design scenarios is not always trivial. There might exist countless alternatives for scheduling applications and distributing application execution but only a few solutions achieve optimal performance. While experienced system designers do solve these issues, it is likely not optimally solved and time-consuming for complex platforms and applications. The solution to this is commonly referred to as DSE, often assisted by software tools [6].

1.1.1 Design Space Exploration

Conceptualizing DSE with an analogy could be seen as decorating a living room with some personal preferences. Placing the sofa to block the doorway, painting on the floor, and hanging a carpet from the ceiling are options for possible interior design, but are generally not desired. Incorporating preferences may state that the sofa must be red and that the furniture should be placed along the walls to the greatest extent to allow open space in the center of the room. The different layouts of the living room and assessing usefulness, style, and spaciousness result in a final decision on how to decorate the room. This analogy translates into exploring how to distribute embedded applications (furniture) on an embedded platform (living room) when having a set of *constraints* (red sofa) and *optimization objectives* (place furniture along the walls) that the resulting system design strives for, whereas room and furniture dimensions represent the *design space*. Constraints guide whether a design alternative is valid, such as assuring that memory limitations are not exceeded. What is optimal for the DSE depends on the problem instance and can target minimization of power consumption and maximization throughput for example [6]. The design space is constructed from system specifications that define parameterized hardware components and component interconnection along with application tasks, inter-task dependencies, and task demands.

1.1.2 ForSyDe

For the last two decades, embedded system design have been researched within the Formal System Design (ForSyDe) methodology to support a streamlined Correct-by-Construction system design process from initial requirements to the final implementation [7]. The proposed design process should follow the Correct-by-Construction development technique which strives to incorporate tools that can, systematically, produce correct artifacts [8]. Within ForSyDe, the design process should consist of a sequence of automated refinement steps via Correct-by-Construction tools. Once system specifications are derived from the initial requirements as an initial *manual* design step, the remainder of the process is handled under automation by tools.

ForSyDe IO and IDeSyDe are tools that follow the ForSyDe methodology [9, 10]. The former provides a component library for programmatic construction of system specifications representing embedded platforms and applications. The latter can perform automated DSE for several system specifications; one being ForSyDe IO. These two tools have been used across various *homogeneous* design spaces of software-implemented applications and CPU-platform architectures and have been proven successful [5].

1.1.3 Related Work

Extensive DSE research has been conducted on MPSoC platforms. The construction of optimal Neural Networks (NNs) architectures and how the NN layers should be distributed on the MPSoC is one example, although the design space is constructed manually and likewise the performance through benchmarking [11]. Automated DSE, distinguished by its automatic construction of the design space and evaluation of design alternatives, has also been researched through a case study on deep neural networks on a Deep Learning Processor Unit (DPU) with multiple objectives [12].

Despite what has been done, there is unexpectedly little research on automated DSE that considers the heterogeneity of processing units on MPSoC platforms where application functionality can be implemented on hardware-programmable and software-programmable processing units. An almost 30-year-old study presents this partitioning problem based on profiled software behavior and a hardware performance estimator for each task before DSE [13], making the design process cumbersome for large design spaces.

1.2 Problem

Saab, the host company, is aware of the potential of DSE as demonstrated by recent work and is interested in investigating if tools can cover exploration of the heterogeneous Zynq UltraScale+ MPSoC platform and realistic applications—both of which are relevant to their business. In this way, they can evaluate the possibilities of using DSE in their system development to possibly omit time-consuming aspects of manual design and ensure correctness in the implementations.

Due to the complexity of heterogeneous platforms, inferred by numerous different types of processing units, compared to homogeneous CPU architectures, DSE becomes a design step to consider using. If two dependent applications execute on separate CPU cores, swapping the core allocation is redundant for performance but this does not hold for heterogeneous processing units. Distributing applications over heterogeneous processing units can introduce unexpectedly high communication costs, and may require prioritization of what should be executed where since resources are not infinite in combination with parallelization which often increases performance.

1.3 Purpose

The work in this thesis aims to produce a great foundation for facilitating the design process of MPSoC systems. From a research perspective, the derived modeling capabilities could enable extensions the design space to consider greater detail and consequently yield better precision in the DSE results.

A design process that is based on the Correct-by-Construction development technique allows system designers to focus on the system specifications since subsequent design steps are almost completely assisted by tools and ensured to produce a correct system. System modeling and systematic creation of the design space is also a one-time effort as partial design spaces can be reused for multiple design scenarios—for example, using the same underlying MPSoC platform when exploring new application functionality. These two major benefits of using DSE help to identify issues and derive insights early in the design process instead of finding them after time-consuming system implementation.

1.4 Goals

What is most important for the project is the DSE ability to decide how applications should be distributed to hardware-programmable (FPGA) and software-programmable (CPU) processing units located on an MPSoC. In terms of ForSyDe support, this makes the FPGA processing unit and functional implementations on the FPGA constitute the additions to modeling in ForSyDe IO and the DSE in IDeSyDe. Goals that do not contribute to the proof-of-concept of hardware-software mappings are marked as "Optional" in the following list.

G1: Decide parameters that can model the key characteristics of the FPGA processing unit and functional implementations on the FPGA.

G2: Develop a software application for easy interaction with the ForSyDe IO library to create the platform and application specifications.

G3: Extend ForSyDe IO and IDeSyDe to interpret the new platform characteristics and account for additional mapping possibilities.

G4: Create test cases that define application and platform parameters that have expected DSE solutions in the context of the novel modeling.

G5: **(Optional)** Create a realistic application specification that aligns with the given sample functionality from Saab to evaluate the DSE capabilities.

G6: **(Optional)** Develop a method for assessing the validity and correctness of the non-trivial DSE solutions.

G7: **(Optional)** Perform DSE on more test cases to conclude design alternatives.

1.5 Project Methodology

This project concerns extended DSE capabilities split into FPGA modeling and the FPGA as part of the system exploration. For the project to be carried out effectively and provide valuable results, there are important cornerstones in its methodology.

Modeling and DSE

Establishing a theoretical foundation of the FPGA as a platform component together with an understanding of the philosophy behind IDeSyDe and ForSyDe IO is crucial. Combining these two literature study areas will contribute to adequate platform and application modeling decisions, respecting realistic aspects of the FPGA, FPGA programming and the effort to extend ForSyDe tools which must be respected time-wise. With the resulting DSE support, test case system specifications of an MPSoC as the target platform should demonstrate the proof-of-concept of DSE with heterogeneous processing units as part of the design space.

Development

Throughout the project, there is a need for a sophisticated development workflow and software, to easily construct and reconstruct the platform and application models for various test cases but also testing during development. For this reason, an application interfacing with the ForSyDe IO library should be developed.

1.6 System documentation

Since this project is tightly connected to industry and how existing design tools can be adapted to support industry tasks, it is of high value to document the necessary modifications to ForSyDe IO and IDeSyDe. In addition to what is covered in the Method chapter, step-by-step instructions for the tool extensions will be supplied in the appendix.

1.7 Structure of the Thesis

The rest of this thesis is structured as follows:

Chapter 2 presents background information about DSE, the ForSyDe tools, and relevant application and platform characteristics. Chapter 3 presents the methodology to achieve DSE results for the target system with modeling, tool extensions, and test cases. Chapter 4 evaluates which goals were met and the results of DSE test cases, while Chapter 5 discusses modeling results, integration to the design tools, and the quality of the extended DSE support. Chapter 6 presents insights following the project and an outlook on future work regarding hardware-software DSE.

Chapter 2

Background

This chapter provides background information about DSE, system modeling, the relevant hardware platform, and relevant reference applications. DSE and system modeling are covered from a general perspective, but also from the perspective of the ForSyDe tools in practical DSE work.

2.1 Design Space Exploration

DSE is a technique used to systematically search through the design alternatives for a given embedded system problem. After evaluating the complete set of alternatives, optimal solutions are presented based on optimization objectives. These solutions present how application execution should be distributed (execution mappings) across processing units, which memories should be utilized (memory mappings), and how applications should be scheduled. Two general topics need to be decided before performing DSE [14]:

1. A method for systematical search of the design space
2. A method to evaluate and compare different DSE solutions

2.1.1 Design Space

An embedded system's design space theoretically consists of all existing design alternatives for executing applications on a target platform. This is generally enormous, ranging from comparing lines of assembly code to assigning a real-time task to a specific CPU core in terms of detail. It is important to emphasize that the design space is determined by the given system

specifications, which can result in varying sizes of the design space depending on the level of detail. Typically, the input to DSE is divided into specifications of the platform, applications, and constraints that are nearly independent as it has been proven to be very effective for handling complex problems [15].

The platform specification is an abstraction of the embedded platform, outlining its components in terms of processing units, such as CPUs, hardware accelerators, Digital Signal Processors (DSPs), DPUs, and FPGAs, memory units, and the platform's interconnection network. In addition to the platform specification, the applications are usually specified as several independent dependency networks of tasks where each task specifies some required resources. To complete the system specification, a set of constraints should be defined. These constraints could be to require a valid task schedule, not using more memory than the platform provides, and that the application's data throughput is higher than some specified value. By introducing constraints, the goal of complete independence between system specifications breaks. This is unavoidable but also desired, as one constraint may include some property of the platform combined with the application demands—essential to producing correct design solutions. Most important is that the platform and application specifications are independent to enable modular reuse through e.g. platform-agnostic applications.

2.1.2 Searching the Design Space

There are two possible high-level approaches when considering DSE search. One is to perform the search manually and the other is to automate the search with some applied software tool. The design space can be constructed manually by selecting some design alternatives that are expected to be performant. In practice, this has been done through having a set of possible application execution mappings onto an MPSoC based on varying parallelization and optimization techniques [11]. Then, analysis tools designed for the specific MPSoC estimate performance metrics to finally conclude the findings. This manual approach is highly effective for evaluating a smaller set of design alternatives, although unsuitable for larger design spaces considering the manual effort required to use the tools for the result collection and rebuilding the systems for benchmarking. In cases where the design space has non-trivial constraints to meet or where the DSE should find the most optimal solutions, only automated DSE approaches are realistic to consider. It has been concluded that a NN application cannot feasibly be explored manually due to the large design space of mappings and

optimizations for each of the NN layers [16]. Automated DSE streamlines the representation of large design spaces through mathematical formulations that, regardless of the design space, do not impose any higher time demand from the user side. Another benefit of automated methods is the potential discovery of unexpectedly good solutions, typically not included when the design space is constructed manually.

2.1.3 Pareto Points

Optimality in the context of DSE can refer to a range of design objectives. These can include minimizing power consumption and maximizing application throughput among others [6]. As these objectives are not comparable since they measure different metrics, optimization with multiple objectives does not necessarily produce a single most optimal solution. Figure 2.1 depicts a scenario where one solution is better in both objectives but in other cases, multiple solutions may arise as a consequence of solutions that are optimal in a subset of the objectives. This group of solutions is called a Pareto front [6]. A solution x_1 can only replace another solution x_2 if the solution has at least one dominant objective and equal performance to the rest of the objective variables in x_2, formally defined as [6]:

$$\begin{aligned} x_1 \prec x_2 \iff \forall i \in \{1, 2, \ldots, n\} : f_i(x_1) \leq f_i(x_2) \wedge \\ \exists i \in \{1, 2, \ldots, n\} : f_i(x_1) < f_i(x_2). \end{aligned} \tag{2.1}$$

Pareto points are used in DSE to collect the most optimal solutions based on the optimization objectives that the DSE uses. In this way, if a newfound solution dominates one or more points in the current Pareto front, these are replaced, and if the solution is only dominant in some subset of the objectives it is instead added to the Pareto front.

2.2 Field-programmable Gate Arrays

Having fully customizable hardware circuits is an important component in improving application efficiency, and this is where the FPGAs chip excels. An FPGA contains a finite number of logic blocks that are independently programmable to implement functionality with the help of digital logic gates in hardware [18]. These hardware implementations are flexible as logic blocks can be reprogrammed to realize other functionality. For this reason,

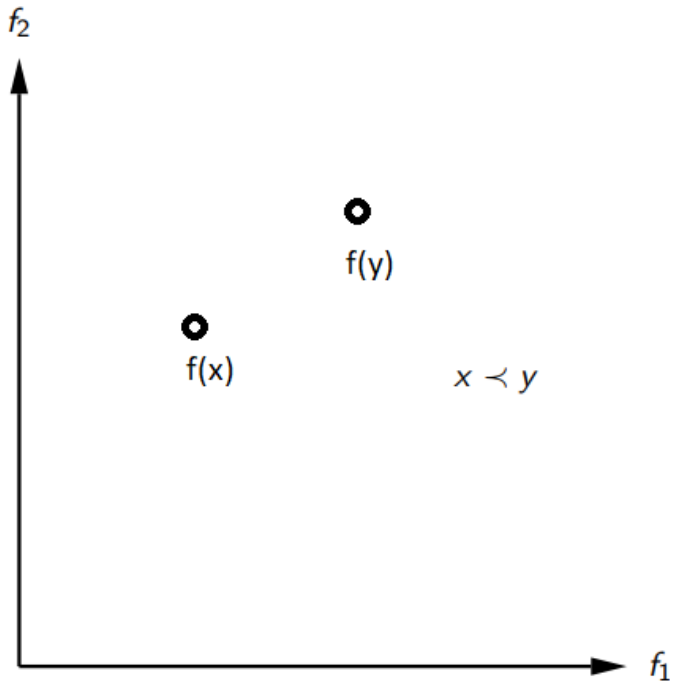

Figure 2.1: Two solutions in a two-dimensional optimization space. In this example $f(x)$ dominates $f(y)$ in both dimensions, hence solution $f(y)$ can be discarded. Source: [17].

FPGAs are highly preferable when developing Application Specific Integrated Circuits (ASICs) since prototyping can be done on an FPGA before fixing the functionality for mass production [19]. Regular use cases also include widespread hardware acceleration for improving the throughput of systems, allowing remote upgrades to hardware functionality.

Memory Accesses

Memory access and data transfers between the FPGA implementations are usually handled by the on-chip, statically located, Block Random Access Memory (BRAM) [20]. This memory ranges between a few kilobytes to a few megabytes depending on the FPGA model and is scattered around the chip in small chunks. It can be used as synchronous memory for normal reads and writes or in First-in First-out (FIFO) fashion [21]. Another type of on-chip memory is Distributed Random Access Memory (DRAM), which is not as commonly found on FPGAs as BRAM but delivers better memory performance. This memory is not statically located on the chip and is

created by consuming logic blocks. For memory requirements that exceed on-chip memory capacity, the FPGA can access external memory through external connections but these operations are off-chip and thus degrades the performance.

Clock Regions

To provide more flexibility for implementations across the chip, FPGAs are commonly divided into a finite number of clock regions that divide the FPGA resources [22]. This feature allows FPGA implementations to not share a common clock frequency by being implemented in different clock regions. When having different clock frequencies synchronous memory is not achievable but the usage of FIFOs makes it possible to buffer data between clock regions [23].

Design Considerations

The general performance of functionality implemented on the FPGA is affected by the overall speed grade of the device that is determined once manufactured and tested [24]. This grading implies a minimal performance expectation from the device including the maximum clock frequency and timing constraints. Overall, this influences the flexibility in using multiple clock regions and decides possible operating frequencies that on-chip memory can assume.

Thousands of logic blocks must be programmed to realize logic functions depending on the selected design since one logic block contains a small amount of digital circuitry [13]. Considering that logic blocks on the FPGA are allocated for the implementations, this becomes an important factor for FPGA development. Generally, applications run faster on dedicated hardware, but the dilemma of choosing between space and performance becomes inherent here. The naive approach of designing FPGA is to ensure that all hardware implementations fit the available logic blocks, although other design considerations are also relevant in realistic design scenarios.

A common design effect for FPGAs is clock skew, caused by how fast digital clock signals propagate to various devices depending on link lengths. This phenomenon can lead to erroneous behavior or decreased performance due to required frequency reduction [25]. The necessary steps to reduce the clock skew involve reducing the logic block requirement of FPGA implementations and shortening physical link lengths which affect the

hardware design.

Expanding on this concept, the average temperature of the FPGA board during runtime, resulting from transistor switching, is also crucial for performance. High temperatures can affect the switching time of transistors, potentially disrupting clock signals and causing clock skew [26]. While moving functions to software is an obvious solution for reducing the FPGA's temperature, techniques like thermal-aware logic partitioning are more relevant to addressing the problem.

A slightly different design constraint involves the arithmetic accuracy of the digital circuits on the FPGA. It is given that any precision can be achieved with sufficient digital logic, but it quickly becomes costly in terms of logic blocks as they are limited. Approximations for the expensive sigmoid function, often used in artificial intelligence, have been attempted but some methods may not always provide sufficiently low error bound [27].

The suitability of using a fully FPGA-programmed convolutional neural network has been investigated to see whether it can achieve a performance boost. However, it faces limitations due to insufficient on-chip memory [28]. On the contrary, the FPGA is found to be useful for digital signal processing with high performance [29, 30]. It can be concluded that implementing functionality on the FPGA is not possible, or beneficial, for all applications and must be decided whether to program it in software or hardware.

2.3 Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit

The initial project proposal by Saab requested the hardware-software DSE to be performed for the Zynq UltraScale+ MPSoC ZCU102 evaluation kit featuring the Zynq UltraScale+ XCZU9EG MPSoC chip. The Zynq UltraScale+ chip family features several hardware- and software-programmable processing units, a fast interconnection network, multiple interfaces for external connections, on-chip memory, and external memory which generates a platform complexity suitable for this project [31]. Because of the chip's performant components combined with various security measures and power management plans, it conforms with industry-standard requirements on complex and demanding applications.

Apart from the common UltraScale+ architecture and the division of processing units between the processing system (software-programmable) and the programmable logic (hardware-programmable), each evaluation board is

configured slightly differently with available resources. Some specifications of the XCZU9EG MPSoC includes [32, 33]:

Processing System Specifications:

- Quad-core ARM Cortex A53 Application Programming Unit (APU) - high performance CPU despite low power usage, operating at 1.5GHz.
- Dual-core ARM Cortex R5 Real-time Programming Unit (RPU) - used for safety-critical real-time systems thanks to deterministic behavior
- Private Tightly Coupled Memory (TCM) of 128kB for each RPU core - fastest and most deterministic memory available to the RPU cores.
- ARM Mali-400 Graphics Processing Unit (GPU) with pixel and geometry processor.
- DDR4-2666 SODIMM memory of 4GB with a 64-bit wide data bus [31]. The memory is operating at 333MHz [34].

Programmable Logic Specifications:

- 600'000 logic blocks with 8 look-up tables and 16 flip-flops each.
- 32.1 Mb BRAM - static memory.
- $\leq$ 8.8Mb DRAM - optional memory created by look-up tables.
- DDR4-2666 SDRAM memory of 512MB with a 16-bit wide data bus [31]. The memory is operating at 333MHz [34].

Board Specifications:

- AXI4 buses of 64-bit and 128-bit widths. The operating clock rate is typically 200 MHz or less [35].
- On-chip Memory (OCM) of 256kB RAM - low-latency access for RPU operating at 600 MHz.

2.4 Synchronous Data Flow

Modeling real-time applications is important in the design process, where abstraction and the possibility of analysis are desirable modeling aspects. A Data-flow Graph (DFG) is defined by a set of nodes and directed edges, forming a directed graph [36]. Such graphs are suitable for representing an embedded application's constituent tasks, with nodes as tasks and edges representing communication. However, the DFG was later specialized into Synchronous Data Flow (SDF) graphs to enable a static analysis of an application's schedulability through mathematical formulas [37].

Figure 2.2 shows an example of an SDF graph. Each node is named an *actor* which executes a specific function as part of the task chain. Edges are called *channels* and represent an abstraction of data buffers. The unique aspect of the SDF graph compared to DFG modeling is the addition of annotations on both ends of the channels, referred to as *tokens*. These numbers specify the proportions of data usage between actors and have no data type. The value at the source of the channel signifies the actor's *production* and the value at the destination the *consumption*. Both values are required for all channels in the SDF graph for the schedulability analysis to work, except the input and output channels of the data flow which only link to one actor.

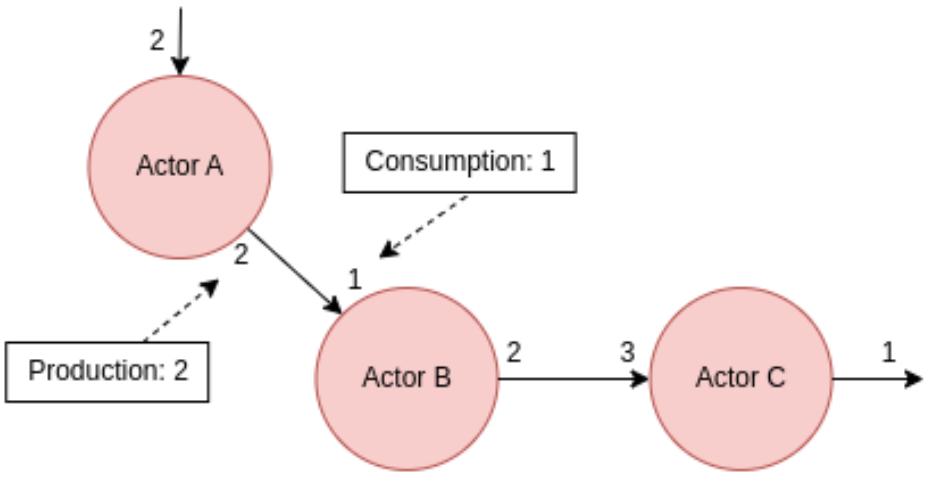

Figure 2.2: An SDF graph with three actors and explicit production and consumption rates. Tokens are infinitely streamed into the system via the top edge, and out via the rightmost edge.

2.5 Reference Applications

Along with the initial project proposal from Saab to model and perform DSE on a heterogeneous MPSoC, they also requested the incorporation of a realistic 4K@30Hz RGB video processing application for evaluating the DSE's usefulness. While not specifying any details, the application should contain the following types of operations:

- Pixel operation(s): Operations that transform a single RGB pixel into some new RGB pixel.
- Stencil operation(s): Operations that transform a single RGB pixel based on values from a set of neighboring pixels, e.g. 3x3 or 5x5 grids.
- Neural Network(s): Neural networks that perform some type of artificial intelligence task.

2.5.1 Memory Constraints

The input stream is a 4K resolution, i.e. 4096x2160 raw RGB pixels, and should support a 30Hz update rate. Given that red, green, and blue require one byte respectively, adding the parameters together yields a raw frame size of ≈ 26.5MB ($4096 \times 2160 \times 3$) and a bandwidth requirement of ≥ 795MB/s for a 30Hz update rate. However, compressing the stream with H.264 encoding, the frame size shrinks to ≈190kB and requires a bandwidth of ≥ 5.7MB/s [38]. Translating this into the actual application, the latency for processing one frame through the complete video stream must be less than $\frac{1}{30}$s. This latency assumes the sequential case where no pipelining or parallelization is applied, which is common in reality.

2.5.2 Throughput Constraints

Image processing does not always process pixels in a FIFO fashion as is the case for independent pixel operations. When operating on pixel grids, storing complete sets of rows in memory is required to enable random access to adjacent pixels. Figure 2.3 showcases this scenario. In other cases, mirroring or rotating a frame requires a large memory allocation to buffer the incoming pixels to transform a column into a row in the resulting frame.

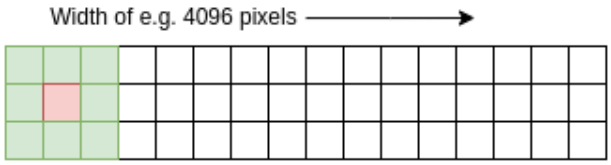

Figure 2.3: Example of grid operations. In this case, a target pixel (red) is dependent on itself and its surrounding pixels (green) resulting in a requirement of loading at least three rows, equivalent to 12288 pixels, from the target frame.

2.6 ForSyDe

The ForSyDe methodology is based on a system-level design methodology presented which emphasizes the separation of concerns between the components of the system-level design to manage system complexity best [39, 15]. Most important is the orthogonality between functional specifications and functional mapping to a System on Chip (SoC) platform with included communication. In practice, this aims to decouple the platform-agnostic application functionality from the specific embedded platform and let the mappings be produced as a subsequent step when the platform is combined with the application. ForSyDe adopted this system design philosophy and refined it further. Figure 2.4 shows a modernized view of the ForSyDe methodology since it was initially published, although the idea of separation of concerns is highly intact. ForSyDe underscores the use of formal system models and applying a sequence of refinements to step-wise close the gap between initial requirements and the final implementation.

The Correct-by-Construction technique is another key aspect of the ForSyDe methodology that aims to ensure that a tool, instead of humans, systematically and automatically constructs artifacts of the system design process. It avoids heavy user-testing of the produced designs based on the theory that correctly user-specified system specifications are used as input to the tools [8]. Instead, the single-effort task of validating that Correct-by-Construction tools are correctly implemented is required, which is negligible time-wise compared to the manual testing needed for each new system. With the technique, system designers are solely required to adequately model the desired system at a relatively high abstraction level in contrast to traditional system construction including low-level implementation details.

Since the millennial shift, the ForSyDe methodology has continuously

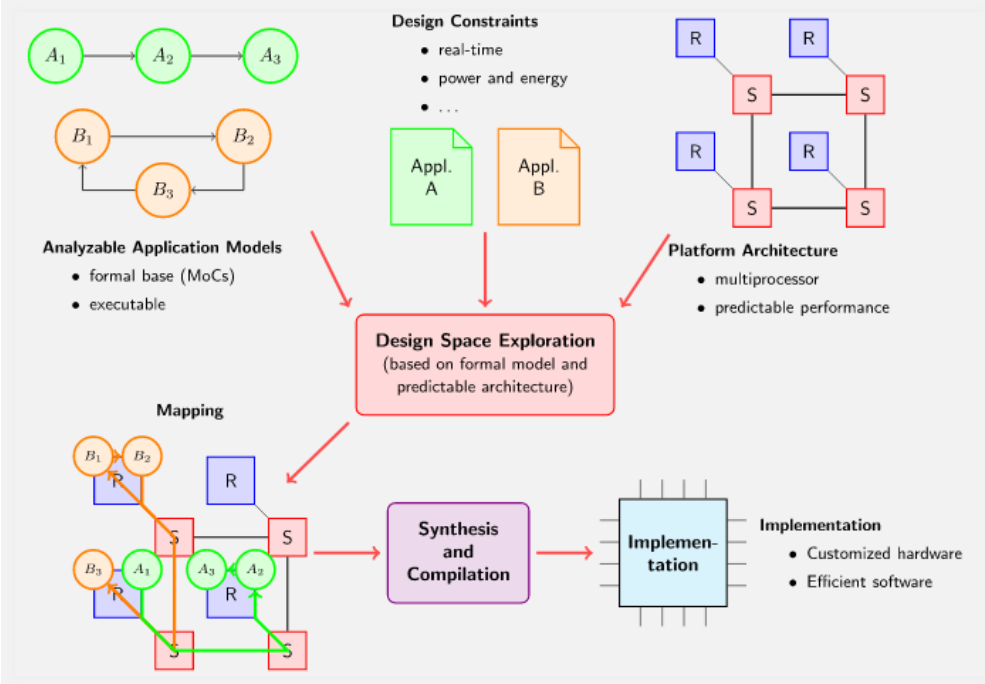

Figure 2.4: The ForSyDe design process with clear separation of concerns with specifications and constraints, design space exploration, and implementation synthesis. Source [7].

evolved by developing new tools that align with the design flow. For this project, the formal, and flexible, modeling tool ForSyDe IO is specifically important together with IDeSyDe which targets the intermediate design step of optimizing the design based on the system specification, i.e. DSE.

2.6.1 ForSyDe IO

The need for formal system representations is apparent in the ForSyDe methodology [39]. Although the focus is to have some model that satisfies this, providing ease of integration with other system models is equally important to favor the longevity and flexibility of tools used in the design process. This served as the motivation for defining the programming language agnostic modeling framework presented in [40], also realized in practice with the ForSyDe IO library [7]. The framework's key characteristic is its integration approach, which utilizes a common graph-based system model and offers a straightforward method to integrate tools originally adhering to different formats. Integration of other tools therefore becomes a matter of creating an

adapter that converts a tool's system representation to ForSyDe IO's lenient framework directives.

Each vertex in the system graph represents a system component such as a processing unit, network switch, or SDF actor. Edges denote the connections between these components. The framework implements a *multi-view* technique that applies to the vertices. A view in this context represents a subset of specifications for a system component, while other views can define other information for the same system component. This technique simplifies tool integration, as potentially new properties can reside in an independent view, thus requiring minimal integration effort. The simplicity of integrating other modeling tools with existing specifications has been demonstrated by creating an adapter for AMALTHEA [40], a popular embedded systems modeling framework [41].

2.6.1.1 Practical Usage

The system model can be constructed with the ForSyDe IO library, which provides an Application Programming Interface (API) for programmatic specification of the system models [9]. Typically the overall system model is divided into two disjoint models, one representing the applications and one representing the platforms. The library allows the user to construct a system model, by adding vertices and edges, and apply *traits* to them as per the multi-view technique. Six fundamental traits are relevant when modeling embedded applications and platforms like the ones described in Section 2.5 and Section 2.3:

Platform Traits

- `GenericProcessingModule`
- `GenericMemoryModule`
- `GenericCommunicationModule`
- `AbstractRuntime`

Application Traits

- `SDFActor`
- `SDFChannel`

Combining the first four traits in the list can abstractly describe any embedded platform. Those cover computation (`GenericProcessingModule`), storage of data (`GenericMemoryModule`), communication switches for data transfers (`GenericCommunicationModule`), and runtime for application scheduling (`AbstractRuntime`). The remaining two traits are used for SDF application modeling to represent SDF actors and SDF channels. Necessary specifications of actor production and consumption, as described in Section 2.4, are available as properties for the `SDFActor` trait. The creation of actual edges in the system model is separated from the concept of traits and only describes the vertex interconnection topology. It should also be noted that the system models created with the ForSyDe IO library can have essentially any information associated with its vertices without verification. The user has full control of the modeling for better or worse.

The previous listing of traits is quite abstract in specifying the system components because the traits are quite shallow. More information is needed to create a meaningful design space to explore. This includes actor requirements and processing module specifications which can combined during DSE to derive execution times. Some specifications required to derive execution times for a CPU-like processing module are to specify the supported instruction set with instructions per cycle and the operating frequency. That information is located in the `InstrumentedProcessingModule` trait attached to the same vertex as the `GenericProcessingModule` defining the operating frequency. See Listing 2.1 for a code example of using the multi-view technique.

```
1 var V = sGraph.newVertex(puName)
2 var pu = GenericProcessingModule.enforce(sGraph, V);
3 pu.operatingFrequencyInHertz(FREQUENCY);
4 var instr =
     InstrumentedProcessingModule.enforce(sGraph, V);
5 instr.modalInstructionsPerCycle(Map.of(
6     "Instruction Set", Map.of(
7         "Integer Addition", 0.5
8     )
9 ));
```

Listing 2.1:
Specification of the operating frequency as part of GenericProcessingModule and instruction definitions are part of InstrumentedProcessingModule.

Creating an SDF actor as part of an SDF application is presented in Listing 2.2. The actor specifies production and consumption requirements, ports for

connecting to SDF channels, and the instructions for its functionality.

An aspect to consider for later calculation of execution times during DSE is that an actor's needed instructions must match name-wise with an instruction supported by some processing module in the platform specification. For instance, the actor in Listing 2.2 would not be able to execute on the processing module in Listing 2.1 as the actor requires "Floating Point Addition" and the processing module only supports "Integer Addition".

```
1  Vertex V = sGraph.newVertex("Actor A");
2  SDFActorViewer actorA = SDFActor.enforce(sGraph, V);
3  actorA.consumption(Map.of("s_in", 2));
4  actorA.production(Map.of( "s1", 2));
5  var instr = InstrumentedSoftwareBehaviour.enforce(
6      sGraph, V
7  ));
8  instr.computationalRequirements(Map.of(
9      "Required Instructions", Map.of(
10         "Floating Point Addition", 1000L
11     )
12 ));
```

Listing 2.2: Specification of an SDF actor with computational requirements using two traits: SDFActor and InstrumentedSoftwareBehaviour.

2.6.1.2 Output formats

Although modeling is efficiently done with the component API that ForSyDe IO provides, visual and textual outputs are also necessary to validate and proceed with the design process in another tool. After the ForSyDe IO graph is constructed it can be fed to handlers that process the graph arbitrarily and provide its specialized output format.

First, the formalized output of ForSyDe IO is `fiodl` files which are compatible as input to IDeSyDe. Secondly, visualized models are the result of a handler that converts ForSyDe IO graphs to `kgt` files representing a `KGraph` [42]. The `kgt` format is a textual representation of `KGraphs`, which can be viewed graphically with the `KIELER` extension available in VS Code [43].

2.6.2 IDeSyDe

IDeSyDe is the latest DSE framework developed for the ForSyDe design suite. When comparing IDeSyDe to other DSE tools, such as Sesame [44], its novelty lies in its underlying Design Space Identification (DSI) approach. DSI allows multiple DSE abstractions to coexist in a modular way and be merged into a combined design space automatically without requiring direct user intervention [45]. DSE abstractions are mathematical representations of a complete or partial design space, which may vary between systems but can potentially be partially reused. The method proposed by IDeSyDe consists of four steps, see Figure 2.5 and the following sections for details on each step.

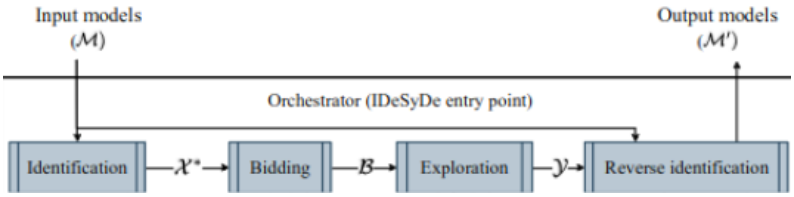

Figure 2.5: The DSE process in IDeSyDe involves four stages: identification, bidding, exploration, and reverse identification, all managed by an orchestrator process. Source: [45].

Identification

The DSI method searches for traits and their contained information in the given input system specifications, also called *design models*, to convert into *decision models*—the refined input for later exploration. Figure 2.6 illustrates the given identification process. Decision models automatically define boundaries for their corresponding design space through parameters derived from input design models. During identification, *identification rules* identify decision models from system specifications and also from combining already identified decision models [45]. An identification rule defines the necessary characteristics to satisfy a decision model. When considering application specifications through SDF graphs, all actor must have defined their production, consumption, and functional requirements among other information. With sufficient information, the `SDFApplication` decision model is created with associated design space. Figure 2.6 illustrates an identification process that uses the `SDFApplication` decision model as a part of the combined system-level decision model named `SDFToTiledMultiCore`.

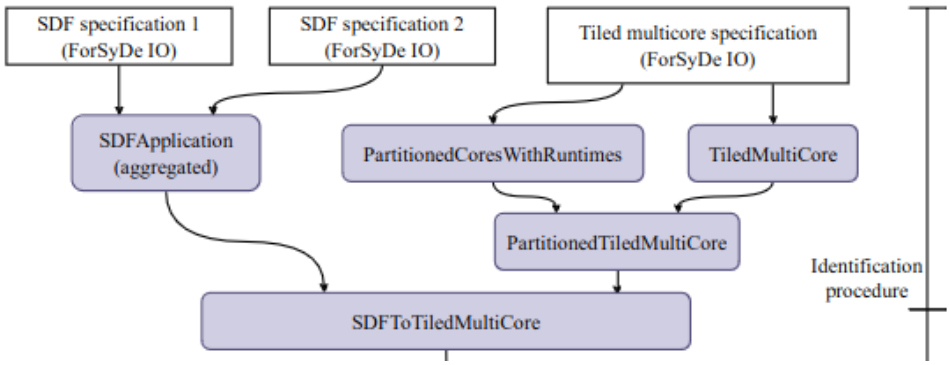

Figure 2.6: Illustration of ForSyDe IO models being step-wise converted into one analyzable decision model in IDeSyDe. Source: [45].

Although the system specification and DSE steps are independent at their core, using them together requires adaptations. Because of the identification process, the input design models must conform to the existing identification rules to be analyzable. Therefore, if there are mistakes in the input specifications, the expected identification is not successful, requiring correction. Also, depending on the specifications in the input design models, IDeSyDe may need to be extended to identify and explore new decision models.

Bidding

The bidding stage aims to distribute the identified decision models from an orchestrator to the available explorers and retrieve *biddings* that indicate if an explorer is capable of exploring the decision model [45]. IDeSyDe currently has two explorers, one using meta-heuristics and the other using Constraint Programming (CP). While the identification rules are the same for all explorers, there is no guarantee that the explorers overlap in which DSE abstractions they can solve.

Exploration

During this stage, the actual exploration occurs. Explorers, selected based on the biddings, search the design space formed by decision models to find optimal solutions. In cases where both explorers can solve the same decision model, solutions found in either explorer are sent to the other so that the exploration is done cooperatively and thus overcome one another's weakness of local Pareto optimal solutions (meta-heuristics) and time demand (CP)

[45]. Upon completion of the exploration, a Pareto front of design solutions is presented. Solutions are captured in extended versions of the input decision models, specifying information about the solutions including where actors should execute, which memories to use for shared memory, and how each processing module should schedule the actors it executes. There is also the case when no solutions are produced due to solution constraints unable to be solved within the given design space.

Reverse Identification

Design solutions, in the form of decision models, are difficult to compare with input design models due to their differing formats. Therefore, the identification steps are appropriately reversed into design model representations [45]. Input specifications do not contain the solutions for obvious reasons. Hence, input specifications are modified to relate design solutions to the original input components.

2.6.3 Choosing a Platform Model

IDeSyDe has two system-level decision models where applications and platforms have been combined into one decision model:

- AperiodicAsynchronousDataflowToPartitionedMemoryMappableMulticore
- SDFToTiledMultiCore

These two decision models are distinguished by private and shared memory [46]. Tiled architectures are represented by independent, bus-interconnected tiles with one-to-one mapped processing modules and memories, and shared memory platforms have multiple processing modules connected to the same memory.

Selecting one of the two architectures depends on the desired platform characteristics to include in the exploration. Platform specifications having shared memories will make the exploration account for the latency caused by memory operations. In contrast, tiled platforms assume Direct Memory Access (DMA), allowing memory operations to occur concurrently with CPU executions of other workloads.

Chapter 3

Method

This chapter provides a thorough introduction to the applied method for collecting project results. The method is divided into two rather disjoint focus areas: *System Modeling and DSE*, and an *Automated Workflow* to effectively derive the necessary textual and visual results.

System Modeling and DSE

Modeling is the major focus of the method. Deciding the appropriate level of abstraction for modeling to derive useful DSE results involves three disjoint areas of literature study:

- Finding a good abstract model of the XCZU9EG MPSoC with relevant components, resources and constraints.
- Gaining insight into the essential design considerations and implementation practices for an FPGA.
- Investigating the support of ForSyDe IO and IDeSyDe to gain insight into the current state of the tools and how they can be extended to cover specifications of an FPGA on the platform and an FPGA implementation alternative for actors.

Combining these three literature study areas creates a foundation for deciding what the system modeling and DSE should cover. A compromise between modeling the reality and the time demand for tool extensions of existing tools is expected, yielding a model for the hardware-software DSE that may not capture all relevant details. Abstracting the reality is also highly necessary as the time demand for the DSE quickly grows with more detailed design spaces.

Automated Workflow

Construction of system specifications, DSE, and the collection of results should be automated to the greatest extent to minimize overhead. ForSyDe IO and IDeSyDe are compatible but no tool exists to automate the interaction. Extending tools is a manual task, but automation can solve data passing between tools, intermediate result collection, and final result compilation, during experiments.

3.1 Selecting an Abstraction

Figure 3.1 presents the general design process for embedded systems spanning from the initial requirements to the final implementation (horizontal axis). The annotations in the figure relate the tools with the design steps: IDeSyDe performs the DSE using the ForSyDe IO representation format as input and output. Although this project does not cover the final step of implementation synthesis, there is a tool that can theoretically tie together the envisioned design process [47].

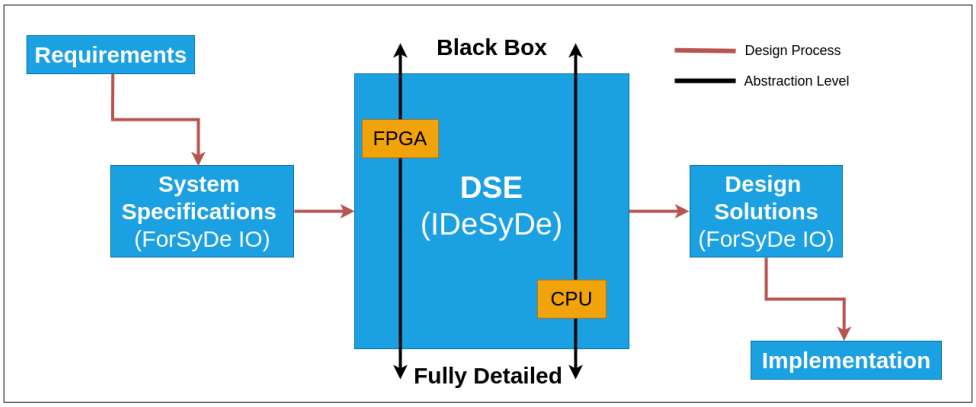

Figure 3.1: System modeling and DSE as part of the complete design process from initial requirements to the final implementation. Inspiration from:[7].

The DSE container in Figure 3.1 features only a subset of the components available on the MPSoC to give a reference of how the level of abstraction for individual system components can vary (vertical axis). Any component can be arbitrarily modeled but of course, models will differ in how well they capture reality. Only the modeled parameters can be accounted for in the design space and directly influence the usefulness of DSE solutions. For instance, modeling

an FPGA as a many-core CPU to represent parallel functionality will yield DSE solutions that respect FPGA parallelism but could also be misleading. For example, CPUs can handle numerous tasks, affecting only the speed of task completion, whereas FPGAs have finite resources that restrict the extent of functionality they can realize. With this in mind, the FPGA should find a suitable abstraction level for the platform model, specifically the FPGA, in ForSyDe IO that should be used in IDeSyDe to produce realistic DSE results.

3.2 Abstraction of the XCZU9EG MPSoC

The platform model in Figure 3.2 presents an overview of the XCZU9EG MPSoC based on datasheets [31, 33, 22, 21, 32]. The model abstracts the platform, capturing key details that define its components and constraints without representing low-level details. It aims to include memories and the interconnection network with links and switches, however, the selection of processing units is limited to the RPU, APU, and FPGA. Additional processing units are irrelevant for this thesis as the FPGA and CPUs are sufficient for hardware-software DSE.

As can be observed in Figure 3.2 there are six processor cores available and an FPGA available for application execution. The three heterogeneous processing units are distributed between two power domains: the Full Power Domain (FPD) (APU, FPGA) and the Low Power Domain (LPD) (RPU). RPU cores have fast and undisturbed access to the private TCM, and rather fast access to the OCM as the OCM is also located in the LPD. In contrast, the APU and FPGA communication must go through the FPD main switch to reach the OCM. The APU and RPU are part of the processing system which gives access to higher capacity memory through the processing system's DDR4 memory and the programmable logic has access to its DDR4 memory. Memories on the platform are annotated with `(clock frequency, data width)`, apart from the BRAM as it is determined by the highly dynamic FPGA requirements. Communication between the system resources is handled with AXI4 interfaces of varying bus widths. Channels supporting a 64-bit data bus are marked with blue color and are used in the LPD, while the FPD uses 128-bit data buses, marked with black in the figure. Depending on system configuration, the operating frequency of the AXI4 buses can vary, but it is recommended to clock it at 200 MHz [35]. There are three types of edges representing communication in the figure:

- **Directed Edge** One way communication path.

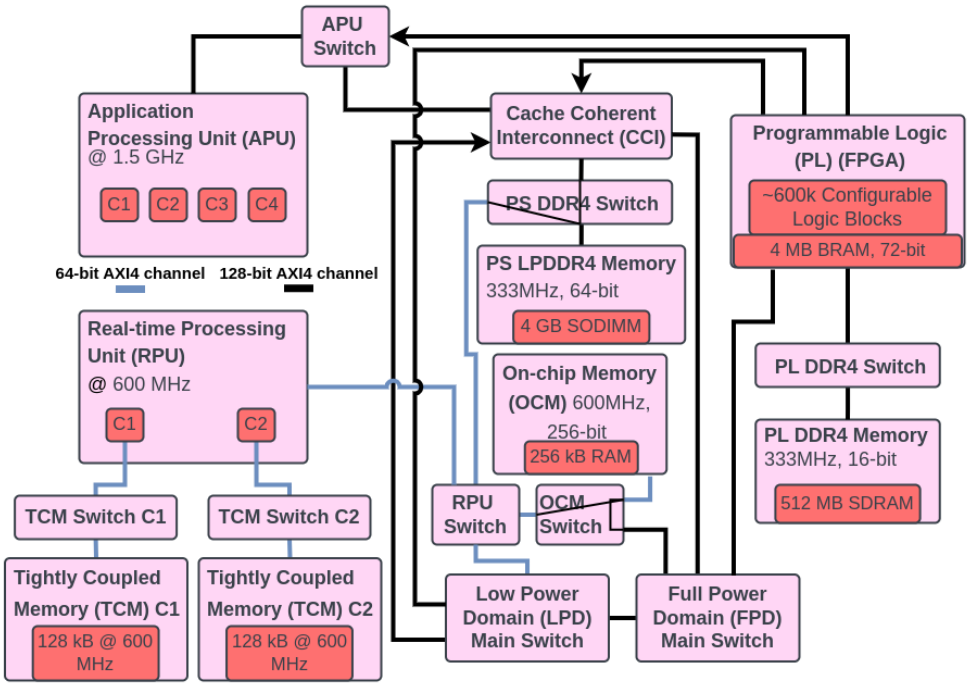

Figure 3.2: The platform abstraction for the XCZU9EG MPSoC, where the most relevant system resources are present. Resources are connected with AXI4 switches that support varying data bus widths. Memory components are annotated with their operating frequency and data width.

- **Undirected Edge** Two way communication path. An abstraction of the reality where there are multiple paths with unidirectional communication paths.
- **Edges within switches** Available routes between ports of switches. Switch ports are fully connected if no edges are modeled within the switch.

3.3 Idea of FPGA Modeling and DSE

The abstract platform model of the XCZU9EG MPSoC, presented in Section 3.2, is for the most part compatible with the available ForSyDe IO components and can be identified as the `MemoryMapped` decision model in IDeSyDe. To properly model the FPGA, it is crucial to incorporate its characteristic features of fixed hardware implementations and limitations on logic blocks BRAM. This work utilizes existing tool support and some tool extensions to achieve

the desired modeling.

At the basic level of modeling and inclusion of the FPGA during exploration, a global constraint must enforce DSE solutions to avoid exceeding the FPGA's resource capacities. If we let LB_{FPGA} represent the logic blocks capacity of the FPGA, define $a1, a2, \ldots, an$ as actors part of the application model, let $a1_{FPGA}, a2_{FPGA}, \ldots, ak_{FPGA}$ represent actors mapped to the FPGA and let A_cost$(a1_{FPGA})$, A_cost$(a2_{FPGA}), \ldots,$ A_cost(ak_{FPGA}) represent the logic block demand of implementing an actor on the FPGA. Taking inspiration from the floor planning problem of optimally placing rooms while not exceeding the total area of the floor [48], the desired global constraint can be formulated as a simplification to one dimension as:

$$\sum_{i=1}^{k} \text{A\_cost}(ai_{FPGA}) \leq LB_{FPGA} \tag{3.1}$$

Similarly, the BRAM capacity of the FPGA must be respected and enforced during exploration. Let the total BRAM capacity be represented by M_{FPGA}, and let M_cost(ai_{FPGA}) represent BRAM demand of ai_{FPGA} while re-using the actor mapping notation from Equation 3.1. The resulting equation can be formulated as:

$$\sum_{i=1}^{k} \text{M\_cost}(ai_{FPGA}) \leq M_{FPGA} \tag{3.2}$$

3.3.1 Modeling in ForSyDe IO

The following tool extensions are open source and available at `https://github.com/forsyde/forsyde-io`.

Through the addition of a `LogicProgrammableModule` trait to ForSyDe IO, the FPGA component can be represented. The trait properties include the available logic blocks, the size of available BRAM, and the common operating frequency for the BRAM operations. Here, the operating frequency can represent the reasonable clock frequency derived from the speed grade of the FPGA. With this in place, the parameters LB_{FPGA} and M_{FPGA} used in Equations 3.1 and 3.2 respectively are satisfied.

The completion of parameters for Equations 3.1 and 3.2 is done by adding the `InstrumentedHardwareBehavior` trait. It exposes properties related to what an actor requires to be implemented in hardware, in this case, on an FPGA. The hardware requirements are highly similar to the software

requirements shown in Listing 2.2, with the slight difference in naming the property `resourceRequirements`. Since modeling in ForSyDe IO is almost entirely the user's responsibility, there are no constraints on what can be specified in the trait's property. It simply provides a structure that *should* hold requirements of hardware-specific resources. In this way, it becomes natural to declare the required logic blocks and required BRAM for an actor's FPGA implementation (cost(ai_{FPGA})). The latency for one actor execution is specified in the same trait but as a separate parameter—possibly derived from $\frac{cyclesrequirement}{frequency}$. With this said, specifying an actor's FPGA implementation is a task for the user, but can sometimes be found in literature [49]. A more reliable option may be to consult FPGA implementation libraries, often readily available. These reusable implementations declare information about execution latency and required hardware resources, including BRAM and logic blocks [50].

3.3.2 DSE in IDeSyDe

The following tool extensions are open source and available at `https://github.com/forsyde/IDeSyDe`.

Let, for simplicity, the `AperiodicAsynchronousDataflowToPartitionedMemoryMappableMulticore` decision model be referred to as the "large" decision model. This decision model contains a `MemoryMappable` platform (described in Section 2.6.3) and SDF applications which have been analyzed to extract computation times and memory requirements to include in the decision model.

As described in Section 2.6.2, combined decision models are identified through a series of identifications producing a final system-level decision model that cannot be identified further. Extending this identification process to add an FPGA and actors implemented on the FPGA to the design space requires extraction of the constraint variables defined in Equations 3.1 and 3.2. Identification of the `LogicProgrammableModule` and the limits on logic blocks and BRAM for the FPGA, is achieved with a slightly extended version of the `MemoryMappableMulticore` identification rule and corresponding decision model. This is to utilize the already existing `MemoryMappable` design space including CPUs, memories, and communication which is still relevant after adding an FPGA. The new identification rule and decision model are *copies* of the original `MemoryMappableMulticore` and are intentionally not extended in

place. This is to preserve the design space of shared memory/multi-core platform architectures. Therefore, subsequent identification rules and decision models are necessary to propagate the new decision model throughout the identification chain through mostly only naming changes. In contrast, actors specifying a hardware implementation alternative are identified at the lowest level in the identification hierarchy because of their independence from other system specifications. These specifications are handled by a completely new identification rule and corresponding new decision model as there is no suitable decision model to extend. These two extensions for the platform and application traits are sufficient for the final system-level decision model which contains adequate information about the global constraints to be used during exploration.

IDeSyDe's bidding step acknowledges that the new system-level decision model, which includes the FPGA constraints, can be explored. The exploration is handled by a *new* CP explorer specifically created for this thesis using the optimization objectives and constraints for the original large design space together with the new FPGA constraints. During exploration, proposed design alternatives for mapping actor execution to the FPGA are constrained. With this in place, DSE solutions are ensured to respect the logic block and BRAM capacities of the FPGA.

Finishing the IDeSyDe process is handled in the reverse identification step. Already existing reverse identification rules for the large decision model are kept intact while making a copy with extended support to differentiate between an actor mapping to a software or hardware processing module.

3.4 Application Modeling

Application modeling is integral in validating that the DSE solutions are working correctly. It is also relevant to model a realistic application to evaluate the practical usage in realistic design scenarios.

3.4.1 Ensuring Correct DSE Solutions

SDF applications with two actors are designed to test and evaluate the novel DSE support. In the defined test cases, an actor's software implementation states its required instructions and how many of each, while hardware implementations specify logic block demand, BRAM usage, and execution latency. The code size of software implementations is omitted from the test

case specifications as they will undoubtedly fit somewhere on the platform's ≈ 4.5 GB of memory. The following test cases should be evaluated:

TC1: Separated HW/SW Mappings

Specifying one actor to only support a hardware implementation and the other to only support a software implementation and expecting them to map to hardware and software respectively.

The actor parameterization is arbitrary, as the test case targets different execution domains. Figure 3.3 visualizes the test case.

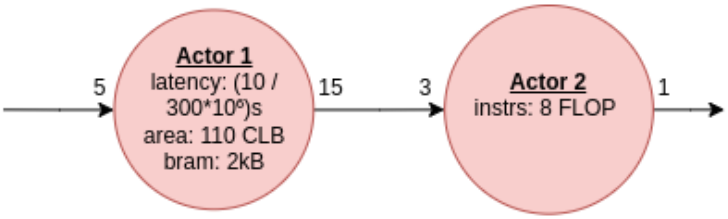

Figure 3.3: The SDF graph for the first test case. "Actor 1" can only be implemented in hardware and "Actor 2" can only be implemented in software.

TC2: Preferable HW Implementations

Specifying favorable hardware implementations compared to software implementations and expecting both actors to be mapped to hardware.

What is favorable is subjective; however, parameterization can be adjusted to have one implementation outclass another by large fractions of a second. In this test case, "Actor A" requires 8K Floating Point Operations (FLOPs), which executes at fastest in $\frac{8000*2}{1.5*10^9} \approx \frac{1}{93'750}$ seconds, given that a FLOP takes 2 cycles on one of the APU cores clocked at 1.5GHz. Compared with the execution latency of the hardware execution ($\frac{10}{200'000'000}$), the hardware implementation is more than 2000 times faster than the software implementation. Figure 3.4 visualizes the test case.

TC3: Exceeding FPGA Resource Limits

Specifying hardware implementations for both actors that require more resources than the FPGA provides and expecting no DSE solutions. This is divided into three scenarios: only exceeding BRAM capacity, exceeding the logic block capacity, and exceeding both capacities.

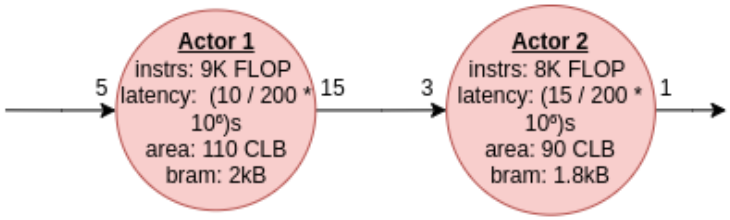

Figure 3.4: The SDF graph for the second test case. The hardware implementations of both actors produce results more than 2000 times faster than the corresponding software implementations.

Given that the FPGA on the XCZU9EG MPSoC has 600K logic blocks and 4MB of BRAM, exceeding any, or both, of these capacities is sufficient. Here, "Actor 1" specifies more BRAM usage than available 1 GB $>$ 4MB, and "Actor 2" specifies more logic blocks than available 601K $>$ 600K. Figure 3.5 visualizes the test case. The other variations of this test case adapt numbers as required to fit or not fit the FPGA limits.

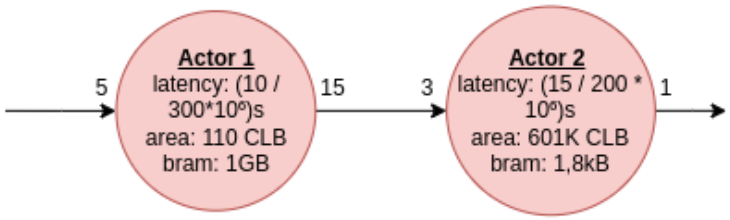

Figure 3.5: The SDF graph for the third test case. "Actor 1" exceeds the BRAM capacity and "Actor 2" exceeds the logic block capacity.

Evaluation of the DSE should also target differences in the platform. Since there is a clear distinction between the processing system and programmable logic it is expected to impose higher bandwidth than communication within one of the sides. The following test cases are to be evaluated:

TC4: Increasing Communication Overhead

Giving one actor a favorable hardware implementation and the other a favorable software implementation, while specifying low bandwidth for the switches connecting the board's processing system and programmable logic. This test case should evaluate how mappings

can change due to bandwidths by comparing its solutions to TC5—where the bandwidth is unchanged. Both actors expect to be mapped to only hardware or software to overcome the communication overhead. Figure 3.6 visualizes the test case's actors.

It is observed in Figure 3.2 that the LPD switch connects the RPU to the programmable logic and the FPD switch connects the APU to the programmable logic while the FPGA can use both the "CCI" switch and FPD switch to reach the processing system. Specifying that these switches require one second to send one "bus width" of data (e.g. 64-bit or 128-bit) and lowering their operating frequencies generates the prerequisites needed for mapping both actors to the same side of the chip. Actors can utilize the embedded BRAM or external DDR4 memory in the programmable logic for sharing data without suffering from the decreased switch bandwidths.

Further specifying hardware and software implementations for both actors is a question of what is favorable yet again. To isolate the test case to evaluate mappings based on communication bandwidth, "Actor 1" requires one second to execute in hardware, and "Actor 2" requires 800K FLOPs in software. Both also have a secondary implementation alternative that takes nearly no time and is thus highly preferable.

Relating this configuration to the exploration objectives of IDeSyDe, the communication overhead must reduce the application throughput more than mapping an actor to its "undesirable" processing element. Otherwise, there are two solutions: mapping to the desirable processing element (FPGA, CPU) with throughput t and mapping to the same processing element with throughput $< t$.

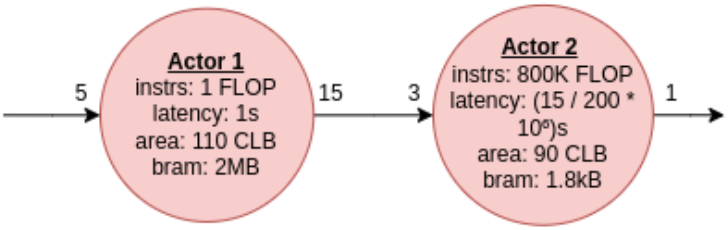

Figure 3.6: The SDF graph for the fourth and fifth test cases. Most focus is on the platform parameterization, but the actors imply one *unrealistically* favored implementation alternative each.

TC5: Normal Communication Overhead

Same actors as TC4, leaving the switches unchanged from the original specification. Figure 3.6 visualizes the test case's actors. It is more likely that the execution can execute on their preferred processing unit, but most important is the resulting difference to TC4.

3.4.2 Realistic application

The implementation of DSE support for mapping and scheduling applications on both FPGAs and CPUs is the critical goal of this project. However, once validated, the video streaming application in Figure 3.7 should be used as input to the DSE to evaluate a realistic scenario using the novel extensions. It contains the three aspects of Saab's application requirements presented in Section 2.5: pixel operations, stencil operations, and a Convolutional Neural Network (CNN).

The input stream to the SDF application is sourced from a 4K camera module capable of streaming a compressed Motion JPEG (MJPEG) format in 4K@30Hz through certain interfaces [51]. Before manipulating the pixels in the video stream, a pre-configured FPGA Intellectual Property (IP) block compatible with the XCZU9EG MPSoC decodes the MJPEG format without losing any bandwidth of the stream [52]. All proposed SDF actors in Figure 3.7 have software and hardware specifications that naturally differ but are intended to have maximal equivalence in practical functionality. One exception is the synchronization actor which only runs in software to collect and place grayscale pixels in certain memory.

- **Pixel Operation** - RGB to Grayscale conversion
  Software: Assuming the weighted RGB to grayscale conversion formula $gray = (0.3{*}R){+}(0.59{*}G){+}(0.11{*}B)$ [53], 5 floating point operations are required. An additional 3 floating point operations are required to convert input RGB values ranging from $[0, 255]$ to $[0, 1]$. The memory demand for the software implementation is lightweight, $\leq$ 100 bytes, and no internal buffer is needed.
  Hardware: 300MHz circuit, requiring approximately 100 logic blocks, converting one pixel to grayscale in $\approx$ 1 cycle, approximated from the specification operating on complete frames [54]. No memory is required during computation.

- **Stencil Operation** - 3x3 Sobel Filtering on grayscale input
  Software: The bare-metal 3x3 sobel filtering C implementation mainly divides the heavy computation into two sections, each requiring 9

integer operations resulting in a total of 18 integer operations [55]. The memory demand is a buffer storing data for at least three adjacent rows and columns required to compute the 3x3 filter on a target pixel. As described in Section 2.5.1, this is approximated to $3 *$ <4K_width> $= 3 * 2160\text{bytes} = 6.4kB$.
Hardware: 300MHz circuit, requiring 132 logic blocks and 7kB BRAM and computes the 3x3 sobel kernel for one pixel in $\approx$ 1 cycle [56]. A cycle requirement approximation is derived from the specification operating on complete frames with a rigorous pipeline allowing for parallelism. The memory demand for the hardware implementation is equal to the software implementation demand of $6.4kB$ since pixel buffering is inevitable.

- **Convolutional Neural Network**
  Important note: The proposed CNN does not operate on the full-size 4K frame. Instead, an image down-size application can be assumed to execute before the CNN is given its input, but the resize itself is omitted to not over-complicate the scenario.
  Software: The LittleNet CNN, referenced as hardware implementation, uses 19 convolution blocks with a kernel size of 3x3 and is quite small to fit an embedded context with limited resources [57]. Modeling the number of FLOPs required for a convolution network is dependent on the type of block and parameters, thus very complex [58]. Therefore, the ShuffleNetv2 CNN is used to approximate a low-demand network (like LittleNet) as it requires the least, "only" 21 million, FLOPs in the given study [58]. The memory demand should not differ significantly from the hardware implementation, thus the CNN model can be assumed to require $\approx$ 756kB.
  Hardware: 300MHz circuit on the DPU as part of the programmable logic located on the XCZU9EG MPSoC, requiring approximately 5650 logic blocks and performs object detection at 53 FPS on $\approx$ 550x550 pixel images [57, 59]. This gives an average cycle count of $\approx$ 5.6 million per frame. The memory demand for the FPGA implementation requires $\approx$ 756kB of BRAM blocks to host the trained model and make predictions [59].

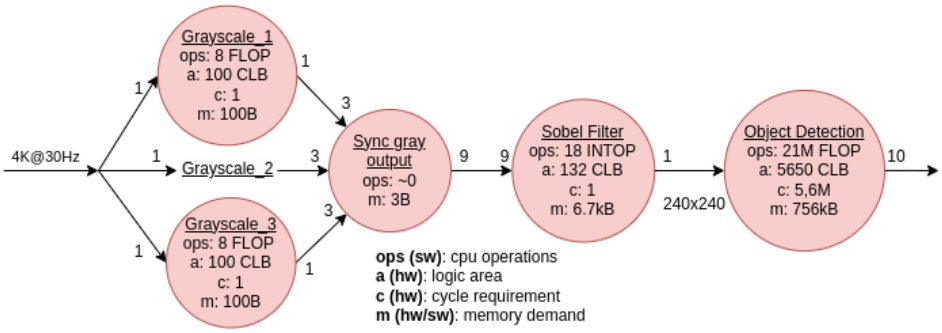

Figure 3.7: The proposed realistic embedded application that streams 4K resolution video frames and performs gray-scaling, the Sobel filter, and object detection. Before sobel filtering, a software implementation arranging the parallelized grayscale output is assumed. A resize operation before the object detection is also assumed.

3.5 Development Setup

The following project setup is available at `https://github.com/saab/dse-for-mpsoc-thesis-2024/`.

System modeling in ForSyDe IO is performed through a Java project, created through Gradle [60], which defines the ForSyDe IO library as a project dependency. This application serves the purpose of interfacing with the library and providing utilities for managing the specifications for different purposes. The main focus is on three submodules of ForSyDe IO:

- `java-core` - Provides classes to create the system graph, interfaces that define the outline for concrete system views, and reading models from, or writing them to, the file system.
- `java-libforsyde` - Provides a library of traits for application onto the system graph.
- `java-graphviz` - Used for creating a visualizable format of the system model

3.5.1 Managing Verbosity

System modeling with `libforsyde` traits is quite verbose, as shown in Listing 3.2, and also repetitive once the system specifications grow large.

Removing the implemented loop required to create a multi-core CPU further proves the point of verbosity. Introducing a set of wrapper functions is thus very valuable for creating the system model. Wrapper functions enable more concise specifications without loss of flexibility; arguably also making it easier to modify the specifications since unimportant assumptions and information are hidden. Listing 3.1 shows the wrapper function implementing the code in Listing 3.2. This wrapper function, along with similar functions for other components, is part of the API. Further, defining these functions as non-static members of a custom class makes specifications less error-prone thanks to internal validation. Consistent maintenance of the custom object is also crucial for the ease of interaction with the ForSyDe IO library, for example, connecting system components by their unique identifiers.

```
platform.AddCPU(
    RPU_NAME,
    RPU_CORES,
    (long) 600 * Units.MHz,
    Map.of(
        Requirements.SW_INSTRUCTIONS,
        Map.of(
            Requirements.FLOP, 2,
            Requirements.INTOP, 1
        )
    )
);
```

Listing 3.1: Example of a wrapper function for creation of an `InstrumentedProcessingModule` that simplifies the system graph creation.

```
for (int i = 0; i < numCores; i++) {
    String coreName;
    if (numCores > 1) coreName = name + "_C" + i;
    else coreName = name;
    var core = InstrumentedProcessingModule.enforce(
        sGraph, sGraph.newVertex(coreName));
    this.viewers.put(coreName, core);
    this.greyBox.addContained(
        Visualizable.enforce(core));
    core.maximumComputationParallelism(1);
    core.operatingFrequencyInHertz(frequency);
    core.modalInstructionsPerCycle(
```

```
13          instructionSet.entrySet().stream()
14              .collect(
15                  Collectors.toMap(
16                      iSet -> iSet.getKey(),
17                      iSet -> iSet.getValue().entrySet()
18                          .stream()
19                          .collect(
20                              Collectors.toMap(
21                                  inst -> inst.getKey(),
22                                  inst -> 1.0 /
                                      inst.getValue()
23      ))));
24      var runtime = SuperLoopRuntime.enforce(sGraph,
25          sGraph.newVertex(coreName + "_Scheduler"));
26      this.greyBox.addContained(
27          Visualizable.enforce(runtime));
28      runtime.addManaged(core);
29      this.CreateEdge(core, runtime);
30  }
```

Listing 3.2: The native interaction with the ForSyDe IO library that is executed when calling the wrapper function shown in Figure 3.1.

3.5.2 Experimental Setup

The system specifications are easily constructed using the Gradle application and integrated with the experimental workflow shown in 3.8. Since interaction with the ForSyDe IO library may have different purposes, the Java application accepts command-line arguments. The following operations are supported:

```
- build <platformType> <applicationType>
- to_kgt <path>
- parse_solution <solutionPath>

platform types: 'mpsoc',
application types: 'tc1', 'tc2', 'tc3', 'tc45', 'real'
```

Specifically, building various test cases in rapid succession is greatly facilitated by having the platform and application types available as command-line arguments. In addition to creating the system specifications, the application supports the conversion of `.fiodl` files to `.kgt` files and the

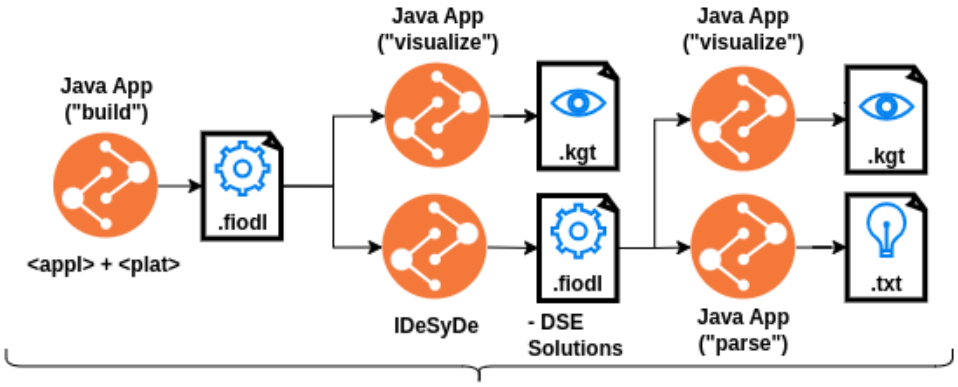

Figure 3.8: A visualization of the experimental workflow executed by a bash script. Visualizations are provided through `.kgt` files while `.fiodl` files are used for further processing.

parsing of DSE solutions. The workflow can thus execute automatically with a bash-script, assisted by the multi-operational support from the Java application. Artifacts from execution include specifications and visualizations of the original input models and DSE solutions presented in the terminal and created text files.

Chapter 4

Results

This chapter presents only the results, leaving the discussion for Chapter 5. First, the fulfillment of the goals outlined in Section 1.4 will be addressed. Secondly, the DSE results for the test cases and the realistic application are presented.

4.1 Fulfillment of Goals

Out of the seven goals defined for the project, all mandatory goals (G1-G4), and one (G5) out of three optional goals were successfully met. The following list breaks down the goals and reasons why they were met (green) or unmet (red).

G1: Decide parameters that can model the key characteristics of the FPGA processing unit and functional implementations on the FPGA.

The FPGA component, as part of an MPSoC, has been modeled with the following parameters:

- Logic block available on the FPGA chip
- Available BRAM embedded on the FPGA chip
- Required logic blocks for implementing an actor in hardware
- Required BRAM for implementing an actor in hardware
- Execution latency in hardware for a functional specification

These parameters target the FPGA as one type of processing unit located in the programmable logic. However, representing the hardware-programmable DPU also found in the programmable logic is partly supported due to the generalization of ForSyDe IO trait specifications.

More design considerations for FPGAs that were not further investigated are presented in Section 2.2.

G2: Develop a software application that provides easy interaction with the ForSyDe IO library to create the platform and application specifications.

A Java application focusing on convenient interaction with the ForSyDe IO library has been created. It provides a suite of functions to facilitate the creation and interconnection of platform components and application processes, focusing specifically on later integration with IDeSyDe. The project can be accessed at `https://github.com/saab/dse-for-mpsoc-thesis-2024/`.

G3: Extend ForSyDe IO and IDeSyDe to interpret the new platform characteristics and account for additional mapping possibilities.

The parameters required to model an FPGA (G1) were successfully added to the ForSyDe IO library and IDeSyDe was extended to respect the new characteristics in the exploration. Most important are the tool extensions to support Equations 3.1 and 3.2.

G4: Create test cases that define application and platform parameters that have expected DSE solutions in the context of the novel modeling.

Section 3.4.1 defines five test cases that evaluate fundamental aspects of the added modeling and DSE support. Details of performed DSE on the test cases are found in Section 5.4.

G5: **(Optional)** Create a realistic application specification that aligns with the given sample functionality from Saab to evaluate the DSE capabilities.

Section 3.4.2 defines a realistic video streaming application including image processing and object detection. Details of performed DSE are found in Section 5.4.

G6: **(Optional)** Develop a method for assessing the validity and correctness of the non-trivial DSE solutions.

Throughout the development of extensions, it became clear that the code base for IDeSyDe's identification and exploration procedures was too large and complex. The complexity primarily arose from the abstract information exchange between the programming languages constituting different parts of IDeSyDe. Thus validation of the results must rely on,

e.g., the schedulability proof from [5] which is also already incorporated in IDeSyDe. However, validating the correctness of input system specifications and DSE calculations is not in place.

G7: **(Optional)** Perform DSE on more test cases to conclude design alternatives.

The realistic application designed for goal "G5" theoretically counts towards achieving this goal but is insufficient in quantity.

4.2 Experimental Results

Evaluation of test cases TC1 through TC5, defined in Section 3.4.1, ensured that the tool extensions properly integrate the novel characteristics in the DSE. Furthermore, the realistic video streaming application in Figure 3.7 did not produce any solution in under two hours but still showed its usefulness and realism with only software implementation. All conducted experiments use a ForSyDe IO representation for the abstract platform model from Figure 3.2 and visualized experimental results use a less verbose version of the platform abstraction, see Figure 4.1. One slight modification is that the BRAM is separated from the FPGA component to make it clearer for mappings.

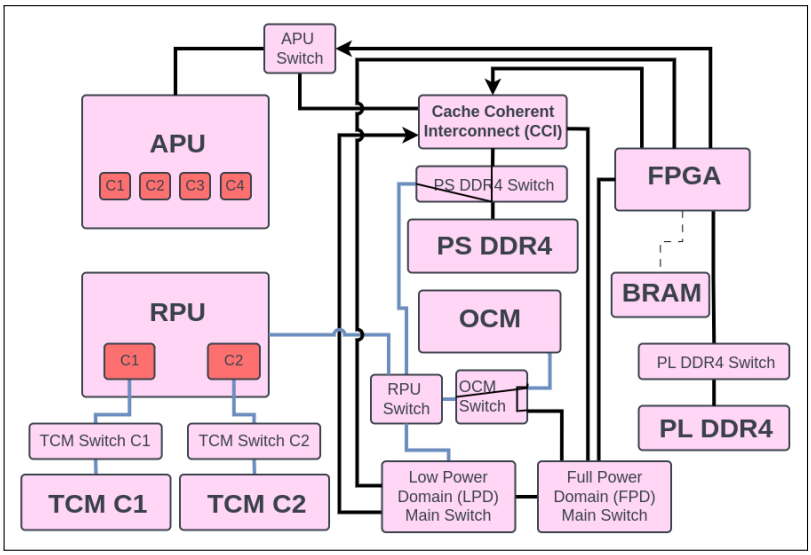

Figure 4.1: A less verbose representation of the platform, compared to the original representation in Figure 3.2, used for visualizations of experimental DSE results.

Setup for IDeSyDe

Experiments are performed with the newly created CP explorer using the MiniZinc constraint modeling language [61]. Unless explicitly stated, no timeout is set for the exploration and thus it runs until the entire design space is exhaustively searched. Also, the results presented are limited to including memory and execution mappings for the test cases to focus on the extensions. Some test cases present throughput when the Pareto front is relevant. Scheduling is irrelevant given the size of the application models.

The experiments are run with a computer provided by Saab, a Dell Precision 3581 with specifications including:

- Intel Core i9-13900H Processor [62]:
  - 6 performance CPUs @ $\leq$ 5.4 GHz
  - 8 efficiency CPUs @ $\leq$ 4.1 GHz
  - 24 MB processor cache
- 2 x 16 GB SODIMM memory

4.2.1 TC1: Separated HW/SW mappings

The DSE for this test case presents one Pareto point after exhaustively searching the design space in little to no time. Most important is that the actor execution is delegated to the expected processing units, see Figure 4.2. The inter-actor data channel is mapped to the BRAM, located inside the FPGA. Execution mappings are as expected, thus the test case is considered **passing** although the implementation would not work in practice.

Through repeated execution of the test case, it is observed that the memory mappings for the A1-A2 buffer and the code storage for relevant actor A2 changes. From what has been collected, the mapping seems random between targeting the BRAM and DDR4 memories.

4.2.2 TC2: Preferable HW implementations

The DSE for this test case presents one Pareto point after exhaustively searching the design space in little to no time. Mappings in Figure 4.3 show the actors being mapped to the FPGA and the inter-actor data channel is placed on the BRAM, located inside the FPGA. Mappings to the FPGA are as expected, thus the test case is considered **passing**.

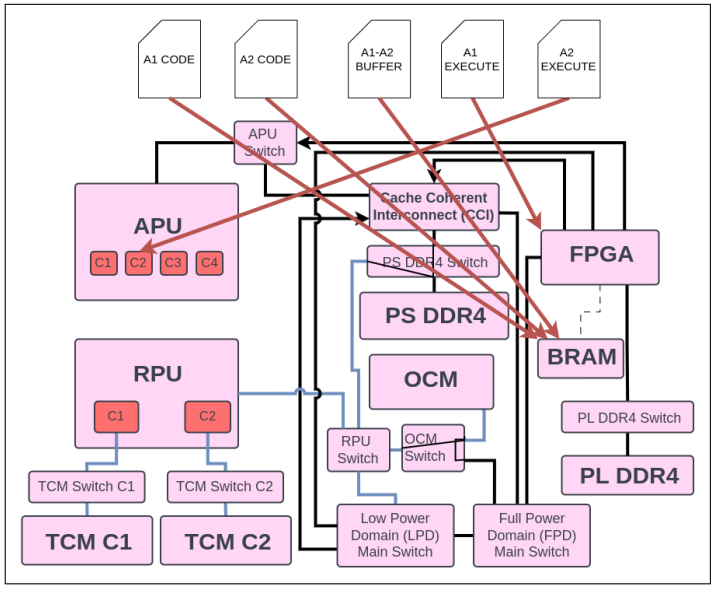

Figure 4.2: Visualization of the DSE result for TC1. The actors A1 and A2 are mapped to the FPGA and APU respectively, and BRAM is used for the inter-actor data channel.

Through repeated execution of the test case, it can be observed that the memory mapping for the A1-A2 buffer does change over execution. From what has been collected, the memory mapping is seemingly random between all memory elements when selecting the placement of the communication channel.

4.2.3 TC3: Exceeding FPGA Resource Limits

IDeSyDe exits almost instantly and provides no solutions when a resource violation occurs. The test case is **passing** since all variations in exceeding FPGA resources yield the expected result.

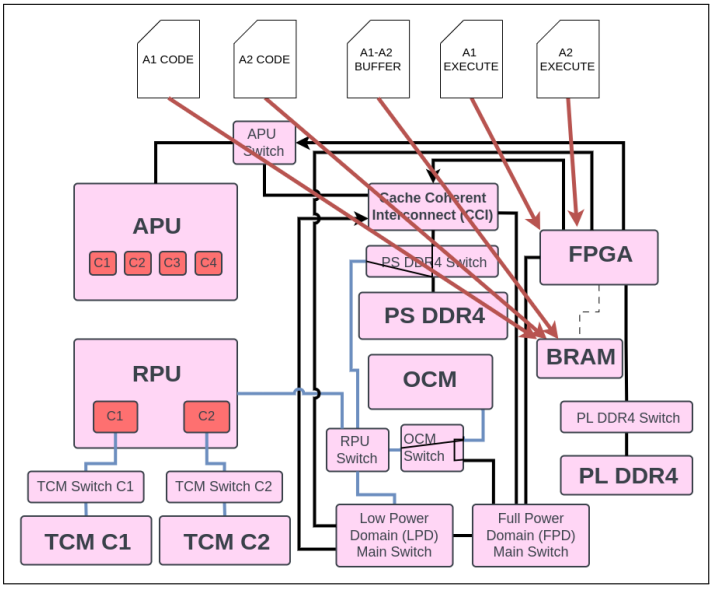

Figure 4.3: Visualization of the DSE result for TC2. Both actors are mapped to execute in hardware and the inter-actor data channel is mapped to BRAM.

4.2.4 TC4: Low PL-PS Bandwidth

The DSE for this test case presents one Pareto-dominant solution after exhaustively searching the design space in little to no time. However, over multiple runs, only the execution mapping, to the FPGA, is deterministic while the inter-actor data channel either maps to the BRAM (Figure 4.4) or the processing system's DDR4 memory (Figure 4.5). Other test runs map the data channel to TCM. The variation in memory mappings arises from what is believed to be equally good solutions with an assumed randomized selection but the throughput varies. Table 4.1 highlights these differences. Since execution mappings are solely on the FPGA, the test case is considered **barely passing** given that the inter-actor data channel sometimes maps to memories that are further away than others and the communication passes through the switches with impaired performance.

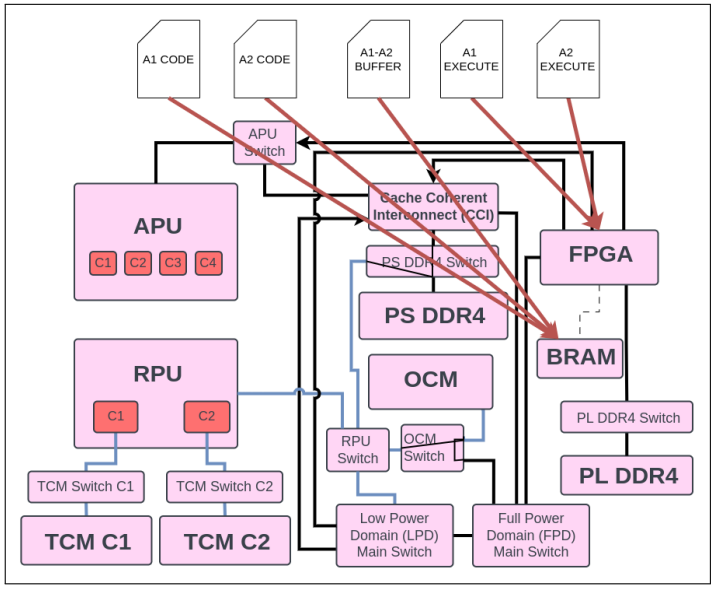

Figure 4.4: Visualization of the Pareto-dominant solution produced by a run of TC4. It maps the inter-actor data channel to BRAM and actors to the FPGA.

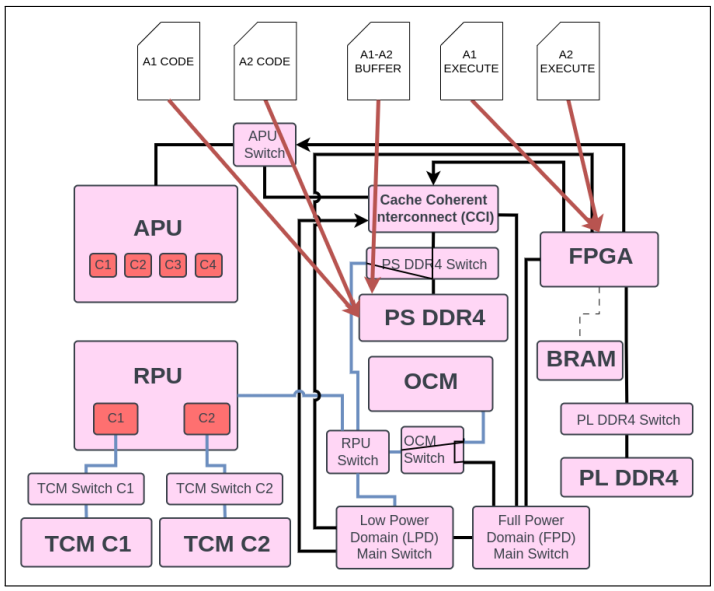

Figure 4.5: Visualization of the Pareto-dominant solution produced by another run of TC4. It maps the communication channel to the programmable logic's DDR4 memory and both actors are mapped to the FPGA just like the solution in Figure 4.4.

	Buffer	Throughput A1	Throughput A2
Example 1	BRAM	1B/s	5B/s
Example 2	TCM	≈ 0B/s	≈ 0B/s
Example 3	PL DDR4	1B/s	5B/s

Table 4.1: Memory mappings for the inter-actor data channel for TC4 over multiple runs with inconsistent throughput.

4.2.5 TC5: Normal PL-PS Bandwidth

The DSE for this test case presents two Pareto-dominant solutions, shown in Figures 4.6 and 4.7 and compared in Table 4.2, after exhaustively searching the design space in little to no time. Two solutions are found as they dominate one objective each: throughput and used processing elements. This is expected because of the favorable two-processor mappings inferred from the test case while one processing unit is also possible. The first solution maps both actors to one core of the RPU and uses the OCM for the communication channel. The other solution maps the actors to their desired processing unit, leading to increased throughput but more processing units, and maps the communication channel to the BRAM.

The test case is considered **passing** since the solutions differ reasonably from TC4 in Section 4.2.4. Reverting the switch performances to their original specifications yields throughput good enough on the actors' preferred processing units and produces another Pareto-dominant solution.

	A1	A2	Buffer	Throughput_A1	Throughput_A2
Sol 1 (Fig 4.6)	RPU_C1	RPU_C1	OCM	37B/s	185B/s
Sol 2 (Fig 4.7)	RPU_C1	FPGA	BRAM	≈12.6 MB/s	≈62.9 MB/s

Table 4.2: Metrics for the two solutions to TC5, one solution having dominant throughput while the other dominates the number of used processing units.

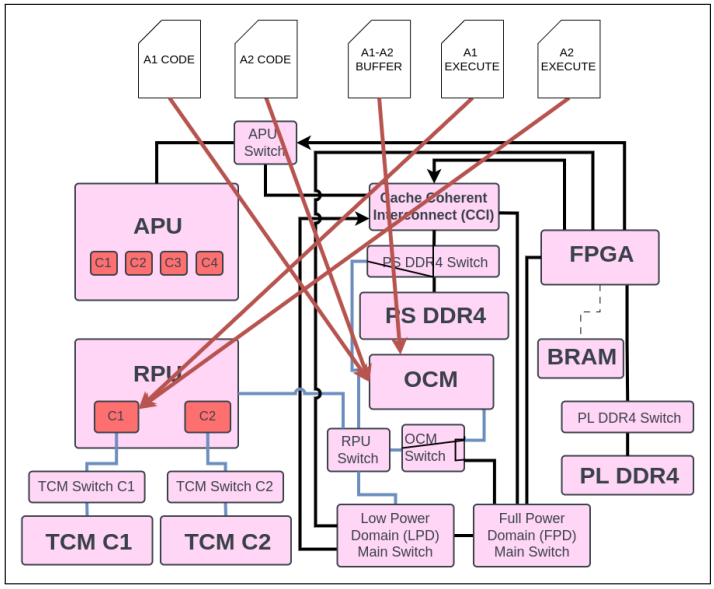

Figure 4.6: Visualization of one Pareto-dominant solution to TC5 with fewer processing units used. It maps both actors to core 1 of the RPU and uses the OCM for the inter-actor data channel.

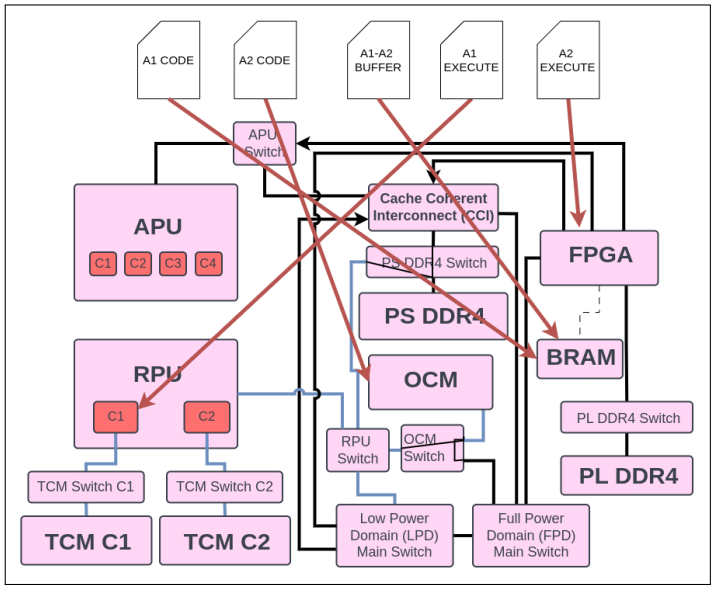

Figure 4.7: Visualization of the other Pareto-dominant solution to TC5 with higher application throughput. It maps the actors to their preferred processing unit and uses the BRAM for the inter-actor data channel.

4.2.6 Realistic application

IDeSyDe did not produce any solutions within a timeout of two hours for the realistic video streaming application presented in Section 3.4.2. The identical SDF application with only software-programmable actors could be explored in IDeSyDe with meta-heuristics and produces roughly 20 solutions in under 60 seconds, of which some are Pareto-dominant. It can safely be concluded that the computational complexity of the novel hardware-software exploration does not scale well with problem size. Altering parameters for the various actors to require less memory, specifying that some actors are only software-programmable, and even reducing the number of actors still did not yield results near the time for meta-heuristics. The upcoming section, Section 4.2.7, shows results for some related benchmarking to this identified issue.

4.2.7 Benchmarking the Explorer

Based on the findings in Section 4.2.6, a set of benchmarks was selected (non-systematically). Despite that no real method was used for defining them, they present interesting insights about the implemented CP explorer.

The benchmarked applications comprise an arbitrary-length actor chain, where all actors have `production=5` and `consumption=5`. All actors in the chains have software implementations, whereas some also have hardware implementations. This is based on the fact that the meta-heuristics explorer was fast for software implementations. Therefore, by increasing the number of hardware implementations it can be expected that the design space will grow in size and lead to decreased performance due to double mapping alternatives for each actor.

The benchmark, presented in Figure 4.8 shows how the execution time (y-axis) varies for a variable-length actor chain (x-axis). Each chain of length n is measured n times to record how 1-n actors having hardware implementation alternatives (annotations) affect the execution time. These benchmarks only ran for one iteration and the time is recorded when the first solution is found.

A linear increase in execution time is observed for >3 actors, translating into an exponential trend since the y-axis is logarithmic. Only one configuration for eight actors found a solution in under two hours. Investigation of the annotations also shows that execution time is not affected in any particular way by the number of actors having hardware implementations. Given the primitive SDF graphs, the exploration is assumed to be overly demanding for “normal” applications because they have different production and consumption, and the graphs are often not represented as a chain—which

introduces complexity.

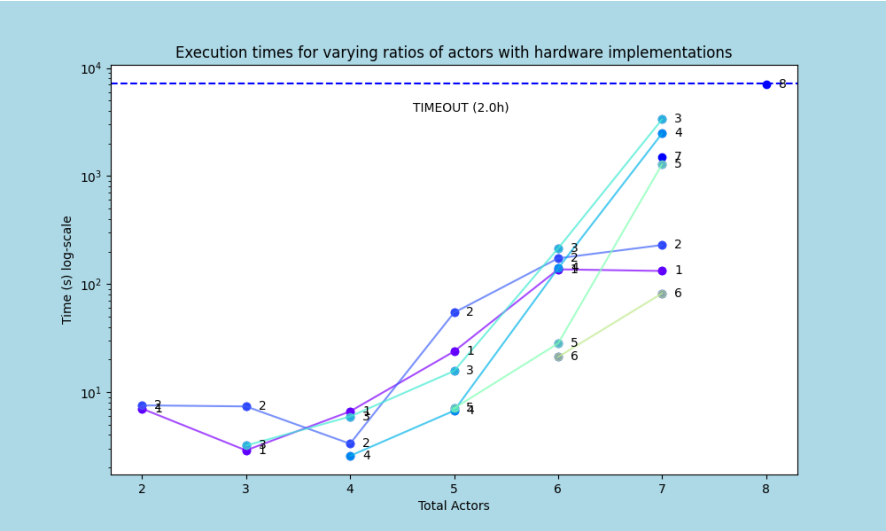

Figure 4.8: Benchmark for 2-8 actors with a time limit of two hours on the exploration. All actors have specified software implementations, and the data annotations indicate how many actors also have hardware implementations for the given number of actors.

The benchmark, presented in Figure 4.9, uses the same data as Figure 4.8 but hardware implementations and actors have changed places. In this way, it becomes easier to investigate patterns in how the execution times change with additional hardware implementations. However, there is no pattern observed as the execution time both increases and decreases while strictly increasing the number of hardware implementations.

Based on results shown in Figures 4.8 and 4.9 both four actors and six actors present either small or large variance in the results. New benchmarks were therefore conducted to investigate these cases deeper for one fixed number of hardware implementations, using ≈300 data points each. The result in Figure 4.10 presents a normal distribution of execution times with some outliers; a highly probable effect of simply varying workloads on the host computer. For six actors, the results depict tendencies of a Poisson distribution as the occurrences decrease gradually to the highest execution times. This presents the possibility of a factor within the search process that normally does not affect results but still sometimes. What this factor might be is however unknown.

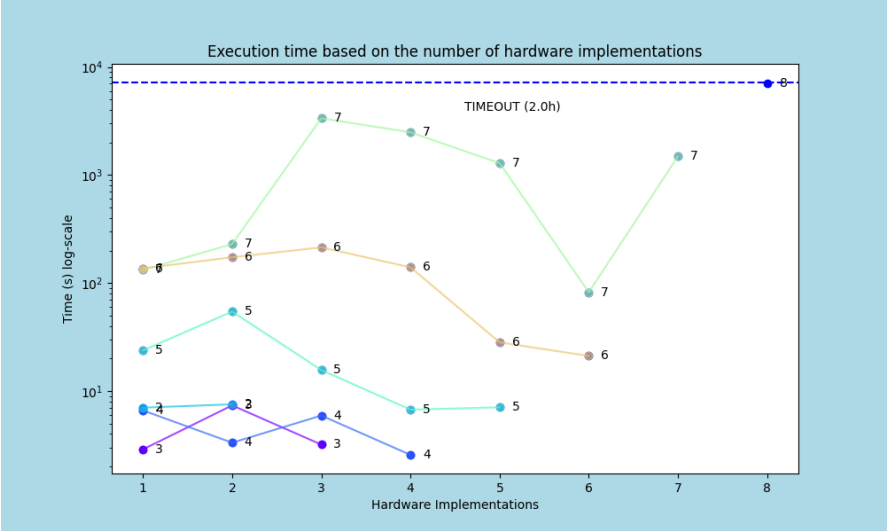

Figure 4.9: The same benchmark data as Figure 4.8. The difference is that the x-axis now contains the number of hardware actors used and the data annotations show the number of actors for the given data point.

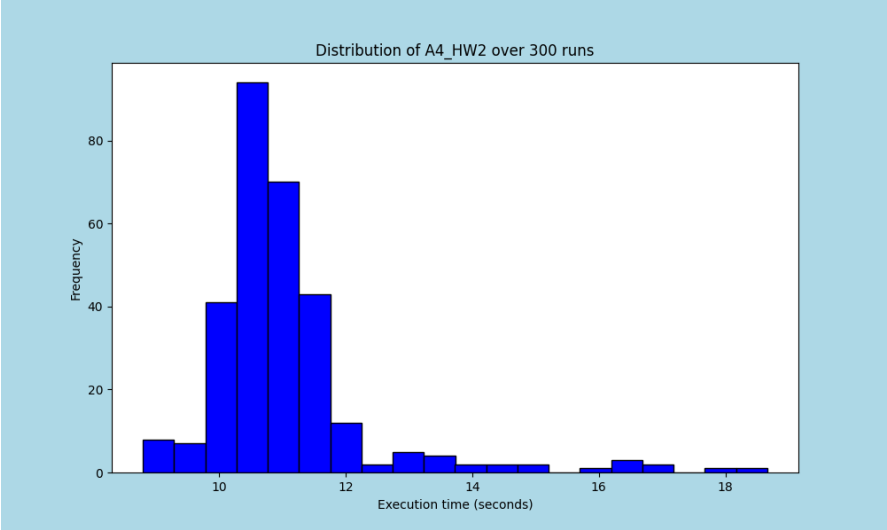

Figure 4.10: Execution time benchmark for four actors where two have specified hardware implementations. The benchmark ran for 300 iterations.

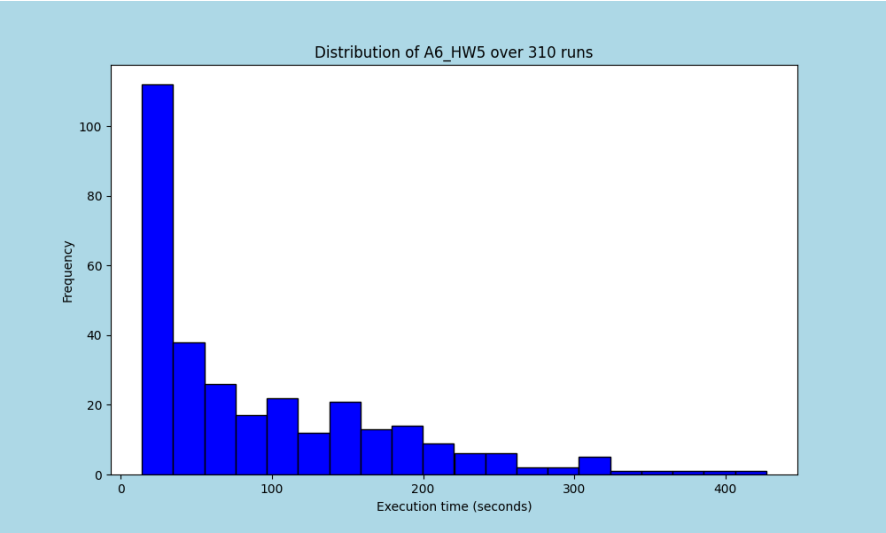

Figure 4.11: The same benchmark as Figure 4.10, but with the difference that the input application has six actors and five who have specified hardware implementations.

Chapter 5

Discussion

Significant insights follow the integration of the platform and application modeling with the design tools. Important topics are also derived from the DSE results, which provide a basis for discussing the potential of using IDeSyDe in practical design scenarios.

5.1 Abstract Platform Model

While modeling aims to represent complex concepts abstractly, simplifications in the abstract platform model are also caused by difficulties in finding relevant information in technical specifications, as exemplified in Figure 5.1. This limits the amount of detail to be reasonably included in the abstract model, making it necessary to prioritize what to model given that specifications outside the tool domains require non-trivial programming efforts. Therefore, the modeling primarily focused on finding the CPU and FPGA parameters necessary to extend ForSyDe IO and IDeSyDe. The secondary modeling priority was the collection of parameters based on what is already supported by the tools, such as memory and network switch instrumentation.

Considering that there was no previous work to extend when creating the abstract model, the modeling result shown in Figure 3.2 is viewed as a good initial model of the XCZU9EG MPSoC chip. It successfully captures *high-level* parameters for the CPUs, FPGA, platform memories, and interconnection network. Related discussion on the topic is found in Section 5.2.1.

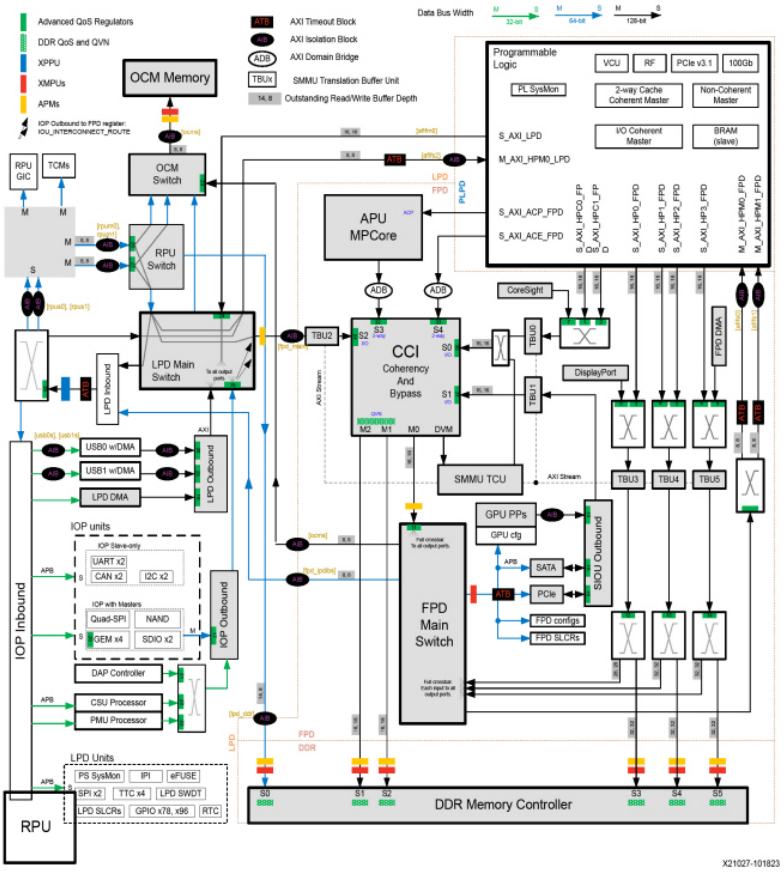

Figure 5.1: A sample datasheet showcasing the information complexity required to comprehend before pursuing modeling. Source [33].

5.2 System Representations

This section covers how accurately application and platform specifications, with both old and new modeling aspects, can model reality.

Instruction Specifications

Specifying an actor's software implementation for execution on CPUs requires specific ForSyDe IO traits to later calculate the execution times during the DSE. The following lists showcase some of the necessary parameters: the types and quantities of required instructions, and the number of CPU cycles each instruction requires.

- Software Implementation

  Necessary instructions for one task execution:

  - 9x Memory read
  - 3x Memory write
  - 23x FLOPs

- CPU

  Required clock cycles to execute different instructions:

  - Memory read: 2 cycles
  - Memory write: 3 cycles
  - FLOP: 4 cycles

This is inconvenient since some recent CPU architectures, like the ARM Cortex A53 (APU) on the XCZU9EG MPSoC, do not specify the number of cycles required per instruction, *possibly* due to a very dynamic behavior in the execution pipeline. This implies the usage of an unofficial method to derive these parameters. Approximations have been made for the ARM Cortex A7 through benchmarking due to the lack of official documentation [63]. Those results were used for the CPU specifications for the XCZU9EG MPSoC, but this is unreliable since the A53 and A7 have different architectures.

Also, specifying the necessary instructions for a software implementation should ideally be based on generated assembly code to be precise. This task was considered too complex for software specifications of actors in the realistic application as it would require C code to be compiled for execution on the APU and RPU. Instead, approximations were made which reduces the precision and reliability. In comparison, hardware execution times were decided to be specified by the user due to the many factors influencing FPGA implementations. With this reasoning, instruction specifications for CPUs and software implementations could be replaced in favor of pre-defined execution times to overcome the noted complications. However, such execution times add overhead in the form of practical benchmarking, and can not provide any guarantees due to unpredictable influence from other platform workloads.

There is a clear trade-off between benchmarking and possibly having difficulties getting the instruction specifications correct. Using design tools should ideally automate the design process to its full extent. While this might be an identified limitation, it could also implicate potential for tool improvements. The specification of hardware implementation latency could

be assumed for software implementations to at least increase the modeling accuracy. In case of unreliable instruction specifications, benchmarking would help. Generally, it can be argued that benchmarked software and hardware execution times produce good results as they are very close to reality and capture parameters that are challenging to model.

Mismatches in Application Modeling

For reference, typical FPGA implementations employ an execution pipeline for even the primitive operations that enable the processing of new data before the completion of previous data [64]—a feature not found in software implementations due to sequential machine instructions. Figure 5.2 shows an FPGA implementation of subtraction using a three-stage pipeline: `subVI A`, `subVI B`, `subVI C`, instead of just one block.

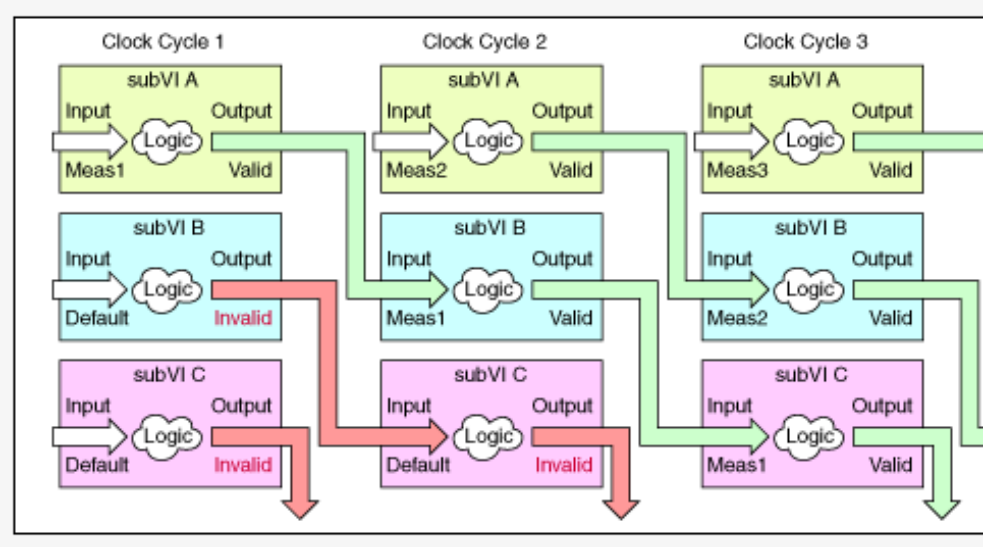

Figure 5.2: An FPGA implementation using a three-stage pipeline for its logic. Source: [64].

Considering that an SDF graph should define actors based on the separation of functionality, having a shared SDF graph for hardware and software implementations is not optimal but also unavoidable with current support. Hence "software implementation optimal" graphs will cause an underestimation of the realistic throughput of hardware implementations. This surely is an application modeling limitation but specifications of hardware latencies can be tuned to justify their throughput better. Therefore, the

Grayscale and Sobel actors were defined to operate on a single pixel, and one grid respectively, which is assumed to help better model the realistic application's throughput. One notable limitation in modeling the realistic application was the inability to model a hardware pipeline for the convolution layers in the object detection actor. This is because a corresponding software implementation obviously cannot predict with partial images as input.

While the SDF graph does not consider actual implementations, its usage in ForSyDe IO's application modeling implicitly makes it insufficient.

5.2.1 Platform Representation

The ability to convert the abstract platform model into ForSyDe IO was, on the one hand, quite straightforward since the model was constructed with parameters already supported by ForSyDe IO in mind. On the other hand, adding support for the FPGA and some auxiliary communication semantics required fairly time-consuming development efforts—elaborated on in Section 5.3.

The platform model is a high-level representation of the MPSoC because several parameters that affect the platform performance are missing. For example, CPU caching, read and write latencies, and parallel reads and writes, among other factors, influence memory performance. Modeling these characteristics was unfeasible time-wise and requires separate investigations for each aspect to unravel which naturally applies to all components represented. However, it is reasonable to omit these parameters since they generally increase the performance and a model should ideally provide some worst-case guarantees. Therefore, the platform model is considered satisfactory although no guarantees are formally proven.

A small programming effort, which barely makes a difference, was to add modeling and analysis support for port connections. What is illustrated as thin lines in the grey interconnect switches in Figure 5.1, was not originally supported by the tools—only connections *between* components. Thus, by adding support for specifying which switch ports connect internally, the interconnect routing was successfully constrained for better representation.

The support provided for FPGAs and FPGA implementations in ForSyDe IO and IDeSyDe is considered satisfactory. An FPGA is modeled with parameters that set limits on available resources, while hardware implementations specify resource requirements and execution latencies which is a good model of reality. These specifications collectively contribute to the constraints outlined in Equations 3.1 and 3.2. In addition to satisfying

the constraints, a memory component representing BRAM was added to the platform specification for modeling easy-access data sharing within the FPGA. This was not as successfully modeled since memory mappings did not consider it private to the FPGA. There is a notable gap between the FPGA and its complex reality with many design considerations. However, tools like ModelSim [65], can simulate FPGA chips. Therefore, DSE solutions based on underlying IP blocks can be simulated in ModelSim to test the solution's feasibility, ensuring a minimal abstraction gap from reality.

While the modeling appears to be sufficient, IP blocks may also list a target frequency, DRAM usage, and required DSPs which are additional discrete resources that should ideally have corresponding constraints enforced during the DSE. In particular, DRAM was omitted from the modeling due to its dynamic behavior, as it requires the allocation of logic blocks. Consequently, during DSE, the available logic blocks and DRAM must be dynamically adjusted, introducing complexity. DSP resources were not part of the added constraints due to an oversight—although adding new constraints was straightforward after initial constraint support was implemented in the new explorer. One last simplification in the model representation concerns clock regions on the FPGA. Neither the limitation of a maximum number of clock regions nor clock region compatibility was enforced which makes FPGA implementation use any number of target frequencies. These limitations are again the responsibility of ModelSim.

5.3 Comments on Tools

Much time was spent on extending modeling and DSE support in ForSyDe IO and IDeSyDe. As these tools are primarily used in novel research, they lack extensive documentation unlike documentation typically found for commercial tools. Therefore, some extensions required assistance by PhD Rodolfo Jordão—the tool developer. His involvement included clarifying general code base concepts, giving ideas, reviewing extensions, and personally contributing to the extensions.

It can be concluded that the tools require varying levels of understanding for beginner usage. For extensions similar to this work, knowing both, and how they cooperate, is necessary.

- **ForSyDe IO**
  Extensions to ForSyDe IO are generally relatively simple and do not require expert knowledge of the tool's source code. They are highly stand-alone thanks to the multi-view philosophy behind the system specifications which effectively decouples tool extensions from already existing code. What primarily depicts the development pace instead are the underlying modeling decisions unrelated to coding—which parameters are needed to capture component characteristics. Once the user is fairly familiar with ForSyDe IO, adding support for a new modeling aspect and incorporating it into system specifications is quite simple. Integrating ForSyDe IO specifications with IDeSyDe introduces a slight difficulty that forces specifications to assume specific data structures to construct an arbitrarily sized design space efficiently. However, this integration is mostly a complex aspect caused by IDeSyDe and is not an issue with extensions themselves to ForSyDe IO.

- **IDeSyDe**
  The orchestrator architecture used in IDeSyDe makes the initial understanding of the tool "unnecessarily" hard to follow as opposed to a programming-language-specific tool. Basic knowledge of how identification rules work is enough to implement similar rules and how decision models should separate system characteristics. The bidding and exploration steps within IDeSyDe were implemented by PhD Rodolfo Jordão as they were considered too time-demanding. Hence, some implementation specifics for identification rules and decision models remain unknown because they are only used within the exploration step. Experiences regarding bidding and exploration remain unknown for the same reason. The time required to extend bidding and exploration would be equivalent to teaching the basics of these steps, making eliminating the dependency on his contributions unfeasible.

Further information about tool extensions guided by practical coding examples is found in Appendix A.

5.4 DSE results

Evaluation of test cases indicates that the desired modeling goals are achieved and show potential for future extensions. While there are some questionable memory mappings, the overall results of the exploration extensions are

promising. In this somewhat indecisive context, the memory mapping is fully inherited from previous builds of IDeSyDe, thus not a primary responsibility of the added extensions. Despite this rather strong claim, finding this limitation is important for knowing the status of IDeSyDe's capabilities. The experimental DSE clearly shows that the extensions work by:

- Making sure applications (actors) can specify either a software or hardware implementation, or both, for the DSE to decide the best overall mapping. Proved by the results from TC1.
- Application actors are correctly distinguished via application modeling whether they are more suitable for hardware- or software-programmable processing units. Proved by the results from TC2.
- Enforcing FPGA constraints to represent realistic limitations that hardware-programmable processing units present compared to CPU-based execution that is essentially unlimited. Proven by the results from TC3.
- Taking adequate decisions for *execution mapping* based on interconnection layout. Proved by the results from TC4 and TC5.
- As can be seen in the specification for the realistic application, adapting realistic application demands to ForSyDe IO modeling is reasonable by utilizing parameters found in libraries for FPGA implementations and sample bare-metal software implementations in C.

While the extensions seemingly work, some general aspects of the DSE in IDeSyDe are undesirable, resulting in an incorrectly interpreted platform model and sub-optimal DSE solutions. This includes:

- The novel DSE extensions are not scaling well for certain configurations and problem sizes. As mentioned in Section 4.2.6, the DSE could not produce results for the realistic application. Likely this is a consequence of the DSE approach which expands the SDF application into a large graph, thus resulting in an exponential increase in computational complexity. Additionally, the benchmarks presented in Section 4.2.7 imply that some exploration parameters may disrupt the exploration efficiency sometimes.
- Memory mappings acting highly non-deterministic. While execution mappings are working as expected, the DSE results do not show any

particular respect for the bandwidth presented by the interconnection network. This is specifically apparent in TC4, where the impairment of switches does not appear to affect the buffer mapping. The mapping to BRAM, shown in Figure 4.4, is reasonable but the mapping to TCM is unjustified since those memories are private to the RPU.

- Results for TC4 show that the throughput varies over exploration instances. It showcases that, despite *identical* input specifications, IDeSyDe sometimes misses more optimal solutions and thus does not always produce the most optimal Pareto front. One reason may involve how parameters from the specification are handled in the design space. If computational optimizations are applied, it may transform specifications differently throughout DSE runs.

- Actors mapped to hardware are still memory mapped on the platform as "empty code" with size zero. This is mostly a formality since it does not cause memory occupation (size zero) that would affect other mappings. Additionally, for instances where actors are implemented in software, their code storage cannot reasonably be placed in BRAM, as mentioned in the results for TC2.

- Investigation of the design space in IDeSyDe shows that platform interconnection paths use processing units for intermediate hops. This would mean in practice that processing units are executing code for moving data but no actual program is accounted for this is part of the design space. This is observed in the results for TC4, where TCM has been used for the purely FPGA-based solution since no path of only switches exists between the FPGA and TCM. It is unknown why IDeSyDe uses processing units as part of interconnection paths but it can be assumed there is a reason behind it.

- IDeSyDe assumes no additional communication costs inferred from physical links. Communication times are thus only affected by the worst-performing switch in an interconnection path, resulting in solutions where memory is incorrectly mapped further away from processing units measured in the number of hops. Some scenarios may favor a memory further away due to switch performances and link capabilities, but this does not apply to the general case.

- DSE solutions which solely distribute application execution to the FPGA may use external memory despite not requiring the extra capacity

that external memories provide. This is highlighted in TC2, where one subsequent run of the test case records to the inter-actor data channel to map to the TCM. As BRAM is the second-fastest memory for FPGAs, after embedded DRAM, this is an incorrect mapping. This is rooted in the previous bullet regarding how paths are evaluated and can be solved by link parameterization.

- DSE solutions which are placing the inter-actor data channel in BRAM do not account for a BRAM controller that needs to be implemented as a "helper" in the programmable logic and naturally requires logic blocks, not only BRAM. This is highlighted in TC1 where the inter-actor data channel is mapped to BRAM without further implications.

While the list of negative insights on the DSE results is longer than the positives, it is good to remember what was said in Section 2.1: results are at most as good as the design space constructed by input models. These negatives are the consequences of a platform model and design space at a selected abstraction level. If the requirements state no relevancy in providing accurate communication and memory modeling, the corresponding models can be simple and only account for a portion of the otherwise necessary aspects. Most bullets are also inherited from the build of IDeSyDe that was later extended, meaning that previous exploration requirements differ from the hardware-software DSE on the XCZU9EG MPSoC. The software-based DSE in IDeSyDe is generally good as it has provided good design solutions historically, making the usefulness a matter of the input problem [66].

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Creating a design space and having DSE support in IDeSyDe for the hardware- and software-programmable Zynq MPSoC platform is mostly successful. Test cases have demonstrated the novel support for hardware-programmable applications, paving the way for further extensions in the hardware domain. While there are quite a few weaknesses in the design space for the MPSoC platform, the work in this project serves as a fully operational experimental setup that has taken the first steps towards modeling MPSoC platforms. Flexible interaction with ForSyDe IO is established (Section 3.5), the abstract platform model is created and details relevant documentation (Section 3.2), and a practical guide for further extensions to ForSyDe IO and IDeSyDe is provided in Appendix A. Having these three components in place, additional work on arbitrarily extending the modeling in ForSyDe IO and the DSE in IDeSyDe is greatly simplified.

6.2 Limitations

Difficulties in deriving necessary parameters from documentation and data sheets for the platform components, along with time-consuming tool extensions, impacted the modeling quality. Also, finding a platform and application model that would provide mathematical guarantees for the DSE solutions is a crucial aspect of the modeling that was not handled which limits usefulness. Some weaknesses found with the platform representation when running the test cases imply significant gaps between the design space and reality. This is elaborated in Future Work—Section 6.3.

6.3 Future Work

Generally, it makes sense to broaden the modeling scope by adding more parameters and components for future work, such as the DSP constraint mentioned in Section 5.1. However, slightly more prominent future work topics are presented in the upcoming sections.

Application Modeling

Application modeling with SDF graphs is excellent in structuring constituent sub-tasks and captures interaction details between its actors for automated analysis. However, when merging software and hardware application structures, achieving a clear division of actors becomes hard due to their underlying implementation differences in parallelism. While latency for respective implementation can be nicely represented, throughput analysis will incorrectly estimate at least one implementation alternative. Solving this would require a new technique for specifying application functionality. Software and hardware representations of the same functionality must exist side-by-side without interfering. Not retaining SDF graphs is acceptable, but preserving similar analysis capabilities is of utmost importance to DSE. Finding an adaptation to SDF graphs could potentially involve changing the graphs to be two-layered, having embedded SDF graphs within current SDF actors. This would make the current SDF graphs centered around abstract functionality rather than task-based actors and the second layer would potentially contain two independent paths. A direct consequence of this layering is the enormous computational complexity that must be handled carefully. Another one is that this adaption is not particularly useful on its own, and requires investigation of how it can be integrated in the construction of design spaces. Another potential solution is to adopt an existing model of computation that abstracts away the communication model in favor of focusing on the functionality [67].

DSE Guarantees

The DSE solutions do not guarantee a worst-case throughput. For what is known, the DSE solutions may overestimate performance and thus produce false positives. Although this project's video streaming application is approximated and platform parameters are in the pessimistic direction, no combined mathematical formulation ensures a lower bound on throughput.

This is a critical aspect of modeling as no *truly* reliable information can be derived. There is a mathematical guarantee for throughput in IDeSyDe [68], but it does not cover user-defined parameters in this project. A good initial guarantee could be an overly pessimistic model, performing at most n times worse than reality. One such pessimistic assumption could involve not modeling memory cache, which then intentionally underestimates memory reads. Omitting parallel accesses to memory and network switches also contributes to a lower bound on throughput. Through a pessimistic baseline model, subsequent refinements based on mathematical proof can decrease the gap to reality and provide trustworthy DSE results.

Extending DSE Objectives and Constraints

Current optimization objectives in IDeSyDe include the number of used processing elements and the overall throughput of the application. The number of used processing elements is useful as it can reduce complexity by limiting the heterogeneity of implementations. Throughput is also useful to maximize, although overpowering the performance is never a desired factor. If the DSE solution can meet the requirement of processing images at 30Hz, achieving 40Hz would only waste resources apart from some performance margins. Thus the objective of maximizing throughput t could better be formulated as minimizing $t_{solution} - t_{target} \geq 0$, with t_{target} as a given constraint. Another optimization objective that would increase the solution precision is to model energy consumption and minimize it in situations where an application merely has to execute. A final objective to consider is the cost that defines how time-consuming implementations are relative to one another. As the number of optimization objectives increases, the Pareto front likely does too, making compromises more challenging for decision-makers. One way of reducing the Pareto front's complexity involves converting objectives to constraints when applicable. For example, the proposed throughput objective in this section could be converted into constraining solutions to satisfy $t_solution > t_target$. IDeSyDe also supports selecting which objectives to use for exploration, allowing flexibility based on the input problem.

Filling in the Gaps

Some project goals were not achieved and Section 5.4 presents a list of limitations with the DSE solutions for the XCZU9EG MPSoC and target applications. Goal G6, which focuses on finding a method to assess DSE

solutions to ensure the model's correctness, is related to the limitations. The limitations present modeling improvements as G6 would, which is the final artifact but the assessment should still be systematized and thorough. This could include using formulas to theoretically determine the ideal solution to a given problem in advance.

Overcoming the limitations would make the modeling more realistic. Hence, goals G5 (realistic application) and G7 (conclude design alternatives) would implicitly be solvable by corrections to the modeling. Most limitations are assumed to be similar in complexity to the work done in this work, or easier. One correction that stands out is to reduce the computational complexity in larger hardware-software DSE explorations. It would be suitable to perform performance profiling of IDeSyDe after identifying which parts of the system specifications yield a high increase in computational cost.

References

[1] E. C. Hall, "GENERAL DESIGN CHARACTERISTICS OF THE APOLLO GUIDANCE COMPUTER," Massachusetts Institute of Technology, Tech. Rep., May 1963. [Page 1.]

[2] P. Nagrath, J. Alzubi, B. Singla, J. Rodrigues, and A. Verma, *Smart Distributed Embedded Systems for Healthcare Applications*, 1st ed. CRC Press, 2023. [Online]. Available: https://www.perlego.com/book/3856589/smart-distributed-embedded-systems-for-healthcare-applications-pdf [Page 1.]

[3] V. Mazzia, A. Khaliq, F. Salvetti, and M. Chiaberge, "Real-Time Apple Detection System Using Embedded Systems With Hardware Accelerators: An Edge AI Application," *IEEE Access*, vol. 8, pp. 9102–9114, 2020. doi: 10.1109/ACCESS.2020.2964608 [Page 1.]

[4] M. Kathiresh and R. Neelaveni, Eds., *Automotive Embedded Systems: Key Technologies, Innovations, and Applications*, 1st ed., ser. EAI/Springer Innovations in Communication and Computing. Springer Cham, 2021. ISBN 978-3-030-59896-9 Hardcover ISBN: 978-3-030-59896-9, Softcover ISBN: 978-3-030-59899-0, eBook ISBN: 978-3-030-59897-6. [Online]. Available: https://doi.org/10.1007/978-3-030-59897-6 [Page 1.]

[5] R. Jordão, M. Becker, and I. Sander, "IDeSyDe: Systematic Design Space Exploration via Design Space Identification," *ACM Trans. Des. Autom. Electron. Syst.*, feb 2024. doi: 10.1145/3647640. [Online]. Available: https://doi.org/10.1145/3647640 [Pages 1, 3, and 43.]

[6] A. D. Pimentel, "Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration," PISCATAWAY, pp. 77–90, 2017. [Pages 2 and 9.]

[7] "ForSyDe - A Methodology for Formal System Design," accessed 2024-02-26. [Online]. Available: https://forsyde.github.io/ [Pages xi, 3, 17, and 26.]

[8] T. Bordis, T. Runge, A. Kittelmann, and I. Schaefer, "Correctness-by-Construction: An Overview of the CorC Ecosystem," *Ada Lett.*, vol. 42, no. 2, p. 75–78, apr 2023. doi: 10.1145/3591335.3591343. [Online]. Available: https://doi.org/10.1145/3591335.3591343 [Pages 3 and 16.]

[9] "About | ForSyDe IO," accessed 2024-02-26. [Online]. Available: https://forsyde.github.io/forsyde-io/about/ [Pages 3 and 18.]

[10] "IDeSyDe | A Framework for Modeling Heterogeneous Systems," accessed 2024-02-26. [Online]. Available: https://forsyde.github.io/IDeSyDe/ [Page 3.]

[11] M. Ali, P. Amini Rad, and D. Göhringer, "RISC-V Based MPSoC Design Exploration for FPGAs: Area, Power and Performance," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, F. Rincón, J. Barba, H. K. H. So, P. Diniz, and J. Caba, Eds. Cham: Springer International Publishing, 2020, pp. 193–207. [Pages 3 and 8.]

[12] R. Kedia, S. Goel, M. Balakrishnan, K. Paul, and R. Sen, "Design Space Exploration of FPGA-Based System With Multiple DNN Accelerators," *IEEE Embedded Systems Letters*, vol. 13, no. 3, pp. 114–117, 2021. doi: 10.1109/LES.2020.3017455 [Page 3.]

[13] V. Srinivasan, S. Radhakrishnan, and R. Vemuri, "Hardware software partitioning with integrated hardware design space exploration," in *Proceedings Design, Automation and Test in Europe*, 1998. doi: 10.1109/DATE.1998.655833 pp. 28–35. [Pages 3 and 11.]

[14] T. Taghavi, A. D. Pimentel, and M. Sabeghi, "VMODEX: A novel visualization tool for rapid analysis of heuristic-based multi-objective design space exploration of heterogeneous MPSoC architectures," *Simulation modelling practice and theory*, vol. 22, no. March, pp. 166–196, 2012. [Page 7.]

[15] K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1523–1543, 2000. doi: 10.1109/43.898830 [Pages 8 and 16.]

[16] M. de Prado, A. Mundy, R. Saeed, M. Denna, N. Pazos, and L. Benini, "Automated Design Space Exploration for Optimized Deployment of DNN on Arm Cortex-A CPUs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 11, pp. 2293–2305, 2021. doi: 10.1109/TCAD.2020.3046568 [Page 9.]

[17] "Pareto Order Dominated - Pareto Efficiency - Wikipedia," accessed 2024-04-15. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Pareto_efficiency&oldid=1234567890 [Pages xi and 10.]

[18] "FPGA - Configurable Logic Block - Digilent Blog," accessed 2024-03-20. [Online]. Available: https://digilent.com/blog/fpga-configurable-logic-block/ [Page 9.]

[19] "What is FPGA? - Arm," accessed 2024-03-20. [Online]. Available: https://www.arm.com/glossary/fpga [Page 10.]

[20] "VHDL and FPGA terminology - Block RAM," accessed 2024-04-08. [Online]. Available: https://vhdlwhiz.com/terminology/block-ram/ [Page 10.]

[21] *UltraScale Architecture Memory Resources - User Guide*, AMD Xilinx, 02 2018, rev. 1.9. [Pages 10 and 27.]

[22] *UltraScale Architecture Configurable Logic Block - User Guide*, AMD Xilinx, 02 2017, rev. 1.9. [Pages 11 and 27.]

[23] "VHDL and FPGA terminology - Clock domain crossing," accessed 2024-04-08. [Online]. Available: https://vhdlwhiz.com/terminology/clock-domain-crossing/ [Page 11.]

[24] "Virtex-5 FPGA Data Sheet - Viewer - AMD Technical Information Portal," accessed 2024-05-04. [Online]. Available: https://docs.amd.com/v/u/en-US/ds202 [Page 11.]

[25] K. Zhu and D. Wong, "Clock skew minimization during FPGA placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 4, pp. 376–385, 1997. doi: 10.1109/43.602474 [Page 11.]

[26] B. C. Schafer and T. Kim, "Hotspots Elimination and Temperature Flattening in VLSI Circuits," *IEEE Transactions on Very Large Scale*

Integration (VLSI) Systems, vol. 16, no. 11, pp. 1475–1487, 2008. doi: 10.1109/TVLSI.2008.2001140 [Page 12.]

[27] I. del Campo, R. Finker, J. Echanobe, and K. Basterretxea, "Controlled Accuracy Approximation of Sigmoid Function for Efficient FPGA-Based Implementation of Artificial Neurons," *Electronics Letters*, vol. 49, no. 25, pp. 1598–1600, 2013. doi: 10.1049/el.2013.3098 [Page 12.]

[28] G. Dinelli, G. Meoni, E. Rapuano, and L. Fanucci, "Advantages and Limitations of Fully on-Chip CNN FPGA-Based Hardware Accelerator," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020. doi: 10.1109/ISCAS45731.2020.9180867 pp. 1–5. [Page 12.]

[29] Y. Li, Y. Du, X. Ye, and Z. Cai, "Doppler radar real-time signal processing based on FPGA," in *2016 IEEE International Conference on Signal and Image Processing (ICSIP)*, 2016. doi: 10.1109/SIPROCESS.2016.7888301 pp. 443–446. [Page 12.]

[30] J. Wang and K. Liu, "High-frequency Active Sonar Real-time Signal Processing System Based on FPGA," in *2018 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*, 2018. doi: 10.1109/ICSPCC.2018.8567799 pp. 1–4. [Page 12.]

[31] *Zynq UltraScale+ MPSoC Data Sheet: Overview*, AMD Xilinx, 11 2022, rev. 1.10. [Pages 12, 13, and 27.]

[32] *ZCU102 Evaluation Board - User Guide*, AMD Xilinx, 02 2023, rev. 1.7. [Pages 13 and 27.]

[33] *Zynq UltraScale+ Device - Technical Reference Manual*, AMD Xilinx, 12 2023, rev. 2.4. [Pages xiii, 13, 27, and 56.]

[34] "DDR4 SDRAM - Wikipedia," accessed 2024-05-03. [Online]. Available: https://en.wikipedia.org/wiki/DDR4_SDRAM [Page 13.]

[35] "Constraining the Core - Reader - AMD Technical Information Portal," accessed 2024-05-03. [Online]. Available: https://docs.amd.com/r/en-US/pg067-axi-chip2chip/Constraining-the-Core [Pages 13 and 27.]

[36] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Commun. ACM*, vol. 19, no. 3, p. 137, mar

1976. doi: 10.1145/360018.360025. [Online]. Available: https://doi-org.focus.lib.kth.se/10.1145/360018.360025 [Page 14.]

[37] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, 1987. doi: 10.1109/TC.1987.5009446 [Page 14.]

[38] "Online Video Size Calculator By Format and Device," accessed 2024-05-08. [Online]. Available: https://www.videoproc.com/edit-4k-video/video-size-calculator.html [Page 15.]

[39] I. Sander and A. Jantsch, *Transformation based communication and clock domain refinement for system design*. New York, NY, USA: Association for Computing Machinery, 2002, p. 281–286. ISBN 1581134614. [Online]. Available: https://doi-org.focus.lib.kth.se/10.1145/513918.513992 [Pages 16 and 17.]

[40] R. Jordão, F. Bahrami, R. Chen, and I. Sander, "A multi-view and programming language agnostic framework for model-driven engineering," in *2022 Forum on Specification & Design Languages (FDL)*, 2022. doi: 10.1109/FDL56239.2022.9925666 pp. 1–8. [Pages 17 and 18.]

[41] C. Wolff, L. Krawczyk, R. Höttger, C. Brink, U. Lauschner, D. Fruhner, E. Kamsties, and B. Igel, "AMALTHEA — Tailoring tools to projects in automotive software development," in *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 2, 2015. doi: 10.1109/IDAACS.2015.7341359 pp. 515–520. [Page 18.]

[42] "KGraph Meta Model - XWiki," accessed 2024-03-24. [Online]. Available: https://wiki.rtsys.informatik.uni-kiel.de/bin/view/KIELER/Discontinued%20Projects/Infrastructure%20for%20Meta%20Layout%20%28KIML%29/KGraph%20Meta%20Model/ [Page 20.]

[43] "KIELER VS Code - Visual Studio Marketplace," accessed 2024-03-23. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=kieler.keith-vscode [Page 20.]

[44] A. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels,"

IEEE Transactions on Computers, vol. 55, no. 2, pp. 99–112, 2006. doi: 10.1109/TC.2006.16 [Page 21.]

[45] R. Jordão, M. Becker, and I. Sander, "IDeSyDe: Systematic Design Space Exploration via Design Space Identification," *ACM Trans. Des. Autom. Electron. Syst.*, feb 2024. doi: 10.1145/3647640 Just Accepted. [Online]. Available: https://doi.org/10.1145/3647640 [Pages xi, 21, 22, and 23.]

[46] "IDeSyDe - Usage | IDeSyDe," accessed 2024-03-18. [Online]. Available: https://forsyde.github.io/IDeSyDe/usage [Page 23.]

[47] "ForSyDe-Deep | A Deep-Embedded Synthesizer for ForSyDe Models," accessed 2024-03-13. [Online]. Available: https://forsyde.github.io/forsyde-deep/ [Page 26.]

[48] R. Maculet and J.-F. PERROT, "Archipel : Intelligence artificielle et conception assistee par ordinateur en architecture. representation des connaissances spatiales,algebre de manhattan, et raisonnement spatial avec contraintes," Ph.D. dissertation, Pierre and Marie Curie University, 1991, thèse de doctorat dirigée par Gondran, Michel Sciences appliquées Paris 6 1991. [Online]. Available: http://www.theses.fr/1991PA066564 [Page 29.]

[49] K. Javeed and X. Wang, "Low latency flexible FPGA implementation of point multiplication on elliptic curves over GF(p): FPGA, ELLIPTIC CURVE CRYPTOGRAPHY (ECC), MODULAR MULTIPLIER," *International Journal of Circuit Theory and Applications*, vol. 45, 12 2016. doi: 10.1002/cta.2295 [Page 30.]

[50] "Bit Depth Conversion - Vitis Libraries - Reader - AMD Technical Information Portal," accessed 2024-05-15. [Online]. Available: https://docs.amd.com/r/en-US/Vitis_Libraries/vision/api-reference.html_2_11 [Page 30.]

[51] "AR1335 based 13MP 4K camera module," accessed 2024-05-09. [Online]. Available: https://www.e-consystems.com/4k-camera-module.asp [Page 35.]

[52] *logiJPGD-LS Lossless MJPEG Decoder*, logiBRICKS by Xylon, 02 2022, v1.03. [Page 35.]

[53] “Grayscale - Wikipedia,” accessed 2024-05-09. [Online]. Available: https://en.wikipedia.org/wiki/Grayscale [Page 35.]

[54] “RGB to GRAY - Vitis Libraries - Reader - AMD Technical Information Portal,” accessed 2024-05-09. [Online]. Available: https://docs.amd.com/r/en-US/Vitis_Libraries/vision/api-reference.html_2_23_20 [Page 35.]

[55] “CUDA Programming: Sobel Filtering implementation in C,” accessed 2024-05-09. [Online]. Available: http://cuda-programming.blogspot.com/2013/01/sobel-filter-implementation-in-c.html [Page 36.]

[56] “Sobel Filter - Vitis Libraries - Reader - AMD Technical Information Portal,” accessed 2024-05-09. [Online]. Available: https://docs.amd.com/r/en-US/Vitis_Libraries/vision/api-reference.html_2_93 [Page 36.]

[57] Machura, Michal and Danilowicz, Michal and Kryjak, Tomasz, “Embedded object detection with custom littlenet, finn and vitis ai dcnn accelerators,” *Journal of Low Power Electronics and Applications*, vol. 12, no. 2, 2022. doi: 10.3390/jlpea12020030. [Online]. Available: https://www.mdpi.com/2079-9268/12/2/30 [Page 36.]

[58] C. Patel, D. Bhatt, U. Sharma, K. Patel, R. Patel, A. Patel, U. Bhatt, S. Pandya, K. Modi, N. Cholli, H. Ghayvat, M. Zuhair, S. A. Shah, S. Majumdar, and M. Khan, “DBGC: Dimension Based Generic Convolution Block for Object Recognition,” *Sensors*, vol. 22, 02 2022. doi: 10.3390/s22051780 [Page 36.]

[59] *DPUCZDX8G for Zynq UltraScale+ MPSoCs Product Guide (PG338)*, AMD Xilinx, 01 2023, version 4.1. [Page 36.]

[60] “Gradle Build Tool,” accessed 2024-03-26. [Online]. Available: https://gradle.org/ [Page 37.]

[61] “MiniZinc,” accessed 2024-06-16. [Online]. Available: https://www.minizinc.org/ [Page 44.]

[62] “Intel Core i9-13900H Processor,” accessed 2024-06-16. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/232135/intel-core-i913900h-processor-24m-cache-up-to-5-40-ghz/specifications.html [Page 44.]

[63] “Cortex-A7 instruction cycle timings - Hardwarebug,” accessed 2024-05-25. [Online]. Available: https://hardwarebug.org/2014/05/15/cortex-a7-instruction-cycle-timings/ [Page 57.]

[64] “Optimizing FPGA VIs Using Pipelining - NI,” accessed 2024-05-15. [Online]. Available: https://www.ni.com/docs/en-US/bundle/labview-fpga-module/page/optimizing-fpga-vis-using-pipelining-fpga-module.html [Pages xiii and 58.]

[65] “ModelSim HDL simulator | Siemens Software,” accessed 2024-05-15. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/modelsim/ [Page 60.]

[66] R. Jordão, M. Becker, I. Sander, and I. Söderquist, “Design space exploration for safe and optimal mapping of avionics functionality on IMA platforms,” in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, 2023. doi: 10.1109/DASC58513.2023.10311316 pp. 1–9. [Page 64.]

[67] I. Sander and A. Jantsch, “Formal system design based on the synchrony hypothesis, functional models, and skeletons,” in *Proceedings Twelfth International Conference on VLSI Design. (Cat. No.PR00013)*, 1999. doi: 10.1109/ICVD.1999.745170 pp. 318–323. [Page 66.]

[68] R. Jordão, I. Sander, and M. Becker, “Formulation of Design Space Exploration Problems by Composable Design Space Identification,” in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021. doi: 10.23919/DATE51398.2021.9474082 pp. 1204–1207. [Page 67.]

Appendix A

Extensions in Practice

The extensions for ForSyDe IO and IDeSyDe span multiple files throughout their repositories. Since the ForSyDe IO modeling is theoretically independent of IDeSyDe it will be presented separately. The presentation of IDeSyDe extensions will then cover how the ForSyDe IO modeling was incorporated into the design space. This appendix section intends to present a meaningful guide on how to reason when extending the tools.

A.1 ForSyDe IO

The modeling from Section 3.3.1 required one trait for the FPGA, `LogicProgrammableModule`, and one trait for specifying an application's hardware implementation, `InstrumentedHardwareBehavior`. New files represented these traits in the repository's `java-libforsyde-hierarchy/src/main/java/forsyde/io/lib/hierarchy` subfolder—the common folder for all traits.

A.1.1 LogicProgrammableModule

The trait's source code is located here.

Since this trait represents a hardware component, it is reasonably placed under `.../hierarchy/platform/hardware`. The file structure is best copied from an existing trait in the same folder but carefully select the class it derives properties from through Java subclassing. This trait uses DigitalModule to retrieve the operating frequency parameter. However, since

the parameter is unused in the design space and does not represent clock regions, subclassing could have been done directly from HardwareModule.

Resource limits are represented by normal integer declarations, such as `int availableLogicArea();` (logic blocks) annotated with `@Property`. During the build process of ForSyDe IO, this annotation yields a getter and setter for the property to use in system specification. This also applies to the BRAM.

A.1.2 InstrumentedHardwareBehavior

The trait's source code is located here.

This trait is used for specifying how a system entity is implemented and behaves, it is placed under `.../hierarchy/behavior/implementation/functional` alongside the existing `InstrumentedSoftwareBehavior` trait. The file structure for this trait is also best copied from an existing, specifically the corresponding software trait.

The resource requirements are represented by the Java `Map<String, Map<String, Long>>` to support independent requirements for multiple FPGAs by the top-level keys. Additionally, the execution latency is specified through two properties, a numerator and a denominator, instead of a single property. The latency is captured as a rational number as hardware components commonly describe their latencies in multiples of clock frequencies, which are exact rational numbers; otherwise, precision is needlessly lost if latencies are captured with floating point properties. These properties are annotated with `@Property`. Lastly, for convenience purposes, getting specifically the hardware implementation area (number of logic blocks) from the resource `Map` is defined as a default function as part of the interface: `default long requiredFPGAHardwareImplementationArea() {...}`.

A.1.3 CommunicationModulePortSpecification

The trait's source code is located here.

This trait's modeling idea is captured best as an extension to the `GenericCommunicationModule` trait as it further defines the communication module's parameters—similar to how the `InstrumentedCommunicationModule` defines how fast data is handled in the switch. The new trait

uses the same structure as the instrumentation trait and is also found under `.../hierarchy/platform/hardware`.

The trait declares `Map<String, List<String» portConnections();` annotated with `@Property`. The `Map` can in a simple way define how port `a` (`String`) connects to ports `[b, c]` (`List<String>`). If there are no restrictions on port connections the `@Property` annotation creates an empty `Map`, although the trait is obsolete then and should not be used in the first place.

A.1.4 Saving Changes

To use the changes made to the ForSyDe library, they must be pushed to the GitHub repository. The necessary code generation based on annotations is done automatically on the remote. However, ForSyDe IO should be built locally before that to ensure no broken code on the remote. Building is done by running `gradle build` in the terminal when standing in the repository's root directory. Further inspecting if the build was successful is done on the GitHub Actions page for the repository.

A.1.5 Using Changes

Changes are available to an interfacing project, such as the one described in 3.5 once the GitHub Actions have been completed. Adding ForSyDe IO as a project dependency can be found in the `build.gradle` file (link) that was used for this project's experimental setup. The ForSyDe IO library can be referenced in various ways depending on branches or specific commits, see the JitPack documentation.

A.2 IDeSyDe

Updating IDeSyDe to interpret the three above-listed ForSyDe IO traits required many new identification rules and adaptations to the exploration process. This documentation is limited to the identification step as the other steps are mostly black boxes to this thesis project's practical work. Three identification rules are included to give an overview of the development process. Since system specifications were modeled in ForSyDe IO, the `java-bridge-forsyde-io` sub-project in IDeSyDe had to be updated to identify the new parameters. Furthermore, `rust-common/src/irules.rs` was updated to know the Java-based decision

models. Similarly, new decision models were added to `java-common` and `rust-common/src/models.rs`.

A.2.1 MM_MCoreAndPL_IRule

The identification rule is located here and the decision model here.

This is the first core addition to the decision models in IDeSyDe as it captures the platform with both hardware- and software-programmable processing units. It has copied the same identification steps as the platform without programmable logic. The core part of this extension is to use `<component>.tryView(...)` to try to identify the new ForSyDe IO trait named `LogicProgrammableModule` among the design models vertices. Before creating the `MM_MCoreAndPL` decision model, it has to be ensured that at least one `LogicProgrammableModule` has been found; otherwise, the decision model is not produced which is also the case if no such component has declared the available logic blocks. These declarations are accessible via getters, e.g. `LogicProgrammableModule.availableLogicArea()`.

A.2.2 HardwareImplementationAreasIRule

The identification rule is located here and the decision model here.

The second core addition to decision models is to identify an application's hardware implementation demand. This scenario required adding a decision model and identification rule from scratch. It could however use identical structure and the common way of `<component>.tryView(...)` for extraction of parameters and traits that must exist.

A.2.3 PartitionedMemoryMappableMulticoreAndPL

The identification rule is located here and the decision model here.

This decision model does not identify anything from the input design models but combines already identified decision models into a larger one. This requires finding necessary decision models in the current set of identified decision models. This specified decision model consists of `RuntimesAndProcessors` and `MM_MCoreAndPL`, where `MM_MCoreAndPL` is

an extension to the decision model without programmable logic that is implemented in Rust. Therefore, the `MM_MCoreAndPL` decision model, identified and created in Java, must still be defined in Rust to enable the decision model merge. This was therefore added by declaring the same properties of `MM_MCoreAndPL` in a Rust struct.

A.2.4 Using Changes

The easiest option to use changes made to IDeSyDe is to compile the tool locally. This is done by the build script, available for Linux among other OSes. After successful compilation, the `idesyde` executable is created and can be used with the new changes in place.

TRITA – EECS-EX 2024:0000
Stockholm, Sweden 2024

www.kth.se

€€€€ For DIVA €€€€

{
"Author1": { "Last name": "Larsson",
"First name": "Ludvig",
"Local User Id": "u1l7bgbh",
"E-mail": "lular@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
}
},
"Cycle": "2",
"Course code": "DA248X",
"Credits": "30.0",
"Degree1": {"Educational program": "Master's Programme, Embedded Systems, 120 credits"
,"programcode": "TEBSM"
,"Degree": "Master's degree"
,"subjectArea": "Technology"
},
"Title": {
"Main title": "Automated Design Space Exploration for Hardware and Software Implementations on Heterogeneous MPSoCs",
"Language": "eng" },
"Alternative title": {
"Main title": "Automatiserad Designrymdsutforskning för Hårdvaru- och Mjukvarumimplementationer på Heterogena Multiprocessorsystemschip",
"Language": "swe"
},
"Supervisor1": { "Last name": "Bahrami",
"First name": "Fahimeh",
"Local User Id": "u1bsi6is",
"E-mail": "fahimehb@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "DIVISION OF ELECTRONICS AND EMBEDDED SYSTEMS" }
},
"Supervisor2": { "Last name": "Karlsson",
"First name": "Ludwig",
"E-mail": "ludwig.karlsson2@saabgroup.com",
"Other organisation": "Saab AB, Stockholm"
},
"Examiner1": { "Last name": "Sander",
"First name": "Ingo",
"Local User Id": "u1l8osh2",
"E-mail": "ingo@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "DIVISION OF ELECTRONICS AND EMBEDDED SYSTEMS" }
},
"Cooperation": { "Partner_name": "Saab AB"},
"National Subject Categories": "10201, 20207",
"Other information": {"Year": "2024", "Number of pages": "1,83"},
"Copyrightleft": "copyright",
"Series": { "Title of series": "TRITA – EECS-EX" , "No. in series": "2024:0000" },
"Opponents": { "Name": "William Asp"},
"Presentation": { "Date": "2024-06-05 10:00"
,"Language":"eng"
,"Room": "via Zoom https://kth-se.zoom.us/j/900310997"
,"Address": "Kistagången 16"
,"City": "Stockholm" },
"Number of lang instances": "2",
"Abstract[eng]": €€€€

Identifying embedded system design solutions that utilize platform resources effectively is often difficult. Due to the demands of modern applications and the complexity of heterogeneous platforms, tailoring the system design to maximize throughput, minimize power consumption, or meet other design goals may pose a highly demanding task for system designers. Design Space Exploration (DSE) is a technique commonly applied within the embedded system design process that utilizes a systematic search to find optimal design solutions. These solutions include where applications should be executed, how applications should be scheduled, and where application data should be stored. The design space is the core of DSE, defining the possible design alternatives based on platforms and application specifications.

This thesis explores how DSE can evaluate design spaces where applications are mapped to hardware- or software-programmable processing units. In this work, *IDeSyDe* is the targeted DSE tool, that originally only supported DSE of software-programmable platforms, and *ForSyDe IO* is used for creating the system specifications. These tools were adapted to support hardware-programmable platforms and hardware implementations, wherein analysis was necessary to decide relevant parameters. A quality assessment of these novel DSE capabilities was performed on a model of the heterogeneous Zynq UltraScale+ XCZU9EG Multi-processor System on Chip through test cases and a realistic video streaming application.

An FPGA is a hardware-programmable processing unit providing hardware resources to implement functionality. Extensions to the tools mainly focus on providing support for specifying FPGA components and applications implemented on an FPGA. The achieved DSE support covers FPGA implementations through the specification of execution latency, required block RAM, and required logic blocks, relevant for FPGA modeling due to its characterizing resource limitations. These parameters are suitable as they are commonly found in libraries for reusable FPGA implementations. Likewise, the FPGA component is modeled with the availability of block RAM and logic blocks.

The project's goals were met and can thus be considered successful. Assessment of the test cases shows that the novel extensions are well incorporated into *IDeSyDe*. However, *IDeSyDe* did not find a solution for the realistic application within a reasonable time due to high computational demands, and the modeling did not adequately account for certain real-world aspects of the platform. Also, although the platform model assumed rather pessimistic performance, the lack of performance guarantees between modeling and reality results poses a significant limitation and is left for

future work. €€€€,
"Keywords[eng]": €€€€
Design Space Exploration, Embedded System Design, System Modeling, Multi-Processor System on Chip, Field Programmable Gate Array
€€€€,
"Abstract[swe]": €€€€
På senare tid har designprocessen för inbyggda system ökat markant i komplexitet vilket leder svårigheter i att hitta resurseffektiva och optimala lösningar. Detta är en direkt följd av de krav som ställs av moderna tillämpningar som vill verka i teknikens framkant och att inbyggda plattformar besitter alltmer heterogena resurser för ökad beräkningskapacitet. Designprocessen för resurs- och prestandakrävande applikationer kan hanteras med hjälp av resignrymdsutforskning genom skapandet av en designrymd utifrån systemparametrar för applikationer och den underliggande plattformen. Denna designrymd kan därefter systematiskt genomsökas för att hitta de bästa designlösningarna givet optimeringmål och krav på systemet. Dessa designlösningar kan beskriva hur applikationsexekvering ska distribueras över plattformens beräkningsenheter och hur applikationer ska schemaläggas samt vilka fysiska minnen på plattformen som ska användas för applikationsdata.
Detta examensarbete syftar till att möjliggöra designrymdsutforsking med en designrymd som utgörs av applikationer som kan exekvera på en plattform med både mjukvaru- och hårdvaruprogrammerbara beräkningsenheter. För att åstadkomma detta tillämpades designrymdsutforskingverktyget *IDeSyDe* som behövde utökas från att bara ta hänsyn till implementationer på mjukvaruprogrammerbara beräkningsenheter. Detta krävde analys av hur hårdvarubaserade beräkningsenheter och dess applikationsimplementationer bäst kan modelleras genom parametrisering, vilket utgjorde grunden för nödvändiga utökningar till verktyget *ForSyDe IO* som används för generell systemmodellering. En kvalitetsutvärdering av de utökade modelleringsmöjligheterna utfördes på en modell av det heterogena multiprocessorsystemschippet Zynq UltraScale+ XCZU9EG med testapplikationer och en verklighetsanpassad videoapplikation.
Den huvudsakliga utvecklingen av designverktygen tillägnades på-plats-programmerbara grindmatriser (eng. FPGA)—en hårdvaruprogrammerbar beräkningsenhet, och tillhörande applikationsimplementationer för grindmatriser. En grindmatris karaktäriseras främst av dess begränsade resurser som avgör mängden funktionalitet som får plats. Utökningarna tillåter specificering av grindmatrisens resursbegränsningar för logiska block och block-RAM samt resursbehov för hårdvaruimplementationer och tillhörande exekveringstid i hårdvara. Behjälpt av att det finns bibliotek som tillhandahåller återanvändbara gridmatrisimplementationer som specificerar dessa parametrar anses modelleringen vara välanpassad. Projektets huvudmål uppfylldes och kan därmed anses vara lyckat. Trots att den skapade plattformsmodellen har pessimistisk prestanda är avsaknaden av garantier på hur designlösningar förhåller sig till verkligheten en kritisk aspekt. Utvärderingen av specifikt den utökade designrymdsutforskningen visade på goda resultat. Däremot kunde inte *IDeSyDe* hantera videoapplikationen då beräkningskomplexiteten blev för stor och i allmänhet visade designlösningarna på vissa svagheter i att respektera realistiska aspekter gällande plattformen.
€€€€,
"Keywords[swe]": €€€€
Designrymdsutforskning, Design av Inbyggda System, Systemmodellering, Multiprocessorsystemschip, På-plats-programmerbar Grindmatris
€€€€,
}

acronyms.tex

```
%%% Local Variables:
%%% mode: latex
%%% TeX-master: t
%%% End:
% The following command is used with glossaries-extra
\setabbreviationstyle[acronym]{long-short}
% The form of the entries in this file is \newacronym{label}{acronym}{phrase}
%                                     or \newacronym[options]{label}{acronym}{phrase
    }
% see "User Manual for glossaries.sty" for the  details about the options, one
    example is shown below
% note the specification of the long form plural in the line below
% \newacronym[longplural={Debugging Information Entities}]{DIE}{DIE}{Debugging
    Information Entity}
% %

% % note the use of a non-breaking dash in long text for the following acronym
% \newacronym{IQL}{IQL}{Independent Q^^e2^^80^^91Learning}

% \newacronym{KTH}{KTH}{KTH Royal Institute of Technology}

% \newacronym{LAN}{LAN}{Local Area Network}
% \newacronym{VM}{VM}{virtual machine}
% % note the use of a non-breaking dash in the following acronym
% \newacronym{WiFi}{Wi^^e2^^80^^91Fi}{Wireless Fidelity}

% \newacronym{WLAN}{WLAN}{Wireless Local Area Network}
% \newacronym{UN}{UN}{United Nations}
% \newacronym{SDG}{SDG}{Sustainable Development Goal}

\newacronym{AGC}{AGC}{Apollo Guidance Computer}

\newacronym{ForSyDe}{ForSyDe}{Formal System Design}
\newacronym{DSI}{DSI}{Design Space Identification}
\newacronym{DSE}{DSE}{Design Space Exploration}
\newacronym{CP}{CP}{Constraint Programming}
\newacronym{MILP}{MILP}{Mixed-Integer Linear Programming}
\newacronym{DFG}{DFG}{Data-flow Graph}
\newacronym{SDF}{SDF}{Synchronous Data Flow}

\newacronym{LPD}{LPD}{Low Power Domain}
\newacronym{FPD}{FPD}{Full Power Domain}
\newacronym[shortplural={MPSoCs}, firstplural={Multi-processor System on Chips (
    MPSoCs)}]{MPSoC}{MPSoC}{Multi-processor System on Chip}
\newacronym{FPGA}{FPGA}{Field Programmable Gate Array}
\newacronym[shortplural={DPUs}, firstplural={Deep Learning Processor Units (DPUs)}]{
    DPU}{DPU}{Deep Learning Processor Unit}
\newacronym[shortplural={DSPs}, firstplural={Digital Signal Processors}]{DSP}{DSP}{
    Digital Signal Processor}
\newacronym{SoC}{SoC}{System on Chip}
\newacronym{DMA}{DMA}{Direct Memory Access}
\newacronym{APU}{APU}{Application Programming Unit}
\newacronym[shortplural={ASICs}, firstplural={Application Specific Integrated
    Circuits (ASICs)}]{ASIC}{ASIC}{Application Specific Integrated Circuit}
\newacronym{RPU}{RPU}{Real-time Programming Unit}
\newacronym{GPU}{GPU}{Graphics Processing Unit}
\newacronym{RAM}{RAM}{Random Access Memory}
\newacronym{BRAM}{BRAM}{Block Random Access Memory}
\newacronym{DRAM}{DRAM}{Distributed Random Access Memory}
\newacronym{FIFO}{FIFO}{First-in First-out}
\newacronym{OCM}{OCM}{On-chip Memory}
\newacronym{TCM}{TCM}{Tightly Coupled Memory}

\newacronym[shortplural={NNs}, firstplural={Neural Networks}]{NN}{NN}{Neural Network}
\newacronym{MJPEG}{MJPEG}{Motion JPEG}
\newacronym{CNN}{CNN}{Convolutional Neural Network}
\newacronym{IP}{IP}{Intellectual Property}
\newacronym[shortplural={FLOPs}, firstplural={Floating Point Operations (FLOPs)}]{
    FLOP}{FLOP}{Floating Point Operation}
\newacronym{API}{API}{Application Programming Interface}
```