Platform-based design for high-performance mechatronic systems

Platform-based Design for High-Performance Mechatronic Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op woensdag 29 april 2015 om 16.00 uur

door

Raymond Mathias Wilhelmus Frijns

geboren te Tilburg

Dit proefschrift is goedgekeurd door de promotor en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr.ir A.C.P.M. Backx
promotor:	prof.dr. H. Corporaal
copromotoren:	dr.ir. J.P.M. Voeten
	dr.ir. S. Stuijk
leden:	prof.dr.ir. M.J.G. Bekooij (UT)
	prof.dr. B.H.H. Juurlink (TU Berlin)
	prof.dr.ir. W.P.M.H. Heemels
adviseur:	dr.ir. R.R.H. Schiffelers (ASML)

Platform-based Design for High-Performance Mechatronic Systems

Cover design: StudioLIN - www.studiolin.nl

Printing: Gildeprint Drukkerijen - www.gildeprint.nl

A catalogue record is available from the Eindhoven University of Technology Library. ISBN: 978-90-386-3826-3

Abstract

Platform-based Design for High-Performance Mechatronic Systems

The digital control domain has traditionally relied on the flexibility and scalable performance of general purpose processors (GPPs) to meet application performance demands. For decades, the rapid advances in processor frequency scaling could easily match the increasing scale and complexity of high-performance digital control applications. Since several years however, this steady performance increase has stagnated, up to the point that application demands are rapidly catching up and control application designers even have to do concessions to yield feasible timing constraints, both in terms of latency as well as throughput. In order to meet the timing constraints of next generation control applications, it is paramount that application and platform form a close union that exploits the available parallelism effectively and at the right granularity. To this end, platforms are required that closely match the structure and heterogeneity of the application, as well as design methodologies that facilitate this close tailoring by cohesive HW/SW design. This thesis presents a novel heterogeneous multiprocessor platform, and a platform-based design methodology that facilitates the design of tailored heterogeneous execution platforms targeting digital control applications. The thesis discusses the trajectory from the typical application domain model to implementation on a platform instance, including analysis and synthesis.

Control engineers typically design digital control applications by modeling them in the z-domain. These untimed declarative models are not suitable for implementation, since they cannot be used to verify timing constraints. Therefore, we propose to transform such models into dataflow models, which can be used for both analysis and synthesis. This thesis shows how to go from a z-domain model to a timed operational dataflow model with a set of timing constraints. The design flow facilitates application mapping onto instances of a novel FPGA-based architecture platform presented in this thesis, which consists of several interconnected Application-Specific Instruction-set Processors (ASIPs). All ASIP types in the template have the same basic architectural features to match the structure of typical DSP functions, but the datapath of each specific ASIP type is tuned towards a specific family of DSP functions. The ASIPs synchronize on input

data-availability according to the (Homogeneous) Synchronous Dataflow semantics. This enables easy application distribution over the platform and provides a solid basis for timing analysis and synthesis.

Application, mapping and platform instance models can be used for both synthesis and analysis. The synthesis trajectory generates VHDL code, platform configuration data and code for the different ASIPs. The analysis trajectory extracts a timed (acyclic) task graph model from the specifications, which is used as input for dataflow analysis. We present a novel algorithm that calculates the worst-case execution timing of tasks in an acyclic task graph given a First-Come-First-Served scheduling policy on each resource. The analysis takes the best-case and worst-case execution timing into account to iteratively produce conservative approximations of the contention experienced by each task. The outcome is guaranteed to be conservative for any possible controller execution, and thus provides bounds on e.g. the best-case and worst-case controller latency.

The applicability of this work is demonstrated in a realistic case-study, in which two complex motion controllers of a lithography machine are mapped onto our platform template. One of those controllers is a traditional PID controller, while the other is a next generation state-space controller. The controllers are interdependent and both have to meet the same latency and throughput constraints. We show how to apply our analysis and synthesis flow to this case, and how to methodically find a suitable platform instance that is able to meet all timing constraints. The derived platform instance and mapping result in a solution that, contrary to an 8-core GPP implementation, is able to meet all timing constraints. This thesis shows that with the presented design methodology and platform template, it is possible to quickly instantiate and generate efficient platforms, and find application mappings that are able to meet the timing constraints of next generation control applications.

Contents

Chapter 1

Introduction

The digital revolution has drastically changed the way we experience our everyday lives. It has made the world a smaller place by connecting billions of people all over the world, accelerating the global exchange of ideas, information and goods at an unprecedented scale, and allowing one to speed up the pace of life by automating many of our everyday tasks. Computing devices were mainly large room-sized mainframe systems or very expensive enthusiast devices in the 1970s. In the following decades the size and cost of these devices were reduced to such an extent that nowadays most people own multiple mobile computing devices, and the world around us is filled with computers that are deeply embedded in the objects they control. It is envisioned that in the coming decade all these embedded devices will be further integrated into an Internet of Things [7], bringing new technologies, application areas and continuous growth to a market that already has a yearly world-wide turnover of 300 billion dollars [25].

This large-scale technology revolution has been made possible by continuous rapid advances in semiconductor technology, which enable chip manufacturers to put more and more functionality into a single integrated circuit (IC) at lower cost per transistor. As a result, applications can grow larger and more complex, encompass more functionality and perform this functionality in a shorter amount of time. The main research and development effort of semiconductor industry focuses on down-scaling the size of transistors to enable higher transistor densities, an increased performance per Watt and a lower cost per transistor. In 1965 Gordon Moore, co-founder of Intel, predicted that the transistor density at minimum cost per transistor will double roughly every 18-24 months [59]. This prediction was popularized into Moore's law, and is still in effect today: where a typical IC in 1971 contained about 2300 transistors, current state-of-the-art IC designs can contain several billions [63] up to even 20 billion transistors [89].

Transistor size is mainly determined by the quality of and accuracy at which structures that realize a transistor can be applied to a silicon substrate. This is usually done by means of a photolithographic process step, in which light patterns

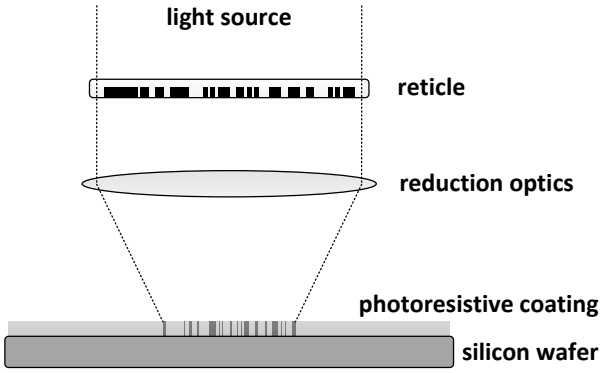


FIGURE 1.1: *Principle of photolithography. Patterns are imprinted on a silicon substrate (wafer) coated with a photosensitive layer by projecting an image onto it. The image is formed by passing light through a patterned quartz plate (reticle) and a column of lenses. The reticle pattern is a 4:1 scaled image of an IC layer.*

are imprinted on a silicon substrate with a photosensitive coating. The width and accuracy of the projected image features mainly determines the minimum transistor size. Hence, Moore's law is for a great deal driven by advances in photolithography tools.

1.1 Photolithography

Figure 1.1 shows the principle of photolithography. Light from a light source passes through a patterned quartz plate called a reticle. The reticle contains a 4:1 scaled image of the IC layer pattern that is to be applied. The image passes through a reduction lens and is then projected onto a circular silicon substrate called a wafer. The wafer surface has a photosensitive coating (photoresist) that hardens when exposed to ultra-violet light. As a result, after exposing the wafer to the light the image is imprinted on the wafer as a pattern of hardened photoresist.

In subsequent production steps the wafer is developed by rinsing away any photoresist that has not hardened, prior to chemically etching away the areas of the wafer toplayer that have no remaining photoresist. Finally, also the hardened coating is removed. To create any subsequent layers, a new metal, oxide or semiconductor layer is applied to the wafer, as well as a new photoresist coating. Then, the cycle of exposure and development can be performed again. This can be repeated for up to 40 times, building up an IC layer by layer.

There exist mainly two variations of the highly complex machines that perform the lithographic process step in semiconductor factories: wafer steppers and, more recently, wafer scanners. A wafer stepper fully exposes the complete reticle, such that the reticle pattern is applied to the full IC area in one step. Wafer scanners on

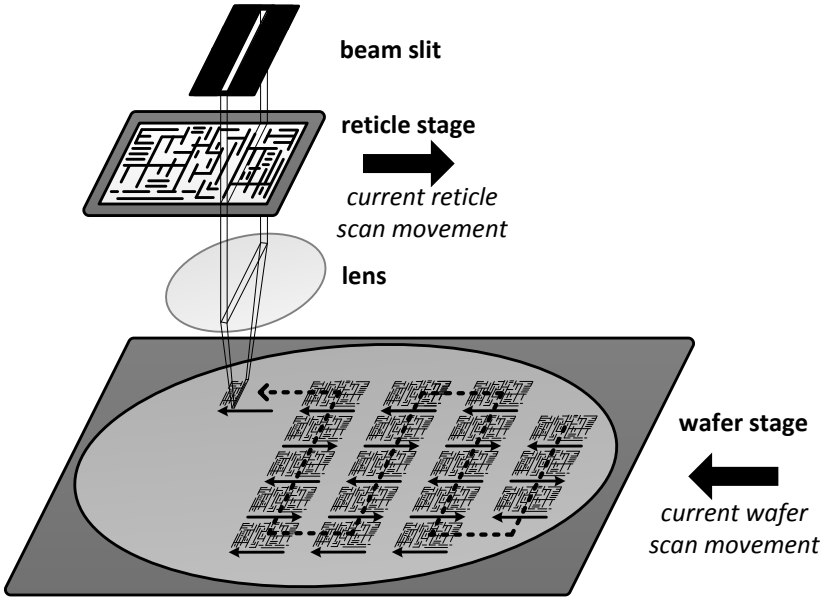


FIGURE 1.2: *Wafer scanner operation. The light is fed through to a slit, such that the light stripe that remains is as wide as the reticle but only a fraction of its length. The IC area is exposed by moving the reticle and wafer in opposite directions.*

the other hand do not expose the whole IC area at once, but instead only expose a narrow stripe that is as wide as the reticle but only a fraction of its length. This reduces the complexity and cost of optics and allows for larger IC exposure areas [81]. Exposure of the full IC area is achieved by scanning, i.e. moving the reticle and wafer in opposite directions in a synchronized way, as shown in Figure 1.2. The figure shows the reticle and wafer placed on stages that can be moved in multiple degrees of freedom (DoF), typically at nanometer accuracy and with accelerations exceeding 20G [10]. The light stripe on the wafer follows the distinct step-and-scan pattern. This zig-zag pattern consists of a synchronized scanning motion (solid arrows) that scans the whole reticle while exposing the area of one IC on the wafer, and the step motion that returns the reticle to its initial position while positioning the wafer such that the light stripe is ready for a scan motion to expose the next IC area (dotted line).

Transistor size and cost strongly depend on the quality of the pattern projections on the wafer surface. In a wafer scanner this does not only depend on the quality of the optics, but also on the accuracy of the scanning motion of the stages and their relative positioning [10]. Current state-of-the-art scanners can accurately project reticle patterns onto a wafer with a precision of a few nanome-

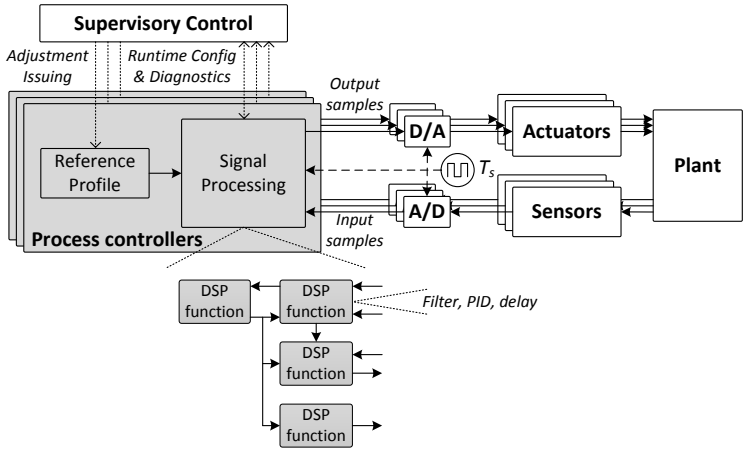


FIGURE 1.3: *Typical structure of a mechatronic system. The process controllers are usually implemented in software, hence, the grey parts can be considered a mechatronic application.*

ters. To achieve smaller transistor feature sizes it is paramount to obtain better control over the motion of wafer and reticle (and other processes that affect the operation of the scanner) by means of high-performance control systems.

1.2 Control Systems

Figure 1.3 shows a schematic of a control system consisting of a supervisory controller, one or more process controllers and a set of sensors and actuators. The goal of a control system is to perform high-level actions, e.g. apply a patterned beam of light to the surface of a moving wafer, or to position a wafer to a specific position under a lens in order to prepare it for the next exposure. It does so by coordinating a set of subsystems (e.g. wafer stage, reticle stage, light source) that need to work together to perform the high-level action. To this end, the supervisory controller translates high-level actions into a set of timed adjustment commands, which it sends to the different process controllers that control the relevant physical processes of these subsystems.

The supervisory controller also collects status information from the process controllers and can change their behaviour. The latter is required to e.g. perform different types of motions that are part of the step-and-scan pattern that a wafer scanner follows during exposure of a wafer. The scan motion must be very tightly synchronized, accurate and extremely resilient to disturbances, while the step motion is required to be fast rather than accurate. The corresponding process controller behaviour, i.e. control modes, are set by the supervisory

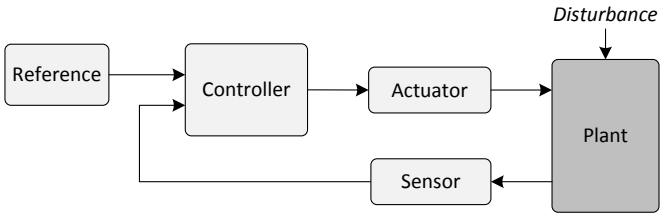


FIGURE 1.4: *Control loop principle. The objective of the control loop is to follow a reference input as good as possible while rejecting external disturbances.*

controller during machine operation by changing algorithmic parameters of the process controllers.

The process controllers control the dynamic state of a physical process (position, speed, temperature, fluid level) in the plant. Figure 1.4 shows the basic structure of a process controller. Its purpose is to alter the behaviour of the controlled process variable, such that this variable follows a reference input as good as possible while rejecting external disturbances. To this end, the controller calculates its output by feedforward, i.e. output adjustment based on knowledge of the plant dynamics and an estimate of future disturbances or a trajectory to follow, and by feedback, i.e. output adjustment based on the difference between its reference input and actual measurements of the process variable through sensors at the plant. An actuator transforms the controller output into a physical action (e.g. a force) that changes the value of the corresponding process variable of the plant.

Control systems are typically implemented digitally, since this provides several advantages (noise sensitivity, accuracy, cost) over their analog counterparts. To this end, controller implementations employ A/D conversion between the sensors and the controller input, and D/A conversion between controller output and the actuators, as shown in Figure 1.3. The AD/DA conversion is typically time-triggered with a fixed sampling period and a time offset between the A/D and the D/A trigger. A digital process controller periodically follows a *sense-compute-actuate* cycle, in which the controller is triggered by the arrival of new sensor samples, to which it applies a set of digital signal operations to compute the controller output, which is subsequently sent to the actuator after DA conversion.

The ability of a process controller to control the dynamic behaviour of a plant is not only dependent on its functional behaviour and the physical properties of the plant, but also on the timing with which the control output is applied to the plant [10, 21]. The sooner the control action can be applied to the plant in reaction to a measurement, the more effective the control action is. Controller timing is expressed in the sampling frequency, i.e. the rate at which it can process new input samples, and IO-delay, i.e. the delay between the arrival of an input sample and the availability of the corresponding output sample. The targeted controller

performance specified during the design of a control system implicitly sets sample frequency and IO-delay constraints on its implementation. The underlying execution platform must in turn be able to execute the controller computations and data communications in such a way that these constraints are guaranteed to be met.

1.3 Trends in Wafer Scanner Control

ASML [6] is the world's leading supplier of wafer scanners and steppers, whose technology leadership role has set the standards for lithography tools for many years. The general development of ASML wafer scanners focuses mainly on improving accuracy and productivity. As a result, the minimum feature size of ASML scanners has steadily decreased from 350 nm in 1994 to less then 20 nm today. In that same time, the wafer surface area has doubled, and machine throughput has tripled [10].

Since the introduction of the TWINSCAN^{TM} lithography platform in 2000, ASML scanners feature dual wafer stages that decouple wafer measurement and exposure by performing these processes in parallel. The resulting improvement in system throughput has made such systems industry-standard for the 300 mm wafer market. Immersion lithography is another revolutionary innovation that was introduced in 2005. It employs a layer of water between the lens and the wafer to improve optical performance such that feature size can be decreased further. This layer of water is maintained during the wafer motion under the static lens, hence its vibrations and volume have to be controlled by additional controllers that are integrated into the overall control system. Recently, EUV (Extreme Ultra-Violet) scanners are being introduced to the semiconductor market. These scanners use a light source that emits light at an extremely short wavelength, typically only a few nanometers. Due to the different light properties, such systems use reflective optics and are operated under vacuum conditions which, besides the already extremely complex EUV light source, considerably raise overall system complexity.

A possible future development is an increase in wafer diameter from 300 mm to 450 mm [84]. To maintain rigid body behaviour of the corresponding larger 450 mm stages, their stiffness must be increased, which results in significant additional stage mass. The larger forces required to accelerate these heavy stages in turn scales up motors and power electronics, as well as power consumption and heat dissipation. As an alternative, less stiff stages can be used [46]. Such lightweight stages do not act as a single rigid body under acceleration, but tend to deform and vibrate. These mostly high-frequent vibrations can actively be suppressed by over-sensing and over-actuating, i.e. by the addition of control loops in more degrees of freedom than those of the actual stage. Due to the high-frequent nature of the non-rigid body vibrations, vibrational suppressors typically require high sample frequencies. This, in combination with the extra degrees of freedom to control, results in a significant computational load.

Besides these specific system innovations, generic developments to achieve higher scanner accuracy and throughput include improved motion control at higher wafer and reticle stage accelerations. Reducing feature size generally requires more disturbances to be rejected. Higher stage accelerations improve system throughput, but also introduce additional vibrations that need to be suppressed in order to maintain accuracy. As a result, developments in accuracy and throughput typically increase the number, complexity and sample frequencies of the process controllers, and requires decreased IO-delays.

Typical implementation platforms for wafer scanners consists of a set of general-purpose processors (GPPs) connected by a high-bandwidth interconnect and dedicated electronic boards that interface these GPPs with sensors and actuators. For years, GPP platforms could provide abundant performance due to rapid advances in processor frequency scaling. Since the stall of processor frequency scaling at around 2005, the growing computational demands of high-performance digital control applications are rapidly catching up with available GPP platform performance. As a result, the implementation effort required to meet application timing constraints is increasing, even to such an extent that performance concessions have to be made to make ends meet.

Trends in the domain of high-performance mechatronic systems can be summarized as follows:

- A rapid increase of complexity and scale of computational demand due to rising system complexity.
- Increasingly strict application timing requirements as a result of increasing system accuracy and throughput requirements.
- The use of general-purpose processor platforms for the real-time execution of digital controllers.

1.4 Next-Generation Wafer Stage Controller

Figure 1.5 shows the task graph of a next-generation wafer stage controller of a wafer scanner. It consists of a current generation Long-Stroke wafer stage controller that controls the wafer stage movement in 6-DoF at micrometer accuracy for long ranges, and a concept of a next-generation Short-Stroke wafer stage controller that controls stage movement in 11-DoF at nanometer accuracy for short ranges. Both controllers operate in tandem to position the wafer under the exposure lens. The Short-Stroke controller is drafted by ASML mechatronics research as possible implementation candidate of a 450 mm wafer stage controller.

The Long-Stroke controller, comprising the right side of the graph is a classic PID-type controller (see Chapter 2), and the Short-Stroke controller that comprises the left side of the graph is a state-space controller (see Chapter 2) with integrated non-rigid body vibration suppression. The Long-Stroke controller contains tasks that are mostly sequential, while the Short-Stroke controller contains

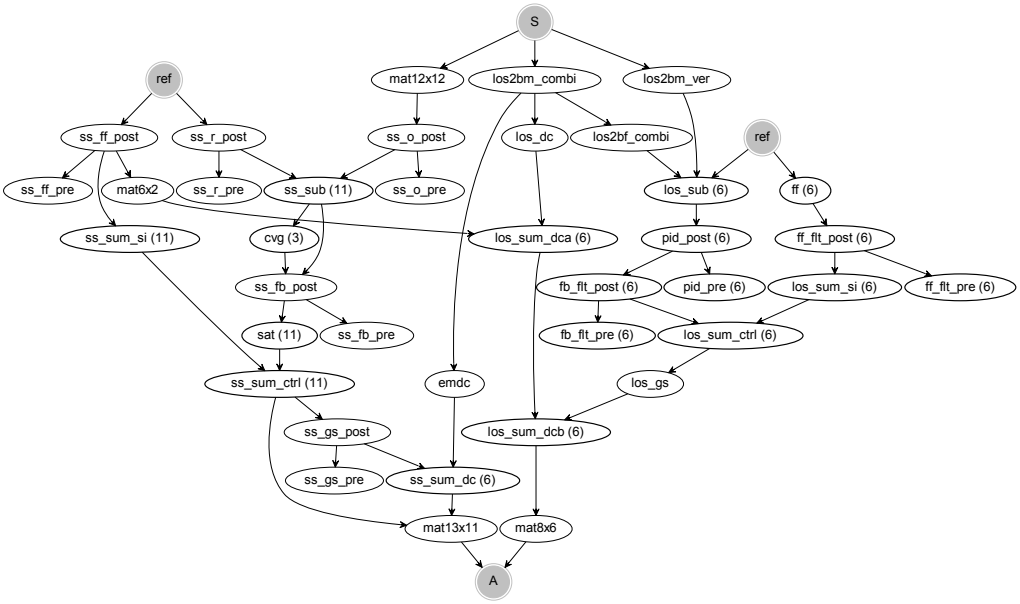


FIGURE 1.5: *Task flow graph of a next-generation wafer stage controller.*

mostly tasks with significant data-level parallelism. The required sample frequency and IO-latency for these controllers is 100kHz and 10μs respectively. The IO-latency includes 6.5μs required to communicate data between IO-boards and the processors, leaving 3.5μs to calculate an output sample.

When mapping these controllers to a GPP platform consisting of a single high-end octo-core (Freescale P4080) that is typically used for wafer scanner controllers, it is not possible to meet these timing requirements. Timing analysis calibrated with average-case profiling data from actual measurements of most of the constituent tasks, and extrapolated calibrations for tasks for which no measurement is available, has shown that the best-case achievable sampling frequency and IO-delay are 20 kHz and 4.4μs respectively. This analysis assumes that the GPP cores can communicate instantaneously. Even adding a second octo-core processor to map this controller will not result in sufficient performance. In this dual octo-core case, larger blocks have been split over multiple cores with zero overhead, and also the two octo-cores are assumed to communicate instantaneously. Then, the achievable sampling frequency and IO-delay are 40 kHz and 3.9 μs respectively.

These numbers reflect a best-case situation calibrated with average-case execution times. One would like to be able to provide guarantees that the system is in any case able to meet a certain guaranteed minimum performance, which

is equal to or better than the application timing constraints. The guaranteed performance of the two GPP implementations is actually much lower than the numbers obtained with the timing analysis mention above, making the situation in fact even worse.

1.5 Problem Statement

The increasing accuracy and throughput requirements for the high-performance mechatronics systems that drive Moore's law have skyrocketed the scale, complexity and time-criticality of the digital control applications that control them. It is getting more and more difficult to meet the strict application timing requirements when mapping these control applications to the general-purpose platforms typically used in this application domain. At some point it will affect product cost and time-to-market through the required additional development effort, or product performance will slip due to concessions on mechatronic performance as an alternative to alleviate the strict application timing constraints.

With the ever tighter timing constraints of digital control applications, the mismatch between these applications and GPP architectures is becoming more evident. GPP architectures are optimized for average-case performance, and have to be able to run any program that can be expressed in a generic high-level programming language like C. As a result, the GPP architecture is very complex, contains all kinds of speculative and hierarchical hardware, and its ability to thoroughly exploit fine grained levels of parallelism is limited. This is in stark contrast with the requirements of real-time digital control applications. For these applications a low latency is important rather than average throughput, and the large fluctuations in execution time that are common with GPP execution have an adverse effect on control performance [12].

GPP platforms are typically composed by a set of homogeneous GPP processors that consist of one or multiple (typically up to 8) homogeneous cores. The processors are connected by a high-bandwidth interconnect, for which inter-processor communication latencies in the order of 0.1-1μs are not uncommon. The cores within a processor are able to communicate with a much lower latency through shared cache memory. Complex digital controllers can consist of hundreds or even thousands of tasks, typically with considerable task-level parallelism. However, the relatively high inter-processor communication latencies of GPP platforms limit the application mapping freedom, because typical application partitioning is, out of necessity, based on minimizing inter-processor communication. As a result, GPP platforms can only exploit a small part of the task-level parallelism present in the application.

Digital control applications typically consist of a heterogeneous mix of tasks with different amounts and levels of parallelism, ranging from simple additions to large matrix calculations or non-linear operations. GPPs are unable to match the heterogeneous nature of these applications with their one-size-fits-all archi-

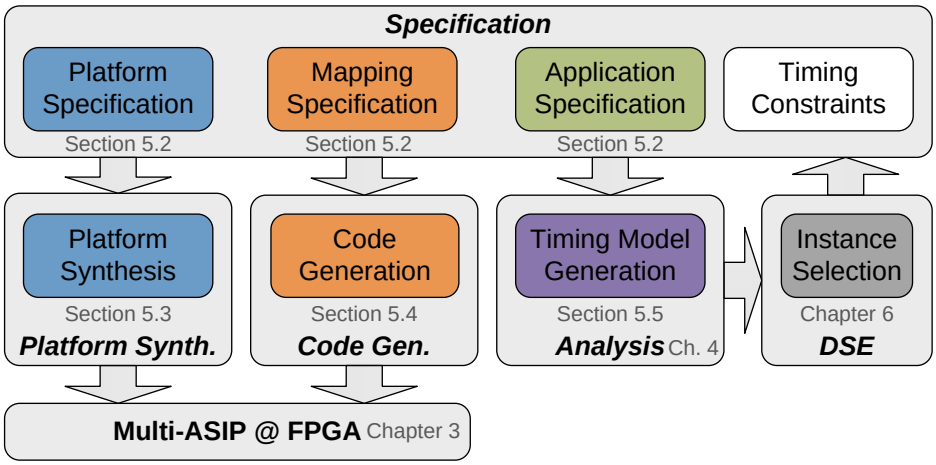


FIGURE 1.6: *Overview of the platform-based design flow.*

tecture. For instance, many GPP architectures have hardware support for vector operations [17] by means of instruction set extensions in each of the GPP cores. With these extensions, all the available GPP cores have the same limited vector operation support, typically limited to 128 bit packed data types. As a result, none of the cores can effectively exploit data-level parallelism for wide vectors.

The difficulty of GPP platforms to meet the strict application latency demands is further exacerbated by their inefficient synchronization methods. Typical processor synchronization in GPP platforms is implemented with trigger signals sent over the interconnect network [43]. The arrival of these time-triggered synchronization signals is necessarily dimensioned for the worst-case, so with the large jitter common to GPP execution this can lead to a considerable over-dimensioning. Also inter-processor and inter-core synchronization in such platforms can result in wasting valuable processor cycles, as such synchronization is typically implemented by polling mechanisms in software.

The tremendous flexibility and ease of programming of GPPs is one of the major reasons that these architectures have become so popular in the digital control domain. When platform performance was still far exceeding the requirements of the applications, GPP architectures could easily be programmed by developers without any specialized platform knowledge or code optimization skills. Now, with increasingly tight latency constraints, more and more effort needs to be put in code optimization in order to meet these constraints. As a result, programmers need to know more low-level architecture details, and hence the ease of programming advantage of GPPs is quickly vanishing.

The lack of predictability of GPP systems makes it very difficult to guarantee that the performance observed with timing analysis will also be met in the

corresponding implementation. The performance numbers obtained by e.g. profiling the execution of a controller mapped to an execution platform are mapping-dependent, because the specific mapping determines e.g. the caching behaviour and contention on shared resources. As a result, model-based evaluation of different design alternatives is only indicative for the actual performance of the implementation. This performance gap between model and implementation often results in under- or over-dimensioning of the solution. As argued above, continuing the sole use of GPP platforms is not viable. There is a need for execution platforms that are better able to meet the ever stricter timing requirements of high-performance digital control applications. In this thesis, we will investigate whether heterogeneous platforms tailored to the application domain are capable of addressing this problem. In addition, the issue of programming and analysis of such platforms is addressed.

1.6 Contributions

This thesis discusses a platform-based approach for digital control applications, in particular high-performance mechatronics, in which control applications are mapped to instances of a heterogeneous platform template. The different elements that form the corresponding design flow are shown in Figure 1.6. The figure and the related thesis contributions will be explained in the remainder of this section.

Heterogeneous Platform Architecture Template for High-Performance Mechatronic Systems (Chapter 3)

Starting at the bottom, the box labeled *multi-ASIP @ FPGA* denotes the mapping target of the design flow shown in the figure. In order to be able to meet the strict timing requirements of modern digital control applications, this thesis presents a heterogeneous platform template tuned to the digital control domain. It aims to alleviate the difficulties posed by the use of GPP platforms as target platform for digital control applications by providing simple processing units (PUs) that focus on the low-latency execution of control tasks. The PUs are Application-Specific Instruction-set Processors (ASIPs), tuned towards a specific set of tasks while still being programmable. The PUs efficiently synchronize according to the (Homogeneous) Synchronous Dataflow [48] ((H)SDF) semantics. This enables easy integration of different PUs into a platform instance with little overhead. The details of this platform are explained in Chapter 3 and have been published in [23].

An Application Timing Analysis Method for Systems with Shared Resources that are Scheduled First-Come-First-Served (Chapter 4)

The proposed architecture template is envisioned as a part of a networked system that connects it with dedicated sensor and actuator electronics and possibly other processor boards. As will be explained in Chapter 2, the data samples calculated by a digital controller have to be present at these time-triggered actuator boards before their triggering moment, otherwise the plant will be actuated with out-of-date actuation data. Therefore, it is important that the performance of the overall system can be analyzed, in order to verify that these timing deadlines can be met in worst-case. Such analysis is typically done by dataflow analysis.

However, existing dataflow-based timing analysis techniques cannot deal with shared resources whose access is arbitrated in a First-Come-First-Served (FCFS) way. Such arbitration is often applied in industry-standard networks. This thesis presents a scalable analysis technique that can provide conservative timing bounds on the execution of applications mapped to platforms that have FCFS-arbitrated shared resources. The application is modeled as a periodically restarted Directed Acyclic Graph (DAG), where the execution times of tasks are specified as intervals that denote their best-case and worst-case execution time. The details of this analysis technique are discussed in Chapter 4 and have been published in [22].

A Platform-Based Design Approach for Mechatronic Systems (Chapter 5)

The use of specialized heterogeneous architectures complicates the task of programming and analyzing the execution platform. Tasks can have multiple different mapping targets that have a completely different datapath, and tasks mapped to different processing units need to communicate and synchronize. This thesis aims to mitigate this complexity, by means of a platform-based design-flow.

The platform-based design paradigm [69] raises the level of abstraction of a system by specifying it in terms of instantiated components from a library and their relation. The central concept in this paradigm is a platform, which offers an abstraction that hides the implementation details of lower layers, and consists of a library of components that can be instantiated and connected together according to some rules to form a platform instance. At the higher platform level, the components are represented by abstract models that only reflect the properties of interest of that particular abstraction level.

In the platform-based design-flow presented in this thesis, both the hardware and the software are abstracted by a platform. The execution platform is specified in terms of its processing units and their connection. Similarly, the application is specified as a graph with nodes corresponding to software tasks and edges corresponding to inter-task communication. Together with a mapping model, our design-flow automatically bridges the implementation gap by generating an HDL implementation of the platform model together with binary code that configures and programs the PUs. This corresponds to the boxes labeled *Specification*, *Plat-*

form Synthesis and *Code Generation* in Figure 1.6.

The abstractions provided by our design-flow enable easy programming of the heterogeneous platforms which are generated with our design-flow. Also the design of the execution platform itself is simplified by similar abstractions. The details of our platform-based design-flow are discussed in Chapter 5.

A Proof-of-Concept with an Industrial Case Study (Chapter 6)

The final contribution of this thesis is a proof-of-concept that demonstrates the mapping of the next-generation wafer stage application discussed in Section 1.4 to an instance of the platform template of Chapter 3. With this industrial case study, we show how to find an efficient platform instance in just a few steps, that is able to meet both the IO-delay constraint of 3.5 μs and the throughput constraint of 100 kHz even at worst-case conditions. The details of this case study are presented in Chapter 6.

1.7 Thesis Overview

The remainder of this thesis is organized as follows. The application domain characteristics and their relation to timing performance are discussed in Chapter 2. Chapter 3 introduces a heterogeneous platform template which targets high-performance mechatronic applications. A novel analysis technique that takes into account contention on shared resources is discussed in Chapter 4, followed by a detailed presentation of our platform-based design flow in Chapter 5. In Chapter 6, the application case study is systematically mapped to an instance of the platform template of Chapter 3 using the analysis technique and design flow of Chapters 4 and 5. Finally, Chapter 7 discusses the main conclusions and future work.

Chapter 2

Application

Chapter 1 introduced the basic concepts of a mechatronic system, and explained that the controller, which forces the state of one or more physical processes of the system to some desired value, is typically implemented in software. This chapter defines the scope of such mechatronic applications as considered in this thesis, shows the important application characteristics, and discusses application modeling from a mechatronics as well as an embedded systems point of view.

2.1 Application Characteristics

Figure 2.1 shows a mechatronic application as part of a mechatronic system. The process controllers, which calculate the stimuli required to achieve a certain state of a physical process under control, are by far the most time- and performance critical part of the controller. Therefore, in the remainder of this work, we consider the mechatronic application to encompass these process controllers up to and including their interaction with the supervisory controller.

The mechatronic application defined as such is shown in grey in Figure 2.1. Each process controller consists of a reference profile generator that generates a sequence of setpoints based on (relatively infrequent) adjustment requests received from the supervisory controller, and a signal processing part that calculates plant stimuli based on these setpoints, controller state and plant measurements. The next subsections discuss the important characteristics of the process controllers and their constituents.

2.1.1 Controller Structure and Execution

The primary building blocks of the application are the digital signal processing functions, called blocks. Blocks have a strictly local scope, operating only on their own inputs, possible internal state and (often parameterized) coefficients. Each

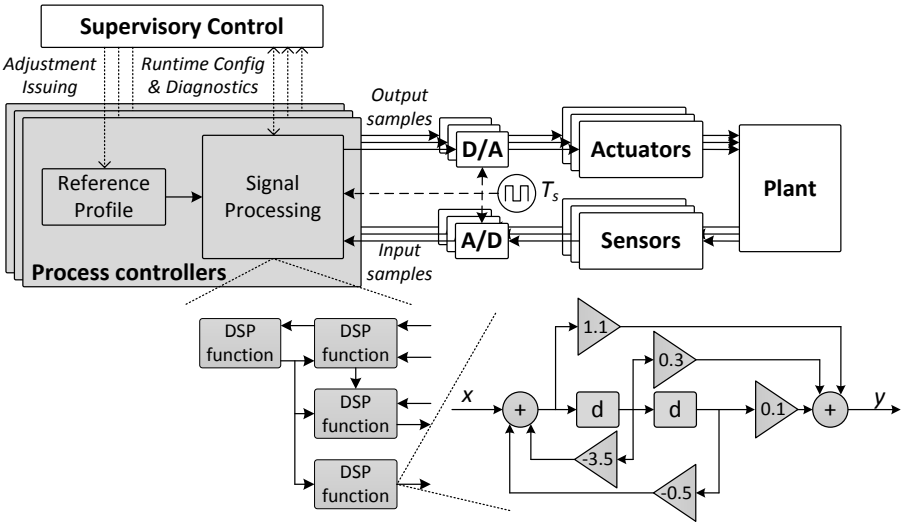


FIGURE 2.1: *Typical structure of a mechatronic system. The process controllers are usually implemented in software, hence, the grey parts can be considered a mechatronic application.*

block receives input values through explicit communication on its input ports prior to the start of its execution. When a block finishes executing it sends a set of output samples using explicit communication on its output ports.

Because of the local scope of the DSP blocks and the explicit communication between them, mechatronic applications are usually represented with a graph structure, where the vertices correspond to signal processing blocks and the edges correspond to block communication. The structure of connected blocks offers a natural application partition for exploitation of task-level parallelism.

Block execution is strictly periodic, i.e. new input samples that arrive at equidistant time instances trigger the process controller to execute. Typically, in an implementation this triggering is time-based, by a trigger signal that arrives at the controller just after the worst-case arrival time of input data. After getting triggered, the controller typically executes it constituent blocks in a fixed order that is determined at compile-time.

In this work, the application is assumed to be single rate, which means that all the blocks in the application run at the same sample frequency, and thus all consume and produce samples at the same rate. Then, each triggering of the controller results in all of its constituent blocks to be executed exactly once. Also, all blocks must have finished their execution before a new input sample arrives, since, as will be explained in the next section (in particular Figure 2.2), pipelining does not improve control performance significantly.

Digital controllers typically consist of a heterogeneous mix of DSP blocks,

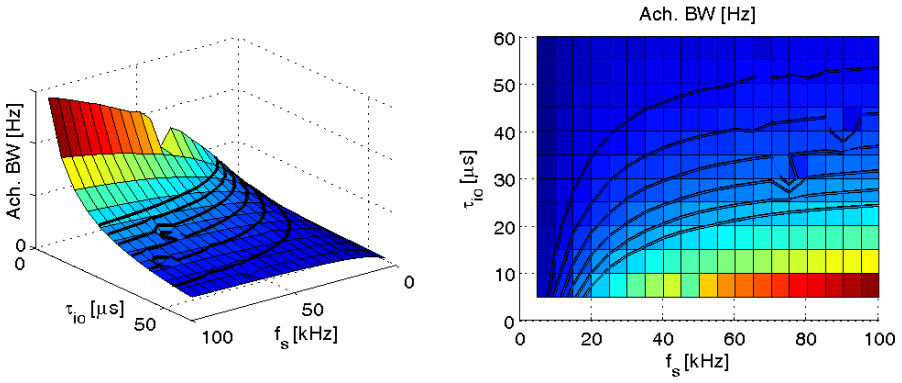


FIGURE 2.2: *Achievable bandwidth as a function of the sample frequency f_s and output delay τ_{io}. Courtesy of M.M.J. van de Wal, ASML.*

with a widely varying level of data-level parallelism ranging from simple signal summation blocks to cascaded filters, and to state-space blocks with state matrices of several tens of thousands of elements. Section 2.6 shows some typical control blocks.

2.1.2 Timing and Performance

The process controllers calculate plant stimuli such that the plant property under control tracks the generated reference values as best as possible, while rejecting any unwanted external or internal disturbances. An important metric that expresses the ability to do so in terms of frequency response is system bandwidth, i.e. the highest frequency at which a system is able to have its output track a sinusoïdal input in a satisfactory manner [21]. The achievable bandwidth not only depends on the functional behaviour of the digital controller, but also on timing properties like sample frequency and IO-delay.

The sample frequency is the frequency at which new sensor input data becomes available for processing. Since all signal processing functions need to be executed before the next sample arrives, the sample frequency imposes a timing constraint for the execution of all the signal processing functions. The IO-delay is the delay between the arrival of input sensor data and the availability of output actuator data. This delay imparts a separate timing constraint on a subset of the signal processing functions, i.e. the signal processing functions that contribute directly to the output.

Figure 2.2 shows a typical plot of the achievable bandwidth as a function of sample frequency and delay. It shows that the achievable bandwidth is increasing when the sample frequency increases, but the performance gain is rapidly decreasing at higher I/O delays, implying that application pipelining is not very useful to

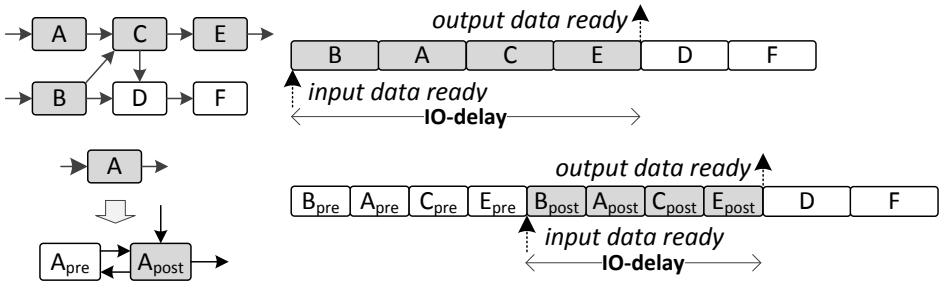


FIGURE 2.3: *IO-delay optimization. Blocks that contribute to the output (critical blocks, grey-shaded) are executed first (top right). Splitting critical blocks into a post and a pre part can further optimize IO-latency (bottom).*

gain control performance. This relation between realizable control performance and application timing constraints imposes a fundamental trade-off between control performance and the performance requirements on the underlying execution platform.

2.1.3 Application Optimization

The top left of Figure 2.3 shows an example DSP network consisting of 6 blocks. Blocks A, B, C and E, colored grey, are denoted 'critical', since they contribute to the output samples of the controller and are therefore also contributing to the IO-delay. To minimize overall IO-delay of the controller, the execution of these critical blocks has priority over that of the non-critical blocks, i.e. they show up early in a block execution schedule, as shown in the top right of Figure 2.3.

To further optimize the IO-delay of an application, critical blocks can be split into two sub-blocks denoted 'pre' and 'post'. The 'pre'-block contains all the computations that are not directly dependent on the inputs of the original block (e.g. state-updating operations), while the 'post' contains only computations that are directly dependent on the inputs of the original block and on the results of the 'pre' block. This is shown in the bottom left of Figure 2.3.

The 'pre' block can be executed prior to the arrival of new input, so when the new input arrives only a minimal amount of computation needs to be performed to calculate the block output. This can greatly reduce the IO-delay compared to the execution of the original blocks, as shown in the bottom right of Figure 2.3. Due to the overhead and possible extra data communication associated with block splitting, the combined execution time of the 'pre' and 'post' block is usually larger than that of the original block.

When critical blocks are split in this way, the 'pre' blocks are executed before the triggering of the controller, i.e. when input sample n arrives, the 'pre' that is required to compute output sample n has already been executed during the

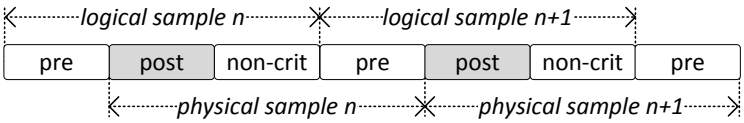


FIGURE 2.4: *Physical vs. logical sample.*

controller triggering caused by input sample $n-1$. This is shown in Figure 2.4, which shows the executed blocks of multiple samples with the 'post' blocks shaded in grey. The figure shows the distinction between the so-called *logical* sample, and the *physical* sample. The logical sample consists of all the blocks that are related to a particular input sample, and the physical sample consists of all blocks that are executed as a result of a controller triggering. The start of the physical sample is marked by the arrival of new input data. The 'post' blocks, which contribute to the IO-delay, are executed first, followed by non-critical and 'pre' blocks. Hence, in the physical sample, the 'post' and 'pre' terminology might look contradictory, because there the 'post' blocks are executed *before* the 'pre' blocks. However, from a logical sample perspective, which starts at the execution of the 'pre' blocks, followed by the 'post' and the non-critical blocks, the terminology is more intuitive.

2.2 Mechatronic Application Model

A mechatronic system is typically modeled as a network of connected blocks whose behaviour is expressed as a transfer function in the s-domain. Since the time-domain interactions between systems and signals are quite complex, analysis in the s-domain provides the necessary abstractions to reason about the dynamic behaviour of larger and more complex systems. The complex interactions in the time domain transform into simple algebraic relations in the s-domain, providing a comprehensible means to calculate the overall system response of composite structures like e.g. feedback loops or the summation of different signal paths in a system [21, 30].

Even though most controllers are implemented digitally, typically both controller and plant are modeled and designed in the continuous s-domain, due to the multitude of well-known design techniques in that domain. When the controller model is tuned such that a satisfactory dynamic plant performance is achieved, the effects of digitization are taken into account, as shown in Figure 2.5. Each controller block in the continuous s-domain model is approximated by its discrete counterpart in the z-domain, e.g. by using a bilinear approximation [21]:

$$H(z) = H(s)\big|_{s=\frac{2}{T_s}\frac{z-1}{z+1}} \tag{2.1}$$

The substitution in Equation 2.1 accounts for the effect of synchronously sam-

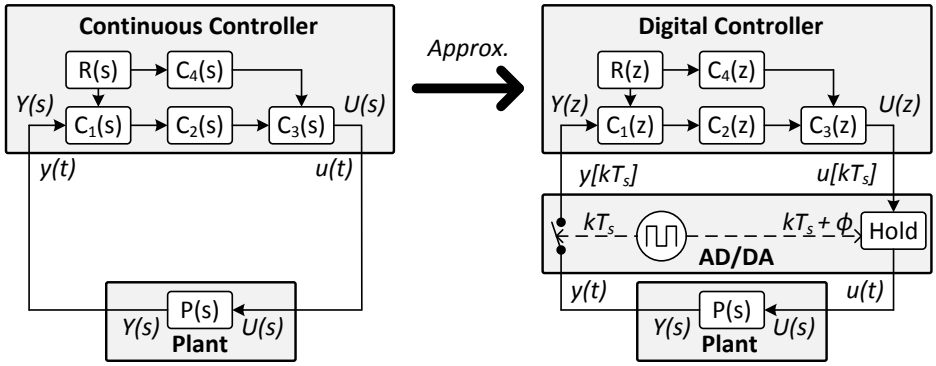


FIGURE 2.5: *Continuous model in the s-domain (left) and its approximate discrete counterpart in the z-domain (right).*

pling the controller inputs with a sample period T_s. In an implementation, the controller output samples are converted to a continuous signal by a DAC by passing them through a sample and hold circuit. To represent the effects of this hold circuit, a block with the appropriate transfer function is added to the model (e.g. $\frac{1-e^{-sT_s}}{s}$ for a zero-order hold).

2.3 Modeling Distributed Execution

A discrete-time controller model in the z-domain, like the one on the right of Figure 2.5, specifies how to calculate the ordered sequence $u[0], ..., u[k-1], u[k]$ corresponding to output u, given the ordered input sequence $y[0], ..., y[k-1], y[k]$ corresponding to input y. The controller periodically receives input samples from the ADC, which trigger the controller to execute its blocks. When the required blocks have been executed, the controller sends an output sample to the DAC. Both ADC and DAC are time-triggered, i.e. every kT_s seconds the ADC sends a new sample to the controller, and after some time offset ϕ, the DAC converts the last received controller output sample to a continuous signal and sends it to the actuators every $kT_s + \phi$ seconds. Due to the periodic sampling of the inputs and outputs of the controller, the sequence relations specified by the z-model impose strict timing constraints on the execution of its constituent blocks.

Figure 2.6 shows an example of digital controller timing in its z-domain model and its implementation. In the z-domain model, shown in the top left of the figure, the operations required to calculate the output of each block are assumed to execute in zero-time. In an implementation however, when the controller is mapped to a (distributed) execution platform, both the execution of blocks and their communication take a non-zero amount of time. This is shown at the top right of the figure, where execution delay is represented by the bars in the figure.

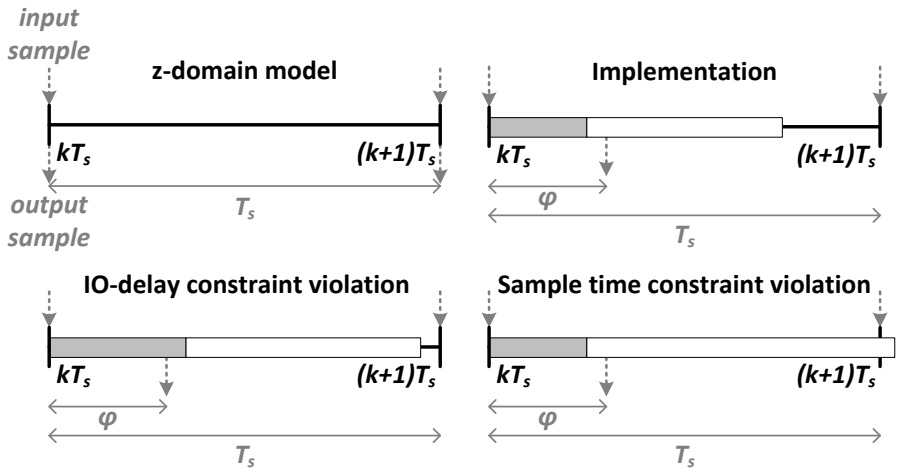


FIGURE 2.6: *Execution timing of a digital controller. The z-domain model (top left) assumes that blocks execute instantaneously. In an implementation, block execution takes a non-zero amount of time (top right). Violation of IO-delay or sample time constraints is a violation of model semantics (bottom left and right).*

The grey shading of these bars corresponds to the grey shading of Figure 2.4, denoting the 'post', non-critical and 'pre' blocks of the physical sample. In the controller implementation, some fixed offset ϕ is used as a trigger for the DAC, so the output samples are effectuated at the plant every $kT_s + \phi$ seconds.

To respect model semantics (and the corresponding dynamic performance), all transfer function calculations contributing to the k-th output sample (represented by the light grey bars in the figure) must therefore be completed before $kT_s + \phi$, since otherwise the $(k-1)$-th output is used by the DAC instead. This IO-delay constraint violation is shown in the bottom left of Figure 2.6. Also, the controller must complete the calculations of its complete set of blocks (light-grey + dark-grey bars in the figure) within one sample period T_s to be able to keep up with the arrival rate of the inputs. The bottom right of Figure 2.6 shows a sample time constraint violation, which delays the start of the calculations of the next sample.

Due to its lack of timing semantics, the z-domain controller model is not suited for timing verification of its future implementation. For this purpose, a model is needed in which the execution times of the transfer blocks are explicitly modeled, with an operational semantics that enable timing analysis of their distributed execution, like the Synchronous Data Flow (SDF [48, 72, 74]) model of computation.

Table 2.1 shows a comparison of model properties in the z-domain and the SDF domain. A z-domain model has a purely declarative semantics that relates an ordered stream of input samples to an ordered stream of output samples (based on the initial states of blocks). Only the relative ordering between input and output

TABLE 2.1: *Comparison of z-domain and SDF model properties.*

Property	**z-domain**	**SDF**
Declarative semantics	✓	✗
Operational semantics	✗	✓
Timed	✗	✓
Ordered streams	✓	✗
Timing constraints	✓	✗

samples is specified, there is no notion of time and thus no assumptions on the sampling moment, execution times or communication delays. The model has implicit timing constraints on the production and consumption of samples. Not meeting these timing constraints means that outdated samples are used, which is a violation of the model semantics.

The SDF domain on the other hand has an operational semantics. It explicitly models when to start and finish the firing of actors, the consumption of time and the transfer of (symbolic) data through the model. SDF does not make assumptions on token ordering: conceptually an actor can be firing concurrently an unbounded number of times, and any of these firings may complete earlier than the upper bound provided by its execution time. A firing that is started later might therefore produce its tokens earlier than a firing that started earlier, so the order of tokens produced at the output of an actor is not related to the order in which tokens are consumed at its input. SDF has no notion of timing constraints; not meeting a timing constraint is not a violation of model semantics.

We propose to use the timed Synchronous Data Flow (SDF [48]) semantics to model and analyze the timing of (distributed) controller executions, as shown in Figure 2.7. The z-domain model (shown in the left of the figure), which assumes execution times to be zero, is transformed into an SDF graph (Figure 2.7, middle) that explicitly models the execution time required to compute the output of each transfer block in the z-model. The periodic controller sampling can be represented by a source actor and sink actor actor with a self-edge and an execution time T_s (Figure 2.7, right). The original timing constraints associated with the z-model can be translated in throughput constraints on the SDF graph, and verified with throughput analysis techniques. Alternatively, the token timestamps of the source and sink actor can be compared to directly verify the end-to-end timing properties of the model.

With SDF analysis, it can be verified that sample times and IO-delay constraints will be met in a future implementation. The next section explains the transformation from z-domain model to SDF model in more detail.

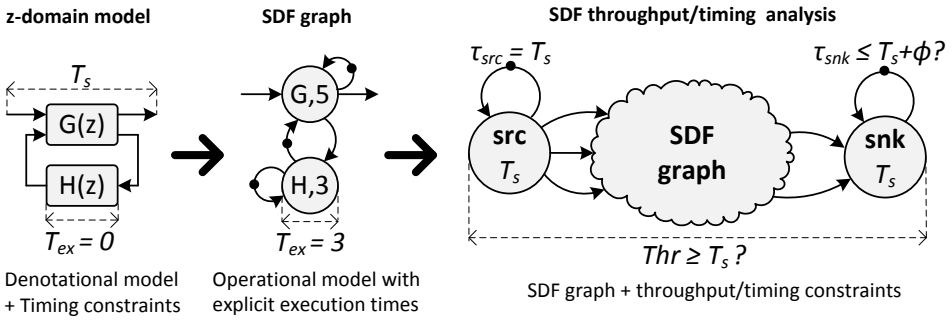


FIGURE 2.7: *Timing verification by SDF analysis on an SDF graph obtained from the z-domain model.*

2.4 Model Transformation

In order to analyze the execution timing of the transfer network model, each transfer block is transformed into one or more SDF actors that model its individual execution. For timing analysis, the computational delay of a transfer function block can be represented by a single actor, since in an implementation, it will be atomically executed as a single block of code. For synthesis purposes, a more fine-grained transformation could be used (see Section 2.5). The computational delay of each block depends on the type of processor it is mapped to, and on a finer scale even to which specific instructions its operations are mapped to on that processor. The specific delay values can be calibrated by profiling data or, on a predictable architecture, derived from program code. To preserve token ordering and prevent auto-concurrency, each SDF actor representing a transfer function block gets a self-edge with an initial token.

Each dependency in the z-domain model is substituted for an equivalent edge in the SDF model that connects the substitutes of its source and sink block. For each transfer function block, the number of initial tokens on the outputs of its SDF model substitute is determined by calculating the number of samples it takes from its input signals before it first starts contributing to its output signals. This can be directly derived from the transfer function of a block. A Single-Input-Single-Output (SISO) discrete transfer function can be expressed as a quotient of polynomials of the following form:

$$H(z) = \frac{p_0 + p_1 z + \cdots + p_u z^u}{q_0 + q_1 z + \cdots + q_v z^v}, \tag{2.2}$$

with $u, v \in \mathbb{N}$ and $u \leq v$. Dividing by the highest order denominator z-term $q_v z^v$ with $q_v \neq 0$ results in a polynomial fraction of the form:

$$H(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_m z^{-m}}{1 + a_1 z^{-1} + \cdots + a_n z^{-n}}, \tag{2.3}$$

with $m = -(u - v)$ and $n = -v$. The denominator of (2.3) has a zeroth-order z-term, so the corresponding difference equation $u[k]+a_1u[k-1]+\cdots+a_nu[k-n] = b_0y[k] + b_1y[k-1] + \cdots + b_my[k-m]$, has a term $u[k]$. Any d-th order term in the numerator corresponds to a term $y[k-d]$ in the difference equation. So the lowest order term in the numerator polynomial with a non-zero coefficient, $d_{min} = \min\{d \mid b_d \neq 0\}$, reflects the least delayed input sample that contributes to the current output $u[k]$. Since the d_{min} first output samples can be calculated without any input having arrived, the output of the corresponding SDF actor will get i_{min} initial tokens.

For a Multiple-Input Multiple-Output (MIMO) block, each input-output pair can be characterized as a SISO transfer function. The transfer matrix for a MIMO block with i inputs and j outputs is given by

$$H_{MIMO}(z) = \begin{pmatrix} H_{11} & H_{12} & \cdots & H_{1i} \\ H_{21} & H_{22} & \cdots & H_{2i} \\ \vdots & \vdots & \vdots & \vdots \\ H_{j1} & H_{j2} & \cdots & H_{ji} \end{pmatrix} \tag{2.4}$$

where H_{vw} denotes the SISO transfer function that specifies the relation between output v and input w.

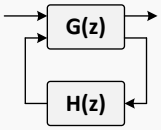
The number of initial tokens at the outputs of a MIMO block can be different for each output. Each block output is now related to a row vector of transfer functions that define its relation to each of the block inputs. The number of samples that the output leads relative to a particular input, corresponds to the lowest order z-term in the corresponding row entry. The number of initial tokens to be placed on an output is then the minimum of the number of samples in the corresponding row of H_{MIMO}.

Note that the placement of initial tokens on edges in the model in this manner allows for timing and schedulability analysis, but is not suited for distributed synthesis. The reason for this is that the tokens represent an ordering relation. In an analysis approach as sketched in this section, the actors only represent time; they abstract from any behaviour. For a synthesis approach, the actors represent behaviour as well, so then the placement of a token in an outgoing edge of an actor means that its actual behaviour is changed.

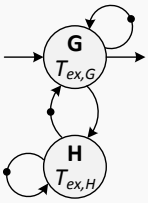
We have shown a simple transformation with which an untimed z-domain model can be transformed to an SDF model that represents its execution timing. As discussed in Section 2.1, digital controllers typically do not have iteration overlap, and we assume them to be single-rate controllers. It is therefore sufficient to analyze only a single iteration of the SDF graph, which then reduces to a periodically re-started Directed Acyclic Graph (DAG). In the subsequent chapters of this work, we will represent the timing of digital controllers by DAGs that follow Homogeneous SDF semantics. Section 5.5 will discuss in more detail how to decorate such an application DAG in order to be able analyze its execution timing when mapped to a particular execution platform.

Example 2.1

Consider the discrete transfer function network in the figure below.



$$G(z) = \begin{pmatrix} 1 & 1 \\ \frac{1.5z^{-1}+0.25}{1+0.5z^{-1}} & 1 \end{pmatrix}$$
$$H(z) = \frac{0.2z^{-2}+0.6z^{-1}}{0.2z^{-1}+1}$$



The two transfer blocks $G(z)$ and $H(z)$ each have an SDF actor that represents their execution time ($T_{ex,G}$ and $T_{ex,H}$), and all dependencies in the transfer network are present in the SDF model as well. Both G and H have a self-edge with initial token to prevent auto-concurrency. The outputs of G have no initial tokens, since the minimum lowest order z-terms of the numerator polynomials in both columns of $G(z)$ is zero. For $H(z)$ the lowest order numerator z-term is 1, so $H(z)$ has a single initial token on its output. The corresponding SDF model is shown on the right of the figure.

Example 2.2

Consider the filter structure at the left of Figure 2.8. Assume its transfer function $H(z)$ is given by (2.3), with $m = n = 2$ and coefficients a_1, a_2, b_0, b_1, b_2 non-zero. To reduce the IO-delay of this filter, it can be split up into two blocks (see Figure 2.3), one block $H_1(z)$ consisting of the grey parts of the figure, and one block $H_2(z)$ consisting of the white parts of the figure. Upon arrival of new input, only H_1 needs to be executed at the moment that new input is available, while $H_2(z)$ can be executed after the time-critical computations of $H_1(z)$ have been completed. The structures of the two separate blocks are shown at the right of Figure 2.8. Their transfer functions are given by:

$$H_1(z) = \begin{pmatrix} b_1 & b_0 & b_0 \\ 0 & 1 & 1 \end{pmatrix} \text{ and } H_2(z) = \begin{pmatrix} b_1 z^{-1} + b_2 z^{-2} \\ -a_1 z^{-1} - a_2 z^{-2} \end{pmatrix} .$$

The corresponding SDF models of the single block filter and its split equivalent are shown in Figure 2.9. For the single-block filter, the lowest non-zero order of z in the denominator of its transfer function is 0, since coefficient b_0 is non-zero, so its output channel gets no initial tokens. For the y output of H_1, the minimum z-order in the first row of $H_1(z)$ is 0. For *out_H1* the minimum order in the second row of H_1 is also 0. So, both outputs of the post-calc block do not get any initial tokens.

The transfer function corresponding to the first output *out1_H2* of H_2 has a 2^{nd} order as well as a 1^{st} order term. Since the lowest order is decisive, the corresponding channel gets a single token. The same holds for *out2_H2*, so its channel will also get a single token.

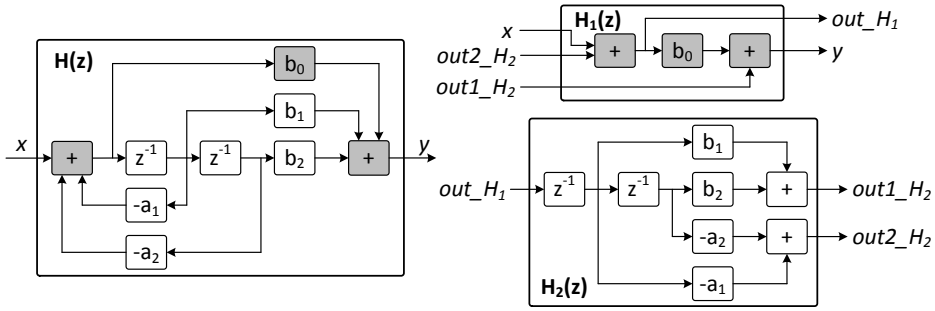


FIGURE 2.8: *Second order filter structure of Ex. 2.2 (left) and its equivalent split into two separate blocks (right) to optimize its IO-delay.*

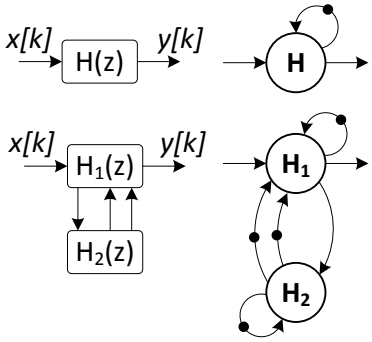


FIGURE 2.9: *SDF models of the filter structures of Figure 2.8.*

2.5 Analysis vs Code Generation

Section 2.4 showed a model transformation with which a discrete transfer network model expressed in z is transformed in an SDF model that can be used for timing analysis of the execution of this transfer network. To this end, a coarse-grained single-actor model suffices to represent a transfer block. With a finer-grained transformation into an SDF model that explicitly models the internal structure of the transfer block, the model can be used for code synthesis and optimization. Instead of the transfer block as a whole, its structural representation is transformed. Figure 2.10 shows an example of such a fine-grained transformation. On the left, it shows the standard Direct Form II representation [21] of the transfer function of Eq. 2.3. The resulting SDF model, shown on the right of the figure, is obtained by instantiating a single actor for each operator block in the structural model, and putting an initial token in each output of a z^{-1} block. Actors connected to an input get a self-edge with an initial token, which is sufficient to

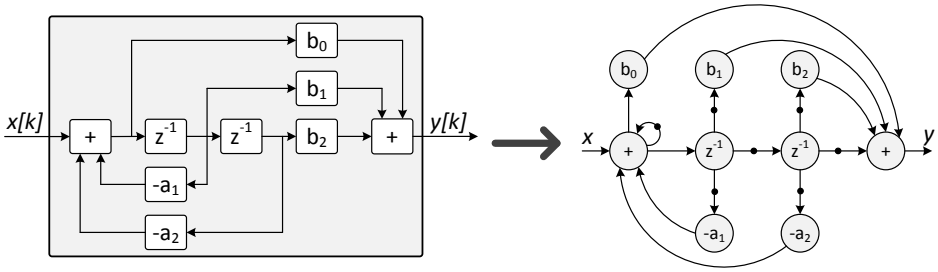


FIGURE 2.10: *Structural representation [21] of the transfer function of Eq. 2.3 (left) and a fine-grained SDF model that could be used for e.g. re-timing or code synthesis.*

prevent auto-concurrency for the complete substitution model.

The resulting graph can be used to perform graph retiming [93], in order to try e.g. to maximize the number of calculations that can be performed independent of the current input sample to reduce IO delay. The same graph can also be used to explore distributed implementations, to e.g. find cuts through the graph that represents the complete z-domain network, in order to find optimal groups of operations to map to a specific computational unit. The graph cuts can then be used as input for the subsequent code generation steps.

2.6 Typical Control Blocks

This section discusses typical control blocks found in mechatronic controllers.

2.6.1 PID-controllers

A PID controller combines a proportional, derivative and integral action together. Given an error input signal $e(t)$, a PID controller computes output signal $u(t)$ by summing its three distinctive terms:

$$\begin{aligned} u(t) &= K_p e(t) + K_i \int_0^t e(\tau)\mathrm{d}\tau + K_d \dot{e}(t) \\ &= K(e(t) + \frac{1}{T_i}\int_0^t e(\tau)\mathrm{d}\tau + T_d \dot{e}(t)), \end{aligned} \tag{2.5}$$

where $K = K_p$, $K_i = K/T_i$ and $K_d = KT_d$.

With Euler's backward method assuming a sampling period T_s, a derivative can be approximated by $\dot{x}(t) \approx \frac{x(t_k) - x(t_{k-1})}{T_s}$. Applying this approximation to the three individual terms of Equation 2.5 and applying the z-transform yields:

$$\frac{U(z)}{E(z)} = K_p \left(1 + \frac{T_s}{T_i}\frac{z}{z-1} + \frac{T_d}{T_s}\frac{z-1}{z}\right) \tag{2.6}$$

Usually, the derivative term is followed by a low-pass filter to make it less noisy. Alternatively, the complete PID output can be followed by a low-pass filter. The integrator output and PID output are usually connected to a saturator (see next section).

2.6.2 Filters

The general transfer function of an n-th order filter is given by Equation 2.2. They are typically constructed by cascading first- and second order filters:

$$H(z) = g\frac{\prod_{m=1}^{M}(1 - z_m z^{-1})}{\prod_{n=1}^{N}(1 - p_n z^{-1})} = g z^{N-M}\frac{\prod_{m=1}^{M}(z - z_m)}{\prod_{n=1}^{N}(z - p_n)} \tag{2.7}$$

2.6.3 Non-linear blocks

Saturation and threshold

Saturation and thresholding are two common non-linear operations. They can be used to prevent signal accumulation (integrator anti-windup), to constrain an input or output. Given an input signal $i(t)$, a low saturation limit L^- and an upper saturation limit L^+, the saturator output $o(t)$ is given by:

$$o(t) = \begin{cases} L^- & \text{if } i(t) \leq L^- \\ L^+ & \text{if } i(t) \geq L^+ \\ i(t) & \text{otherwise} \end{cases} \tag{2.8}$$

With symmetric saturation, the lower saturation limit L^- and the upper saturation limit L^+ are equal. Similarly, given an input signal $i(t)$ and a threshold limit L, the threshold output $o(t)$ is given by:

$$o(t) = \begin{cases} 0 & \text{if } -L \leq i(t) \leq L \\ i(t) & \text{otherwise} \end{cases} \tag{2.9}$$

Thresholding usually only occurs in a symmetric way.

Variable gain

A variable gain is a non-linear operation that changes its gain based on the value of some input signal. Given an input in, parameters L and E, and a function $F(in)$, the output out is given by:

$$out(t) = in(t) + F(in(t)) \cdot in(t) \tag{2.10}$$

For a saturating variable gain, function $F(in(t))$ is defined as:

$$F(in(t)) = \begin{cases} E(t) & \text{if } |in(t)| \leq L \\ \frac{E(t) \cdot L}{|in(t)|} & \text{otherwise} \end{cases} \tag{2.11}$$

A dead-zone variable gain is realized by:

$$F(in) = \begin{cases} 0 & \text{if } |in| \leq L \\ -\frac{E(t) \cdot L}{|in(t)|} + E(t) & \text{otherwise} \end{cases} \tag{2.12}$$

Non-linear compensations

Typically, non-linear dynamics due to e.g. electromagnetic or centrifugal forces are linearized by bilinear interpolation from a lookup table.

2.6.4 State-space representations

The behaviour of a dynamic system can be expressed by a set of difference equations. In a state-space representation, these equations are ordered as a set of first-order difference equations in the vector-valued state of the system. Given state vector $X = [x_0, x_1, \ldots x_{n-1}]$, excitations by input vector $U = [u_0, u_1, \ldots u_{m-1}]$ result in a change of the state vector and the output vector $Y = [y_0, y_1, \ldots y_{p-1}]$ of the system. This is expressed in the following matrix-vector equations:

$$\begin{aligned} X[k+1] &= \mathbf{A}X[k] + \mathbf{B}U[k] \\ Y[k] &= \mathbf{C}X[k] + \mathbf{D}U[k] \end{aligned} \tag{2.13}$$

For an n-th order system, state vector X has n elements. The change of the state vector due to input excitation and the current value of the state is characterized by the $n \times n$ system matrix $\mathbf{A}$ and the $n \times m$ input matrix $\mathbf{B}$. Correspondingly, the system output response to state and input is characterized by the $p \times n$ output matrix $\mathbf{C}$ and the $p \times m$ direct transmission matrix $\mathbf{D}$.

The matrices in a state-space system are sparse. Depending on the particular canonical form used, the matrices have a different structure of non-zero elements [21]. Any system expressed by a set of transfer functions can be characterized by a state-space representation as well. Design techniques operating directly on the state-space representation are generally applicable, but are particularly useful for MIMO systems.

2.6.5 δ-domain representation

For systems with high sample rates, the z-operator suffers from numerical issues due to the finite word length of variables [57]. To alleviate such inaccuracies and round-off errors, the δ-operator can be used instead of the z-operator. The delta-operator is defined as:

$$\delta = \frac{z-1}{T_s}, \tag{2.14}$$

with T_s the sampling time.

2.7 Summary

This chapter discussed the main application characteristics of high-performance digital control applications. We explained the typical structure of a digital control system, en discuss the relation between control performance and application execution timing.

Digital controllers are typically modeled in the z-domain. This is an untimed declarative specification, which is not directly suited for performance analysis. In this chapter, we showed how such specifications can be transformed into a dataflow model with timed operational semantics. Such models can be used in performance analysis, and can be synthesized into an implementation that meets the worst-case performance found with this analysis, as shown in subsequent chapters.

Furthermore, this chapter discussed common operations found in digital control applications. The execution platform presented in Chapter 3 is tuned towards efficient execution of these operations.

Chapter 3

Heterogeneous Multi-ASIP Platform

This chapter introduces an FPGA-based heterogeneous multiprocessor platform template targeting digital control applications. Section 3.1 motivates the choice of the top-level architecture and implementation technology for the platform template. An overview of the template is discussed in Section 3.2, followed by a detailed explanation of its architectural elements in Section 3.3. Section 3.4 discusses related work, and the chapter concludes with an outlook and a summary in ections 3.5 and 3.6. The case study presented in Chapter 6 maps the controller application discussed in Section 1.4 to instances of the platform template discussed in this chapter.

3.1 Motivation

General purpose processors (GPPs) are in widespread use as execution platform for digital control applications, as they provide abundant performance, flexibility and programmability. With processor frequency scaling reaching its limits several years ago, application performance demand is rapidly catching up with the offered GPP performance, even to such an extent that GPP platforms will not be able to provide sufficient performance for next generations of digital control applications [82].

GPPs are optimized for average performance of generic applications, at the cost of tremendous complexity, unpredictability and inefficiency. Due to their generic instruction sets, GPPs are unable to closely match the heterogeneity found in digital control applications. Their architectures contain optimizations like hierarchical memories, branch prediction and out-of-order execution, which greatly improve average processor performance, but not worst-case performance, which is needed in real-time applications. Branch mispredictions and cache misses cause

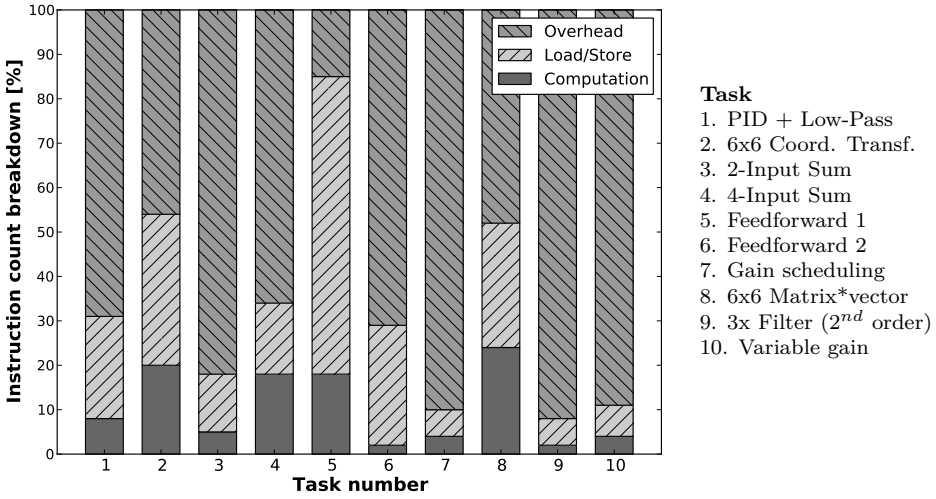


FIGURE 3.1: *Instruction count breakdown for GPP execution of several control blocks.*

jitter, which actually has an adverse effect on control performance [12].

Figure 3.1 exemplifies some of this GPP execution overhead by showing an instruction count breakdown of several digital control blocks from an industrial design library. The C-implementation of these blocks has been assembled into PowerPC assembly, and the instructions have been labeled according to their contribution to the actual functionality. Floating point instructions required to realize the core functionality of the blocks are labeled *computation*. Load and store instructions that load or store variables are labeled *load/store*. Loads and stores that are part of a function call, integer arithmetic for address calculation and other instructions are labeled *overhead*. On average, only 11% of the instructions contribute to the actual floating point calculations in the considered blocks.

With specialized architectures, the amount of data movement overhead can be greatly reduced, and calculations can be performed more efficiently with extensive parallelization and a close match between hardware and algorithm. However, the reduced programmability due to specialization impairs flexibility, since these architectures are less able to deal with changing or expanding application functionality. Figure 3.2 shows a performance-programmability trade-off of the GPP architecture and several other architectures. Specialized architectures, like Application-Specific Integrated Circuits (ASICs), can easily improve the execution platform performance by two orders of magnitude [31]. However, the superior performance offered by ASICs comes at the cost of programmability, as they are specifically tuned towards a narrow range of applications. Less specialized architectures like Application-Specific Instruction set Processors (ASIPs), offer a good

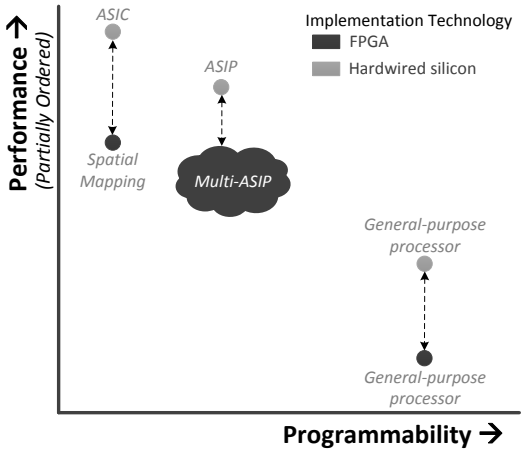


FIGURE 3.2: *Programmability vs performance trade-off.*

trade-off between performance and programmability [41]. In contrast to ASICs, ASIPs have an instruction set tuned towards a *class* of applications, giving them the flexibility to better deal with changing or expanded application functionality.

Both ASICs and ASIPs are typically implemented in hardwired silicon. The associated high manufacturing costs render designs in this technology only feasible for high volume markets. Due to their architecture specialization, ASICs, and to a lesser extent ASIPs, often target application domains that are too narrow to be cost-effective using hardwired silicon technology. Also, the design and implementation process in hardwired silicon takes months, and includes significant corporate risk. Therefore, reconfigurable hardware such as FPGA technology, is an interesting alternative, despite showing about an order of magnitude lower performance compared to its hardwired counterpart [45, 87].

A common FPGA approach taken by e.g. MathWorks [53] is to implement the application directly in FPGA hardware, comparable to an ASIC design mapped to reconfigurable FPGA fabric. As this basically realizes a spatial mapping in FPGA with a limited extent of time-multiplexing (as opposed to a time-multiplexed mapping with limited spatial-multiplexing as used in GPPs), such an approach lacks scalability and flexibility, since small changes in the application will require a complete re-run of the time-consuming FPGA mapping flow.

A programmable ASIP architecture in FPGA shows better resource re-use, and allows decoupling of the compilation flow from the FPGA mapping flow. This offers a more scalable and flexible solution compared to direct spatial mappings, at the cost of performance. An FPGA-based multiprocessor platform consisting of multiple different ASIPs can match both heterogeneity and task-level parallelism present in digital control applications, and still allows the platform the be tuned

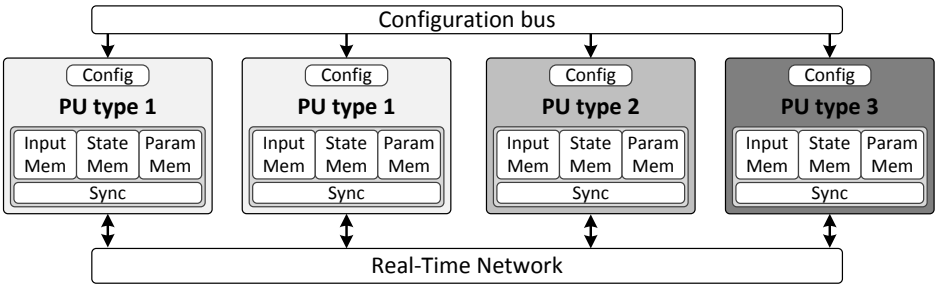


FIGURE 3.3: *Execution platform template.*

towards the target application at design time. Since a multi-ASIP in FPGA approach seems to offer a good balance between flexibility, programmability and scalable performance, we have chosen this architectural concept as the basis of the execution platform presented in this chapter.

To reduce the complexity of composing a tailored platform and to facilitate design-reuse, we employ the platform-based design paradigm [69] to our hardware and software architecture. In the platform based design paradigm, the platform (HW + SW) is presented to a designer at a higher level of abstraction. A platform consist of a set of abstract component types that can be instantiated and integrated into a larger component or system. Different platform designs comprise different compositions of these components.

To facilitate platform-based design, this chapter presents a platform template that consists of processing units (PUs) connected together in a systematic way. The internals of the PUs are mostly fixed, except for a tuning space that allows to select different memory and datapath dimensions, and some limited instruction set specializations. PUs are connected together by a network with a uniform communication protocol. The PUs enforce a dataflow model of computation onto the platform, which enables also the application to be presented in a similar abstract way. This chapter discusses the internal structure of our platform template in detail. The different HW and SW abstractions presented to a designer are discussed in Chapter 5.

3.2 Platform Template

The platform template presented in this chapter aims to provide improved real-time performance by employing architecture specializations that reduce the huge overhead associated with GPP execution, and by enabling instantiation of tailored platforms consisting of a composition of differently specialized cores. Figure 3.3 shows the heterogeneous multi-ASIP platform template. It consists of a set of different parameterizable ASIP processing units (PUs), connected by a packet-

switched network composed of one or more parameterizable switches. Different ASIP types have different architecture specializations, that allow them to e.g. exploit data-level parallelism, or provide direct hardware support for specific operations. The composition of several of these PUs into a multiprocessor allows the platform to exploit task-level parallelism as well. The platform template targets digital control applications, and offers a good balance between flexibility, programmability and scalable performance by means of the following features:

- **Tuned ASIP processing units.** The internal structure of each PU type is closely matched to that of typical Simulink blocks. The datapath of different PU types is tuned towards different families of digital control blocks. This better match between application and platform results in better worst-case performance and jitter reduction.
- **Dataflow task synchronization.** Tasks mapped to PUs synchronize on data availability according to the Homogeneous Synchronous Dataflow (HSDF) model of computation. This enforces a clear execution semantics, and since the graph is application can be modeled by a Directed Acyclic Graph without iteration overlap (see the application model discussed in Chapter 2), it is also well analyzable (see Chapter 4).
- **Low-latency interconnect.** Since we aim at a distributed approach, where control applications are mapped to a set of PUs, it is crucial to have a good mechanism for fast inter-PU communication. To this end, our platform employs a low-latency packet-switched network with static routing to connect different PUs together in any structural composition.
- **FPGA-based template.** Mapping to FPGA technology allows design-time reconfiguration. By defining a platform template, parameterizable PUs can be instantiated from a set of design library elements and connected together into a platform instance. In this way, the execution platform can be composed in a reconfigurable way, similar to how digital control applications are composed in Simulink. This enables a close tailoring of application and platform.

The use of programmable cores in FPGA ensures that changes in application can be handled in software to a large extent, while keeping the hardware unchanged. This is particularly useful in case time-consuming certification is required upon hardware changes. Only major application changes require instantiation of extra or different types of cores. The common dataflow interface shared among PUs allows easy integration of new ASIP types into the design library.

With the different specialized processing units and the composable switched interconnect in the template, platform instances can be constructed that are able to closely match the structure, granularity and heterogeneity of a wide range of different control applications. The following subsections present the different elements that constitute the template in more detail. First, the the generic

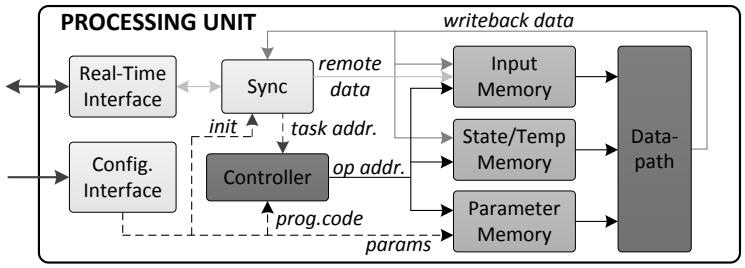


FIGURE 3.4: *Common internal structure of the processing units in our template.*

processing unit structure is explained in Section 3.2.1, followed by the synchronization mechanism in Section 3.2.2 and the interconnect that is used to connect PUs together in Section 3.2.3. Finally, the architecture of the different PUs in our current design library are discussed in Section 3.3.

3.2.1 Processing Unit Structure

The primary entities mapped to the PUs in the platform template, called *tasks*, have a one-to-one correspondence with Simulink blocks (like PID blocks, filters, etc.). To enable fast execution of tasks, the internal PU structure matches that of a Simulink block.

Figure 3.4 shows the general structure of the PUs that are part of the platform template. To reduce data movement overhead, the PUs have a flat distributed memory layout consisting of three separate memories. Each PU has a dedicated input memory connected to a real-time interconnect interface, a parameter memory which can be updated independently, and a state/temporary memory for the internal variables that are used by a block. Note that these are separate *logical* memories; to reduce resource requirements, in a physical implementation, any of these logical memories can be mapped together to a single physical memory (e.g. map the state and temporaries memories to the same physical memory).

The three separate memories decouple communication from computation, i.e., an ASIP can perform computations on data stored in its parameter and state memory while new input data is received (provided that the physical memory mapping provides sufficient ports to facilitate this concurrency). In addition, they provide the processor with a large memory bandwidth enabling low-latency data access. Both aspects enable a fast execution of tasks which is key for meeting the timing constraints. The PUs use direct addressing, i.e. instructions contain the complete operand addresses. This complicates processor runtime-reconfiguration, and the wider instruction results in a larger memory footprint, however, direct addressing avoids lots of address calculation overhead, in time and in resources, since the address calculations are performed at compile-time by the code generator

(see Section 5.4).

The controller and datapath are unique for each ASIP type, and specialized for execution of a family of control blocks. In principle, a PU can have a datapath of any data width, and can be fixed-point, integer, floating point, or mixed. However, in our current set of processing units, we only employ single-precision floating point datapaths, since this provides sufficient accuracy and dynamic range for a wide range of applications. The different PU architectures currently available as instantiable element in the template are explained in Section 3.3.

3.2.2 Processing Unit Synchronization

To ensure the easy integration of heterogeneous units into the platform and to keep a clear distributed execution semantics, PUs need to follow a consistent task synchronization semantics. Kahn Process Networks (KPNs) are widely used to model streaming multi-media and digital signal processing applications. In the KPN model of computation, tasks synchronize on the availability of input data arriving through one or more communication channels. However, since actors produce and consume a run-time decided number of tokens, the execution of KPNs requires run-time buffer management and scheduling [26], which can be costly to implement in software as well as in hardware. In the Synchronous Dataflow (SDF, [48]) model of computation, tasks also synchronize on input data availability, but in SDF tasks consume a fixed compile-time known number of tokens. At the cost of expressiveness, this restriction makes SDF an analytically tractable model of computation, and its semantics are efficiently implementable in either hardware or software.

Chapter 2 showed that the typical digital control applications can be modeled as SDF graphs, and thus do not need the more expressive KPN model of computation. Therefore, SDF is chosen as the model of computation to be enforced onto the architecture template and the applications running on it. Since digital control applications are typically not pipelined, we can assume unpipelined homogeneous (all channels produce and/or consume the same number of data tokens) SDF, implying that starting a task always consumes *all* available input tokens. This significantly simplifies the implementation of the synchronization mechanism, as a simple count of the total number of available input tokens for each task suffices. Since a software implementation will inevitably result in considerable overhead when tasks are small, we have chosen to implement synchronization in hardware.

Figure 3.5 shows the synchronization unit. Data from the RT network is passed through the Real-Time (RT) network interface to the synchronization unit. The data format received from the NI is shown on the top left of the figure. Besides the data payload, it contains addressing fields that relate the data to the task it belongs to (*tID*) and which specific input of that task is targeted (*iID*). Each task has a *tID* that is unique on the PU it is mapped to. The *tID* and *iID* fields are used as write address to store the payload data in the PU input memory.

The *tID* field is also used to index a lookup memory to lookup execution

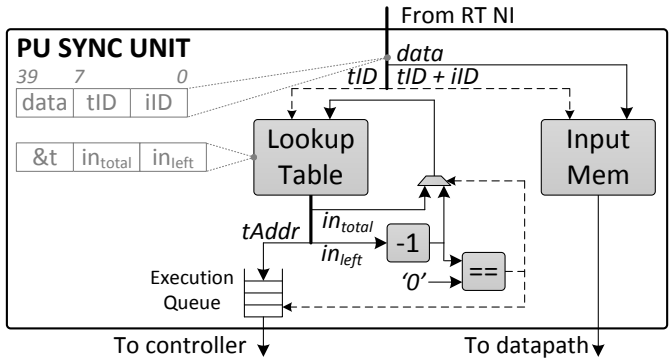


FIGURE 3.5: *Hardware synchronization mechanism enforcing HSDF semantics onto tasks running on a PU.*

precondition information (middle left of Figure 3.5) for the corresponding task, consisting of the address of the first instruction of the target task (*tAddr*), its number of inputs (in_{total}), and the number of inputs it still needs before being eligible for execution (in_{left}). Task input data arrives either externally from the interconnect or internally from a write-back bus. When new input arrives, in_{left} is decremented for the corresponding task. If in_{left} reaches zero, the task's program counter *tAddr* is sent to the *Execution Queue*, and in_{left} is reset to in_{total}. This implements a first-come-first-serve task scheduler. A static order scheduler can be implemented with a minor adjustment; the lookup memory then contains an ordered sequence of task program counters, with a valid bit for each entry. When a task has all its inputs available, its valid bit is set; program execution blocks on a missing valid bit. The size of the *Execution Queue* is dimensioned to the worst-case number of tasks mapped to the corresponding PU.

3.2.3 Interconnect

PUs send and receive data through their real-time interface, which connects to a low-latency packet-switched network. This network can be used to realize any hierarchical composition of PUs, allowing a designer to exploit the specific communication patterns of the application. The routing tables of each switch are configurable at design-time and are fixed at run-time. In this way, different parts of the network can be kept separated, to minimize contention. Each switch can be configured to use a First-Come-First-Served (FCFS) scheduling policy, or a Fixed-Priority scheduling policy where the priorities are based on the switch port number. Currently, our analysis method only supports FCFS-scheduling.

To minimize communication overhead, most PUs have support for implicit data sending in their instruction set. Instead of executing an explicit communication instruction, a PU can select a remote target in its normal (arithmetic)

instructions by setting specific bits in the output operand address. These bits result in the creation of a network packet, whose data payload is filled with the output operand. The network ID of the receiving PU, and the task ID and input ID of the receiving task, which are encoded in the instruction, are set as addressing fields of the packet. The PU then pushes the data packet into a switch input buffer, where it is further handled by the switch hardware.

A separate configuration interface handles the uploading of parameters and initialization of PUs. Since this interface uses only uni-directional traffic, from a single host acting as master to the PUs acting as slaves, this interface is implemented as a memory mapped bus. At run-time, this interface can be used to change parameter values without affecting the real-time inter-PU communication.

3.3 Processing Units

The PU design library that is part of the platform template currently consists of three different ASIP types, whose instruction sets match with common operations found in control tasks (see Chapter 2).

3.3.1 Scalar Processing Unit

The Scalar Processing Unit (SPU) is a basic signal processing unit intended for tasks that have a mostly sequential structure, and (non-linear) tasks that require comparison-based operations.

To facilitate efficient saturation operations, the SPU has a *clipsym* instruction that performs symmetric saturation with a single instruction that loads operands, performs the operation and stores back the result or sends it to a remote destination PU. The instruction has two source operands A and B, with A acting as input and B acting as comparison limit. The result of the instruction is $Y = A$ if $|A| < B$, or $Y = sign(A) \cdot B$ if $|A| \leq B$.

Example 3.1 shows different implementations of asymmetric clipping targeting a GPP and the SPU.

Example 3.1

Asymmetric saturation is typically implemented by the following C-code:

```
void asym_sat ( float low_limit, float high_limit, float *const
    signal_ptr)
{
    if UNLIKELY((*signal_ptr) > high_limit){
       *signal_ptr = high_limit;
    }
    else if UNLIKELY((*signal_ptr) < low_limit){
       *signal_ptr = low_limit;
    }
}
```

The corresponding PowerPC assembly code is given below. The actual instructions required to perform saturation are shown in boldface, the other instructions can be considered overhead.

```
// Setup and clean up stack frame
01 stwu  r1,-48(r1)   05 stfs  f2,28(r31)   30 lwz r11,0(r1)
02 stw   r31,44(r1)   06 stw   r3,32(r31)   31 lwz r31,-4(r11)
03 mr    r31,r1                             32 mr  r1,r11
04 stfs  f1,24(r31)                         33 blr
// Saturation function body
07 lwz   r9,32(r31)   15 stfs  f0,0(r9)     23 lwz   r9,32(r31)
08 lfs   f13,0(r9)    16 b     30           24 lfs   f0,24(r31)
09 lfs   f0,28(r31)   17 lwz   r9,32(r31)   25 stfs  f0,0(r9)
10 fcmpu cr7,f13,f0   18 lfs   f13,0(r9)    26 b     30
11 bgt-  cr7,13       19 lfs   f0,24(r31)   27 lwz   r9,36(r31)
12 b     17           20 fcmpu cr7,f13,f0   28 li    r0,0
13 lwz   r9,32(r31)   21 blt-  cr7,23       29 stw   r0,0(r9)
14 lfs   f0,28(r31)   22 b     27
```

The SPU can execute such asymmetric saturation in two instructions:

```
00 0 load low_limit
01 0 clip_asym signal_in high_limit signal_out
```

The first instruction loads the lower clipping limit to a special-purpose register. The second instruction performs the actual clipping by comparing the input to the lower and higher limit concurrently.

Asymmetric clipping operates on two different limits. Due to the required third source operand, asymmetric clipping requires two instructions, since one source operand has to be moved to a special purpose register. To this end, the lower saturation limit operand must be moved to this special purpose register with a *load* instruction prior to executing a *clipasym* instruction. Similarly, the SPU can perform symmetric thresholding by executing a single *thressym* instruction. The variable gain discussed in section 2.6 can also be implemented using the *clipsym* instruction, as shown in Example 3.2.

Example 3.2
The dead-zone variable gain operation is defined as:

$$out = \begin{cases} in & \text{if } |in| \leq L \\ in + E \cdot in - \frac{E \cdot L \cdot in}{|in|} & \text{otherwise} \end{cases}, \text{ where } \frac{in}{|in|} = sign(in).$$

On the SPU, this operation can be implemented using the *clipsym* instruction. The following SPU pseudo-assembly implements the dead-zone variable gain.

```
00 t0 = clipsym(in, L)
01 t0 = sub(in, t0)
02 t0 = mult(t0, E)
03 out = add(t0, in)
```

First, *clipsym* realizes $t0 = \begin{cases} in & \text{if } |in| \leq L \\ sign(in) \cdot L & \text{otherwise} \end{cases}$. Subsequently subtracting this term from *in*, results in $t0 = \begin{cases} 0 & \text{if } |in| \leq L \\ in - sign(in) \cdot L & \text{otherwise} \end{cases}$. By multiplying with E and finally adding *in*, the dead-zone variable gain is realized. A saturating variable gain is implemented by omitting the subtraction.

Consider the saturation and thresholding operations discussed in Section 2.6. Typically, on a GPP platform, these operations are implemented using *if-then-else* expressions. These expressions not only suffer branching overhead in terms of potential pipeline stall cycles, but also the number of (sequential) GPP operations required to execute the simple saturation operation is already quite large. Example 3.1 shows a typical C-implementation of a saturation function and its corresponding PowerPC assembly code. The number of instructions to execute this function will vary between 19 and 24, depending on the branch taken. The saturation operation itself requires two comparisons and four branches, the other instructions can be considered overhead.

The complete instruction set of the SPU is summarized in Table 3.1. Besides the special-purpose instructions that implement symmetric and asymmetric saturation and thresholding, it supports the common *add*, *sub* and *mult* instructions. The *multinv* instruction is used in coordinate transformations where some specific inputs are negated. The hardware overhead of this instruction is minimal, since negation in floating-point representation is performed by simply inverting the sign bit. The single-instruction multiplication and negation prevents the necessity to implement such operations with two dependent instructions.

The *copy* instruction provides efficient multi-destination communication. To minimize communication overhead, external communication is explicitly encoded in the operand addresses (see Section 3.2.3). Specific bits in a destination operand address trigger the network interface unit to send the data to the network. When

TABLE 3.1: *SPU instruction set.*

Instruction	Operation
clear Y	$Y = 0$
copy Y	$Y = PREV$
add A B Y	$Y = A + B$
sub A B Y	$Y = A - B$
mult A B Y	$Y = A \cdot B$
multinv A B Y	$Y = A \cdot -B$
load A	$L = A$
clipsym A B Y	$Y = A$ if $\|A\| \leq B$
	$Y = sign(in) \cdot B$ otherwise
clipasym A B Y	$Y = A$ if A < B and if A > L
	$Y = B$ if A > B
	$Y = L$ if A < L
thressym A B Y	$Y = 0$ if $\|A\| < B$
	$Y = A$ otherwise
thresasym A B Y	$Y = 0$ if $L < A < B$
	$Y = A$ otherwise

tasks need to send the same data to different destinations, the *copy* instruction sends the previous ALU output, stored in special purpose register $PREV$, to a different destination specified in the Y operand.

The micro-architecture that implements this instruction set is shown in Figure 3.6. The local distributed memories (the input memory is considered part of the real-time network interface here), configuration interface and real-time network interface that all processing units have in common are shown on the left. The SPU-specific datapath is shown on the right. It comprises an 8-stage pipelined single-precision floating point unit with an adder, a multiplier and two comparison units, and dedicated special-purpose registers L and $PREV$. The floating-point functional units are generated as FPGA vendor-specific IP cores that are configured with deep pipelines in order to support high frequencies (targeting 200 MHz). The register labeled FW in the middle of the figure (connected to the data buses) is used for forwarding when an instruction uses the output operand of a previous instruction as one of its input operands.

Figure 3.7 shows the 64-bit SPU instruction format. SPU instructions have three 16-bit operand addresses, followed by three 4-bit instruction fields *stall*, *opcode* and *cid*. The operand addresses consist of a 2-bit selection to select which of the local distributed memories to target, and a 14-bit memory address. The Y-operand address can target a location on a different PU. Since the selection bits and remote address do not fit in the 16-bit $addrY$ field, the *cid* field is used to store the first 4 bits of the remote address when a remote PU is targeted.

The SPU has a pipelined floating point datapath, whose intermediate results cannot be forwarded because they are incompatible in between pipeline stages (only in the last stage the results are combined into a valid floating point number). Therefore, hazard detection is required to stall the pipeline in case of data hazards.

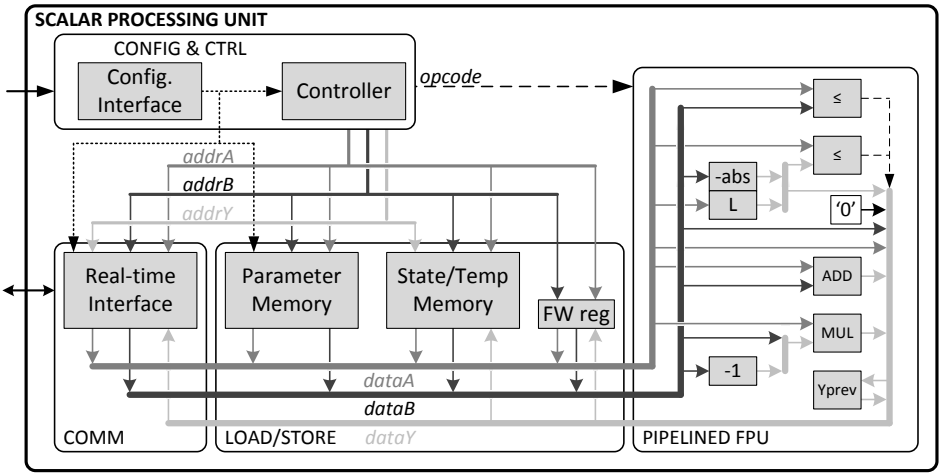


FIGURE 3.6: *Micro-architecture of the Scalar Processing Unit.*

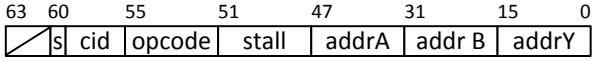


FIGURE 3.7: *SPU instruction format.*

Since the SPU is implemented in FPGA-technology, the floating point pipeline is typically 5 (multiply) to 8 (add) stages deep for a 200Mhz target design [3]. For designs targeting higher frequencies, the pipeline depth can even go up to 14 for a floating point adder. This considerable pipeline depth makes a hardware hazard detection unit very expensive, so a software solution has been implemented. To this end, the SPU pipeline can be stalled for up-to 15 clock cycles by setting the 4-bit *stall* instruction field.

The shallow, directly addressed distributed memory structure, specialized instructions and explicit communication of the SPU enable it to efficiently execute common sequential blocks like PID controllers and sum blocks, as well as comparison-based non-linear blocks like saturation and input-dependent gain. For example, the execution of the saturation function of Example 3.1 is realized by executing a *loadasym* followed by a *clipasym* instruction, which load, compare, set and send the data. Similarly, the SPU executes a two-input sum block and sends its output with just a single *add* instruction, while otherwise it would require 17 GPP instructions to realize the same functionality.

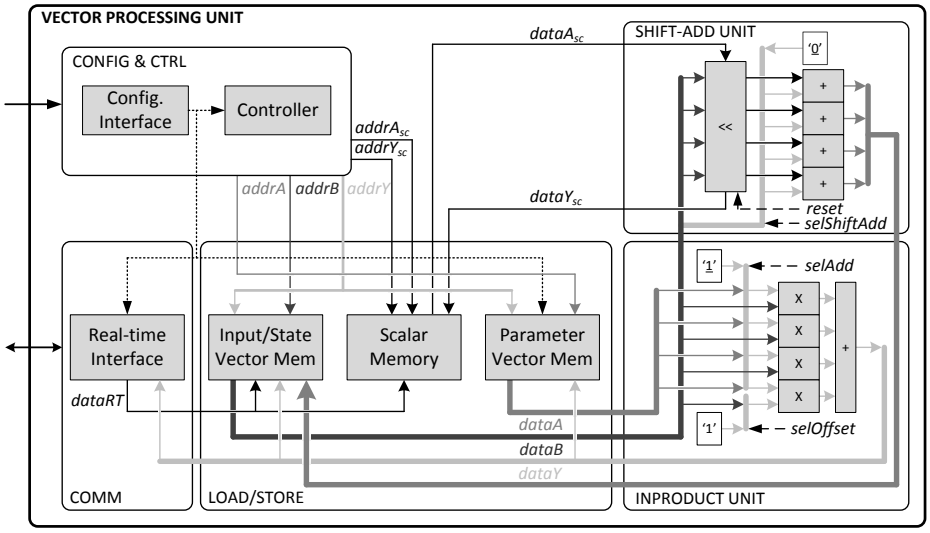


FIGURE 3.8: *Vector Processing Unit block diagram.*

3.3.2 Vector Processing Unit

The VPU is a basic signal processing unit for state-space calculations, filter state updates and coordinate transformations (see Section 2.6). There is ample data-level parallelism in these kinds of operations, which can be exploited by providing wide vector instructions.

Figure 3.8 shows the micro-architecture of a VPU with 4-way vector support ($vpuWidth = 4$). The left side of the figure shows the real-time network interface, configuration interface and synchronization unit (with input memory), which are common for all processing units. The distributed memories of the VPU are implemented as vector memories, as wide as the VPU issue width. A read access on these memories reads a whole vector of $vpuWidth$ elements. A write access can either write a whole vector, or a single element on a specific location in a vector. The VPU also has a dedicated scalar memory, which is used for e.g. shifting new data samples into or out of a vector that implements a delay line.

The VPU-specific datapath, shown in the right of the figure, consists of a vector add-shift unit and an in-product unit. The in-product unit utilizes an optimized adder tree using folding, where the floating point is only aligned once for the whole vector, and the rounding is only done at the end of the tree.

Table 3.2 shows the instruction set realized by the VPU implementation. The *add*, *inprod* and *inprodoffset* instructions use the in-product unit. The *inprod* calculates the in-product of two vectors. The *sum* instruction adds the elements of a vector by calculating the in-product of the A-operand and a vector with

TABLE 3.2: *Vector Processing Unit instruction set.*

Instruction	Operation
r sum B Y	$Y = \sum_{i=0}^{vpuWidth-1} b_i$
r inprod A B Y	$Y = \mathbf{A} \cdot \mathbf{B}$
r inprodoffset A B Y	$Y = \sum_{i=0}^{vpuWidth-1} a_i \cdot b_i$, with $b_{vpuWidth-1} = 1$
r shift A_{sc} B Y_{sc}	$\mathbf{B} = shift(A_{sc} \rightarrow \mathbf{B} \rightarrow Y_{sc})$ (1)
r shiftadd A_{sc} B Y_{sc}	$\mathbf{B} = \mathbf{B} + shift(A_{sc} \rightarrow \mathbf{B} \rightarrow Y_{sc})$
r clear B	$\mathbf{B} = \mathbf{0}$
r loop $incA$ $incB$ $incY$	Execute next instr. $r + 1$ times, while updating operand addr.: $\&A = \&A + incA$ $\&B = \&B + incB$ $\&Y = \&Y + incY$

(1) Subscript sc denotes an operand from scalar memory.

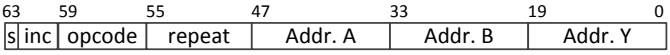


FIGURE 3.9: *Vector Processing Unit instruction format.*

all entries set to 1. The *inprodoffset* instruction is another variant of the *inprod* instruction that calculates the in-product of a *(vpuWidth - 1)*-wide vector, and adds an offset. To this end, two vectors A and B are loaded, where the last element of A contains the offset to be added. By setting the last element of B to '1', the in-product with offset is realized.

Two vector shift operations are supported, one for the z-domain and one for the δ-domain [57]. The *shift* instruction can be used to implement a delay line, i.e. a cascade of multiple z^{-1} operations. It shifts the elements in a vector one position from lower index to higher index. On the lower index side, a scalar value A_{sc} from the scalar memory can be shifted in. Similarly, the vector element with the highest index is shifted out, and can be stored in the scalar memory. The *shiftadd* operation, adds a vector to its shifted version. This implements a cascade of multiple δ^{-1} operations.

The *loop* instruction executes the next instruction $r + 1$ times and increments the element indices of the source and destination operands of that instruction independently based on the *incA*, *incB* and *incC* operands (which are interpreted as immediate). This provides hardware loop support without branching and exit testing overhead, similar to zero-overhead looping techniques seen in DSP-processors [19].

The VPU instruction format is shown in Figure 3.9. The 14-bit A and B source operand addresses target a whole vector, the 20-bit Y destination operand address targets a specific element in a vector. The Y-operand is a concatenated vector address and element index. The *opcode* and *s* fields, like in the SPU instructions, specify which operation to execute and indicate that the instruction is the last of

the task (see Section 3.3.1).

An 8-bit *repeat* instruction field and three *inc* bits form an explicit second level of hardware looping support which can be used in conjunction with the explicit *loop*-instruction. The *repeat* field specifies the number of times the current instruction is to be repeated (corresponding to the r in front of the operations in Table 3.2). During repetition of the instruction, the operand addresses A, B and C can be incremented by 1 by setting the corresponding bit in the *inc* field. For Y this means that the next *element* is addressed, and for A and B this means that the next *vector* of *vpuWidth* elements is loaded. The combination of an explicit *repeat* instruction field with fixed-size element increments and an explicit *loop* instruction with variable element increments provides a powerful two-level hardware loop support mechanism, enabling fast iteration through vectors and matrices. This prevents extensive software looping overhead and is much more memory-efficient compared to e.g. loop unrolling.

Typical tasks mapped to the VPU are tasks with data-level parallelism, like the multiplication of an $M \times N$ matrix with an $N \times 1$ vector ($MAT_{M \times N}$). Example 3.3 compares the execution of $MAT_{6 \times 6}$ on a PowerPC GPP and a VPU.

Example 3.3

The PowerPC assembly code below implements matrix-vector multiplication (that is part of e.g. the state-space calculations of Section 2.6) for an $M \times N$ matrix and an $N \times 1$ vector with $M = N = 6$.

PowerPC assembly (Single row-vector in-product)

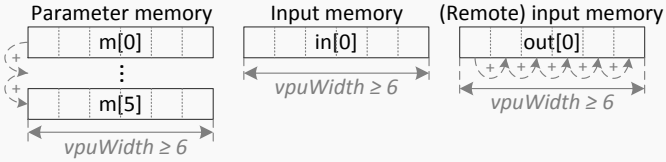
```
01 lwz   r9,24(r31) 14 lwz   r9,24(r31) 28 lwz   r9,24(r31)
02 lwz   r9,12(r9)  15 lwz   r9,20(r9)  29 lwz   r9,28(r9)
03 lfs   f13,0(r9)  16 lfs   f13,0(r9)  30 lfs   f13,0(r9)
04 lwz   r9,8(r31)  17 lwz   r9,8(r31)  31 lwz   r9,8(r31)
05 lfs   f0,4(r9)   18 lfs   f0,12(r9)  32 lfs   f0,20(r9)
06 fmuls f12,f13,f0 19 fmuls f0,f13,f0  33 fmuls f0,f13,f0
07 lwz   r9,24(r31) 20 fadds f12,f12,f0 34 fadds f12,f12,f0
08 lwz   r9,16(r9)  21 lwz   r9,24(r31) 35 lwz   r9,24(r31)
09 lfs   f13,0(r9)  22 lwz   r9,24(r9)  36 lwz   r9,32(r9)
10 lwz   r9,8(r31)  23 lfs   f13,0(r9)  37 lfs   f13,0(r9)
11 lfs   f0,8(r9)   24 lwz   r9,8(r31)  38 lwz   r9,8(r31)
12 fmuls f0,f13,f0  25 lfs   f0,16(r9)  39 lfs   f0,24(r9)
13 fadds f12,f12,f0 26 fmuls f0,f13,f0  40 fmuls f0,f13,f0
                    27 fmuls f0,f13,f0  41 fadds f0,f12,f0
                                        42 lwz   r9,24(r31)
                                        43 stfs  f0,36(r9)
```

Like the PowerPC clipping implementation in Example 3.1, the actual number of floating-point arithmetic instructions is only a fraction of the total number of instructions. Typical GPP architectures can only process a few floating-point operations concurrently.

VPU assembly and memory layout

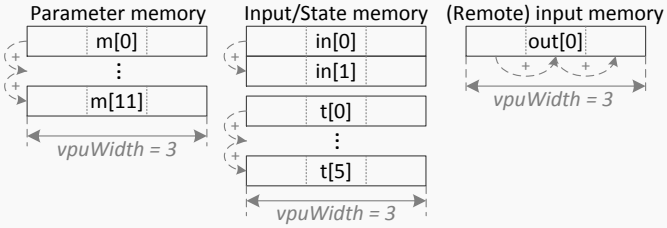
The vector-ALU and adder-tree of the VPU provide acceleration for these in-product operations. A VPU of at least 6 elements wide can perform this operation with a single instruction:

```
5 inprod m+ in out+
```



When executing an $M \times N$ matrix-vector multiplication with a VPU that is less than N elements wide, additional temporary storage and *sum* instructions are required to sum the intermediate partial in-products:

```
00 5 loop 2 0 1
01 1 inprod m[0]+ in[0]+ t[0]+
02 1 loop 0 3 1
03 2 sum t[0]+ out[0]+
```



In general, execution of a $MAT_{M \times N}$ block on a GPP with completely unrolled loops requires $7(M \cdot N) + M$ instructions (excluding stack setup and cleanup instructions); $(2N-1) \cdot M$ *mult* and *sum* instructions, and $5(N+2) \cdot M$ instructions to move the data to and from memory. The PowerPC instructions to process a single matrix row-vector in-product are shown in the top of the figure. For the complete matrix-vector in-product, this instruction sequence has to be repeated M times, totalling to 258 instructions for this particular case (excluding stack management). The VPU can execute the same calculation with 6 instructions.

A VPU with a *vpuWidth* of N (or larger than N) can retrieve a matrix row operand and an input vector operand, calculate their in-product en store the result as an entry in an output vector in a single instruction. In the middle of Example 3.3, the execution of the same block on a VPU with an issue width of 6 is shown. The matrix coefficients are stored as 6 vector parameters of width 6 in the parameter memory. The input and output vectors are available as single vectors in the (remote) input memory. Executing the complete $MAT_{6 \times 6}$ block then requires a single `inprod` instruction that is repeated *M-1* times. An operand followed by a + denotes that the corresponding *inc*-bit for that operand is set.

The `inprod m+ in out+` instruction is repeated 5 times, where at each invocation the incremented `m` yields the next row of the matrix, and the incremented `out` yields the next element in the `out`-vector. The $MAT_{6\times6}$ is executed in 6 instruction on a VPU with an *vpuWidth* of at least 6.

Due to timing, area and memory constraints, it is not always possible to use a VPU as wide as the vectors to be processed. If the issue width of a VPU is smaller than N, it can only partially process a row-vector in-product with each `inprod` instruction, so additional `add` instructions are required to form the final in-product elements from the intermediate results. In general, the number of VPU instructions needed to execute a $MAT_{M\times N}$ block on a VPU of size *vpuWidth* is M if $vpuWidth \geq N$, or $M\lceil N/vpuWidth\rceil + M\lceil\lceil N/vpuWidth\rceil/vpuWidth\rceil$ otherwise.

The execution of the $MAT_{6\times6}$ block on a VPU with $vpuWidth = 3$ is shown at the bottom of Example 3.3. Now, the matrix coefficients are stored as 12 vector parameters of width 3 in the parameter memory, where consecutive even and odd vector addresses contain the first and second half of a matrix row respectively. Similarly, the input is stored as two consecutive 3-wide vectors in the input memory. A complete row-vector in-product is calculated by adding the in-products of the first and second semi-rows and two semi-vectors.

The two intermediates of a complete row-vector in-product are obtained with a single `inprod m[0]+ in[0]+ t[0]+` instruction, with its repeat field set to 1. The `inprod` instruction is then repeated once, incrementing `m`, `in` and `t` by one to access the second half of a row of `m`, the second half of the input vector, and the next element in an intermediate result vector of `t`. By preceding this instruction with a `5 loop 2 0 1` instruction, the once-repeated `inprod` instruction is executed 6 times, where each loop-repetition increments `m` by 2 (thus starting at the even vector addresses of `m`, containing the first semi-rows of the matrix), and `t` by 1, starting at the next intermediate result vector.

After executing the 12 `inprod` instructions, the first two entries in each of the 6 `t` intermediate vectors contain the in-product result of a semi-row and a semi-vector. With the `2 sum t[0]+ out[0]+` instruction, these intermediate entry pairs are added to yield the final row-vector in-product result, which is repeated twice. The preceding `1 loop 0 3 1` executes this sequence twice, producing the 6 final in-product results and placing them in the two output vectors.

With a VPU of width 3, the $MAT_{6\times6}$ is executed in 20 cycles. Alternatively, the small loop overhead of the two explicit `loop` instructions can be avoided by unrolling the two outer loops. Then, the same block can be executed in 18 cycles, by means of 6 `inprod` instructions which are each repeated once, and two `add` instructions that are each repeated twice. In this way, a reduction in execution time is traded for increased code size.

3.3.3 Lookup Unit

The Lookup Unit (LU) is a processing unit intended for tasks that require extensive address calculations, like non-linear estimation through lookup, interpolation

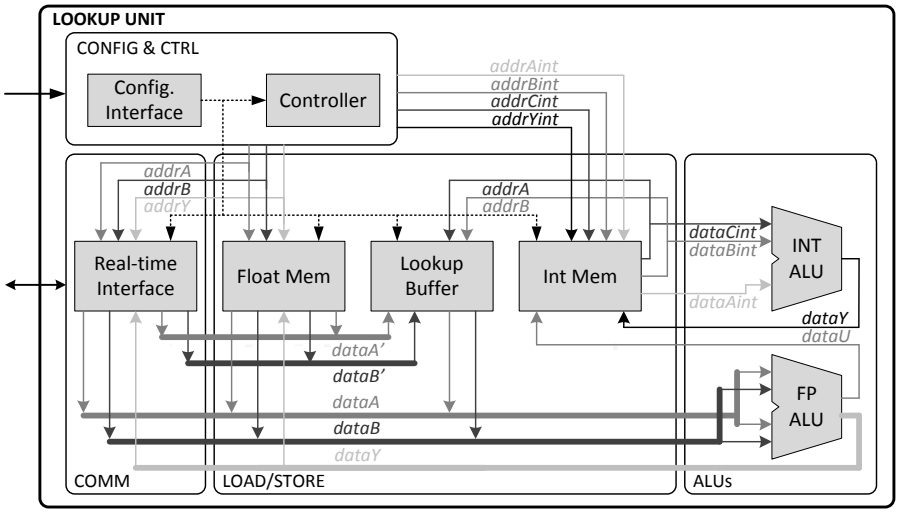


FIGURE 3.10: *Lookup Unit block diagram.*

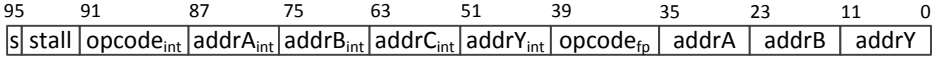


FIGURE 3.11: *Lookup Unit instruction format.*

and cyclic buffers. To this end, it has a dual integer and floating point datapath, organized as a dual issue-slot VLIW processor.

Figure 3.10 shows the micro-architecture of the LU. The configuration interface and real-time interface common to all processing units are shown on the left side of the figure. The LU has two sets of distributed memories. A parameter / state / temporary memory used for floating point calculations is mapped to one physical memory (Float Mem). Another parameter/state/temporary memory, used for integer calculations, is mapped to a second physical memory (Int Mem). A third physical memory is used as a lookup memory (Lookup Buffer), which contains floating-point lookup data (e.g. approximations of non-linear functions).

The LU has two ALUs, one for integer calculations, and one for floating point calculations. The integer ALU consists of a set of comparison units, an adder and a multiplier, which can be used to calculate addresses to indirectly address the Lookup Buffer (LB). The floating point ALU consists of an adder, a multiplier and a floating point to integer converter. It can access data from floating point memory (for temporaries, state and parameters), the input memory, or from the Lookup Buffer. The floating point datapath can be used to e.g. perform interpolation on values obtained by lookup.

The LU instruction set, is shown in Table 3.3. The integer instruction set has

TABLE 3.3: *Lookup Unit instruction set.*

Integer Operations	
Instruction	Operation
add $A\ B\ Y$	$Y = A + B$
sub $A\ B\ Y$	$Y = A - B$
mult $A\ B\ Y$	$Y = A \cdot B$
clip $A\ B\ C\ Y$	$Y = \min(\max(A, B), C)$
wrap $A\ B\ C\ Y$	$Y = B$ if $A > C$
	$Y = C$ if $A < B$
	$Y = A$ otherwise
wrapinc $A\ B\ C\ Y$	$Y = A + 1$ if $A < C$
	$Y = B$ otherwise
Floating Point Operations	
fclear Y_{fp}	$Y = 0$
fadd $A'\ B'\ Y$	$Y = A' + B'$ (1)
fsub $A'\ B'\ Y$	$Y = A' - B'$
fmult $A'\ B'\ Y$	$Y = A' \cdot B'$
fconv $A'\ B\ Y$	$U[\&B] = (int)(A)$ (2)
	$Y = A - (int)(A)$
write $A\ C_{int}$	$LB[C_{int}] = A$ (2)
write2 $A\ B\ C_{int}\ B_{int}$	$LB[C_{int}] = A$
	$LB[B_{int}] = B$
writeout $A'\ B\ Y$	$Out[\&B, \&Y] = A'$ (3)

(1) $A' = A$ if $msb(\&C_{int}) = 0$, else $A' = LB[C_{int}]$
$B' = B$ if $msb(\&B_{int}) = 0$, else $B' = LB[B_{int}]$
(2) LB = Lookup Buffer, U = Uint Memory
(3) $\&B$ and $\&Y$ together contain the remote dest.addr.

add, *sub* and *mult* instructions to implement linear addressing modes, as well as several *wrap*-instructions to facilitate increasing and decreasing circular addressing modes. Besides the common *add*, *sub* and *mult* operations, the floating point datapath can also perform *float* to *uint* conversion and read/write data from/to the Lookup Buffer.

The floating point instruction set contains, next to the common *add*, *sub* and *mult* instructions, an *fconv* instruction that converts and rounds a floating point number to an unsigned integer number. The *write* (*write2*) instruction writes a (two) floating point value into the Lookup Buffer. The *writeout* instruction sends a data value from the Lookup Buffer or the floating point memory to another processing element. Note that this deviates from the general explicit data-sending employed in the ASIPs. To constrain memory usage, the LU memories are more shallow, and instruction operand addresses are smaller to constrain the instruction memory size. These operand addresses are too small to contain a complete output address, so communication is implemented as an implicit instruction in this unit.

Figure 3.11 shows the LU instruction format. The 96-bit wide VLIW instructions consist of a 3-operand floating point instruction and a 4-operand integer instruction. A floating point instruction has three 12-bit operand addresses and a 4-bit *opcode*. An integer instruction contains four 12-bit operand addresses and

a 4-bit *opcode*. A 3-bit *stall* field and a *stop* bit implement pipeline stalling and speed up task switching (Section 3.3.1).

3.4 Related Work

The mismatch between GPP platform performance and the performance requirements of real-time (control) applications is widely understood, and many different approaches have been reported to address this problem. The works in [77, 49, 71] aim to improve the time-predictability of GPP architectures in order to provide better best-case and worst-case performance guarantees. In [77], the sources that result in bad GPP timing predictability are identified and architectural changes are proposed, while in [49] and [71] concrete time-predictable processor architectures are proposed. The main focus is on a single-core GPP-based architecture with predictable caches, memory access and pipeline behaviour.

Our architecture supports a similar philosophy, but besides providing timing-guarantees our main focus is to improve worst-case performance. This is achieved by architecture specialization, reducing complexity and providing a scalable distributed platform. Even though the proposed predictable architectures in [77, 49, 71] greatly reduce jitter and can be accurately modeled and analyzed, they are still single-core general-purpose architectures. As a consequence, they suffer from overhead due to address calculations and cannot thoroughly exploit the different levels of parallelism and heterogeneity present in digital control applications.

Other approaches rely on application-specific architectures [92, 50, 67, 13, 58], mostly targeting FPGA technology. In these approaches, the operations of the control algorithm are implemented directly in FPGA, i.e. a spatial layout of the corresponding data flow graph at operator level with some amount of resource sharing controlled by a FSM. Even though some approaches employ floating-point logic ([67]), typically a fixed-point representation is used because floating-point logic is expensive in terms of FPGA logic and delay. To this end, the dynamic range of the variables in the control algorithm is analyzed and an appropriate fixed-point representation is chosen (e.g. in [50]). The approach in [13] further reduces FPGA resource utilization by employing distributed arithmetic which enables LUT-based multiplication.

Despite the steady growth of gate count and other logic on modern FPGA devices, these application specific architectures are only applicable to small to medium sized control applications due to the limited amount of resource sharing. Even though architectures implemented in FPGA are reconfigurable, the reconfiguration is very slow because upon application changes the complex placement and routing steps have to be performed again. In our particular industrial case the allowed configuration time is constrained to the order of a few minutes, because part of the functionality is decided at boot time of the machine. Finally, due to the lack of abstraction, direct FPGA implementation requires specific skills that are usually not the expertise of control application designers. However, the work

in [60] shows that the advent of High-Level Synthesis tools into the digital control domain can significantly alleviate this issue.

More generic architectures for digital control are proposed in [38, 14, 88, 68]. These ASIP architectures focus on one or more MAC units with some memory for inputs and coefficients, supporting a limited instruction set with which PID and state-space controllers in the z or δ domain can be implemented. The architecture in [38] consists of one or more MACs connected to an input buffer, a coefficient array and an output buffer. The MACs operate on 30-bit fixed-point state variables and 11-bit floating-point coefficients, and are organized in three architectural variants: a single MAC, up to 4 parallel MACs and a larger set of MACs organized in an SIMD way. Similarly, the low cost architecture in [14] consists of a single MAC that operates on 27-bit fixed-point state variables and 11-bit floating-point coefficients stored in a 4-port register file that can be supplied with new input data from I/O devices or new coefficients from a data ROM. The work in [68] integrates a single-MAC architecture onto a System-on-Chip design including a GPP and peripherals. The high-speed low-cost architecture proposed in [88] employs $\Sigma\Delta$ modulation to obtain 1-bit input signals. The inputs are multiplied with 24-bit fixed-point coefficients. The limited instruction set implements few other operations that e.g. read/write data from/to memory or IO.

Our platform targets the same type of calculations as these ASIP architectures. Contrary to these approaches, we use a pipelined 32-bit floating-point datapath in our ASIPs, and the accumulation operation is performed by an adder tree instead of a MAC. The presented architectures employ only a single ASIP architecture specialized for a specific type of calculation. Contrary to the interconnected ASIPs in our platform, these architectures are therefore unable to exploit the available task-level parallelism, and cannot deal with the heterogeneity found in modern control applications.

Heterogeneous MPSoC platforms are popular architectures for real-time embedded applications. The FPGA-based architecture templates proposed in [32, 61, 20] offer a mix of GPPs and special-purpose hardware connected by dedicated FIFO buffers. The different cores synchronize in a distributed way according to specific dataflow variants. The architecture proposed in [51] targets software-defined radio applications. It connects several fixed-point vector DSPs, floating-point scalar DSPs, global memory through two crossbar-based NoCs. Synchronization and scheduling is performed at runtime by a single PU that enforces SDF semantics in a centralized way. Upon the enabling of a task, inputs are copied from global memory to the local memory of the target PU, and copied back when the task execution completes. Such heterogeneous multiprocessors are able to deal with heterogeneity and can exploit several levels of parallelism. However, the target applications for these architectures are more throughput-oriented, contrary to the more latency-dominated digital control domain (see [10]). As a result, they can afford software-based synchronization protocols that result in overhead and communication latencies that are unacceptable for high-performance control applications. The centralized approach in [51] not only results in synchronization

overhead, the copying of data to and from PUs can result in lots of contention on the interconnect. Similar to this approach, our platform employs a combination of specialized vector, scalar and other PUs and synchronizes according to dataflow semantics. However, synchronization in our platform is done at PU-level in hardware with the more restricted HSDF synchronization semantics, and our PUs have a distributed local memory system.

3.5 Outlook

The platform architecture presented in this chapter focuses on the signal processing part of the control application operating in a single controller configuration. In complex systems like e.g. a wafer scanner, a controller can have multiple configurations (control-modes), which are changed at runtime by synchronously setting new parameter values for all constituent controller tasks. To this end, the tasks are to be provided with new parameter values, which are loaded during the course of multiple samples, while the tasks keep operating on their previous parameter values. When all tasks have loaded their new parameters, tasks switch to their new parameters simultaneously, i.e. a double buffering approach similar to that in display rendering is used. Future work could focus on the integration of this functionality into the platform.

In a follow-up study [80], we have shown that the current architecture is capable of supporting such functionality by adding configuration logic with a DDR3 interface for parameter storage. The configuration logic is connected to the configuration bus of a multi-ASIP platform instance. The proof of concept showed that the platform can change its configuration, however, the single configuration bus is proving to be a performance bottleneck for the run-time reconfiguration. Therefore, future maturation of the platform should involve upgrading this configuration interface such that its bandwidth is sufficient to load large sets of new parameter values within the required number of samples. Also, the follow-up study in [80] considered a hybrid GPP-and-FPGA solution, to mitigate the complexity of the interaction between supervisory control and process controllers. Further investigation could focus on supporting the full interaction interface of the supervisor on the FPGA platform, including e.g. status monitoring.

Currently, PU instruction operand addresses comprise the full memory address of the associated variables. As a result, operands can be immediately fetched from memory, but it limits the amount of addressable memory space due to the limited size of instructions. It also complicates runtime controller reconfiguration, since parameter addresses are hard-coded in the instructions. In the current incarnation of the platform, tasks that have multiple parameter sets need to be instantiated twice, with the parameter addresses of the two tasks linked to two different parameter memory spaces to support double buffering. This is quite memory-inefficient, since both the required parameter and instruction memory for each task is doubled. Therefore, modification of the architecture to employ

indirect addressing rather than direct addressing would be a logical step. To this end, for each task a context register could be kept, which contains the base addresses of the different logical memory spaces for that task. Memory addresses can then be constructed either by concatenation or addition of the base addresses and the address offsets specified in instruction operands.

Another focus for follow-up study could be improving the floating point unit latency. An FP addition takes 8 cycles, so there is a significant performance penalty for sequential instructions. The VPU has an even larger latency (depending on its issue width) due to the addertree that is connected to its parallel multipliers. Compared to tasks mapped to a SPU, for tasks mapped to the VPU it is typically easier to hide (part of) this latency due to the high amount of parallelism present in these tasks. Therefore, the marginal performance improvement realized by adding forwarding to all the FPUs and addertree of a VPU is unlikely to weigh up to the added complexity and logic. For the SPU however, the potential performance gain is much higher due to the sequential nature of typical tasks mapped to it, and since it only has a single FPU the amount additional logic is relatively low compared to adding forwarding to the VPU FPU units.

3.6 Summary

This chapter presented a heterogeneous execution platform template targeting high-performance digital control applications. The choice of an FPGA-based template of interconnected ASIP processors was motivated, and the main components with which platform instances can be constructed were discussed. In particular, the generic architecture optimizations and the dataflow synchronization between the different processing units were discussed, and how this contributes to an efficient execution of digital control applications.

The main ASIP processing units that are part of our design library were discussed in detail, with focus on their particular datapaths and instruction sets. Several examples showed the execution of typical operations discussed in Chapter 2 on these ASIPs. Finally, the chapter related the presented platform to other platforms described in literature.

Chapter 4

Analysis

The previous chapters discussed the main characteristics of digital control applications and presented a heterogeneous multi-ASIP platform template on which these applications can be mapped. This chapter presents a timing analysis technique with which the performance of these mapped applications can be predicted. Chapter 2 showed that applications in the digital control domain can typically be modeled as periodically restarted Directed Acyclic Task Graphs (DAGs). In our analysis, these DAGs are assumed to follow the semantics of Homogeneous Synchronous Data Flow (HSDF, [48]). The HSDF model of computation is commonly used in performance analysis and synthesis of e.g. Multi-Processor Systems-On-Chip. With SDF analysis, the impact of mapping and scheduling on application timing and platform memory requirements can be analyzed.

However, there exist no SDF-based analysis techniques that can analyze the execution timing of tasks mapped to shared First-Come-First-Served (FCFS) scheduled resources. Parts of our platform, e.g. the switches that form the Real-Time network between PUs, employ FCFS arbitration. Also, many (interconnect) networks in modern embedded systems, like e.g. FlexRay [52] and RapidIO [24], employ FCFS scheduling. These types of networks are typically used to connect multiple processor boards (including e.g. boards that house an FPGA with a multi-ASIP platform running on it) that form a computational platform.

This chapter presents a scalable analysis technique that can analyze the timing of DAGs consisting of tasks with varying execution times mapped to shared resources under a FCFS scheduling policy. In this approach, timing is expressed in intervals (Section 4.1), which denote the best-case and the worst-case timing properties of tasks. These intervals are propagated through the nodes in the graph taking into account the contention (Section 4.2). By taking into account the best-case timing as well as the worst-case timing, the contention on the shared resources can be estimated (Section 4.3). Scalability is ensured by using a conservative approximation (Section 4.3.1) on the interval bounds of the tasks. An iteration-based approach is then used to obtain conservative bounds on the timing

TABLE 4.1: *Interval operations.*

Operation	Definition
$i_1 + i_2$	$[L(i_1) + L(i_2), U(i_1) + U(i_2)]$, $i_1, i_2 \in I$
$i_1 - i_2$	$[L(i_1) - L(i_2), U(i_1) - U(i_2)]$, $i1, i2 \in I, L(i_1) \geq L(i_2), U(i_1) \geq U(i_2)$
$\max V$	$[max_{i \in V} L(i), max_{i \in V} U(i)]$,$V \subseteq I$, $V \neq \varnothing$, V is finite
$\cup V$	$[min_{i \in V} L(i), max_{i \in V} U(i)]$, $V \subseteq I$, $V \neq \varnothing$, V is finite

of a DAG application model (Section 4.4). We evaluate the analysis method by applying it to an industrial-scale application model in Section 4.5. Related work in Section 4.6 is followed by an outlook in Section 4.7, and the chapter concludes with a summary in Section 4.8.

4.1 Timing Properties

The analysis presented in this chapter assumes a given Directed Acyclic Task Graph (DAG) G, with a set of tasks T and a set of dependencies D. Chapter 2 discussed how digital control applications can be modeled by such DAGs. The tasks in T are allowed to have variation on their execution time, so the execution timing of G will also vary at different executions of the graph. To capture this variation, the timing properties of G are expressed in intervals.

Definition 1. (INTERVAL) *I is the set of closed intervals defined by $I = \{[a, b] \mid a, b \in \mathbb{N}, b \geq a\}$. Intervals are ordered according to a subset ordering, i.e. $(I, \subseteq)$ is a partially ordered set. The lower bound L of an interval is given by $L([a, b]) = a$, the upper bound U is given by $U([a, b]) = b$.*

Table 4.1 defines some basic operations on intervals. With each task $t \in T$ an execution interval $E(t) = [a, b]$ is associated, denoting that any execution of t will require at least a, and at most b time units. Each execution of t can require a different amount of time, but the required time is always within these interval bounds. G is bound to a set of resources R, which can be either private or shared. Resources in R are assumed to employ a *First-Come-First-Served (FCFS)* scheduling policy, where tasks are queued for execution based on the time they are enabled. An interval-timed DAG is then defined as follows:

Definition 2. (INTERVAL-TIMED DAG) *An interval-timed DAG is a quadruple (T, D, M, E) consisting of a finite set of tasks T, and a finite set of dependencies $D \subseteq T \times T$. Tasks in T are mapped to a set of FCFS-scheduled resources R, according to the function $M : T \to R$, which maps each task in T to exactly one resource $r \in R$. Labeling function $E : T \to I$ assigns to each task $t \in T$ an execution interval $E(t)$ that bounds the duration of an execution of t.*

Figure 4.1 shows an example interval-timed DAG G_1 with five tasks $t_1 \cdots t_5$ and six dependencies $d_1 \cdots d_6$, bound to two resources r_1 and r_2. Resource map-

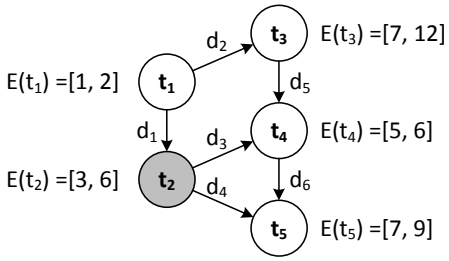


FIGURE 4.1: *Example interval-timed DAG G_1.*

ping M is denoted by task grey shading, and the execution interval labeling E is shown next to the task vertices.

In a concrete execution, interval-timed DAG G is labeled with an execution *time* labeling function E^c, which assigns to each task $t \in T$ an execution *time* $E^c(t)$ for which $L(E(t)) \leq E^c(t) \leq U(E(t))$ holds. Due to the variation in task execution times and the *FCFS*-scheduling of the resources that G is mapped to, G can have many different concrete executions. Any concrete execution time labeling E^c on G uniquely determines (up to non-determinism) the exact execution order of all the constituent tasks in G.

The execution timing properties of G should reflect the properties for any execution of G, and are therefore expressed in intervals as well, with the following additional task labelings:

- $C : T \to I$ maps tasks to their completion interval bounds, i.e. $C(t)$ is the interval at which task t will finish its execution.
- $En : T \to I$ maps tasks to their enabled interval bounds.
- $B : T \to I$ maps tasks to a busy interval. The busy interval of a task reflects the delay between its enabling and its completion time, including both waiting time as a result of contention, and its execution time.[1]

The goal of timing analysis on G is to find conservative bounds on these labeling intervals, such that the concrete timing properties $C^c(t)$, $En^c(t)$ and $B^c(t)$ of *any* possible concrete execution of G are within these interval bounds for any $t \in T$. The busy interval $B(t)$ of some task t is considered to be conservative if both $En(t)$ and $C(t)$ are conservative bounds for any concrete execution. The relations between the different timing interval labelings are discussed in the next section.

[1]Note that the busy interval B is not a response interval, but an algebraic term that, given a best-case (worst-case) enabling of a task, denotes the delay that results in a best-case (worst-case) completion of that task. The worst-case response time of t, i.e. its maximum delay, is not necessarily in $B(t)$; it is bounded by $[L(C(t)) - U(En(t)), U(C(t)) - L(En(t))]$.

4.2 Timing Model

A task in a DAG G is enabled when all its predecessors have completed their execution. Without loss of generality, tasks without inputs are assumed to be enabled at $[0, 0]$. The relation between the enabled interval of a task $t \in T$ of G and its completion interval is:

$$En(t) = \begin{cases} [0,0] & \text{if } pred(t) = \varnothing \\ \max\limits_{t' \in pred(t)} C(t') & \text{otherwise} \end{cases}, \tag{4.1}$$

where $pred(t) = \{t' \in T \mid (t', t) \in D\}$ denotes the set of predecessors of t.

When t is enabled, it enters the execution queue of the resource it is mapped to, where it needs to wait on the completion of all tasks that were queued earlier. Eventually, t exits the execution queue of its resource and starts executing. Completion of t then occurs after some time in $E(t)$. During execution of t, no other task can claim the resource on which t is executing (i.e. there is no preemption). The busy interval $B(t)$ of task t is the period between its enabling and its completion, and thus includes both the time that t spends waiting in the execution queue (which depends on the contention) as well as the time that t is executing. The completion interval of any task $t \in T$ in terms of its busy interval is then given by:

$$C(t) = En(t) + B(t). \tag{4.2}$$

If B is given, e.g. when there is no contention, the timing of a DAG is calculated by propagating the completion- and enabled intervals of Equations 4.1 and 4.2 through the nodes of the graph in topological order, similar to (Homogeneous) SDF-analysis techniques lifted to an interval-timed domain. DAG G_1 in Figure 4.1 is an example of such a DAG where tasks do not contend for resource access. The dependencies in G_1 enforce a unique execution order ($t_1 \rightarrow t_2$ *and* $t_3 \rightarrow t_4 \rightarrow t_5$) in *any* concrete execution, i.e. the tasks in G_1 execute according to a Fixed-Order (FO) schedule. The only tasks that are enabled at the same time are t_2 and t_3, but they cannot contend because they are mapped to different resources. Due to the FO-schedule, any task t will never have to wait on completion of some other task t' still executing on its resource, so it can start executing immediately when it is enabled, so $B(t) = E(t)$ for all $t \in T$.

The timing intervals of G_1, shown in Table 4.2, can then be computed as follows. Initially, only task t_1 is enabled at $En(t_1) = [0, 0]$, since it is the only task that has no predecessors. Execution of t_1 completes at $[0, 0] + [1, 2]$ and enables tasks t_2 and t_3 at $[1, 2]$, since both have only a single incoming dependency on t_1. Subsequently, executions of t_2 and t_3 complete at $C(t_2) = [1, 2] + [3, 6] = [4, 8]$ and $C(t_3) = [1, 2] + [7, 12] = [8, 14]$ respectively. Completion of t_3 enables task t_4, which has multiple incoming dependencies, so $En(t_4) = \max\{C(t_2), C(t_3)\} =$

TABLE 4.2: *Execution timing labels of non-contended DAG G_1.*

Task	En	C	B
t_1	[0,0]	[1,2]	[1,2]
t_2	[1,2]	[4,8]	[3,6]
t_3	[1,2]	[8,14]	[7,12]
t_4	[8,14]	[13,20]	[5,6]
t_5	[13,20]	[20,29]	[7,9]

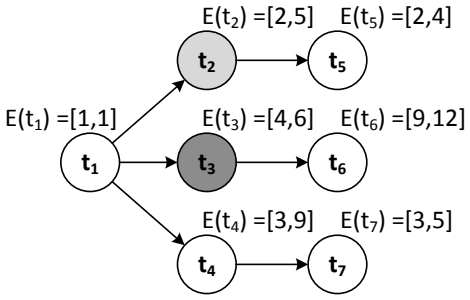


FIGURE 4.2: *Example DAG G_2 with contention on the white resource.*

$\max\{[4,8],[8,14]\} = [8,14]$. Completion of t_4 at $[8,14] + [5,6] = [13,20]$ enables t_5 at $[13,20]$, which finally completes at $[20,29]$.

For DAGs with contention, B is not given but needs to be computed. Since tasks are queued for execution based on their concrete enabling time, the additional task delay in B due to contention can be estimated by analyzing the relation between enabled intervals of different tasks on the same resource. These enabled intervals in turn depend on B. A natural approach to deal with this recursive dependency is to use a fixed-point iteration, where DAG timing is analyzed by iterating on B, starting with a B assuming no contention. In each iteration more contention is taken into account, until a fixed-point is reached. The required contention model that estimates contention given the enabled intervals of the DAG is explained in the next section.

4.3 Contention Model

Figure 4.2 shows another DAG G_2, consisting of seven tasks $t_1 \cdots t_7$ mapped to three resources p_1(white), p_2(light grey) and p_3(dark grey). G_2 has insufficient dependencies between the tasks on resource p_1 to enforce a FO-schedule. As a result, some tasks on that resource will contend for resource access.

Due to the dependency between tasks t_1 and t_4, t_1 will always complete before t_4 is enabled, so they do not contend for resource access. Also, t_2, t_3 and t_4

do not contend, since they are mapped to different resources. For these tasks, $B(t) = E(t)$ holds. Initially, task t_1 is the only enabled task, at $[0, 0]$. Completion of t_1 at $[1, 1]$ enables t_2, t_3 and t_4, which then complete at $[3, 6]$, $[4, 7]$ and $[4, 10]$ respectively. Now, tasks t_5, t_6 and t_7 are enabled at $[3, 6]$, $[5, 7]$ and $[4, 10]$. These tasks suffer from contention, so their busy interval is to be determined.

Figure 4.3 shows a Gantt-chart representation of four different concrete executions of G_2. In the figure, task names are followed by a bracketed number that denotes the concrete execution time for that task. In all executions, t_1 is the first task to execute, followed by concurrent execution of t_2, t_3 and t_4 on the three different resources. The concrete completion times of these tasks determine the execution order and busy times of tasks t_5, t_6 and t_7. In the concrete execution at the top of the figure, t_3 is the first of the three concurrently executing tasks to complete its execution, enabling t_6. At the moment that t_6 is enabled, task t_4 is still executing on the white resource, so t_6 needs to wait for completion of t_4 before it can start executing. Completion of task t_2 occurs after t_3 and before t_4, enabling t_5. At the moment that t_5 is enabled, t_4 is still executing on the same resource, and t_6 is already queued for execution. So t_5 needs to wait for the completion of both tasks before it can start executing. Task t_7 is the last of the three concurrent tasks to finish, enabling t_7. Similarly, t_7 needs to wait for completion of t_4, t_5 and t_6 before it can start executing.

The second concrete execution shown in the figure results in the same execution order, but with a different timing. Now, when t_6 is enabled, task t_4 is just completed, so t_6 can immediately start executing. Despite contention, in this particular execution t_6 does not need to wait. The Gantt charts of the bottom two concrete executions of G_2 in the figure show two other possible execution orders as a result of different concrete enabling times of the contending tasks. In these executions, t_6 is either enabled after t_7 or after t_5 and t_7, so in these cases t_6 needs to wait on the completion of these earlier queued tasks.

These four concrete executions exemplify how contention in the interval domain affects concrete execution timing. In general, the following observations on resource contention can be made:

- Predecessor completion is not a sufficient precondition for the start of the execution of a contending task; completion of any other tasks that precede it in its execution queue is required as well.

- A task that is enabled *strictly* before some other task precedes it in *any* concrete execution. For example, in G_2, t_4 is enabled strictly before t_5 and t_6, so in any concrete execution of G_2, t_4 precedes these tasks.

- A task that has an enabled interval that overlaps with that of another task mapped to the same resource will precede it in *some* executions, while in other executions it can be the other way around (assuming they are independent). For example, the enabled interval of task t_6 of G_2 overlaps with those of t_5 and t_7. In the first two concrete executions of Figure 4.3,

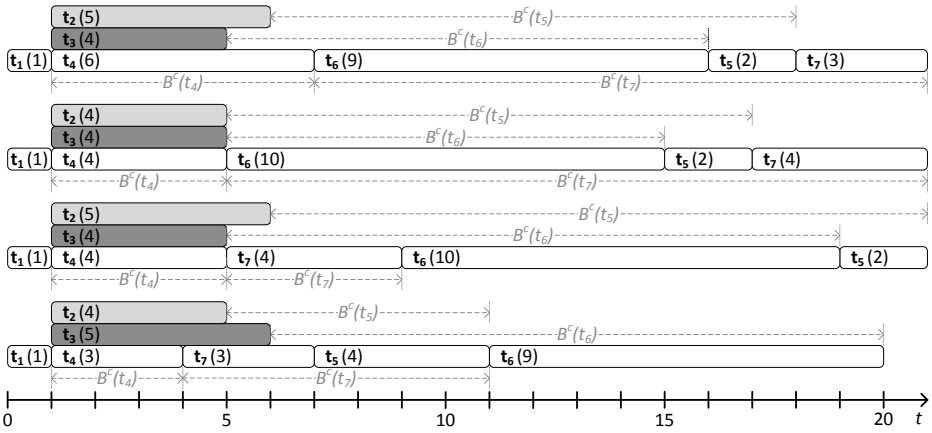
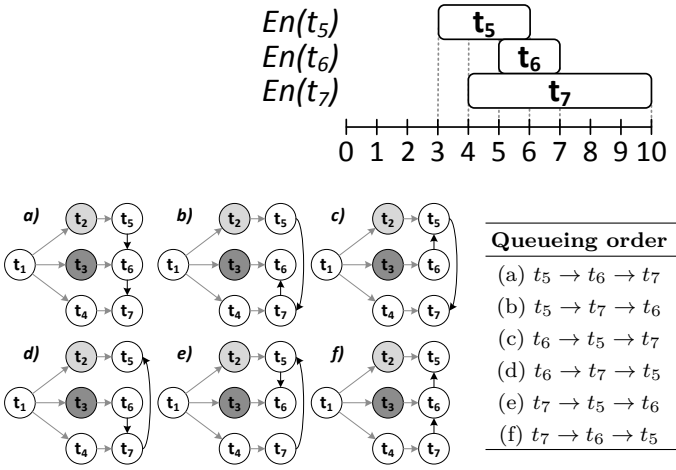


FIGURE 4.3: *Gantt-chart showing four different concrete executions of G_2.*

t_6 precedes t_5 and t_7, while in the bottom two concrete executions, t_6 is preceded by one or more of the other contenders.

For tasks with overlapping enabled intervals both the queueing order as well as the timing of these tasks can be different in different concrete executions. This not only affects the contending tasks under consideration, but also all tasks that depend on them and other tasks mapped to the same resource that have not been executed or queued yet. The tightest interval bounds are acquired by analyzing the timing effects of all these different queueing orders, e.g. by means of state-space exploration. In such an exact approach, each possible queueing order can be modeled as a separate child DAG which is derived from the original DAG by adding dependencies to enforce a particular execution order. The enabled intervals of contending tasks in the original graph can be updated in the child graph according to the choice of execution order that is enforced in that particular child graph. This is shown for G_2 in Figure 4.4.

In G_2, there are three contending tasks t_5, t_6 and t_7 that are enabled at $[3, 6]$, $[5, 7]$ and $[4, 10]$ (shown in the top of the figure). The six child graphs that can be constructed from the original graph are shown on the bottom left and the updated enabled intervals for each child graph are shown in the table on the bottom right. The execution order enforced in a particular child graph is modeled by the additional black-colored dependencies in the figure. In the first child graph (a), this enforced execution order is $t_5 \rightarrow t_6 \rightarrow t_7$. Since t_6 precedes t_7 in this graph, the latter cannot be enabled earlier than $t = 5$, so $En(t_7) = [5, 7]$. In the second child graph (b), the execution order is fixed to $t_5 \rightarrow t_7 \rightarrow t_6$. Since t_7 precedes t_6, it can only be enabled in $[4, 7]$, since t_6 is never enabled later than $t = 7$. Any later enabling of t_7 would not result in the execution order of this particular child graph, so that case is modeled in one (or more) of the other child



Queueing order	En(t_5)	En(t_6)	En(t_7)
(a) $t_5 \to t_6 \to t_7$	[3,6]	[5,7]	[5,10]
(b) $t_5 \to t_7 \to t_6$	[3,6]	[5,7]	[4,7]
(c) $t_6 \to t_5 \to t_7$	[5,7]	[5,6]	[5,10]
(d) $t_6 \to t_7 \to t_5$	[5,6]	[5,6]	[5,6]
(e) $t_7 \to t_5 \to t_6$	[4,6]	[5,6]	[4,6]
(f) $t_7 \to t_6 \to t_5$	[5,6]	[5,6]	[4,6]

FIGURE 4.4: *Different execution orders of the three contending tasks in G_2.*

graphs. For the other child graphs (c)-(f), similar updates to the enabled intervals can be derived.

The child graphs in Figure 4.4 are contention-free, since the contention of the original graph has been resolved by assuming a particular execution order of the contenders in each of the child graphs. However, besides the complexity of generating these child graphs for more complex DAGs, the analysis time and resource utilization of such an approach show factorial growth in the number of overlapping tasks in the DAG, rendering an exact analysis intractable already for relatively small DAGs. To ensure scalability of the timing analysis, instead of exactly analyzing all different task execution orders, we approximate the best-case and worst-case completion time of tasks given their enabled intervals, and then combine these into a single closed interval that provides conservative bounds for any concrete execution of the graph. This is shown in the next section.

4.3.1 Approximation

Given an initial B, the enabled intervals of the tasks in the DAG can be calculated by evaluating Equations 4.1 and 4.2 on tasks in the graph in topological order. If the enabled intervals of all tasks are known, the completion interval of a task t can be conservatively approximated by analyzing the possible delay caused by tasks mapped to the same resource that can be queued in the execution queue before the enabling of t.

Tasks mapped to the same resource as t that are enabled strictly earlier than t precede t in *all* concrete executions. The same holds for tasks mapped to the same resource as t on which t is dependent. This dependency can be either direct,

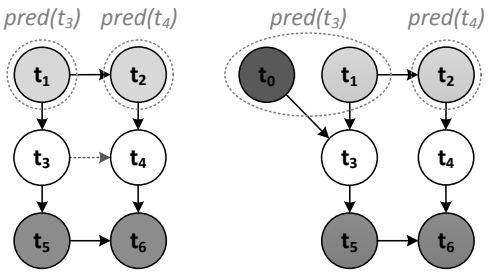


FIGURE 4.5: *Two similar DAGs, with (left) and without (right) an indirect dependency between t_3 and t_4. The dependency relations between the direct predecessors of t_3 and t_4 determines the occurrence of an indirect dependency.*

i.e. there exists a directed subset of the set of dependencies D of graph G that directly connects t' and t, or indirect. An indirect dependency between t and t' is a direct dependency relation between the predecessors of t and t' that results in t' to be enabled before t in any concrete execution, even though there is no direct dependency between t and t'. This is shown in Figure 4.5. The dependencies in the left DAG are such that the tasks in $pred(t_4)$ are dependent on *all* tasks in $pred(t_3)$. Therefore, upon the last completion of the tasks in $pred(t_3)$, at least one task in $pred(t_4)$ has not completed yet, so t_3 will precede t_4 in any concrete execution. In the right DAG, there is at least one task in $pred(t_3)$ on which at least one task of $pred(t_4)$ is not dependent. Here, t_3 is not necessarily queued before t_4 in all concrete executions, since a late completion of t_0 can result in t_4 to be queued before t_3.

When there is a direct dependency from t' to t, t' does not contribute to the delay of t, since it always completes before t is enabled. Tasks with a strictly smaller enabled interval and tasks on which t is indirectly dependent can contribute to the delay of t, since in all concrete executions t needs to wait on completion of these tasks prior to starting its execution. This affects both the best-case as well as the worst-case delay estimation of t. Tasks mapped to the same resource as t with an enabled interval overlapping with that of t will precede t in only *some* concrete executions. These tasks only affect the worst-case delay estimation of t.

Let $ee(t)$ denote the set of tasks which are independent of t, mapped to the same resource and enabled strictly earlier than t, either based on a strictly smaller enabled interval, or because of an indirect dependency. Similarly, $oe(t)$ denotes all tasks which are independent of t, mapped to the same resource, whose enabled interval overlaps with that of t and which are not in $ee(t)$.

The completion interval of t is approximated by considering a best-case as well as a worst-case concrete execution scenario. These two scenarios lead to a conservative best-case and a worst-case concrete completion time for t, which are used as upper and lower bounds for the completion interval of t.

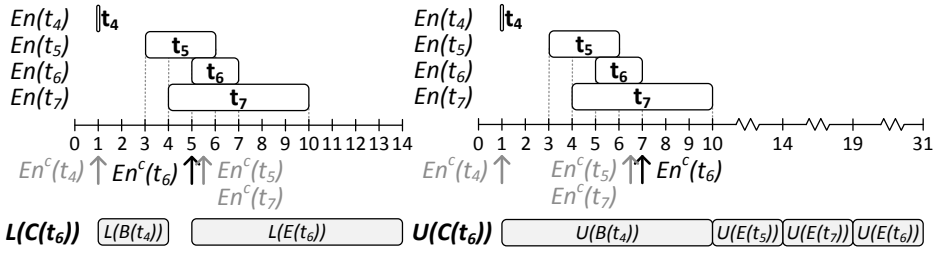


FIGURE 4.6: *Best-case and worst-case completion scenarios for t_6 of G_2.*

The earliest possible completion of t occurs when it is enabled as early as possible, and the start of its execution is delayed as little as possible. This occurs when t is enabled at $L(En(t))$, all tasks in $ee(t)$ complete as soon as possible, and all tasks in $oe(t)$ (except for t itself, which executes at its best-case execution time) are enabled after t. So, in best-case, t can start executing after its best-case enabling and the best-case completion of the last completing task in $ee(t)$.

The latest possible completion of t occurs when it is enabled as late as possible, and the start of its execution is delayed as much as possible. Given t's worst-case enabling, t is delayed most if all tasks in $ee(t)$ complete at the upper bound of their completion interval, and all tasks in $oe(t)$ execute at the upper bound of their execution interval, while they are enabled just before t.

Both completion scenarios are shown in Figure 4.6 for task t_6 in G_2. In G_2, $ee(t_6) = (t_1, t_4)$ and $oe(t) = (t_5, t_6, t_7)$. The best-case completion of t_6, shown in the left of the figure, occurs when t_6 is enabled at 5, the other tasks in $oe(t_6)$ are enabled after t_6, and tasks in $ee(t_6)$ complete as soon as possible. The last completing task in $ee(t_6)$ is t_4, which completes at 4 in the best-case. Since the best case enabling of t_6 is at 5, it is not delayed by any task in $ee(t_6)$, so the best-case completion of t_6 is at $L(En(t_6)) + L(E(t_6)) = 5 + 9 = 14$.

The worst case completion of t_6, shown in the right of Figure 4.6, occurs when t_6 is enabled at 7, t_4 completes at 10, and tasks t_5 and t_7 are enabled just before the enabling of t_6 and execute at the upper bound of their execution interval. The last completion in $ee(t_6)$ is at 10 (i.e. $U(C(t_4))$), so the worst-case completion of t_6 is at $U(C((t_4)) + U(E(t_5)) + U(E(t_7)) + U(E(t_6)) = 10 + 4 + 5 + 12 = 31$.

The conservatively approximated upper and lower bound on completion of t_6 are combined into a single completion interval $C(t_6) = [14, 31]$.

This approximation needs to be refined to prevent counting contributions to the delay of a contending task multiple times. Consider the partial DAG and a representation of its execution queue in Figure 4.7. It consists of 3 independent tasks t_1, t_2 and t_3 mapped to the same resource. The enabled interval of t_1 overlaps with that of t_2 and t_3, and t_2 is enabled strictly before t_3. According to the aforementioned completion time estimation, $U(C(t_2)) = U(En(t_2)) + U(E(t_2)) + U(E(t_1))$. Task t_3 must wait for completion of t_2 before it can start its execution, and

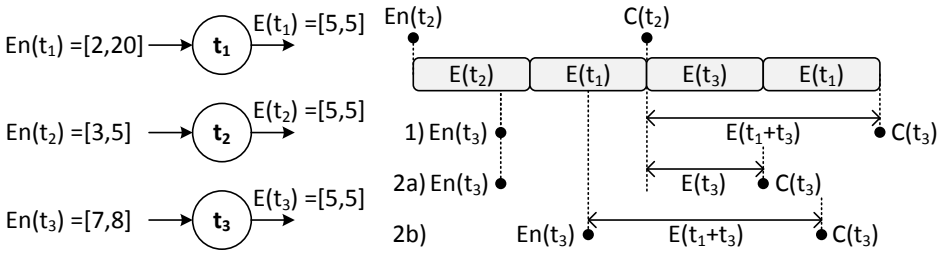


FIGURE 4.7: *Example of part of a DAG where the execution time of t_1 is contributing twice to the completion time of t_3.*

can be preceded by t_1, so then $U(C(t_3)) = U(C(t_2)) + U(E(t_3)) + U(E(t_1))$. However, since $U(E(t_1))$ is already taken into account in $U(C(t_2))$, it is accounted for twice in $U(C(t_3))$ (case 1 in the figure). To correct this, in the estimation of $U(C(t_3))$, only the execution time of tasks in $oe(t_3)$ that are not in $oe(t_2)$ are to be added (case 2a in the figure). However, if $U(En(t_3)))$ is such that $U(C(t_2)) - U(En(t_2)) \leq U(E(t_1))$, not adding t_1 would lead to an estimate that is not conservative, so in that case $U(C(t_2))$ is estimated with $U(En(t_2)) + U(E(t_1)) + U(E(t_2)) + U(E(t_3))$ (case 2b in the figure). So in general, the corrected worst-case completion time estimate is the maximum of case 2a and 2b in the Figure 4.7.

With this refined contention model, the completion interval of a task t given some B is then given by:

$$
\begin{aligned}
&C(B)(t) = \max(\xi, \zeta), \text{ where} \\
&\xi = En(B)(t') + B(t') + \left(\sum_{t'' \in oe(t) \setminus oe(t')} E(t'') \right) \cup E(t) \\
&\text{with } t' \text{ the last completing task in } ee(t) \\
&\zeta = En(B)(t) + \left(\sum_{t'' \in oe(t)} E(t'') \right) \cup E(t).
\end{aligned}
\tag{4.3}
$$

So now, assuming an initial busy interval labeling B, with Equations 4.1 and 4.2 an enabled interval labeling En can be calculated. Based on these enabled intervals, with Equation 4.3 the completion interval of each task can be computed, and by subtracting the corresponding enabled intervals, a new B labeling is obtained. This new B labeling captures contention as seen by the current En labeling, and can then again be used to calculate new enabled and completion intervals. This fixed-point iteration on B is explained in the next section.

4.4 Fixed-point Iteration

This section formulates an iteration on busy intervals, and proves that with this iteration a fixed point will always be found in a finite number of steps. First, a partial ordering on the interval labelings of Section 4.1 is defined.

Definition 3. (POSET ON I^T) *Define relation $\sqsubseteq \subseteq I^T \times I^T$ as follows. For $l_1, l_2 \in I^T$ let $l_1 \sqsubseteq l_2$ if and only if $l_1(t) \subseteq l_2(t)$ for all $t \in T$. It is easy to verify that $(I^T, \sqsubseteq)$ is a poset.*

The iteration will compute busy intervals for each task taking into account the contention on the shared resources. To this end we will define a finite complete lattice on the mappings from tasks to busy intervals. This lattice is contained in poset $(I^T, \sqsubseteq)$ and is defined by explicitly defining a bottom and top element. The bottom element represents the case in which tasks do not contend, while the top element assumes maximal contention.

Definition 4. (FINITE BUSY INTERVAL LATTICE IN I^T) *We define bottom $\bot \in I^T$ and top $\top \in I^T$ by $\bot(t) = E(t)$ and $\top(t) = (\sum_{\{t' \in T | M(t') = M(t)\}} E(t')) \cup E(t)$ for all $t \in T$. The set $B_L \subseteq I^T$ is defined as $B_L = \{B : T \to I \mid \bot \sqsubseteq B \sqsubseteq \top\}$. It is easy to prove that $(B_L, \sqsubseteq)$ is a finite lattice.*

Any $B \in B_L$ maps tasks in T to their busy interval. Given a busy interval labeling $B^i \in B_L$, a new busy interval labeling $B^{i+1} \in B_L$ can be computed by the following progressive function $F_B : B_L \to B_L$:

$$B^{i+1}(t) = \big(C(B^i)(t) - En(B^i)(t)\big) \cup B^i(t), \tag{4.4}$$

where $i1 - i2 = [L(i_1) - L(i_2), U(i_1) - U(i_2)]$ for all $i1, i2 \in I$ with $L(i_1) \geq L(i_2)$ and $U(i_1) \geq U(i_2)$. Given B^i, with Equations 4.1 and 4.2, $En(B^i)$ is computed. Then, $C(B^i)$ can be computed with Equation 4.3, taking into account the contention based on $En(B^i)$. A new busy interval is then obtained by calculating $C(B^i) - En(B^i)$ and taking the union with B^i to ensure progressiveness, which is required to guarantee convergence.

Starting with an initial $B(t) = E(t)$ for all $t \in T$ ($\bot$ of B_L), with Equation 4.4, lattice B_L is traversed from the bottom upwards. At each iteration, busy intervals can grow by taking more contention into account, until a fixed-point is reached. The following theorem shows that the fixed-point algorithm converges in a finite number of iterations.

Theorem 1. *F_B has a fixed-point in B_L given by FIX $F_B = \sqcup \{F_B^n(\bot) \mid n \geq 0\}$, and there is an $m \in \mathbb{N}$ such that $F_B^m(\bot) =$ FIX F.*

Proof. Trivially, $B \sqsubseteq F_B(B)$ holds, since $F_B(B)$ is constructed by taking the union with B. Since B_L is a lattice, and thus a chain-complete partial order, with the Bourbaki-Witt theorem [9, 86], F_B has a fixed-point. Since $\hat{B}$ is finite, this fixed-point will be reached in a finite number of steps. □

4.4.1 Fixed-point Properties

Now that the fixed-point iteration is established, two important properties can be derived. First, the following theorem shows that any fixed-point of F_B is conservative.

Theorem 2. *If B is not conservative, then it is not a fixed-point of F_B.*

Proof. Assume that B is not conservative in the i^{th} iteration. Then there exists a first faulty task t_f with either a faulty concrete completion time $C^c(t_f) > U(C(B^i)(t_f))$ or a faulty concrete enabled time $En^c(t_f) > U(En(B^i)(t_f))$. Since it is the first faulty task, the concrete execution of all earlier tasks are within their respective intervals. Since $En^c(t_f)$ is derived from completion times of predecessors which are within their intervals, $En^c(t_f) \leq U(En(B^i)(t_f))$. Hence, $C^c(t_f) > U(C(B^i)(t_f))$. We show that this will be fixed in the $i + 1^{th}$ iteration such that $C^c(t_f) \leq U(C(B^{i+1})(t_f))$. Referring to Equation 4.4, we distinguish two cases in the concrete execution. The first case is when some tasks in $ee(B^i)(t_f)$ are still executing after $En^c(t_f)$. In the concrete execution, there is a last completing task in $ee(B^i)(t_f)$ followed, without any gaps, by a sequence of zero or more tasks that are not in $ee(B^i)(t_f)$ that ends with the execution of t_f. Since $En^c(t_f) \leq U(En(B^i)(t_f))$, these tasks are included in $eo(B^i)(t_f))$ in the evaluation of ξ given in Equation 4.4. Hence for this case $C^c(t_f) \leq U(C(B^i)(t_f))$. The second case is when t_f is enabled after completion of all tasks in $ee(B^i)(t_f)$. In the concrete execution, there are tasks not in $ee(B^i)(t_f)$ and enabled earlier, ending with t_f. These tasks are included in $eo(B^i)(t_f))$ in the evaluation of ζ given in Equation 4.4. The remaining execution of these tasks is added to $En^c(t_f)$ which is also within its interval bounds. This leads to $C^c(t_f) \leq U(C(B^i)(t_f))$ in both the cases. Since $C^c(t_f) > U(C(B^i)(t_f))$, transitively $U(C(B^i)(t_f)) \leq U(C(B^{i+1})(t_f))$. Thus, B^i is not a fix-point of F_B. □

Finally, Theorem 3 provides bounds on the fixed-point found by the algorithm.

Theorem 3. *For some DAG $G = (T, D)$ and $t \in T$, let $B_{bounds}(t)$ be defined by $[L(E(t), U(\sum_{t' \in indep(t)} E(t'))]$, with $indep(t) = \{t' \in T \mid M(t') = M(t) \wedge \{(t,t'),(t',t)\} \cap D_{TC} = \varnothing\}$ and $G_{TC} = (T, D_{TC})$ the transitive closure of G. If $B(t)$ is bounded by $B_{bounds}(t)$, then $F_B(B)$ is bounded by $B_{bounds}(t)$ as well.*

Proof. $L(F_B(B)(t)) = L(E(t))$ holds trivially due to the union operation in the definition of B in Equation 4.4. The upper bound considers two cases. First case is when $U(\xi) > U(\zeta)$ and $U(F_B(B)(t)) = U\left(En(B)(t') + B(t')\right) - U\left(En(B)(t)\right) + U\left(\sum_{t'' \in \{eo(B)(t) \setminus eo(B)(t')\}} E(t'')\right)$ where t' is the last completing task in $ee(B)(t)$. The term $U(En(B)(t') + B(t')) - U(En(B)(t))$ is only computed using tasks in $indep(t)$. This is because $U(En(B)(t'))$ is derived from tasks that t' depends on, and $U(B(t'))$ is derived from tasks not in $U(En(B)(t'))$ but in $eo(B)(t')$.

Hence, $U(En(B)(t') + B(t'))$ is not computed using duplicates of the execution time of any task. $U(En(B)(t))$ is computed using all tasks having a dependency to t. As a result, the difference $U(En(B)(t') + B(t')) - U(En(B)(t))$ is not computed using any tasks with dependencies to t. It is also not computed using any tasks dependent on t since those have not been enabled yet to contribute to any of the three constituent terms. Tasks in $eo(B)(t)$ can neither have a dependency to t since they are completed before $U(En(B)(t))$, or have dependencies from t since those are enabled only after the completion of t. The term $U\left(\sum_{t'' \in \{eo(B)(t) \setminus eo(B)(t')\}} E(t'')\right)$ adds tasks that are in $eo(B)(t)$ but not in $eo(B)(t')$ thus avoiding duplicates. Therefore, $U(F_B(B)(t))$ is computed using tasks in $indep(t)$, without duplicates. In other words $U(F_B(B)(t))$ is bounded by $U(B_{bounds}(t))$. The second case is when $U(\xi) < U(\zeta)$ and $U(F_B(B)(t) = U\left(\sum_{t'' \in eo(B)(t)} E(t'')\right)$ since $U\left(En(B)(t) - En(B)(t)\right) = 0$. As already explained, tasks in $eo(B)(t)$ are in $indep(t)$, so also in this case $U(F_B(B)(t))$ is bounded by $U(B_{bounds}(t))$. □

4.5 Experimental Evaluation

To show the scalability of the analysis approach presented in this chapter, we apply it to a large-scale industrial analysis problem, and compare the analysis results with results obtained by a static worst-case analysis. The application is from the same application domain as the case application discussed in Section 1.4, mapped to a multi-processor multi-core GPP platform. The analyzed DAG is a 6 degree of freedom digital control application that controls an imaging subsystem in a commercial wafer scanner. It consists of 2285 tasks mapped to a platform with three octo-core GPP processors, where the 8th core on each processor is reserved for other processing purposes. The model is calibrated with time measurements obtained by measuring block execution timing in isolation on the machine. Our analysis approach is used to estimate the contention on the shared cache.

The application DAG contains a set of dependencies that enforces a fixed-order schedule on each processor resource. Additionally, the DAG has lots of dependencies between tasks mapped to different cores. To model core-to-core (c2c) communications, which occur through shared L3 cache, 5377 c2c tasks are automatically added to the DAG and mapped to 3 (FCFS) resources, one for each processor. Each c2c task has an assumed execution interval of $[5, 5]$ns. The decorated DAG is analyzed using our interval analysis, and compared to static analysis that does not use enabled intervals to estimate contention dynamically. To this end, we use a best-case and a worst-case static analysis.

In both cases, for each task the interference of other tasks is estimated, which is then used as a measure to scale up the busy interval of that task correspondingly.

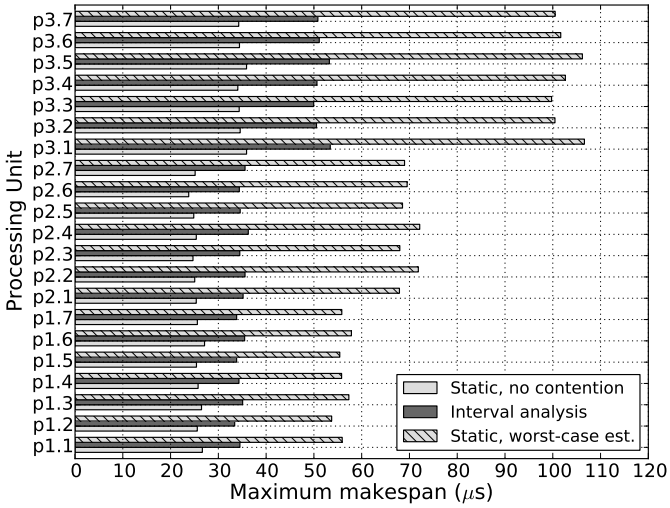


FIGURE 4.8: *Makespans of a DAG modeling an industrial-sized control application in a wafer scanner, mapped to 3 octo-core processors.*

When all tasks have been scaled, their resource binding is removed. By doing so, their busy intervals remain fixed, since there can be no further interference with other tasks. After scaling and re-mapping its tasks, the DAG can be analyzed with the interval analysis algorithm of Section 4.4, which then reduces to a single iteration of max-plus operations on the tasks in topological order. For best-case analysis, where no contention is assumed, each task $t \in T$ is initialized with $B(t) = E(t)$ and its resource binding is removed. For worst-case analysis, tasks are assumed to be interfering if interference can not be ruled out by the direct or indirect dependency relations between tasks.

Figure 4.8 shows the maximum makespan (the worst-case completion of the last scheduled task) for each core, obtained our by interval analysis, static analysis neglecting contention, and static analysis assuming worst-case contention. The model is analyzed in less than 10 seconds. The analysis results show that, on average, the worst-case makespan calculated with interval analysis is 46% lower compared to static worst-case analysis. On average, the makespans obtained with our analysis are 40.2% above the makespans where no contention is assumed. Part of this difference is caused by contention and part by over-estimation; exactly in which proportion is not easy to pinpoint. This experiment shows that our analysis can easily handle large-scale models in the order of thousands of contending tasks, and shows a significant improvement in the provided worst-case bounds compared to static worst-case analysis.

4.6 Related Work

SDF analysis is a well-known approach in the design and analysis of distributed real-time systems. SDF is commonly used in static timing analysis, where tasks are annotated with fixed execution times [72]. Many approaches focus on throughput analysis, either by computing the Maximum Cycle Mean [15] with structural analysis of the graph or by computing the time between the occurrence of equivalent token distributions through analysis of the dynamic execution state of the graph [28, 74]. The analysis technique in [27] computes the minimum achievable latency between any two actors in an SDF graph, in terms of their related firings, i.e. a destination actor firing that consumes at least one token of a specific firing of a source actor. However, these approaches are unable to model shared resources that employ runtime arbitration.

In [91], a resource-aware extension of SDF analysis is used in a design-space exploration technique to dimension multiprocessor systems. Shared resource accesses are explicitly modeled, and their impact is analyzed by considering the state-space of the different executions of an SDF graph. Even with fixed execution times, heuristics are needed to prevent a state-space explosion. The work in [85] considers HSDF graphs that are mapped to shared resources arbitrated by runtime arbiters in the class of latency-rate ($\mathcal{LR}$) servers (e.g. Round-Robin, TDM and priority-based with rate controller). Actors scheduled by these starvation-free $\mathcal{LR}$-servers are modeled as two vertices that model the access rate and latency. These approaches cannot analyze graphs consisting of actors with varying execution times that are scheduled on shared FCFS resources.

Real-Time Calculus (RTC) [76] models event streams by arrival curves, which bound the minimum and maximum number of events seen in a stream for any time window of some size (i.e. characterization in the time-interval domain). The event streams are related to service curves that denote the minimum and maximum amount of service available on a resource for any window of time. The resulting output streams are propagated to subsequent tasks. Since the RTC arrival curves are defined over *any* time window, information about absolute time is lost. Therefore, it cannot distinguish absolute time offsets in streams, resulting in pessimistic bounds when merging e.g. two strictly periodic streams that have a phase shift relative to each other [83]. Also, RTC assumes arrival curves are given. In our approach it is sufficient to know the minimum and maximum task execution times.

The SymTA/s approach [36] combines concepts from RTC and real-time scheduling [79] to calculate the worst-case response times of tasks. Task activations are characterized by a period and an enabling jitter. With local scheduling analysis (Round-Robin, TDM, Rate-Monotonic Scheduling), the worst-case response times of tasks are calculated. To this end, the maximum possible interference experienced by a task is determined based on the maximum number of activations of interfering tasks within the response time of a task. The results are propagated to subsequent system components in order to perform a system-level

analysis. In [70], the accuracy of this work is improved by taking into account the effect of pipelined processing and bursty event arrivals.

Similar to [76], these approaches use enabling characterization in the time-interval domain, and hence suffer from the same inaccuracy problems due to stream correlations that are not taken into account. Our approach also calculates response time bounds based on best-case and worst-case task interference, however, contrary to [36] we use interference estimates based on task enabling in the time-domain. This allows more accurate analysis of FCFS-scheduled resources.

The approach in [34] provides HSDF analysis for run-time schedulers that are not in the class of starvation-free schedulers. Similar to [36], task timing characterization comprises a period and enabling jitter, which are used to calculate task response times by analyzing the best-case and worst-case interference that tasks can experience. With the calculated response times, best-case and worst-case schedules are computed with which the enabling jitter is re-computed. The approach iterates until response times do not increase anymore. The resulting analysis flow broadens the scope of dataflow analysis to include non-starvation-free schedulers. In [33], this work is further extended with support for multi-rate dataflow graphs of any topology, including cyclic dependencies.

The FCFS scheduling we consider is within the class of starvation-free schedulers considered in [34], however, their response time calculation based on the maximum number of interfering activations in the time-interval domain is not suitable for FCFS scheduling, because FCFS interference depends on the enabling of tasks in absolute time. If these absolute timings are not known, the interference that a FCFS scheduled task experiences can only be calculated by assuming that all other tasks mapped to the same resource that have no dependency relation to this task are interfering. We employ a similar fixed-point iteration on response times, which enables analysis of systems with cyclic resource dependencies, but we consider best-case and worst-case task enabling bounds in absolute time.

Analysis methods based on model checking tools, like the popular UPPAAL [47] tool, use Timed-Automata (TA) [4] to verify timing properties [35]. The execution state-space of these TA is exhaustively searched to e.g. find worst-case timing properties. This results in very accurate bounds, but model checking inherently suffers from state-space explosion. Different approaches focus on heuristics and model-reductions to limit the state-space. In [90], UPPAAL is used to analyze the timing of a RapidIO network. By applying heuristics, somewhat larger models can be handled at the cost of accuracy, but analysis of the full system model under high traffic load is not possible without reverting to simulation techniques. The work of [29] combines model checking with Real-Time Calculus. Scalability of the TA-analysis is improved by abstracting part of the system under consideration into a single arrival curve.

4.7 Outlook

The current analysis is limited to Homogeneous Dataflow. An interesting development would be to extend the analysis to support full SDF semantics. This would require a more elaborate estimation of task interference (see Section 4.3), where possible multiple firings of interfering tasks are taken into account. To this end, the production intervals of individual tokens need to be tracked, in order to estimate different enabling intervals for different firings of the same task.

Another direction for further development of the analysis flow could be back-pressure modeling. Currently, the analysis can take into account the delay due to contention on shared resources, but not the additional delay due to back-pressure. Even though we can detect the occurrence of back-pressure (see Section 5.5.1), it would be interesting to extend the current analysis technique to also take into account this kind of delay.

4.8 Summary

This chapter presented a timing analysis technique with which the worst-case timing of an application mapped to a heterogeneous multiprocessor platform can be analyzed. The application is modeled by a Directed Acyclic Graph (DAG) with vertices that represent tasks and edges that represent data dependencies. Tasks are assigned an execution interval that expresses the best-case and worst-case bounds on their execution time. The DAG is assumed to be statically mapped to a platform consisting of resources that employ First-Come-First-Served (FCFS) arbitration, and follows HSDF execution semantics without any iteration overlap.

The chapter showed how to find task timing intervals that conservatively bound the task execution timing of any concrete execution of the modeled application. The iterative approach that was used is explained in detail, together with its main abstractions that improve scalability. Proofs showed that the iteration will converge on a fix-point if one exists, and that the task timing intervals found at that fix-point will be conservative for any possible execution of the graph. Finally, we related this analysis technique to techniques described in literature.

Chapter 5

Design Flow

This chapter discusses the details of the design flow introduced in Chapter 1 (see Figure 1.6). With an application, platform and mapping specification as inputs, the flow generates dataflow models with which the performance of an application mapping is analyzed, and it synthesizes the input models into a corresponding implementation that is able to meet the observed performance. A short overview is presented in Section 5.1, followed by the platform, mapping and application specification in Section 5.2. The steps that synthesize these abstract specifications into an RTL-level platform implementation and binary application code are discussed in Sections 5.3 and 5.4. Section 5.5 shows how to extract a timing model from a system specification and how to use it to obtain conservative timing estimates. After a discussion on related work in Section 5.6, the chapter concludes with an outlook and summary in Sections 5.7 and 5.8.

5.1 Methodology

The application domain, execution platform and timing analysis technique discussed in Chapters 2, 3 and 4 form the pillars of the design flow introduced in Chapter 1. For clarity, an overview of the design flow is shown again in Figure 5.1.

The box labeled *specification* shows the input for the design flow, consisting of a specification of the platform, application and mapping instances, and a set of timing constraints. These specifications can be automatically implemented, as shown in the two leftmost boxes labeled *platform synthesis* and *code generation*, or analyzed as shown in the middle box labeled *analysis*. The performance numbers found by timing analysis can be checked against the specified timing constraints, and can then be used to change the platform and/or mapping in order to search the design-space for a satisfactory solution. This is shown in the box labeled *DSE*.

The synthesis flow generates an FPGA implementation of the input specifications. To this end, it provides a hardware synthesis trajectory that generates HDL

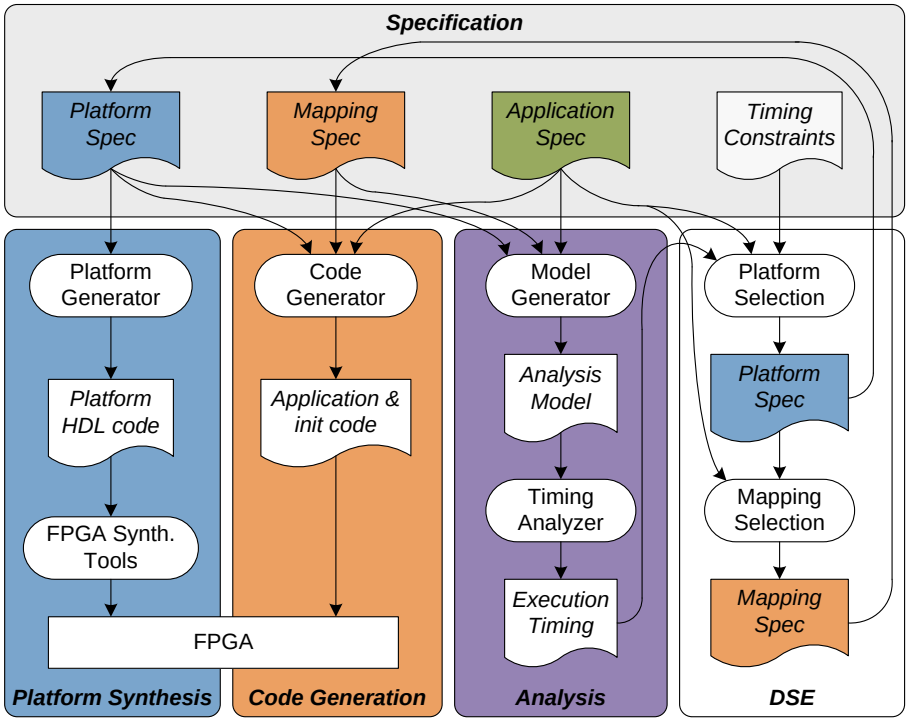


FIGURE 5.1: *Overview of the design flow introduced in Chapter 1.*

code of the platform, and a software synthesis trajectory that generates application binaries that run on the generated ASIP PUs together with configuration code to initialize them. The generated HDL can be synthesized into an FPGA bitstream by commercial FPGA vendor tools. The binary code is uploaded to the FPGA via a configuration interface that is connected to the memories and synchronization hardware of the PUs in the platform.

The analysis flow transforms the input specifications into a task graph that models the application bound to the specified platform instance, modeling both the computational delays on the PUs and the communication delays induced by the interconnect. The execution timing of this model can be analyzed using the timing analysis technique of Chapter 4, which takes into account additional contention delay on shared resources that employ FCFS arbitration. The task execution timings found by the analysis provide worst-case bounds that are guaranteed to be met by the synthesized implementation of the specification.

The DSE trajectory is currently mostly a manual process. Platform instances and mappings are chosen by hand, and task scheduling is either denoted implicitly in the mapping specification, or FCFS scheduling is performed at runtime. Auto-

mated scheduling approaches like the one in [2], which uses a due-date heuristic to schedule control applications onto a homogeneous multi-core multi-processor platform, can be applied to our platform as well, provided that the task bindings are fixed and known to the scheduler. If task bindings are not a-priori known, the approach of [2] needs extension to support heterogeneity before it can be used to schedule tasks on our platform.

The design flow is used in a case study that maps two digital control applications to instances of our multi-ASIP platform (see Chapter 6). The presented mapping approach iteratively selects platform instances with a smaller resource footprint while timing constraints can be met. These systematic steps, guided by simple heuristics, are amenable for automation. Together with the aforementioned automated scheduling technique, this could form the basis for an automated design-space exploration that, given an application specification, aims to find an efficient platform instance and mapping.

5.2 Specification

Following the Y-chart modeling paradigm [42], the input for the design flow of Figure 5.1 consists of a separate *platform*, *application* and *mapping* specification. The orthogonal input specifications of the Y-chart approach enable separation of concerns, allowing the designer to more efficiently explore and understand the impact of changes of either of the input models on the overall performance. Besides these three inputs, a set of timing constraints on the application is also part of the specification. The next subsections discuss the input specifications of the design flow in detail.

5.2.1 Platform Metamodel

Platforms can be instantiated by specifying their composition and dimensions in terms of the platform building blocks. These building blocks and their attributes are defined in a platform metamodel, which is shown in Figure 5.2.

The toplevel entity in the metamodel hierarchy is the *platform*, which consists of *platform resources* and *connections* that connect them together. Each *platform resource* can have *ports*, which act as an attachment point for a *connection* that relates the two connected ports. A *platform resource* can be either a *communication resource* or a *computation resource*. Memories are assumed to be part of the processing units, hence there are no explicit memory resources.

Communication resources are resources that provide facilities for communication. These resources do no execute code, and serve as communication pass-through devices. To this end, a *communication resource* has *forwarding rules*, that relate ports for routing and arbitration purposes.

At this moment, *switch* is the only specialization of *communication resource*, which comprises a simple switch unit that arbitrates its inputs to forward a single

FIGURE 5.2: *Platform meta-model.*

packet each clock cycle, and buffers the inputs which where not selected for output forwarding due to output contention. With the *schedulingPolicy* attribute, the switch arbitration can be set to FCFS or fixed-priority. The number of ports and buffer sizes can be set with the *nPorts*, *inBufSize* and *outBufSize* attributes.

Computation resources are resources that perform computations or conversion between internal and external communication. Application tasks can be mapped to these units, and hence they are the communication endpoints. Therefore, each *computation resource* has a *networkID* attribute, which is used to route packets through the platform interconnect. Since each *computation resource* has a dataflow synchronization unit (see Chapter 3) that triggers the execution of tasks mapped to it, it has attributes *maxInputs* and *maxTasks* that configure the addressing bits for this mechanism. Furthermore, since it executes code, it has an instruction memory, whose size is specified with the *iMemSize* attribute.

Even though *computation resources* have the same logical memory layout, in each specialization these memories can be mapped to different physical memories. The sizes and types of these physical memories are specified by corresponding attributes of these specializations. Currently, there are 4 specializations: the *rioIF* and the three ASIP types discussed in Chapter 3 (i.e. SPU, VPU and LU). The *rioIF* specialization converts between interconnect packets and RIO-packets, and provides an interface to a RIO network external to the platform instance via an implicit task running on this unit. The other specializations correspond to different ASIP types that execute signal processing tasks. They can have attributes for specific architectural features, like the *issueWidth* that sets the vector size for a *VPU* unit, and *typeIS* that selects whether or not a *VPU* also has a shiftadd unit besides its default inproduct unit.

5.2.2 Application and Mapping Specification

Figure 5.3 shows the application meta-model. An application instance consists of *tasks* and *connections*. Tasks have attributes that denote their *name* and *type*. The name attribute is unique to the task instance, and the type attribute matches an entry of a task implementation library (see Section 5.3). Tasks have zero or more *inputs* and zero or more *outputs*, through which they can communicate data if they are related by a *connection*. A connection relates a single task output to one or more task inputs.

A mapping specification relates tasks in an application specification to a computation resources in the platform specification. To this end, the simple meta-model shown in Figure 5.4 is used. A mapping consist of *bindings*, that relate exactly one *task* from the application model to exactly one *computation resource* from the platform model. The order in which the bindings of tasks to PUs are specified can be used to set fixed-order task schedules on these PUs.

Figure 5.5 shows an example application mapped to a platform instance, and the corresponding application and mapping models. The platform instance consists of two *SPU* PUs and a *rioIF* PU connected by a switch.

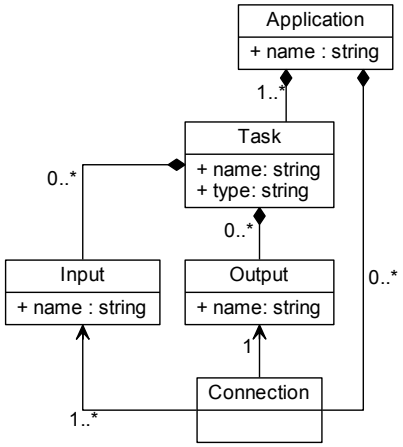


FIGURE 5.3: *Application meta-model.*

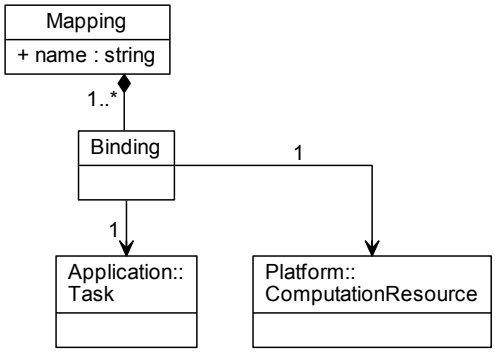
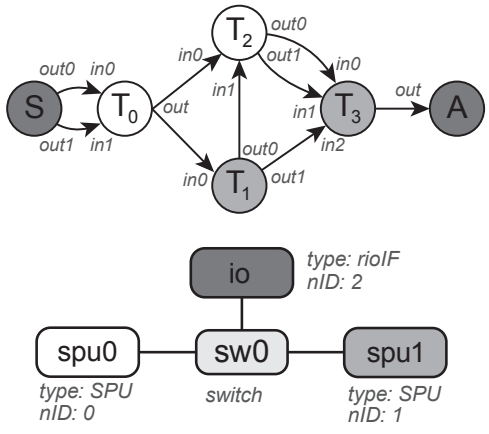


FIGURE 5.4: *Mapping meta-model.*

5.3 Platform Synthesis

With the metamodel discussed in Section 5.2.1, model instances can be created that specify a platform in terms of its connected building blocks. Such a platform model instance can be transformed into a corresponding VHDL [1, 5] implementa-



Application model

```
Task T0; Task T1; Task T2;
Task T3; Task S; Task A;
connect S out0 T0 in0;
connect S out1 T0 in1;
connect T0 out T1 in0;
connect T0 out T2 in0;
connect T1 out0 T2 in1;
connect T1 out1 T3 in2;
connect T2 out0 T3 in0;
connect T2 out1 T3 in1;
connect T3 out A in0;
```

Mapping model

```
spu0 {T0; T2};
spu1 {T1; T3};
io {S; A};
```

FIGURE 5.5: *Example application graph mapped to a platform instance (left). The application contains four tasks $T_0 \cdots T_3$ that interact with sensors represented by S and an actuator represented by A. Task input- and output names are denoted outside the task vertices, and vertex grey shading denotes task binding. The corresponding application and mapping models are shown on the right. Task names, except for S and A, match the names of their corresponding assembly templates.*

tion by the platform generator, as shown in the left of Figure 5.1. The generated VHDL implementation can be synthesized into an FPGA bit-file by generic vendor FPGA synthesis tools.

To transform a platform specification to a corresponding VHDL implementation, the platform generator performs the following transformation steps:

1. **Model parsing.** The platform specification is parsed and transformed into a datastructure consisting of a pair of lists containing the connections and the platform resources with their attributes.

2. **Component instantiation.** The platform resource list is traversed, and a model-to-text transformation is performed for each platform resource. The transformation generates VHDL code that instantiates a component, including the component generic map that is generated based on the values of the corresponding platform resource attributes from the platform instance model. The component port list is left blank, since it will be created in a subsequent transformation step. The types of each instantiated component matches an entry of a library of VHDL implementations of the platform resource types (switch, ASIP types, etcetera) defined in the metamodel (see previous section).

3. **Component connection.** After instantiation of all components, the plat-

form connection list is traversed. For each connection, a model-to-text transformation is applied that generates VHDL code that specifies the component port maps of the two components that the model connection connects. Also, corresponding signal declarations are generated.

4. **Top-level VHDL file creation.** The platform generator creates a toplevel VHDL file. First, the toplevel entity declaration is created, with the relevant external platform connections (memory-mapped configuration bus and optional external interfaces). Then, the toplevel architecture declaration is created with the signals that connect different components. Finally, all component instantiations with their generic maps and port maps are created.

5. **Switch configuration.** Finally, the routing tables for the interconnect switches are generated based on the forwarding rules specified in the platform model instance. For each switch a *mif*-file is generated that contains the data for the (static) switch routing table. The generated files are used by FPGA synthesis tools to initialize memory blocks.

The Eclipse Modeling Framework (EMF, [73]) is used to implement the metamodel defined in the previous section and the HDL code generator. In EMF, a metamodel can be defined in the Ecore-format, which is an UML-like class diagram with *classifiers* and their *associations*. Given an Ecore specification, EMF can automatically generate model code and editor code in Java.

The generated model code consists of a Java-class for each classifier in the Ecore diagram, with corresponding attributes and *get()* and *set()* accessors to these attributes. User code can be added to the generated code to implement additional functionality. User code (in Java) is added to implement the model to text transformations discussed earlier in this section.

The generated editor code can be loaded as an Eclipse environment plugin, where it implements a tree-like user interface to create model instances, with property views on the tree elements to edit their attributes. The plugin also provides facilities for model persistence and validation (w.r.t. conformance with the metamodel). Loading a model instance in Eclipse returns a reference to the toplevel model entity, a *Platform* in our case, which is then used to traverse the model elements to implement the five transformation steps explained earlier.

The metamodel and corresponding HDL code generation can be easily extended. For instance, the addition of a new ASIP type involves, after creating a parameterized VHDL implementation of the ASIP once, the following steps:

1. Addition of a new class in the Ecore diagram for the ASIP type. The class has a specialization relation with the *platform resource* class. Type-specific attributes are added to the class if needed (e.g. to specify the sizes of the physical memories present in the ASIP type).

2. Model code extension. The model code (Java) of the new classifier is extended with a function that performs the model-to-text transformation of

the classifier attributes.

The same steps are required for the addition of a new *communication resource* specialization. All other model transformation code that implements model-to-text transformations or code that controls model traversal remains unchanged.

The aim of the current implementation of the design flow is mainly to provide a proof of concept. Therefore, we have chosen to implement the platform synthesis in a pragmatic way by employing direct low-level model-to-text transformation in Java. Alternatively, the synthesis can be performed at a higher level of abstraction by standardized model-to-text formalisms implemented by tools like Acceleo [18], which are well integrated in the Eclipse environment. Also, due to the experimental state of the tools, no consistency checks on static semantics of input models have been implemented yet (that also holds for the code generator described in Section 5.4). These checks, that verify that e.g. certain attributes or relations are properly set in the input model, can be formalized by specifying a set of verification rules in the Object Constraint Language (OCL, [64]).

5.4 Code Generation

The design flow presented in this chapter constitutes two separate synthesis flows. The hardware synthesis flow, discussed in Section 5.3, consists of the elements in the box labeled *platform synthesis* in Figure 5.1. It generates platform HDL code based on a platform model. The software synthesis flow, consisting of the elements in the box labeled *code generation* in Figure 5.1. It generates the binary code that runs on the computational resources of the generated platform, and configuration code that e.g. initializes the dataflow synchronization units of these resources.

Both flows are similar, in the sense that they rely on the instantiation and configuration of pre-fabricated implementation templates to realize the implementation of the overall application or platform. This concept is known from the platform-based design paradigm [69]. Just like the platform generator, the code generator has access to an implementation library which it can use to instantiate and connect application elements. This application design library contains, for each task that can be instantiated, an assembly implementation for any of the PU types to which that particular task can be mapped to. The code generator assembles and links instances of these implementations in order to generate binary code for the PUs.

To this end it requires a task implementation library, an application specification, and a mapping specification as input. The code generator transforms these input specifications into binary PU code and platform configuration code as follows:

1. **Model parsing.** The platform, application and mapping specifications are parsed and transformed into a data structure *processorMap* that maps each computation resource to a list of tasks mapped to it. Also, based on the

connections in the application specification, a data structure *connectionMap* that maps each task output to a list of connected task inputs is created.

2. **Assembly template instantiation.** For each *processorMap* entry, the list of tasks mapped to that processor entry is retrieved. Each task in the list is assembled and partially linked. To this end, based on the type attribute of a task and the specific specialization of the computation resource, the corresponding task assembly template is loaded from the implementation library. The code generator assigns a task ID that is unique to the resource it is mapped to. This ID is used in conjunction with the processor network ID to address and synchronize task communication. After assigning the task ID, the assembly code of the task is converted to binary code, and its input, state, parameter and temporary memory addresses can be assigned. A data structure *inAddrMap* is created, that maps task input names to input addresses.

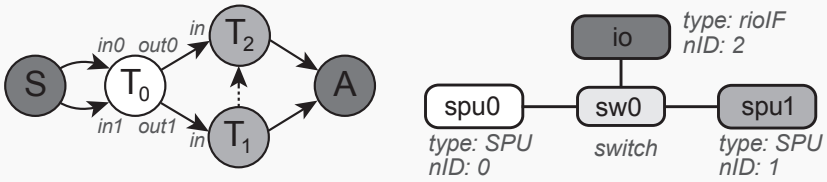
3. **Output address assignment.** After all tasks have been assigned an ID and input addresses for their input variables, the output variable addresses that refer to those inputs are assigned. To this end, the code generator updates the binary code and inserts output addresses based on *inAddrMap*, the destination task ID and the network ID of the destination processor. If a task has an output that is connected to multiple inputs, the corresponding instruction is copied, where each copy has an output address corresponding to the input it is connected to.

4. **Configuration data generation.** Finally, the configuration data for the platform is generated. This comprises instruction memory and parameter memory initialization for each PU in the platform, and initialization data for the dataflow synchronization units. The configuration data is organized into a stream of data that is to be put on the configuration bus of the FPGA that implements the platform.

Example 5.1 shows the generation of code for a small example application.

Example 5.1

Consider the application graph and platform instance below, where grey shading denotes task binding and the dashed edge denotes a scheduling edge.



Assume T_0 is implemented by the following SPU assembly code:

```
input in0, in1;
output out0, out1, out2;
param p0,p1;
temp t0, t1;
0 multinv in0, in0, t0;
7 mult in1, in1, t1;
0 clipsym t0, p0, out0;
0 clipsym t1, p1, out1;
```

This example will show how code is generated for T_0. To this end, the instantiated assembly template that belongs to T_0's implementation on an SPU-type processing unit is transformed into binary code.

The SPU has separate input, state/temp and parameter memories, which are selected by the two highest bits of the 16 bits addresses in the instruction. Their base addresses are 0x0, 0x4000 and 0x8000 respectively. Output variables have a base address of 0x0 for internal targets, and 0x8000 for remote targets. Input addresses are composed of a 5 bits task ID and a 3 bits input number. The opcodes for *mult*, *multinv* and *clipsym* are 0xB, 0x3 and 0xD respectively.

Since T_0 is the first (and only) task on *spu0*, it will be assigned task ID 0 and its variables are assigned the first free entries in their respective memories. As a result, the inputs *in0* and *in1* are assigned address 0x0 and 0x1 respectively, and the parameters *p0* and *p1* are assigned address 0x8000 and 0x8001. Temporaries are assigned addresses from the top of the temp/state memory downwards, since their addresses can be reused by any task mapped to *spu0*. Hence, *t0* and *t1* are assigned addresses 0x7FFF and 0x7FFE. The opcodes and the stall cycles are known from the assembly, and all internal addresses have been assigned, so at this point the first two instructions can be converted to binary. Using Figure 3.7, their binary values will be 0xB0000000007FFF and 0x37000100017FFE. The other two instructions require the input addresses of *spu1* to be known, lest their Y-operands can be set.

Similarly, on *spu1*, tasks T_1 and T_2 are assigned task IDs 0 and 1 respectively. Their inputs are assigned address 0x0 and 0x8 respectively. The other address assignments for *spu1* are irrelevant for this example.

Now that all input addresses of all tasks have been assigned, the *clipsym* instructions on *spu0* can be converted to binary. The Y-operand contains the remote input address and 4 bits of the remote nID. The cid field contains the remaining four bits of the remote nID. The first *clipsym* then transforms into 0xD07FFF80008108. Similarly, the last *clipsym*, which gets its stop-bit set since it is the last instruction of the task, transforms into 0x10007FFE80010100.

5.5 Analysis

The analysis trajectory that is part of the design flow, consists of the elements in the third box in Figure 5.1, labeled *Analysis*. The *model generator* extracts a Directed Acyclic Task Graph (DAG) model from the application specification. With additional input from the platform and mapping specifications, this unbound and untimed graph can be transformed into a timed and bound graph that models the execution timing of the application mapped to the platform instance. The bound graph can be analyzed by the *timing analyzer*, which applies the analysis technique of Chapter 4 to obtain worst-case time bounds of all tasks in the graph taking into account any contention on shared resources.

5.5.1 Modeling Execution Timing

Initially, the *model generator* transforms the application specification into a DAG. To this end, each task in the application specification results in a task node in the DAG, and each connection in the specification results in a dependency in the DAG. Contrary to connections, which relate task outputs to task inputs, the dependencies in the DAG relate source tasks to destination tasks. The resulting DAG is untimed and unbound.

The mapping specification provides task binding information. When tasks are bound to a resource, an execution time can be assigned to them based on a library of task execution times. This timing library can be directly derived from the task implementation library used by the code generator. With additional platform information, the graph can be transformed to model the execution time of the application running on the specified platform instance. The next subsections explain this transformation in detail.

Task Execution Timing

The execution time of a task running on a particular processing unit (PU) depends on the number of instructions to be executed for that task and the pipeline length of the PU it is mapped to. All PUs available in our platform template are pipelined and fetch a single instruction per clock cycle. Therefore, the execution timing of a task consists of a delay d_f that corresponds to the fetching of all instructions of a task, and a delay d_p that corresponds to the propagation of the last instruction of that task through the execution pipeline of the PU. When all instructions of a task have been fetched, the instruction fetching of a different task can already start, i.e. for two tasks t_1 and t_2 executed on the same resource, d_f of t_2 can overlap with d_p of t_1, provided that t_1 and t_2 are independent. If not, t_2 can start only after all instructions of t_1 that produce data that t_2 consumes have been propagated through the execution pipeline.

To transform an untimed task graph into a timed graph that models the execution timing resulting from tasks bound to PUs, each task in the untimed

(a) Untimed task t_1.

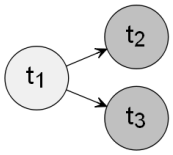
(b) Two timed actors representing the execution timing of t_1 when it is bound to a resource.

FIGURE 5.6: *When transforming an untimed task graph into a timed graph that models the execution timing of tasks bound to PUs, each untimed task in (a) is substituted for two timed actors t_f and t_p (b) that represent the instruction fetching and pipeline delays of that task. These actors get assigned execution intervals d_f and d_p respectively. Grey shades denotes resource binding (white tasks are not bound to a resource).*

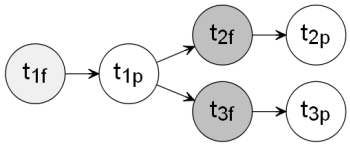
graph can be substituted with two dependent actors t_f and t_p, which are assigned an execution time corresponding to delays d_f and d_p respectively. Since d_f of different tasks mapped to the same resource can not overlap, while d_f and d_p of these tasks can potentially overlap, actor t_f remains bound to a resource in the timed model and actor t_p is not bound to any resource. This is shown in Figure 5.6.

The execution delays d_f and d_p can be derived from the specifications of Section 5.2 and the task implementation library used by the code generator (discussed in Section 5.4). The task implementation library contains for each application task a corresponding assembly implementation for any of the PU types it can be mapped to. Because the PUs do not support conditional branches, and the number of loop iterations and stall cycles are explicitly specified in the assembly code (see Chapter 3), the exact number of executed instructions when running a task on a particular resource is statically known. Therefore, if task t is bound to a resource r, the execution delay d_f of substitute task t_f can be directly derived from the assembly code corresponding to the implementation of t on a resource r, provided that the cycle time of that resource is known. The execution delay d_p for substitute task t_p can be derived from the cycle time and number of pipeline stages of r, which are specified in the platform specification.

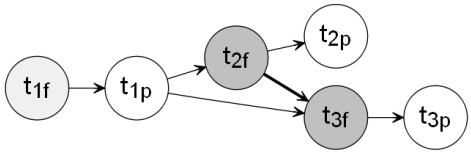
The analysis technique of Chapter 4 operates on task execution time intervals, which specify both the best-case and worst-case execution time of a task, or their conservative estimates. For a pipeline delay task t_p, the best-case and worst-case execution time are always the same, i.e. its execution interval is given by $[d_p, d_p]$. However, this is not the case for an instruction fetch task t_f. Any communicated output from t_f could potentially enable some dependent task, even though t_f has not completed executing yet. Therefore, a lower bound on the best-case completion time of t_f is the moment at which it produces its first output. Accordingly, t_f is assigned a best-case execution time $d_{f,min}$ that corresponds to the delay between the first instruction of the application task modeled by t_f and the first instruction of this task that writes to an output variable, independent



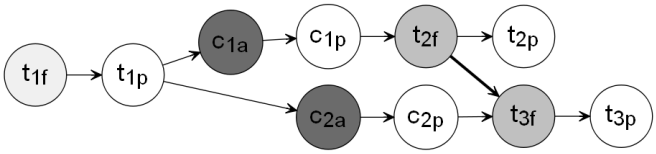
(a) Initial untimed graph.



(b) Timed graph that models the timing of tasks in (a) bound to PUs.



(c) An additional dependency models a fixed-order schedule $t_2 \rightarrow t_3$ on the mid-grey resource.



(d) Two additional tasks c_{1a} and c_{1p} model the communication delay between t_1 and t_2. Accordingly, tasks c_{2a} and c_{2p} model the communication delay between t_1 and t_3 respectively.

FIGURE 5.7: *Transformation of an untimed task graph into a timed graph that models the execution timing of an application mapped to a platform instance. Grey shades denote resource binding (white tasks are unbound).*

whether that output is mapped to a remote task or not. The worst-case execution time $d_{f,max}$ of t_f is the delay between the first and the last instruction of the application task modeled by t_f.

Figure 5.7 shows the steps that transform an initially untimed application task graph (Figure 5.7(a)) into a timed graph that models the execution timing of the application mapped to a platform instance (Figure 5.7(d)), and the modeling of PU-bindings is shown in Figure 5.7(b). The other steps are explained in the next subsections.

Static Order Schedules

When multiple tasks are bound to the same processing unit, their execution order can be enforced by a static order schedule. These static order processor schedules can be modeled by adding dependencies between tasks mapped to the same processing unit, as shown in Figure 5.7(c). Here, the static-order schedule $t_2 \rightarrow t_3$ is modeled by adding a dependency between t_{2f} and t_{3f}.

Static-order schedules can be specified in the mapping specification, where the order of appearance of tasks denotes the execution order. This can be done either manually, or by automated scheduling techniques like the one in [2]. Alternatively, the implicit execution order of the mapping specification can be ignored, to let the PU hardware perform task scheduling. In that case, the tasks are executed in a First-Come-First-Served order, and no additional scheduling edges are generated in the analysis model.

Task Communication

Tasks bound to the same PU communicate directly through shared memory. The corresponding delay is already modeled in the task execution time. Tasks that are bound to different PUs (denoted as remote tasks) communicate via the interconnect network that connects the different PUs. In such a case, a source task injects its output value as a packet into the interconnect network, where it arrives at the destination task after one or more hops through the interconnected switches. At each hop, the packet competes for switch output access with possibly other arriving packets. Currently, our switches employ FCFS arbitration. After arbitration, a packet is either forwarded to an output, or buffered at the switch input if other packets are granted access first. Each switch is pipelined, so after arbitration a packet has to traverse several pipeline stages before appearing at the switch output, while the switch can already process other access requests. Similar to modeling task execution timing, the execution timing of a packet passing a switch can be modeled by two tasks t_a and t_p, where t_a is bound to the switch resource and models the switch access of the packet, and t_p is not bound to any resource, depends on t_a, and models the packet traversing the switch pipeline.

To model communication delay over the interconnect, the edges in a task graph that connect remote tasks are substituted by a cascade of these t_a and t_p substitute pairs, one pair for each network hop between the source and destination PUs. Figure 5.7(d) shows the transformed graph that models the communication between remote tasks t_1 and t_2 and between remote tasks t_1 and t_3, assuming a single switch connects the light-grey and mid-grey resource.

Interconnect switches in our platform template use programmable routing tables that are specified in the platform specification, so the packet routes between PUs are statically known. Hence, the number of cascaded substitute pairs for each edge between remote tasks is known at compile time. Also, the access delay, clock cycle time and number of pipeline stages of each switch are known at compile time, which can be used to determine the execution times of tasks t_a and t_p for each substitute pair. For tasks t_a and t_p, the lower bound is equal to the upper bound of their respective execution time intervals, since switch access latency and switch pipeline delay do not vary.

Networks that have switches with a finite amount of buffer space can exhibit a phenomenon called back-pressure. This occurs when the input buffer of some switch in the network is full. The switch is then unable to accept new data from

the blocked input, so packets in the network that are routed through this blocked input pile up until the blocking input buffer has sufficient space again. Our timing analysis approach is able to take into account the delay due to contention on the network, but not the additional delay caused by back-pressure. Therefore, our analysis is conservative under assumption of the absence of any back-pressure.

With our analysis, it is possible to analyze whether switch buffers are sufficiently large to guarantee that no back-pressure will occur. By counting the number of potentially simultaneously enabled tasks at any moment in time for each switch resource and comparing the maximum count of each switch to its buffer size, the possible occurrence of back-pressure can be detected. To this end, one use two sorted queues Q_{BC} and Q_{WC} containing the best-case and worst-case enabling times of tasks mapped to a switch. The heads of both queues are checked, popping the lowest value from either one of the queues. For each value popped from Q_{BC}, a counter is increased, and for each value popped from Q_{BC}, the counter is decreased. The maximum observed counter value for should then be lesser than or equal to the buffer size of the corresponding switch to guarantee that this particular switch is not causing back-pressure. If this does not hold, either an alternative mapping and scheduling should be selected, buffer sizes should be increased, or an alternative static network routing should be selected in order to prevent the occurrence of back-pressure.

5.5.2 Timing Analysis

Given the bound and interval-timed DAG that is constructed based on the application, mapping and platform specifications, the *timing analyzer* (see Figure 5.1) analyzes the execution timing of this graph, taking into account any possible contention on both communication and computation resources.

As a result of the analysis, each task in the graph is labeled with an enabling interval and a completion interval, which conservatively bound the enabling and completion of that task. These worst-case completion times can be checked against timing constraints on the IO-delay and makespan of the original application.

The assumption in this analysis is that there is no back-pressure in the interconnect network. The occurrence of back-pressure can be detected with our analysis (see Section 5.5.1). Subsequently, either an alternative mapping or schedule can be chosen, or buffer sizes of some switches can be increased in order to prevent back-pressure.

5.6 Related Work

Simulink [56] is the de-facto design tool for (digital) control applications. It provides a rich library of instantiable and configurable components, and a rigid simulation framework with which many different functional aspects of a design can be tuned and verified. Simulink also provides tools to automatically transform

its model instances into a (partial) implementation. HDL Coder [55] generates VHDL or Verilog HDL code from a Simulink model. The generated HDL code constitutes a spatial mapping of the transformed block onto FPGA logic, with parameterizable area/speed optimization. Simulink Coder [54] generates C-code from a Simulink model, which can be executed with or without a real-time operating system on an embedded target. The generated code is single-processor code, and thus requires further integration when used in a multiprocessor environment. Simulink lacks a clear distributed semantics as well as timing analysis and verification tools, which are essential in the design of resource-efficient implementations that will always meet timing constraints.

Many academic approaches focus on drawing non-functional properties like timing early into the design flow. One such approach [16] presents SysWeaver, an automated model-driven environment that enables analysis and multi-node code generation for signal processing applications. Here, a Simulink model that serves as functional input for SysWeaver is enhanced with non-functional properties, according to which multi-node code is generated by gluing together code generated by Simulink Coder.

Our design flow follows the same ideology regarding the use of Simulink models as starting point, and adding timing verification and deployment to the functional model in order to synthesize a distributed implementation. Both approaches rely on dataflow timing analysis to provide mapping and and timing verification. However, where SysWeaver provides software synthesis targeting general purpose platforms only, our approach also provides hardware synthesis, and targets instances of specialized platforms in FPGA. Finally, SysWeaver implements synchronization in software, while our approach suffers less overhead by employing hardware synchronization.

Several other approaches enforce the synchronous distributed semantics of (a discrete subset of) Simulink applications by mapping them to a Time-Triggered Architecture (TTA) [43]. Such architectures rely on the availability of a single global clock, that is typically distributed over a network that connects the different computational nodes. In [65], time-triggered semantics are added to Simulink models by executing them on virtual machines that implement the time-triggered task execution and timed network transfers. In [11], a Simulink model is transformed into a SCADE/Lustre model prior to mapping it to a TTA platform. An MPSoC-based approach that uses a TTA-like synchronization is presented in [66]. Cores start executing after receiving a global heartbeat signal. After executing all computational tasks, the different PUs can communication over the NoC. The communicated data is available at the destination after the next heartbeat. The work in [37] makes the TTA paradigm explicitly visible to the programmer by means of a TTA-oriented programming language, while [49] takes this approach a step further by employing a timing-aware programming language, operating system and processor architecture to map distributed real-time applications.

In our approach, tasks synchronize on the availability of their input data. This results in less overhead compared to the TTA approach, since TTA has to

dimension its triggering for worst-case arrival of data. Also, it may be difficult to synchronize a global clock. Typical TTA platforms consist of general-purpose processors or DSPs connected by some network with a protocol that supports TTA. Our platform provides much more specialization in its processing units, and an interconnect with a much lower latency, giving our approach a big advantage for large-scale low-latency applications such as motion controllers in lithography machines. The approach of [49] reduces jitter partially by delaying execution paths or enforcing round-robin thread-interleaved pipelining without bypassing, whereas in our approach we achieve predictability by employing large register files and omitting conditional execution from the PU instruction sets. We allow unpredictability in the interconnect, and take that into account in our analysis.

There exist many academic design flows that focus on the distributed mapping of (real-time) streaming applications to (heterogeneous) Multi-Processor System-on-Chips (MPSoCs). The design flow presented in [39] combines the analysis and exploration facilities of the SDF3 [75] dataflow analysis tool set with the MAMPS [44] multi-application system synthesis flow. The result is a platform-based design framework that provides automated DSE, throughput and buffer size analysis, and synthesis of homogeneous MPSoC platforms. MAMPSx [20] extends this design flow by adding heterogeneity and a flexible C-HEAP based communication assist that decouples communication and computation.

While the model of computation used in MAPSx is similar to that used in our flow, the analysis techniques in SDF3 and MAMPSx are throughput-oriented. Since latency is not the main optimization goal for MAMPSx, it can afford to employ a high-level C-HEAP based synchronization approach, where node to node communication latencies exceeding $2\mu s$ [20] are not uncommon. Furthermore, the analysis techniques in SDF3 cannot deal with contention on shared resources with FCFS scheduling, while we can compute bounds on such contention. Finally, the MAMPSx platform template differs from ours, targeting a combination of general-purpose processing and fully specialized processing with hardware IPs, while our template targets a heterogeneous mix of specialized but programmable cores.

Daedalus [78, 62] is a platform-based design framework similar to MAMPS, that enables automated system-level design-space exploration and synthesis targeting heterogeneous MPSoCs in FPGA. A Kahn Process Network [40] specification is used as application input for the design flow, either specified directly or automatically transformed from a sequential C-program (with restrictions)by the KPNgen tool. The SESAME architecture exploration tool automatically generates a platform and mapping specification based on this input specification. DaedalusRT [8], a hard real-time adaptation of Daedalus, trades the automated DSE for a platform dimensioning approach based on automated schedulability analysis. To this end, a Cyclo-static SDF model is derived from the KPN specification in order to perform this analysis. With ESPAM, a Daedalus(RT) system specification can be automatically synthesized into an RTL implementation targeting FPGA. Synthesized platforms consist of programmable cores connected to hardware IP blocks, which communicate through FIFO-buffers in shared memory

such that the application respects the Kahn process synchronization semantics.

Our design flow operates on application input that follows the (H)SDF model of computation, which is less expressive but better analyzable compared to KPN. As a result, with HSDF it is possible to calculate processor schedules at compile time whereas KPN requires run-time mechanisms for synchronization and scheduling. Since DaedalusRT uses a CSDF model in its analysis, it is able to calculate processor schedules at compile time. Both Daedalus and DaedalusRT rely on software implementation of their synchronization mechanisms, whereas our approach performs synchronization in hardware and (optionally) static-order scheduling in software. The synthesis flows of both approaches are very similar, they both rely on the instantiation and connection of library components to compose a platform instance expressed in VHDL. The employed platform templates are quite different, however. Our platform instances are compositions of different specialized programmable cores, whereas Daedalus(RT) maps to a template consisting of programmable general-purpose cores and hardware IP co-processors.

SystemCoDesigner [32] is another platform-based design methodology that provides automatic design-space exploration, performance evaluation and system synthesis. The input for the flow is an actor-oriented application specification, expressed in SysteMoc, a synthesizable subset of SystemC. The behaviour of the application actors is controlled by Finite State Machines, with which the (H)SDF, CSDF and KPN models of computation are realized. The flow can perform automated DSE to find a Pareto front in terms of latency, throughput and resource footprint. The performance feedback used for this DSE comes from analytical models for resource utilization, and simulation for latency and throughput. SystemCoDesigner supports automated mapping to a platform template consisting of Microblaze processors and hardware IP blocks, connected by an interconnect that supports FIFO communication (either a buffer or a bus with synchronization at source and sink nodes).

SystemCoDesigner has a very expressive input MoC which also supports the HSDF MoC that serves as input for our flow. However, contrary to our efficient hardware task synchronization, SystemCoDesigner implements task synchronization in software. SystemCoDesigner relies on simulation to get timing performance feedback in its DSE, which is more time-consuming compared to our analytical approach. Also, simulation is unlikely to cover all possible executions of the simulated model. Finally, the platform template used by SystemCoDesigner differs from the one used in our approach. SystemCoDesigner employs either generic RISC processors or fully specialized hardware blocks, while our template consists of ASIPs that are in between those two specialization extremes.

5.7 Outlook

In this prototype version of our design flow, a purely structural application specification is used with an implicit correspondence to a behavioral specification re-

siding in an implementation library. This behavioral task specification comprises a hand-made assembly template for each PU type to which the task type can be mapped to. In a further maturation step, these templates should be generated directly or indirectly from a Simulink application.

Generating these ASIP assembly templates from e.g. C-code generated with Simulink Coder [54] is difficult, since the generated C-code can already contain implicit architecture-specific assumptions. For example, a clipping operation is typically expressed as an if-then-else construction in C, even though such an operation does not necessarily have to be implemented with branches. For a compiler or translation tool, it is very hard to recognize and resolve such implicit assumptions. Therefore, the input for such a translation should be the Simulink model itself, or an implementation-independent high-level DSL. Future work could encompass this automation trajectory and the design of a implementation-independent DSL that allows for efficient compilation to different ASIP types.

Another improvement point is the automation of the design-space exploration part of the flow. Currently, the exploration steps and processor scheduling are performed by hand. In the next chapter, we present several systematic steps that could be used as foundation for an automatic mapping flow. Future work could focus on finding and comparing different heuristics to find an efficient platform instance and mapping given an application specification and timing constraints.

5.8 Summary

This chapter discussed the design flow, briefly introduced in Chapter 1, which maps application instances onto the platform presented in Chapter 3. Following the Y-chart approach, the design flow uses three separate input specifications that model the application, platform and mapping. A set of timing constraints on the application is part of the input specification as well.

The different input specifications were explained in detail, as well as the hardware and software synthesis steps that synthesize them into an implementation. The hardware synthesis flow uses a structural platform specification and parameterized behavioral RTL-specifications of the different structural elements to generate a complete platform implementation on RTL-level, which can be mapped onto FPGA technology by commercial FPGA tools. Similarly, the software synthesis flow generates binary code by instantiating and connecting assembly implementations of tasks according to the structural specification of the application.

The design flow provides an analysis trajectory, that generates a timed Directed Acyclic Task Graph that models the execution timing of the corresponding input specifications. The generated model can be analyzed using the analysis technique of Chapter 4, taking into account contention on shared resources. The resulting execution timings provide worst-case bounds, that are guaranteed to be met by the synthesized implementation, provided that the interconnect network does not suffer back-pressure. The occurrence of back-pressure can be detected

with our analysis techniques.

Performance numbers obtained with timing analysis can be checked against the timing constraints that are part of the input. Based on this comparison, the platform or mapping specification can be adapted in order to perform a structured design space exploration that aims to find a satisfactory solution in terms of timing performance and resource footprint. These steps are currently still performed manually. Chapter 6 and related work provide an outlook on automation of this part of the design flow.

Chapter 6

Case Study

This chapter presents a case study in which the industrial control application discussed in Chapter 1 is mapped to an instance of the platform template presented in Chapter 3 using the design flow from Chapter 5. The goal of this case study is to, given an application task graph and a set of timing constraints, find an efficient platform instance and mapping that meet all application timing requirements. To this end, platform instances and task bindings are selected, and the task graph is transformed to reflect these bindings and the corresponding timings. Subsequently, the resulting task graph is analyzed with the technique discussed in Chapter 4 to verify its timing. Following a simple set of heuristics, an efficient platform instance and mapping that are able to meet application timing constraints are found in a systematic way.

The next section shows the application task graph and the timing constraints that are the input for this case study. Section 6.2 gives an overview of the mapping approach, followed by the concrete mapping and platform selection process in Section 6.3. The chapter concludes with a discussion on the results in Section 6.4 and a summary in Section 6.5.

6.1 Application Task Graph

Figure 6.1 shows the task graph of the case application introduced in Chapter 1. The graph consists of 145 tasks, ranging from simple additions to large state-space blocks with more than 200 states. It models the signal processing part of two connected 6 DoF motion controllers that control the positioning of wafers in a wafer scanner. The left side of the graph is formed by tasks belonging to the Long-Stroke (LS) controller, and the right side of the graph consists of Short-Stroke (SS) controller tasks. The grey nodes in the graph denote the interaction points with the plant and supervisory controller. The task graph is provided with sensor input samples through node S. Reference input samples, generated

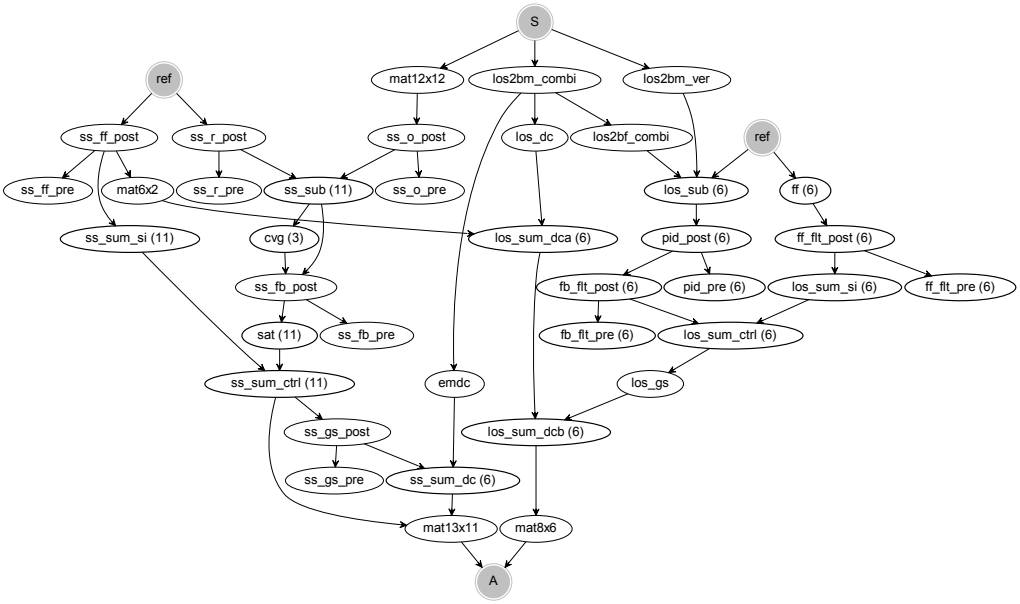


FIGURE 6.1: *Task graph model of the case application.*

TABLE 6.1: *Timing constraints on the task graph of Figure 6.1.*

Task	Type	Completion time
all tasks	throughput	$10\mu s$
mat8x6	latency	$3.5\mu s$
mat13x11	latency	$3.5\mu s$

by a setpoint generator, enter the graph through the *ref* nodes, and actuator output samples leave the graph through node A. The S, *ref* and A nodes are not part of the application considered in this case, but are shown to emphasize the top-to-bottom flow in the acyclic task graph.

6.1.1 Timing Constraints

Table 6.1 shows the timing constraints on the execution of the task graph of Figure 6.1. The application is intended to run at 100kHz, so assuming single-rate execution starting at time instant 0, there is a completion time constraint of $10\mu s$ on each task in the graph. The IO-delay is constrained to $10\mu s$. Assuming a total delay of $6.5\mu s$ for sensing, actuation and moving the data between FPGA and IO-boards, the controller is required to calculate an output sample within $3.5\mu s$. Hence, tasks *mat8x6* and *mat13x11* have a completion time constraint of $3.5\mu s$.

6.2 Approach

The decision problem of finding an optimal mapping and scheduling of a given task graph onto a given platform instance without a-priori binding information is known to be NP-complete [2]. In this case study however, both the platform instance and the task bindings are not a-priori known, and are part of the optimization problem. Now, the decision problem complicates to finding a mapping, task scheduling and platform instance that will require a minimal resource footprint while resulting in a minimal IO-delay and maximum throughput.

Due to the high complexity of this decision problem, we will not follow an analytic approach to find a minimal platform instance and application mapping, but instead we use heuristics to find a platform instance that is efficient in terms of the number and size of processing units. Such an approach can not be guaranteed to find an optimal solution, but shows better scalability in terms of the number of exploration steps. The steps taken in our approach are systematic and follow simple heuristics, so the process is amenable for automation.

In order to find a suitable platform instance and mapping, we start with a platform instance that is unconstrained in terms of FPGA resources. This allows the selection of a platform instance, mapping and processor schedules that enable the exploitation of all task-level and data-level parallelism present in the application. In subsequent steps, based on timing analysis and the structure of the task graph, this platform instance will be reduced by trading timing slack for increased resource sharing. To this end, processor schedules of processing units of the same type are merged if that does not violate a timing constraint. Consequently, processing units with an empty binding are then removed from the platform instance.

At each step, when task bindings and processor schedules have been selected, a dataflow model that models the execution timing of the bound task graph is constructed (see Section 5.5). With dataflow analysis, the starting times and completion times of the tasks can be computed and visualized in a Gantt-chart. In the first few steps, the task communication delay will be ignored. The reason to do so is twofold: since the starting point is a maximally parallel platform with hundreds of PUs, the huge amount of network communication in the first few platform instances will not at all be representative for the amount of network communication in the final solution. Secondly, the design space for an interconnect network that connects hundreds of PUs is huge. The effort of selecting a network is huge, while the resulting network delay will not provide any additional insight that is useful for the next iteration of platform selection. Therefore, we will first reduce the computational resource footprint of the platform using traditional dataflow analysis. When a suitable platform is found that cannot be reduced any further, an interconnect network is added. Finally, with the interval analysis technique presented in Chapter 4, the worst-case timing of the selected platform and mapping can be verified against the set of timing constraints.

The approach of reducing a maximally parallel platform has the advantage

that the process requires only steps that merge processor schedules, and that subsequent timing analysis provides direct feedback on the feasibility of a mapping decision. If in some step the choice to bind a task to a specific PU results in the violation of a timing constraint, it is immediately clear that any further exploration based on this mapping choice cannot result in a feasible solution. Only a single backtracking step prunes that part of the search space, and returns the search process to a feasible solution in terms of timing. From there, the search for a feasible solution with lower resource requirements can be continued.

Since the opposite approach, where a minimal platform is expanded until the timing constraints are met, starts from an infeasible solution, it lacks direct feedback on the eventual timing feasibility during intermediate steps. Such an approach would therefore need to move tasks from a shared resource to a new resource until the timing constraints are met. Subsequently, excess resources can be removed by trying to merge task bindings again. The number of steps required to find a feasible solution would therefore larger for such an approach.

6.3 Platform and Mapping Exploration

Initially, a platform instance with unlimited resources is assumed, as explained in Section 6.2. With this platform, a mapping can be chosen that maximally exploits the available task-level and data-level parallelism, by mapping each task to a private instance of its best mapping target in terms of execution time. If there is a tie on minimum execution time in the selection of a mapping target, the PU type with the lowest resource footprint is chosen.

After binding each task of the task graph of Figure 6.1 to a resource and selecting a static-order schedule for each resource (best-effort, by hand), the graph is transformed and analyzed with the techniques explained in Section 5.5 and Chapter 4. Task execution times are calculated from their corresponding assembly code, assuming a 200Mhz FPGA design. Figure 6.2 shows a Gantt-chart of the execution timing of this initial mapping.

The vertical axis shows the instantiated resources (without annotation) and the horizontal axis denotes time. Each bar represents the instruction fetching period of a single task, and grey shading denotes the type of resource the corresponding task is bound to (see Section 5.5.1). For clarity, only the timing bars of the instruction fetching part of the execution time are shown, the pipeline delay bars are omitted. The initial mapping uses 7 LU-type PUs (dark grey), 112 SPU-type PUs (white) and 26 VPU-type PUs with issue widths between 8 and 64 (light grey).

The two encircled bars near the bottom of the figure correspond to the two tasks with a latency constraint on their completion, *mat8x6* (top) and *mat13x11* (bottom). With this mapping, they complete well within their budgets of $1.490\mu s$ and $1.620\mu s$ respectively. The application makespan is dominated by the execution of 4 specific tasks near the bottom, which correspond to the state-space

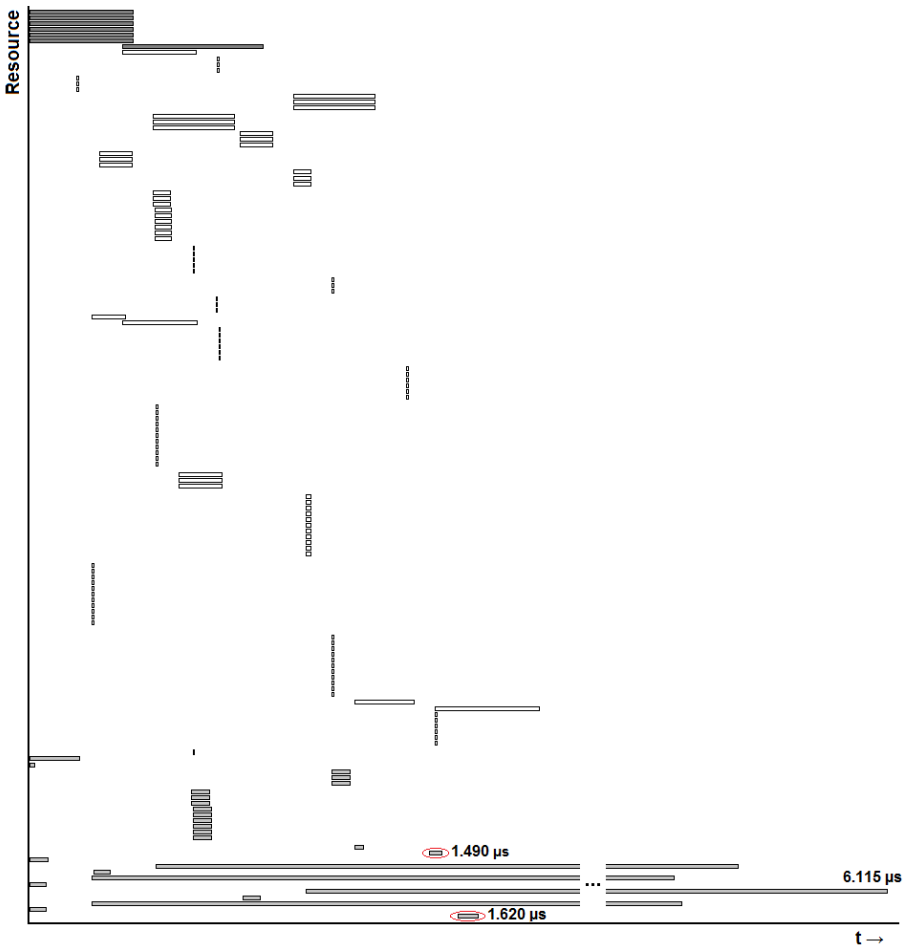


FIGURE 6.2: *Gantt chart showing the execution timing of the initial mapping.*

precalculation tasks of the SS controller. The last state-space precalculation task completes at 6.115μs, which is also well within its budget of 10μs.

6.3.1 Merging Critical Task Bindings

Performance analysis on the mapping chosen in the previous sections shows that there is considerable slack in the completion time of *mat8x6* and *mat13x11*. This slack will be used to reduce the resource footprint of the current platform instance, by increasing the amount of resource sharing. To this end, first the critical tasks will be redeployed to a smaller set of resources where possible, as long as it does not violate any latency constraints. Then, the remaining IO-delay slack and/or

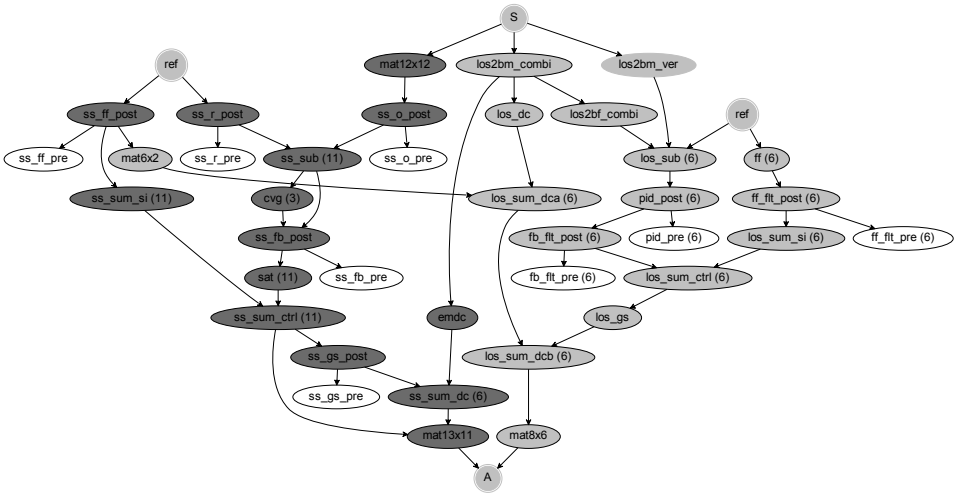


FIGURE 6.3: *Critical tasks in the task graph model. Critical LS tasks are colored light grey, and critical SS tasks are colored dark grey. The uncolored tasks are non-critical.*

remaining throughput slack can be used to redeploy the non-critical tasks to a smaller set of resources.

Since the sets of critical tasks of the LS and SS controller have little overlap, the redeployment will be done first for the LS controller, and then for the SS controller. Starting at the LS sink task *mat8x6* of the application graph, all tasks that belong to the LS that are reachable by only traversing backwards through incoming dependencies are critical for IO-delay, and thus eligible for redeployment in this merging step. Figure 6.3 shows the critical LS tasks (light grey) and critical SS tasks (dark grey).

PU schedules with critical LS tasks that run on the same PU type are merged when possible, removing any PU with an empty schedule from the platform instance. Figure 6.4 shows the task execution timing after merging the critical LS schedules. The makespan and IO-delay of the SS have not changed, but the completion time of *mat8x6* has been almost doubled to $2.950\mu s$. The resource requirements have been reduced to 2 LU (-5), 62 SPU (-50), 13 VPU8 (-3), 6 VPU16 (-0) and 4 VPU64 (-0) processing units.

Similarly, merging the task bindings of the critical SS tasks results in the execution timing shown in Figure 6.5. Now, the makespan has increased to $6.395\mu s$, because the state-space postcalculation tasks and the three *cvg* tasks have been sequentialized by redeployment, so that the state-space precalculation task that dominates the application makespan is started later. The completion of *mat13x11* has increased to $2.030\mu s$ while the completion time of *mat8x6* remains unchanged.

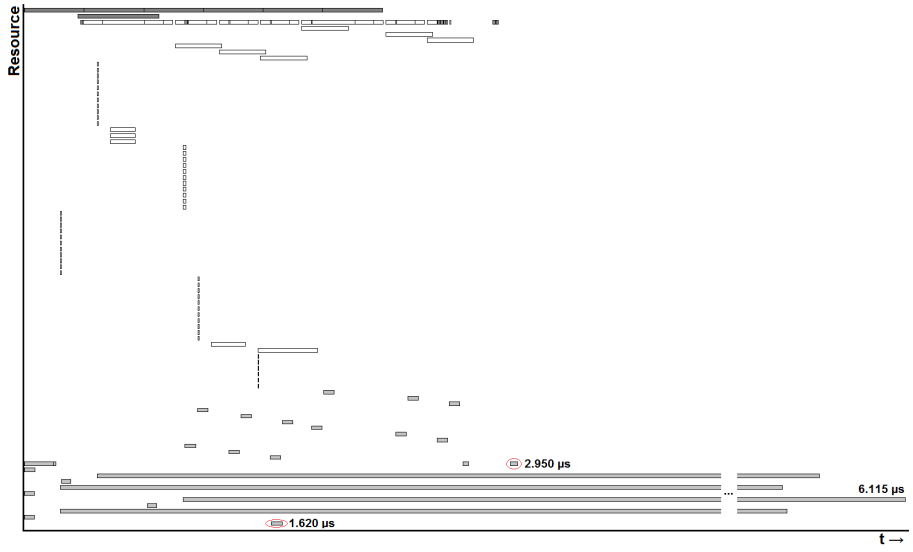


FIGURE 6.4: *Execution timing after merging the task bindings of critical LS tasks.*

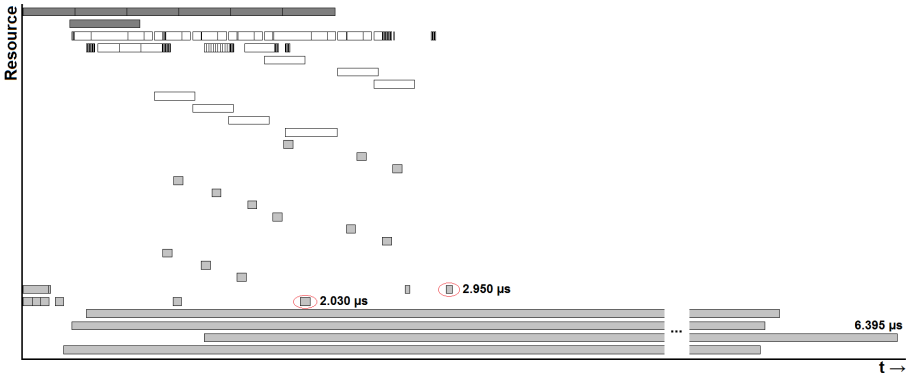


FIGURE 6.5: *Execution timing after merging the task bindings of critical SS tasks.*

The resource footprint after this merging step has been reduced to 2 LU (-0), 9 SPU (-53), 13 VPU8 (-0), 1 VPU16 (-5) and 4 VUE64 (-0) units.

The processor schedules with critical tasks have been merged for the LS and the SS controller separately. In the next step, the critical schedules of both the LS and the SS are merged where possible. At this point, the only critical PU schedules that can be merged without violating the latency constraints are the two LU schedules (top two resources in Figure 6.5). Figure 6.6 shows the execution timing after redeploying the task running at the bottom LU to the the top LU, where it is executed at the end of the schedule.

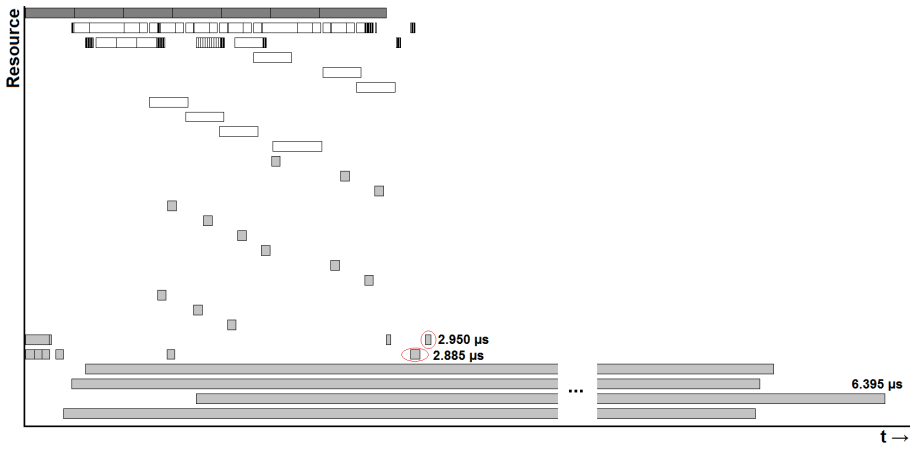


FIGURE 6.6: *Execution timing after merging the task bindings of critical LS and SS tasks.*

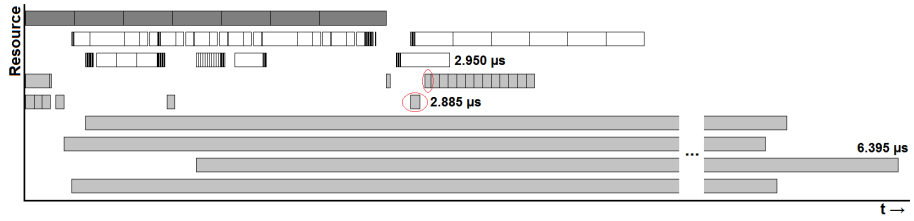


FIGURE 6.7: *Execution timing after merging the task bindings of non-critical tasks.*

The IO-delay tasks *mat8x6* and *mat13x11* now complete at 2.950μs and 2.885μs respectively, and the last completing task completes at 6.395μs. The resource footprint has reduced to 1 LU (-1), 9 SPU (-0), 13 VPU8 (-0), 1 VPU16 (-0) and 4 VPU64 (-0) units.

6.3.2 Merging Non-critical Task Bindings

Now that all critical tasks have been sequentialized where possible, the non-critical tasks are redeployed. Non-critical tasks can be appended at the end of schedules with critical tasks until the throughput constraint is violated, or they can be scheduled in possible timing gaps before critical tasks. Figure 6.7 shows the execution timing after redeploying the non-critical tasks where possible.

The resource requirements have reduced to 1 LU (-0), 2 SPU (-7), 1 VPU8 (-12), 1 VPU16 (-0) and 4 VPU64 (-0) units, and the completion times of the last completing task and the two IO-delay sink tasks remain unchanged at 6.395μs,

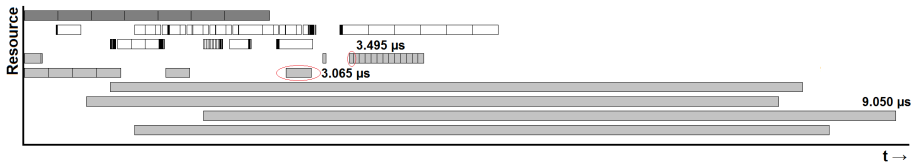


FIGURE 6.8: *Execution timing after reducing the issue width of vector processing units.*

2.950μs and 2.885μs respectively. No schedules can be merged anymore. However, there is still sufficient timing slack to further reduce the resource footprint.

6.3.3 Final Merging Steps

At this point, all schedules mapped to the same resource type have been merged where possible. It is not possible to merge any schedules on resources of the same type any more, since that would either violate the IO-delay constraints (if the SPU schedules were merged), or the throughput constraint (if any of the VPU64 schedules were merged). There is still 3.605μs slack time between the last completing task and the throughput constraint, and also the IO-delay is still well within budget. Therefore, this remaining timing slack is used to reduce the issue width of vector units where possible.

There is sufficient slack for the four VPU64 units (bottom 4 vector-type resources in Figure 6.7) to be reduced to four VPU32 units. Also, the VPU16 (second vector-type resource in Figure 6.7) can be reduced to a VPU8. The resulting execution timing is shown in Figure 6.8.

As a result of reducing the issue widths of these vector units, the resource footprint has reduced to 1LU (-0), 2 SPU (-0), 2 VPU8 (+1), 0 VPU16 (-1), 4 VPU32 (+4) and 0 VPU64 (-4) units, at the cost of an increased maximum makespan and IO-delay of 9.050μs, 3.495μs and 3.065μs respectively. Note that the IO-delay of *mat8x6* is now extremely close to the deadline of 3.500μs, while we do not yet include additional delays due to inter-processor communication. So even though this current solution likely is an infeasible solution if the exploration would stop at this point, we still allow this step since we know that in the immediate subsequent step the IO-delay of *mat8x6* will be reduced by parallelization of several sequential tasks that contribute to its delay.

The reduction of the VPU16 into a VPU8 presents a new candidate for schedule merging. However, it is not possible to completely merge the schedules of the two remaining VPU8 units, since this would result in *mat8x6* violating its completion time constraint. The tasks executing at the beginning the two VPU8 schedules form a sequential bottleneck that prevent a full merging. The other tasks mapped to these units are executing at disjunct time periods, and can thus be redeployed without timing impact. Also, there is a significant idle period on

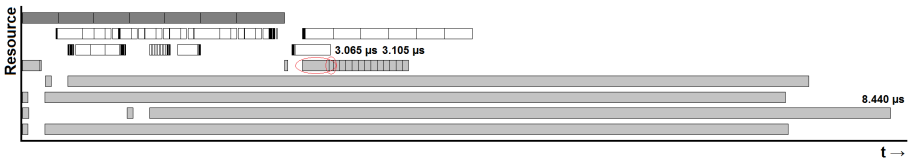


FIGURE 6.9: *Execution timing after parallelizing some critical tasks by redeploying them to existing vector units which are wider than required.*

each of the four VPU32 units prior to the start of the statespace precalculations.

Therefore, in the last resource reduction step, the first four tasks in the schedule of the bottom VPU8 of Figure 6.8, are redeployed to the four VPU32 units. This is inefficient in terms of memory, since now a full 32-wide vector is used for each originally 8-wide vector. However, the tasks that are moved are not very memory-intensive, and this redeployment allows for the remaining tasks in the VPU8 schedule to be merged with the other VPU8 schedule. The resulting removal of a complete VPU8 unit compensates the inefficient memory use of the four tasks moved to the VPU32 units.

The execution timing resulting from this final resource reduction steps is shown in Figure 6.9. The resource footprint has been reduced to 1 LU (-0), 2 SPU (-0), 1 VPU8 (-1) and 4 VPU32 (-0) units, and the resulting makespan and IO-delays are $8.440\mu s$, $3.105\mu s$ and $3.065\mu s$ respectively. Even though all tasks still complete well within budget, it is not possible to reduce the current platform instance any further by merging schedules.

6.3.4 Adding Communication Delay

Now that the platform instance is reduced as much as possible, the inter-task communication is taken into account. To this end, an interconnect network is created by connecting the processing units together by one or more switches. Since the platform is reduced to only 8 processing units, and the application consists of two controllers that are loosely coupled, an interconnect consisting of 2 switches is chosen, and connected as shown in Figure 6.10.

More generally, an interconnect network can be composed by applying some heuristics that e.g. tries to reduce the number of hops per data transfer while trying to keep the load on the switches low. To this end, PUs that communicate frequently are connected to the same switch, with a parameterized upper limit on the number of connected PUs per switch. The resulting disconnected clusters can then be connected by connecting the most frequently communicating clusters to the same switch, again applying an upper limit to the number of connected clusters. This can be repeated until all clusters are connected, forming a tree-like hierarchical network with most traffic concentrated at the leaves of the tree.

The left switch is connected to processing units that mostly have LS tasks

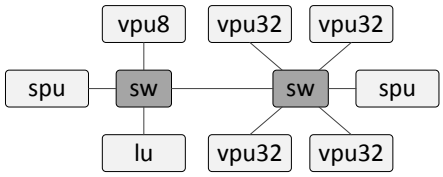


FIGURE 6.10: *Final platform instance with two interconnect switches.*

deployed on them, and the right switch is connected to processing units that execute SS tasks. In this way, most communication can occur in a single hop, and the interference between LS and SS communication remains limited.

The additional delay due to task communication can now be modeled by substituting communication edges between tasks mapped to different processing units with a two-actor model that represents the switch access time and switch pipeline delay for each traversed switch, as explained in Section 5.5.1. Applying the analysis method, which was explained in Chapter 4, on the resulting model shows that the worst-case maximum makespan and IO-delays are $8.695\mu s$, $3.275\mu s$ and $3.235\mu s$ respectively. Taking into account the communication delay raises the makespan by 3% and the completion times of the IO-delay tasks by approximately 5.5%. Hence, the assumption that communication time is small compared to the computation time, based on which we decided to neglect communication delay until the very last step, proved to be valid for this case.

6.3.5 Results Overview

Table 6.2 summarizes the resource requirements and execution timings resulting from the different mapping steps followed in this section. With the presented platform reduction process, starting with a maximally parallel platform, a resource-efficient platform instance and mapping are found with which the application can be executed well within its timing budgets. The final platform instance obtained by following a set of simple heuristics cannot be reduced any further by applying more resource sharing, since merging the schedules of any of the current processing units would result in the violation of a timing constraint. So for this particular application case and its constraints a platform with minimal resource footprint is found, however, this is of course not generally the case.

As shown in Section 1.4, the performance requirements of this case application can not be met by typical GPP platforms used in industry. Even in a best-case estimation where communication and contention are neglected, the same application running on a dual octo-core GPP processor can not meet its requirements. At best, using an overly optimistic estimation, a sample frequency of 40 kHz and an IO-delay of 3.949 μs can be achieved.

Step 8 in Table 6.2 shows the worst-case achievable performance on the corre-

TABLE 6.2: *Resource requirements and their corresponding performance for the mapping steps shown in Section 6.3.*

Step	**Resources**			**Completion time** [μs]		
	spu	*vpu*	*lu*	*all*	*mat8x6*	*mat13x11*
1. initial	112	26	7	6.115	1.490	1.620
2. merge LS critical	62	23	2	6.115	2.950	1.620
3. merge SS critical	9	18	2	6.395	2.950	2.030
4. merge critical	9	18	2	6.395	2.950	2.885
5. merge non-critical	2	6	2	6.395	2.950	2.885
6. reduce vector widths	2	6	1	9.050	3.495	3.065
7. final redeployment	2	5	1	8.440	3.105	3.065
8. add communication	2	5	1	8.695	3.275	3.235
Timing Constraint [μs]	-	-	-	*10.000*	*3.500*	*3.500*

sponding platform instance selected from our platform template. The implementation is guaranteed to meet the observed performance, since worst-case behaviour including contention is taken into account. With the final selected platform instance, the IO-delay constraint of 3.500 μs is easily met, with a time slack of 0.225 μs or more. The control application can be executed at the intended 100 kHz, while the last completing task even has a time slack of 2.305 μs with respect to its deadline.

This case study shows that with our platform template it is possible to instantiate platforms that are able to meet the strict low-latency performance requirements of industrial high-performance controllers, without doing any concessions on the mechatronic performance of the application. We have shown that, given the application model and its performance requirements, we can find an efficient platform instance in a few simple steps.

6.4 Outlook

Since the exploration steps followed in Section 6.3 are simple, and the approach to reduce a maximally parallel mapping ensures direct feedback on the impact of redeployment choices, the presented systematic approach is amenable for automation. The assumption that the timing contribution of communication is relatively small compared to the computation time is valid for this case application. The last two rows in Table 6.2 show that the worst-case completion time difference between a mapping that does not take communication into account and one that does so is less than 0.3μs for the three tasks that determine throughput and IO-delay.

The choice to take communication into account only at the last step does not result in an infeasible solution in this last step. Not taking into account communication delays in the early reduction steps simplifies platform selection, since the interconnect structure does not have to be instantiated. However, it

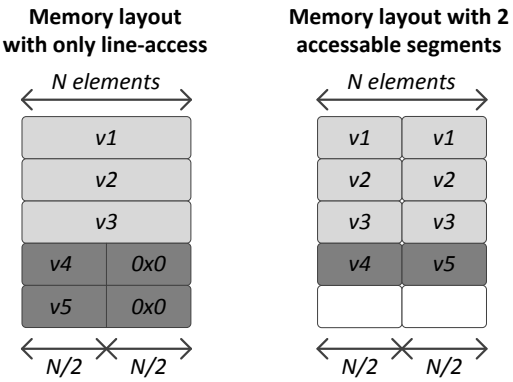


FIGURE 6.11: *Memory layout of two tasks that use a different vector width mapped to the same vector processor. Vectors v1, v2 and v3 belong to some task t_1, and their width is equal to the vector width of the vector memory. Vectors v4 and v5 belong to some task t_2, and their width is only half the width of the memory. Segmented memory access allows packing of vectors smaller than the memory size (shown on the right), which prevents inefficient memory utilization due to zero-padding when only line-accesses are allowed (shown on the left).*

does require sufficient remaining time slack after the final merging step in order to still meet the timing requirements when finally the communication delay is added. If in the last step it turns out that the added communication delay results in a timing constraint violation, it is hard to determine which part of the visited design space to prune in order to return to a point that is likely to eventually result in a feasible platform. On the other hand, keeping too much slack will result in an overly conservative platform reduction. Finding a good rule of thumb for the amount of time slack to keep is therefore an important factor in the search for a feasible and efficient solution.

Especially the last few steps of the platform reduction show several aspects that give rise to considerations for the future development of the hardware platform:

- **Pre-emption support.** First, consider the execution timing shown in Figure 6.9. The top vector unit (a VPU8) has a few relatively short tasks mapped to it, and is otherwise mostly idling. The two critical tasks at the beginning of its schedule can be redeployed to the VPU32 that has an idle period at the start of its schedule. Also, the non-critical tasks mapped to the VPU8 can be redeployed to all four VPU32 units, where they can be appended at the end of the schedules. However, the remaining critical tasks on the VPU8 unit cannot be redeployed, since their enabling and deadline are coinciding with the long execution period of the statespace precalculation tasks on each of the VPU32 units. These precalculation tasks running

on the VPU32 cannot be postponed until after the completion of these critical VPU8 tasks, since that would result in violation of the throughput constraint.

By adding preemption support to the VPU hardware, the precalculation tasks on the VPU32 can be interrupted to execute the critical tasks originally mapped to the VPU8. Consequently, all tasks of the VPU8 can be redeployed to the four VPU32 units, and the VPU8 unit can then be removed from the platform instance. This can be realized with a simple static priority-based preemption system, where critical tasks get a higher priority than the non-critical tasks. The processing units employ FCFS scheduling, so with priority based preemption, a VPU can run a critical schedule in conjunction with a separate non-critical schedule. However, the interval analysis technique of Chapter 4 cannot handle preemption yet.

- **Segmented vector memory access.** The vector units now have a fixed granularity, i.e. they only support data widths that are equal to their issue width at instantiation. There are multiple occasions in this case study where a task that runs optimally on a 8-wide vector unit is redeployed to a wider vector unit like a VPU32. Besides the additional pipeline delay of 2 cycles for each doubling of the vector issue width, the execution time of the redeployed task will remain the same. However, where each vector was 8 wide in the original task, now a 32-wide vector is used. Since the other 24 entries of the vector remain unused, such redeployments are very memory inefficient.

  For a more efficient mapping of these smaller tasks, it could be beneficial to have support for segmented vector memory access, i.e. to allow the vector memory of wide vector units to be accessed in segments of e.g. 8 entries. In this way, a wide vector memory line can be shared by multiple smaller tasks, as shown in Figure 6.11. This can be achieved by using a bitmask on the write enable of the vector memory, and a bypass from one of the last stages of the addertree to allow each segment to calculate an 8-wide inproduct separately. For vector units that support the *shiftadd*-instructions, the shift unit needs to be changed such that it can also handle one separate *shiftadd*-instruction per segment, or alternatively, the shift unit could be made less wide than the issue width of the vector unit. In such a case, only tasks that are mapped to a specific segment can use the *shiftadd*-instructions. This saves a lot of resources, since typically tasks that require *shiftadd*-instructions are used for updating the state of filters or PID-controllers, and are therefore not very wide.

- **Exploiting subword parallelism on vector units.** One step further would be to enable the parallel execution of multiple smaller tasks on a wide vector unit, similar to the exploitation of subword-level parallelism in e.g. Intel SSE instructions on x86 platforms. Full subword parallelism

support would be too complex to fit into the current architecture and its instruction format. However, a restricted form which allows the parallel execution of multiple instances of the same task type could be both beneficial and feasible. Since these instances have, except for operand values, exactly the same instruction sequence, it is possible to execute them in an SIMD-way.

6.5 Summary

In this chapter, a given task graph that models the industrial control application discussed in Chapter 1 was mapped to an instance of the platform discussed in Chapter 3. Given two latency constraints and a throughput constraint, a suitable platform instance was to be found. Assuming a platform with infinite resources as a starting point, each task in the task graph was mapped to a private instance of its best mapping target in terms of execution time. This maximally parallel mapping was then subjected to resource sharing as long as the timing constraints are met. In a few structured steps, the resource requirements of the platform instance were reduced as much as possible by merging the schedules of processing units of the same type, and reducing the width of vector units where possible. The communication delay was ignored until the last step.

The platform instance that was found after the last resource sharing steps consists of 2 SPU, 1 VPU8, 4 VPU32 and 1 LU units, connected together by two switches. In the worst case, taking possible communication contention into account, the last task in the graph will complete within $8.695\mu s$. The two tasks that are sink nodes of the critical path of the application complete within $3.275\mu s$ and $3.235\mu s$. The deadlines for these tasks were $10\mu s$, $3.5\mu s$ and $3.5\mu s$ respectively.

The mapping choices made in the last steps of the platform reduction, and the execution timing of the final mapping showed that, with some architecture changes, there was room for further improvement and a more efficient mapping. These architecture changes include static priority-based preemption, segmented memory access and restricted sub-word parallelism support for wide vector units.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The enormous momentum of the digital revolution can only be maintained if the semiconductor industry is able to keep up with the fast development pace set by Moore's law. To this end, it is paramount that the developments in photolithography technology consistently provide the industry with better IC fabrication tools that enable a steady shrink of transistor size. This requires more sophisticated controllers that can better control the physical processes in these IC machines, which already operate at nanometer scale accuracy at accelerations in the order of tens of G's.

One of the problematic trends in the development of these controllers is the growing scale and complexity of the control applications that implement them, especially in combination with the increasingly strict real-time requirements on their execution. Two important metrics in this context are sampling rate (i.e. application throughput) and IO-delay (i.e. the latency of the execution of a subset of the application that is more performance-critical). Chapter 1 argued that the current trend of using general-purpose hardware to execute these applications is likely not a viable solution for future high-performance control applications. With a case application that represents a candidate next-generation high-performance digital controller from industry, we showed by a calibrated timing analysis that it is not possible to meet the target 100 kHz sampling frequency and 10 μs IO-delay. As an alternative approach, we have investigated whether a heterogeneous platform tailored to the application domain is better able to address these strict real-time performance requirements, and we have addressed the issue of programming such an architecuture.

Chapter 2 discussed the main characteristics of digital control applications. Even though the execution timing of such applications is paramount for their functional performance, they are typically expressed as untimed declarative mod-

els (in the z-domain). We have argued that typical digital controllers can be modeled by a Directed Acyclic Graph without iteration overlap that follows Homogeneous Synchronous Data Flow semantics. The placement of initial tokens in the HSDF graph can be derived from the z-domain model and actors can be assigned an execution time based on the piece of atomically executed code they represent. The timed operational semantics of the HSDF model extracted from the untimed controller model allow one to reason on distributed controller execution, and to analyze the timing of its distributed execution. Furthermore, this model can also be used as input specification for code generation when implementing the corresponding controller.

An HSDF graph modeling a digital controller can subsequently be mapped to an instance of the heterogeneous platform template presented in Chapter 3. The presented platform is tailored to high-performance digital control applications, facilitating low-latency execution of typical controller tasks. The template consists of a set of ASIP processing units that are connected by a low-latency interconnect. The processing units all have the same flat distributed memory architecture, and are efficiently synchronized in hardware according to the HSDF semantics. Different processing unit types are tuned towards different application task types, with a limited specialization compared to e.g. ASICs in order to maintain flexibility.

Chapter 4 presented an analysis technique that calculates the worst-case performance of a DAG mapped to a set of connected resources that employ First-Come-First-Served arbitration. The input for the analysis is a DAG with tasks that are labeled with an execution interval denoting their best-case and worst-case execution time, and a set of task bindings that associate each task in the graph to a resource. The analysis takes into account the contention on shared resources by employing an iterative approach that is guaranteed to find a fix-point on the timing intervals of the graph. The result of the analysis is a set of timing intervals that conservatively bound the timing of each task in the DAG. With this analysis we can calculate the worst-case timing of a digital controller mapped to instances of our platform.

With the platform-based design-flow presented in Chapter 5 we automatically generate analysis models from a set of models that specify the application, platform and mapping. After analysis has shown that a chosen platform instance and mapping will yield satisfactory results, the input models can be automatically synthesized into an implementation that is guaranteed to be able to meet the worst-case performance found with the analysis. The platform model is automatically synthesized into an RTL implementation, and application and configuration code is generated from the application and mapping models. The high abstraction level of the input models, their corresponding implementation gap being covered by automated synthesis, and the conservative worst-case analysis enables extensive tailoring of platform and application.

This is exemplified in Chapter 6, which discusses a manual design-space exploration that maps the industrial application case of Chapter 1 to an instance of the platform of Chapter 3 using the design flow of Chapter 5. With the followed

approach, an over-dimensioned platform that enables full application parallelism is reduced into an efficient platform instance that is able to meet the application timing constraints. The presented steps are simple and systematic, and hence amenable for automation.

This thesis has shown that a platform-based approach that enables easy mapping of digital controllers to instances of a heterogeneous platform template is a promising direction for future digital controllers. The combination of multiple specialized processing units that can communicate with very low-latency enables a good exploitation of the available task-level and data-level parallelism. The use of an FPGA-based template allows the platform to be tailored to the application, and the conservative worst-case analysis and automated implementation offered by the supporting design-flow ensure that the performance constraints are met in any possible execution of the controller.

7.2 Future Work

Besides the recommendations for future work discussed in the chapter outlook sections, we identify several more generic directions for future work:

- Currently, our approach only supports single-rate controllers, i.e. all controllers mapped to a platform instance run at the same sampling frequency. However, some controllers are less time-critical than others (e.g. the Long-Stroke could be executed at a lower sampling rate than the Short-Stroke). Support for multi-rate control could improve the flexibility and resource efficiency of our approach. A follow-up study could investigate how to support multi-rate control, and how to model and analyze it. As mentioned in the outlook of Chapter 4, one possible future research direction for our analysis technique is to extend it with support for full SDF semantics. In line with this direction, future research could focus on the question how SDF can be used to model, analyze and execute multi-rate controllers.

  With the current HSDF synchronization semantics, the hardware used to synchronize PUs only needs to keep track of the total number of received tokens for each task mapped to a PU in order to determine which tasks are enabled. With support for multi-rate controllers, the production and consumption rates can be different for each actor channel. Hence, to determine which tasks are enabled, the number of received tokens on each channel should be tracked separately instead. To do so for all synchronization units on all PUs would require considerably more logic and memory. Alternatively, restrictions like mapping tasks with different rates only to some designated PUs with more elaborate synchronization support could reduce additional hardware costs. However, this would reduce flexibility and could negatively affect performance. An in-depth exploration of such trade-offs could be another direction for future research.

- The case study of Chapter 6 showed that priority-based pre-emption support could be useful to e.g. start the execution of a non-critical task in the idle time of some resource that has some critical tasks mapped to it that have not been processed yet. Currently, non-critical tasks are scheduled at the end of the fixed-order processor schedules, so that they cannot hinder the critical tasks, which are scheduled at the beginning of the schedule. If critical tasks could pre-empt any non-critical tasks, some non-critical tasks could be scheduled earlier and be executed in the idle time periods between consecutively scheduled critical tasks.

  Support for pre-emption requires changes both the hardware and the analysis method. The performance analysis should be extended to take into account the timing effects of mixed FCFS and priority-based scheduling. To this end, the contention model of Section 4.3 should be adopted to take into account the interference tasks can experience by getting pre-empted by high-priority tasks. Existing work [34] provides a good basis for further exploration in such a direction.

  Assuming two priority levels (critical and non-critical), extending the platform of Chapter 3 with support for fixed-priority pre-emption requires an additional execution queue for the high-priority tasks, logic that handles the queue selection and interruption, and a register to store the current instruction address of an interrupted low-priority task. The interruption itself simply encompasses storing the instruction address of the interrupted task and loading an enabled task from the high-priority execution queue. When no high-priority tasks is available in the high-priority tasks queue, an interrupted low-priority task can be either resumed or started from the low-priority queue.

- The application case used throughout this thesis consists of two controllers that are heterogeneous and compute-intensive. Even though these controllers are some of the most performance-critical controllers in a lithography machine, there are many more interesting candidate-controllers for our approach. For example, the FlexRay^{TM}programmable illumination technology in ASML lithography machines handles pupil-shaping of the light-beam by controlling an array of thousands of individually adjustable micro-mirrors. Such a controller consists of a large set of very similar mirror controllers. An interesting follow-up to this work would be to perform a case-study similar to the one in Chapter 6, targeting this particular controller.

Bibliography

[1] IEEE standard VHDL language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages 1–640, Jan 2009.

[2] S. Adyanthaya, M. Geilen, T. Basten, R. Schiffelers, B. Theelen, and J. Voeten. Fast multiprocessor scheduling with fixed task binding of large scale industrial cyber physical systems. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 979–988, Sept 2013.

[3] Altera. Floating-point Megafunctions user guide, Nov 2011. $http://www.altera.com/literature/ug/ug\_altfp\_mfug.pdf$.

[4] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183 – 235, 1994.

[5] P. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., 2nd edition, 2001. ISBN: 1558606742.

[6] ASML. `http://www.asml.com`. Accessed: December 2014.

[7] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010.

[8] M. Bamakhrama, J. Zhai, H. Nikolov, and T. Stefanov. A methodology for automated design of hard-real-time embedded streaming systems. In *Design, Automation & Test in Europe Conference Exhibition (DATE), 2012*, pages 941–946, March 2012.

[9] N. Bourbaki. Sur le théorème de Zorn. *Archiv der Mathematik*, 2(6):434–437, 1949.

[10] H. Butler. Position control in lithographic equipment [applications of control]. *Control Systems, IEEE*, 31(5):28 –47, Oct 2011.

[11] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: A layered approach for distributed embedded applications. In *Proceedings of the 2003 ACM SIGPLAN Conference*

on Language, Compiler, and Tool for Embedded Systems, LCTES '03, pages 153–162. ACM, 2003.

[12] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K. Arzen. How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. *Control Systems, IEEE*, 23(3):16 –30, June 2003.

[13] Y. Chan, M. Moallem, and W. Wang. Design and implementation of modular fpga-based pid controllers. *IEEE Transactions on Industrial Electronics*, 54(4):1898–1906, Aug 2007.

[14] R. Cumplido, S. Jones, R. Goodall, and S. Bateman. A high-performance processor for embedded real-time control. *Control Systems Technology, IEEE Transactions on*, 13:485–492, May 2005.

[15] A. Dasdan and R. Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(10):889–899, Oct 1998.

[16] D. de Niz, G. Bhatia, and R. Rajkumar. Model-based development of embedded systems: The sysweaver approach. In *Real-Time and Embedded Technology and Applications Symposium, Proceedings of the 12th IEEE*, pages 231–242, April 2006.

[17] K. Diefendorff, P. Dubey, R. Hochsprung, and H. Scale. AltiVec extension to PowerPC accelerates media processing. *Micro, IEEE*, 20(2):85–95, March 2000.

[18] Eclipse. Acceleo. `http://www.eclipse.org/acceleo/`, 2014.

[19] J. Eyre and J. Bier. The evolution of DSP processors. *IEEE Signal Processing Magazine*, 17(2):43–51, 2000.

[20] S. Fernando, F. Siyoum, Y. He, A. Kumar, and H. Corporaal. MAMPSx: A design framework for rapid synthesis of predictable heterogeneous MPSoCs. In *Rapid System Prototyping (RSP), 2013 International Symposium on*, pages 136–142, Oct 2013.

[21] G. Franklin, D. Powell, and A. Emami-Naeini. *Feedback Control of Dynamic Systems*. Prentice Hall PTR, 4th edition, 2001. ISBN: 0130323934.

[22] R. Frijns, S. Adyanthaya, S. Stuijk, J. Voeten, M. Geilen, R. Schiffelers, and H. Corporaal. Timing analysis of first-come first-served scheduled interval-timed directed acyclic graphs. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 288:1–288:6. European Design and Automation Association, March 2014.

[23] R. Frijns, A. Kamp, S. Stuijk, J. Voeten, M. Bontekoe, K. Gemei, and H. Corporaal. Dataflow-based multi-ASIP platform approach for digital control applications. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 811–814, Sept 2013.

[24] S. Fuller. *RapidIO: The embedded system interconnect.* Wiley, 2005. ISBN: 978-0-470-09291-0.

[25] Gartner. Worldwide semiconductor revenue grew 5 percent in 2013, according to final results by Gartner. `http://www.gartner.com/newsroom/id/2698917`, 2014.

[26] M. Geilen and T. Basten. Requirements on the execution of kahn process networks. In *Proceedings of the 12th European Conference on Programming*, ESOP'03, pages 319–334. Springer-Verlag, 2003.

[27] A. Ghamarian, S. Stuijk, T. Basten, M. Geilen, and B. Theelen. Latency minimization for synchronous data flow graphs. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 189–196, Aug 2007.

[28] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij. Throughput analysis of synchronous data flow graphs. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, ACSD '06, pages 25–36. IEEE, 2006.

[29] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele. Timed model checking with abstractions: towards worst-case response time analysis in resource-sharing manycore systems. In *Proceedings of the 10th ACM International Conference on Embedded Software*, EMSOFT, pages 63–72, 2012.

[30] B. Girod, R. Rabenstein, and A. Stenger. *Signals and systems.* Wiley, 2001.

[31] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, volume 10 of *ISCA '10*, pages 37–47. ACM, 2010.

[32] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich. A SystemC-based design methodology for digital signal processing systems. *EURASIP Journal on Embedded Systems*, 2007(1), Jan 2007.

[33] J. Hausmans, S. Geuns, M. Wiggers, and M. Bekooij. Temporal analysis flow based on an enabling rate characterization for multi-rate applications

executed on mpsocs with non-starvation-free schedulers. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '14, pages 108–117. ACM, 2014.

[34] J. Hausmans, M. Wiggers, S. Geuns, and M. Bekooij. Dataflow analysis for multiprocessor systems with non-starvation-free schedulers. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, M-SCOPES '13, pages 13–22. ACM, 2013.

[35] M. Hendriks and M. Verhoef. Timed automata based analysis of embedded system architectures. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, Apr 2006.

[36] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *Computers and Digital Techniques, IEE Proceedings*, 152(2):148–166, Mar 2005.

[37] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, Jan 2003.

[38] S. Jones, R. Goodall, and M. Gooch. Targeted processor architectures for high-performance controller implementation. *Control Engineering Practice*, 6(7):867 – 878, 1998.

[39] R. Jordans, F. Siyoum, S. Stuijk, A. Kumar, and H. Corporaal. An automated flow to map throughput constrained applications to a MPSoC. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18, pages 47–58. Schloss Dagstuhl, 2011.

[40] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, pages 471–475, 1974.

[41] K. Keutzer, S. Malik, and A. Newton. From ASIC to ASIP: The next design discontinuity. In *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pages 84–90, 2002.

[42] B. Kienhuis, E. Deprettere, K. Vissers, and P. Van Der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*, pages 338–349, July 1997.

[43] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, Jan 2003.

[44] A. Kumar, S. Fernando, Y. Ha, B. Mesman, and H. Corporaal. Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA. *ACM Transactions on Design Automation of Electronic Systems*, 13(3):40:1–40:27, July 2008.

[45] K. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):203–215, 2007.

[46] D. Laro, R. Boshuizen, D. Ooom, L. Sanders, and J. van Eijk. Lightweigth 450 mm wafer stages enabled by over-actuation. In *Proceedings of the 10th International Conference of the Europian Society for Precision Engineering and Nanotechnology*, page 4, 2010.

[47] K. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

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

[49] B. Lickly, I. Liu, S. Kim, H. Patel, S. Edwards, and E. Lee. Predictable programming on a precision timed architecture. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 137–146. ACM, 2008.

[50] J. Lima, R. Menotti, J. Cardoso, and E. Marques. A methodology to design FPGA-based PID controllers. In *IEEE International Conference on Systems, Man and Cybernetics, 2006.*, volume 3, pages 2577–2583, Oct 2006.

[51] T. Limberg, M. Winter, M. Bimberg, M. Tavares, H. Ahlendorf, M. Fettweis, H. Eisenreich, and G. Ellguth. A heterogeneous MPSoC with hardware supported dynamic task scheduling for software defined radio. In *Proceedings of the 46rd Annual Design Automation Conference*, DAC '09, 2009.

[52] R. Makowitz and C. Temple. Flexray - a communication network for automotive control systems. In *Factory Communication Systems, 2006 IEEE International Workshop on*, pages 207 –212, 2006.

[53] MathWorks. HDL Coder product page. `http://www.mathworks.nl/products/hdl-coder/`, Oct 2013.

[54] MathWorks. Simulink coder. `http://www.mathworks.nl/products/simulink-coder/`, 2014.

[55] MathWorks. Simulink hdl coder. `http://www.mathworks.nl/products/hdl-coder/`, 2014.

[56] MathWorks. Simulink product page, Feb 2014. http://www.mathworks.nl/products/simulink/.

[57] R. Middleton and G. Goodwin. Improved finite word length characteristics in digital control using delta operators. *Automatic Control, IEEE Transactions on*, 31(11):1015–1021, 1986.

[58] E. Monmasson and M. Cirstea. FPGA design methodology for industrial control systems – a review. *Industrial Electronics, IEEE Transactions on*, 54(4):1824 –1842, Aug 2007.

[59] G. Moore et al. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8):114–117, Apr 1965.

[60] D. Navarro, O. Lucia, L. Barragan, I. Urriza, and O. Jimenez. High-level synthesis for accelerating the FPGA implementation of computationally demanding control algorithms for power converters. *Industrial Informatics, IEEE Transactions on*, 9(3):1371–1379, Aug 2013.

[61] H. Nikolov, T. Stefanov, and E. Deprettere. Automated integration of dedicated hardwired IP cores in heterogeneous MPSoCs designed with ESPAM. *EURASIP Journal on Embedded Systems*, 2008.

[62] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(3):542 –555, March 2008.

[63] NVIDIA. Kepler GK110 whitepaper. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012.

[64] OMG. OMG Object Constraint Language (OCL), version 2.4. `http://www.omg.org/spec/OCL/2.4/`, January 2014.

[65] J. Porter, P. Völgyesi, H. Kottenstette, N. Nine, G. Karsai, and J. Sztipanovits. An experimental model-based rapid prototyping environment for high-confidence embedded software. In *IEEE International Workshop on Rapid System Prototyping*, pages 3–10. IEEE, 2009.

[66] F. Robino and J. Öberg. From Simulink to NoC-based MPSoC on FPGA. In *Design, Automation & Test in Europe Conference Exhibition (DATE), 2014*, pages 328:1–328:4. European Design and Automation Association, 2014.

[67] Z. Salcic, J. Cao, and S. Nguang. A floating-point FPGA-based self-tuning regulator. *IEEE Transactions on Industrial Electronics*, 53(2):693–704, April 2006.

[68] D. Sancho-Pradel and R. Goodall. Targeted processing for real-time embedded mechatronic systems. *Control Engineering Practice*, 15(3):363 – 375, 2007.

[69] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*, 18(6):23–33, Nov 2001.

[70] S. Schliecker and R. Ernst. A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '09, pages 433–442. ACM, 2009.

[71] M. Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, pages 2:1–2:17, Jan 2009.

[72] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000. ISBN: 0824793188.

[73] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition, 2009. ISBN: 978-0-321-33188-5.

[74] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, pages 899–904. ACM, 2006.

[75] S. Stuijk, M. Geilen, and T. Basten. SDF3: SDF for free. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design, ACSD'06*, pages 276–278. IEEE Computer Society, 2006.

[76] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 4, pages 101–104, 2000.

[77] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[78] M. Thompson, H. Nikolov, T. Stefanov, A. Pimentel, C. Erbas, S. Polstra, and E. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS, pages 9–14. ACM, 2007.

[79] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Journal of Real-Time Systems*, 6, 1994.

[80] M. van Broekhoven. Motion controller acceleration by FPGA co-processing. MSc thesis, Eindhoven University of Technology, September 2014.

[81] M. van den Brink, H. Jasper, S. Slonaker, P. Wijnhoven, and F. Klaassen. Step-and-scan and step-and-repeat: a technology comparison. In *Proceedings of the SPIE Symposium on Optical Microlithography IX*, volume 2726, pages 734–753, June 1996.

[82] J. Voeten, T. Hendriks, B. Theelen, J. Schuddemat, W. T. Suermondt, J. Gemei, K. Kotterink, and C. v. Huët. Predicting timing performance of advanced mechatronics control systems. In *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pages 206–210, July 2011.

[83] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System architecture evaluation using modular performance analysis: A case study. *International Journal on Software Tools for Technology Transfer*, 8(6):649–667, Oct 2006.

[84] M. Watanabe and S. Kramer. 450 mm silicon: An opportunity and wafer scaling. *Interface-Electrochemical Society*, 15(4):28–32, 2006.

[85] M. Wiggers, M. Bekooij, and G. Smit. Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *Proceedings of the 10th International Workshop on Software & Compilers for Embedded Systems*, SCOPES '07, pages 11–22. ACM, 2007.

[86] E. Witt. Beweisstudien zum satz von M.Zorn. *Mathematische Nachrichten*, 4(1-6):434–438, 1950.

[87] H. Wong, V. Betz, and J. Rose. Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA, pages 5–14. ACM, 2011.

[88] X. Wu, V. Chouliaras, J. Nunez-Yanez, R. Goodall, and T. Vladimirova. A novel processor architecture for real-time control. In *Advances in Computer Systems Architecture*, volume 4186 of *Lecture Notes in Computer Science*, pages 270–280. Springer Berlin Heidelberg, 2006.

[89] Xilinx. Latest product portfolio sets capacity record with 3D Virtex UltraScale FPGA containing 4.4 million logic cells. `http://www.xilinx.com/publications/archives/xcell/Xcell86.pdf`, 2014.

[90] J. Xing, B. Theelen, R. Langerak, J. Pol, J. Tretmans, and J. Voeten. UPPAAL in practice: Quantitative verification of a RapidIO network. In *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6416 of *Lecture Notes in Computer Science*, pages 160–174. Springer Berlin Heidelberg, 2010.

[91] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal. Iteration-based trade-off analysis of resource-aware sdf. In *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pages 567–574, 2011.

[92] W. Zhao, B. H. Kim, A. Larson, and R. Voyles. FPGA implementation of closed-loop control system for small-scale robot. In *Proceedings of the 12th International Conference on Advanced Robotics*, pages 70–77, July 2005.

[93] X. Zhu, T. Basten, M. Geilen, and S. Stuijk. Efficient retiming of multirate DSP algorithms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(6):831–844, June 2012.

Glossary

ADC	analog-to-digital converter
ALU	arithmetic and logic unit
ASIC	application-specific integrated circuit
ASIP	application-specific instruction set processor
CSDF	cyclo-static dataflow
DAC	digital-to-analog converter
DoF	degrees of freedom
DSP	digital signal processor
FCFS	first-come first-served
FIFO	first-in first-out
FO	fixed-order
FPGA	field-programmable gate array
FSM	finite state machine
GPP	general purpose processor
HSDFG	homogeneous synchronous dataflow graph
HW	hardware
IC	integrated circuit
IP	intellectual property
KPN	Kahn process network
LU	lookup unit
LUT	lookup table
MAC	multiply-accumulate
MIMO	multiple-input multiple-output
MIMD	multiple-instruction multiple-data
MoC	model of computation
MP-SoC	multi-processor system-on-chip
NI	network interface
NoC	network-on-chip
PBD	platform-based design
PID	proportional integral derivative
PU	processing unit
RIO	rapidIO

RT	real-time
SDF	synchronous dataflow
SDFG	synchronous dataflow graph
SIMD	single-instruction multiple-data
SPU	scalar processing unit
SS	state-space
SW	software
TDMA	time-division multiple-access
TTA	time-triggered architecture
VLIW	very long instruction word
VPU	vector processing unit

Acknowledgements

Even though I love to read books, writing one myself proved to be quite a challenge. I would like to spend some ink to acknowledge all the people that have contributed and supported me in the completion of this thesis.

First of all, I sincerely thank my promotor and co-promotors for providing the opportunity to do this PhD project, and for their support and guidance during these years. I would like to thank Henk Corporaal for his critical view on my work, the in-depth discussions, and his keen eye for detail. The unrelenting enthusiasm and support of Jeroen Voeten have been a solid foundation during my PhD project. Many thanks for the fruitful cooperation both at the TU/e and ASML, and for the inspiring whiteboard-sparring-sessions. The weekly discussions with Sander Stuijk have greatly helped me to shape my ideas and polish off the rough edges. You have my sincere thanks for your guidance and close support. Thanks also to Marc Geilen for the many discussions that contributed to Chapter 4.

My thanks to the members of my PhD committee, Ton Backx, Marco Bekooij, Maurice Heemels, Ben Juurlink and Ramon Schiffelers for their participation and contribution to the final version of this thesis.

This research was funded by ASML and TNO-ESI. I would like to thank Marcel Bontekoe, Ramon Schiffelers, Shreya Adyanthaya, Nikola Gidalov, Wouter Tabingh Suermondt, Wouter Aangenent, Marc van de Wal, John Gemei, Jan van Vlerken, Maarten Hoeks, Cees van Huët, Hans Vermeulen, Elmar Beuzenberg, Jan Stoeten, Jan Schuddemat, Frans Beenker, Reinier van Eck and the CARM2G team. In particular many thanks to my former M.Sc. student Twan Kamp. We two hardheads didn't always agree on how things should look like, so we had many good discussions that resulted in great contributions to Chapter 3.

The years as a student, employee and PhD candidate at the ES group have been a great and fun experience. My thanks to Ralph Otten and Twan Basten for their support as chairmen of ES, and for the unforgettable experience called 'Computation'. It was a great joy to teach the first-year students some practical pro-

gramming together with my Computation-buddy Sven Goossens. The hunt for the perpetual cheating attempts, popping some *I-really-know-what-I-am-talking-about*-bubbles, and the great creativity and enthusiasm shown by students at their final programming assignments certainly added to the fun. Furthermore, I would like to thank my office-mates Majid, Yifan and Mark for our daily office chit-chat. The heavier topics were saved for the famous coffee corner at the ninth floor of PT. Its centerpiece, the large oval table, has a warm place in the memories of many PhD students, me included. Many thanks to Marcel, Luc, Roel, Gert-Jan, Maurice, Cedric, Sven, Andrew, Joost, Martijn, Marc, Sander and Yang for being part of its contemporary population, and for all the sense and nonsense discussed there. Of course many thanks to Rian, Marja and Margot for all the good care and the warmth you bring to ES. In particular thanks to Marcel and Luc. We evolved from colleagues into good friends misbehaving both at work and beyond. Thanks for the many good laughs, the games of Gilles-de-la-Tourette-Risk and the ridiculous peer-pressure when playing New Super Mario Bros Wii. I am happy to have you guys as paranymphs during my PhD defense.

Many thanks to my friends for their support and the fun nights where we could blow off some steam. In particular thanks to Coen & Anke, Kenny & Dorien and Daan for the regular coffee-vlaai-and-gossip Saturday afternoons, close support in choosing which particular Belgian beers to buy in the huge beer shop in Poppel, and the Shoarma-with-Family-Guy-nights. I heard there's a new Guitar Hero coming, so maybe it's time to dust off our cheap plastic China-made guitars again!

Thanks to my family and 'in-laws' for their unrelenting interest in my work. No egg can hatch into a proper bird without a warm nest; I am very grateful to my parents and brother for providing that warm nest, and for their unconditional support and involvement. And yes, now I am finally done with my 'afstuderen' :-)

Finally, my thanks to Loes, for her love, support and understanding. Without a complaint you let me trade quality time for work time, you accompanied me on conferences (in sunny Spain, yeah, that must have been very hard for you...), and since you did a PhD yourself, you understood every inch of the road. Even though you don't know the glossary of this thesis by heart yet, it's super that we can always talk tech, and that we share the same sense of bad humour. This allows us to always pop any residual work or other issues off our stacks at dinner talk, which has been a great support for me. Lots and lots of thanks for always being there for me and for reminding me that there is beer at the finish line!

Raymond Frijns
March 2015

Curriculum Vitae

Raymond Frijns was born in Tilburg, The Netherlands, on May 4, 1980. After receiving his B.A.Sc. degree in applied physics at Fontys Hogescholen in Eindhoven in 2003, he started studying electrical engineering at the Eindhoven University of Technology in Eindhoven. He received his M.Sc. degree in 2008, upon completion of his final thesis project on the design and implementation of an SIMD-processor with a dynamic interconnect. This work resulted in a scientific publication.

In May 2008, he started working as a research assistant in the electronic systems group at the department of electrical engineering of the Eindhoven University of Technology, where he designed and implemented mechatronic demonstrators and their embedded control software. From May 2010 onwards, Raymond started to pursue a Ph.D. degree within the electronic systems group and ASML. His work focused on a design methodology for high-performance digital controllers for lithography machines. This work has led, among others, to several scientific publications and this thesis.

Currently, Raymond works as a processor tools engineer at Intel Corporation.

List of Publications

First author

- R.M.W. Frijns, A.L.J. Kamp, S. Stuijk, J.P.M. Voeten, M. Bontekoe, K.J.A. Gemei and H. Corporaal Dataflow-based Multi-ASIP Platform Approach for Digital Control Applications. In *16th Euromicro Conference on Digital System Design, DSD 2013*, proceedings, pages 811-814, 2013.

- R.M.W. Frijns, S. Adyanthaya, S. Stuijk, J.P.M. Voeten, M.C.W. Geilen, R.R.H. Schiffelers and H. Corporaal. Timing Analysis of First-come First-served Scheduled Interval-timed Directed Acyclic Graphs. In *Conference on Design Automation and Test in Europe, DATE 2014*, proceedings, pages 288:1–288:6. IEEE, 2014.

- R.M.W. Frijns, M. Bontekoe, M. van Broekhoven, N. Gidalov, A. van Herk, C. van Huët, S. Jambekar, L. Lemmen, R.R.H. Schiffelers, E. van Uden, S. Stuijk, and J.P.M. Voeten. Towards 120kHz stages using CARM2G, In *15th ASML Technology Conference*, poster, 2014.

- R.M.W. Frijns, M. Bontekoe, K.J.A. Gemei, N. Gidalov, C. van Huët, R.R.H. Schiffelers and J.P.M. Voeten. High-performance heterogeneous architectures for CARM2G. In *14th ASML Technology Conference*, poster, 2013.

- R.M.W. Frijns, W.H.T.M. Aangenent, M. Bontekoe, H. Corporaal, G. Clijsen, W. Tabingh Suermondt, B.D. Theelen, J.P.M. Voeten and M.M.J. van de Wal. Towards a predictable automated design flow for large-scale high-performance control applications. In *ICT.Open Conference*, poster, 2011.

- Raymond Frijns, Hamed Fatemi, Bart Mesman, and Henk Corporaal. DC-SIMD: Dynamic Communication for SIMD processors. In *International Parallel and Distributed Processing Symposium, IPDPS 2008*, proceedings, IEEE, 2008.

Co-author

- Marcel Groothuis, Raymond Frijns, Jeroen Voeten and Jan Broenink. Concurrent Design of Embedded Control Software. In *International Workshop on Multi-Paradigm Modeling, MPM'09*, proceedings, 6 Oct 2009.