

A Scenario-Aware Dataflow Programming Model with support for Fault-Tolerance

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven, op gezag van de
rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 25 juni 2019 om 16:00 uur

door

Jacobus Reinier van Kampenhout

geboren te Beilen

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

voorzitter:	prof.dr.ir. J. H. Blom
1 ^e promotor:	prof.dr. K. G. W. Goossens
copromotor:	dr.ir. S. Stuijk
leden:	prof.dr. A. Kumar (Technische Universität Dresden)
	prof.dr. S. D. Cotofana (Technische Universiteit Delft)
	prof.dr.ir. C. H. van Berkel
	prof.dr.ir. T. Basten

Het onderzoek dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

**A Scenario-Aware Dataflow
Programming Model
with support for Fault-Tolerance**

Reinier van Kampenhout

Committee:

prof.dr. K. G. W. Goossen	Eindhoven University of Technology, <i>promotor</i>
dr.ir. S. Stuijk	Eindhoven University of Technology, <i>copromotor</i>
prof.dr.ir. J. H. Blom	Eindhoven University of Technology, <i>chairman</i>
prof.dr. A. Kumar	Technische Universität Dresden
prof.dr. S. D. Cotozana	Delft University of Technology
prof.dr.ir. C. H. van Berkel	Eindhoven University of Technology
prof.dr.ir. T. Basten	Eindhoven University of Technology

© Reinier van Kampenhout 2019. All rights are reserved. Reproduction in whole or in part is prohibited without the written consent of the copyright owner.

Cover design by Studio LEG.

Printed by ProefschriftMaken – The Netherlands.

A catalogue record is available from the Eindhoven University of Technology Library.

ISBN: 978-90-386-4785-2

Summary

A scenario-aware dataflow programming model with support for fault-tolerance

Real-Time (RT) embedded systems can provide solutions to a broad spectrum of applications in almost every domain. Systems that adhere to a common set of static requirements can be designed with state-of-the art design flows. Advances in functionality and technology however lead to novel dynamic requirements that cannot be met in existing flows. Firstly, applications exhibit dynamic behaviour because they must respond to input data. Secondly, the set of applications that is simultaneously executed on a system must change in response to the user and the environment. Therefore the application sets are dynamic as well. Thirdly, the decreasing feature size of chip designs increases the power density and leads to hot spots which cause intermittent and permanent processor faults. This means that the set of available processors is also dynamic. We see that there is dynamism in the applications, application sets, and available processors. Existing design flows can combine static requirements with only one or two of these new dynamic requirements. In this thesis we extend the existing SDF³ design flow and the CompSOC platform to handle the static requirements along with all three dynamic requirements.

Applications must respond dynamically to changes in the input data. We select the *Finite-State Machine Scenario-Aware Dataflow (FSM-SADF) Model-of-Computation (MoC)* that can capture different application behaviours in separate scenarios, and can provide a bound on the minimum throughput of an application. There is no *Programming Model (PM)* available to implement applications written in this MoC. When a scenario is started at runtime, it may be necessary to process some of the input data before the new application scenario can be identified. It is therefore not possible to decide which scenario graph should be started, we term this the causality dilemma. In Chapter 3 we propose a PM for FSM-SADF that allows to schedule any allowed sequence of scenarios at run-time. In our proposed design flow the scenario graphs are transformed into a scenario sequencing model that features a separate *detector* scenario in which the current scenario is detected. This detector scenario is always executed before a regular scenario, which solves the causality dilemma. The timing behaviour of the sequencing model is identical to that of the original graph and can be used to analyse the throughput of the application. The scenario graphs of the sequencing model can be merged into a scenario execution model. Together with the *Rolling Static-Order (RSO)* scheduling and a number of middleware extensions the execution model can be executed on the CompSOC platform. The design flow was extended to automatically generate and analyse these models. We also supply models of our implementation that capture the exact timing behaviour. We

contribute an FSM-SADF PM that consists of a method, design flow, middleware and analysis model for designing applications that can dynamically respond to input data.

The active set of applications at run-time must change dynamically to allow the system to respond to user and environment events. This means it must be possible to start and stop an application at any time. In this thesis we select a three-layer mapping and scheduling concept that maps applications to *Virtual Processors (VPs)* at design-time, which are deployed (mapped and scheduled) to the physical processors at the application start at run-time. This separates design-time analysis from run-time deployment. The selected architecture, middleware and timing analysis ensure that VPs may be deployed to any processor at run-time if sufficient processor capacity, memory and interconnect bandwidth are available. We propose to employ a heuristic-based resource manager that can start and stop applications at any time. The platform utilisation at the moment at which an application starts cannot be predicted. If the resource requirements are unbalanced across the applications, the probability of success of starting applications will be low. The existing *Design Space Exploration (DSE)* trades the throughput of applications against the total memory size, and attempts to reduce the processor utilisation only after these other parameters are fixed. In Chapter 4 we extend the DSE algorithm to uncover the trade-off between the required processor capacity and the required memory, while meeting the throughput constraint. This results in a set of Pareto-optimal points with which the designer can balance the processor and memory requirements and thus increase the probability of successful deployment in systems that dynamically respond to the user and environment.

Processor faults lead to dynamism in the set of available processors. We propose to employ the resource manager to re-deploy all VPs from a faulty processor to unused capacity on other processors, avoiding the costly reservation of spares. As the utilisation at the moment at which the fault occurs cannot be predicted, applications may fail because insufficient processor capacity is available. Re-deployment is essentially a bin-packing problem. In Chapter 5 we propose two contributions that maximise the probability of successful re-deployment by changing the sizes of the VPs. Firstly, mapping applications to more VPs of a smaller size at design-time comes at the cost of a larger total VP size, but increases the probability that the bin-packing is successful. Secondly, we split and resize VPs at the moment a fault occurs, and possibly target them for a different processor type. We exploit the contribution of Chapter 4 by selecting Pareto points with minimal cost in terms of processor capacity. The experiments show that both strategies increase the probability of successful re-deployment and thereby the fault-tolerance of a system.

In conclusion, we present a design flow and PM for RT embedded systems that allows dynamic response to the input data, the user and environment, and to processor faults. We implemented the concepts in algorithms and middleware libraries that are presented in this thesis, and evaluate each contribution experimentally.

Contents

1	INTRODUCTION TO REAL-TIME EMBEDDED SYSTEM DESIGN	1
1.1	Common requirements for real-time embedded systems	1
1.1.1	Use-case: point-to-point video surveillance	1
1.1.2	Use-case: adaptive cruise control	3
1.1.3	Common requirements	4
1.1.4	Fault-tolerance	5
1.2	Real-time embedded system design	5
1.2.1	The design steps in detail	6
1.2.2	Propagation of real-time requirements through the design flow	7
1.3	Problem statement	8
1.4	Contribution	11
1.4.1	Existing platform and design flow	11
1.4.2	Solutions to the sub-problems	12
1.5	Related work to overall thesis goals	18
1.6	Thesis outline	20
2	BACKGROUND AND TERMINOLOGY	21
2.1	Introduction	21
2.2	Design flow	21
2.3	Model of Computation	24
2.3.1	Comparison to other MoCs	25
2.3.2	Application scenarios	25
2.3.3	Scenario graphs	26
2.3.4	FSM-SADF applications	28
2.4	Mapping and scheduling	28
2.4.1	Storage distributions	28
2.4.2	Mapping applications to VRs	30
2.4.3	Determining the VP size	30
2.5	Programming model	31
2.6	Middleware	32
2.6.1	CoMik μ kernel	32
2.6.2	libFIFO	33
2.6.3	libDataflow	33
2.7	Hardware	33
2.8	Summary	34
3	A SCENARIO-AWARE DATAFLOW PROGRAMMING MODEL	37
3.1	Introduction	37
3.2	The causality dilemma	37
3.2.1	Delayed scenario detection	37

3.2.2	Immediate scenario detection	39
3.2.3	Shared persistent tokens	42
3.2.4	Contributions	42
3.3	Scenario sequencing model	43
3.3.1	Creating a sequencing model for applications with delayed scenario detection	43
3.3.2	Creating a sequencing model for applications with immediate scenario detection	44
3.3.3	Timing analysis of applications with immediate scenario detection	48
3.3.4	Automatic creation of the scenario sequencing model	49
3.4	Scenario execution model	49
3.4.1	Executing a sequence of scenarios	50
3.4.2	Executing schedules of applications with delayed scenario detection	52
3.4.3	Executing schedules of applications with immediate scenario detection	54
3.4.4	Implementation of switch and select actors	54
3.4.5	Implementation of shared persistent tokens	57
3.5	Extended Binding Aware Graph	58
3.6	Experimental evaluation	60
3.6.1	Delayed scenario detection	60
3.6.2	Immediate scenario detection	61
3.6.3	Results	62
3.7	Related work	62
3.7.1	Non-dataflow PMs	63
3.7.2	Non-dynamic dataflow PMs	63
3.7.3	Dynamic dataflow PMs	64
3.8	Summary	65
4	TRADING VIRTUAL PROCESSOR SIZE AGAINST BUFFER SIZE	67
4.1	Introduction	67
4.2	Inter-application deployment	68
4.2.1	Design-time deployment	68
4.2.2	Run-time deployment	68
4.2.3	Hybrid deployment and deployment	68
4.3	WCRT analysis for intra-application mapping	69
4.3.1	Platform preliminaries	70
4.3.2	WCRT analysis	70
4.4	Design Space Exploration	73
4.5	Trade-off: VP size against buffer size	75
4.6	Experimental evaluation	77
4.7	Related work	82
4.8	Summary	82
5	FAULT-TOLERANT DEPLOYMENT	85

5.1	Introduction	85
5.2	Recovering from processor faults	85
5.3	Mapping dataflow graphs	88
5.4	Resource manager	89
5.5	Fault model	89
5.6	Fault-tolerance concept	90
5.6.1	Re-deployment	90
5.6.2	Resize and split	91
5.7	Experimental evaluation	92
5.7.1	Preliminary	92
5.7.2	Mapping	94
5.7.3	Re-deployment	96
5.7.4	Resize and split	98
5.7.5	Trade-offs	99
5.8	Related work	100
5.8.1	Re-deployment	100
5.8.2	Resource manager	102
5.9	Summary	102
6	CONCLUSIONS AND FUTURE WORK	105
6.1	Dynamic response to input data	105
6.2	Dynamic response to the user and environment	106
6.3	Dynamic response to processor faults	107
6.4	Future work	107
	APPENDICES	109
A	USE-CASE REQUIREMENTS	111
B	ALGORITHMS FOR FAULT-TOLERANT DEPLOYMENT	117
	BIBLIOGRAPHY	121
	LIST OF ACRONYMS	139
	INDEX	141
	ACKNOWLEDGMENTS	143
	ABOUT THE AUTHOR	145
	LIST OF PUBLICATIONS	147



1 Introduction to real-time embedded system design

1.1 Common requirements for real-time embedded systems

Embedded systems are electronic systems that are designed for a specific purpose and are integrated into their environment. Contemporary embedded systems are digital systems where software applications are executed on a hardware platform. Such systems can provide solutions to a broad spectrum of applications in domains such as health-care, transportation, and automation. In order to do so systems must offer complex functionality with high reliability at an acceptable cost, resulting in a complex set of requirements.

We now give two representative use-cases and distill the system requirements that they have in common. The use-cases as a whole are fictional, but each requirement is derived from a source that is supplied as a reference. For convenience the relevant excerpts from these sources can be found in Appendix A.

1.1.1 Use-case: point-to-point video surveillance

The first use-case is a point-to-point video surveillance system for health-care applications, where a person under observation can be monitored by multiple staff members that carry wireless devices. An overview is presented in Figure 1.1 Each monitor device decodes the received video stream. The video stream for each monitor device is produced on a multi-processor platform at the patient's location by an encoder application. This use-case must satisfy the following requirements:

- UC-1.1 the target frame-rate is 25 *Frames Per Second (FPS)* and the worst-case latency of the stream must be under 300 milliseconds [17, 127];
- UC-1.2 the system must function in a temperature range from 0° C to +85° C [61];
- UC-1.3 the battery of the hand-held monitor devices must last for 32 hours [27];
- UC-1.4 the cost must be competitive with comparable devices available on the consumer electronics market [5, 18];
- UC-1.5 the number of monitor devices must be adjustable at any time [16];
- UC-1.6 to enhance the quality of each stream a compressed video format with different frame types must be used [152];

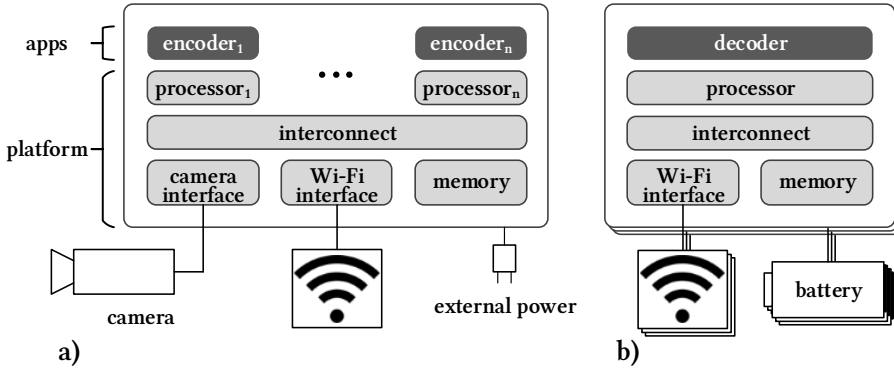


Figure 1.1: An overview of the first use-case. a) The multi-processor platform at the patients location is connected to a camera and WiFi device, and contains n processors that run n encoder applications. b) Multiple single-processor battery-powered monitor devices each run one decoder application.

UC-1.7 an alarm must be raised if the system encounters a fault, and the system should try to recover while not interrupting functional streams [30].

We can sort the requirements in four categories: *Real-Time (RT)* constraints, resource constraints, dynamic behaviour, and fault-tolerance. Timing requirement UC-1.1 means this system falls in the category of RT embedded systems that must produce their results before a certain point in time, the deadline. A consequence of UC-1.2 is that the maximum power must be sufficiently low in order not to overheat. On top of that the energy consumption and thus the average power of the device must be constrained as dictated by UC-1.3. The cost of the system largely depends on the hardware components that are used, so in order to satisfy UC-1.4 the memory, interconnect and number of processors should be minimised. Power, energy and system cost are all part of the *resource* constraints. *Dynamic behaviour* is suggested by UC-1.5: the platform at the patient side must be able to start and stop encoder applications on-the-fly depending on the number of monitors. To meet UC-1.6 a video format must be selected that enhances the quality of the stream through video compression techniques. Commonly such formats transfer one full video frame followed by a number of delta frames that describe only the changes relative to the full frame. When a decoder receives a frame, it must decode the header to determine which type it is and call the correct decoder function. This implies dynamism within the application rather than in the application set. Lastly, some form of *fault-tolerance* must be implemented to meet UC-1.7.

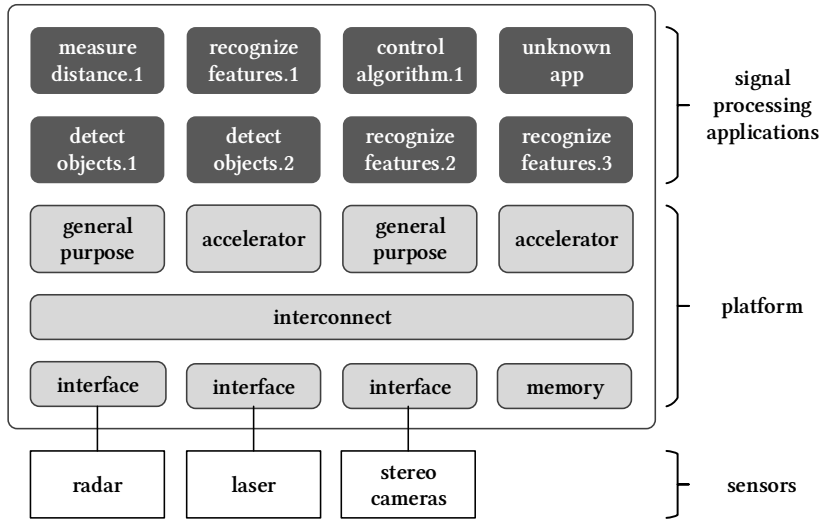


Figure 1.2: An overview of the second use-case: a heterogeneous platform with two general-purpose processors and two accelerators is connected to multiple sensors. The sensor data is the input to three signal processing applications executed on the platform, each consisting of one or more tasks. There is also one task from an unknown application.

1.1.2 Use-case: adaptive cruise control

The second use-case that we consider is an adaptive cruise control for automotive vehicles, see Figure 1.2. Such systems use the input from multiple high-resolution sensors including stereo cameras, lasers and radar. These data are analysed by signal processing applications, whose results are the input to a control application that runs on the same platform. The following requirements must be satisfied in this use-case:

- UC-2.1 the control loop must have a frequency of at least 1000 Hz [103];
- UC-2.2 the system must function in a temperature range from -40°C to $+125^{\circ}\text{C}$ [61];
- UC-2.3 the application shares a heterogeneous multi-processor platform with other applications, each of which is restricted to its own energy budget [87];
- UC-2.4 the platform should host as many applications as possible to keep the cost low and compete on the automotive market [81];
- UC-2.5 the number and types of other applications that share the platform is unknown at design-time and may change with each software update [1];

UC-2.6 the cruise control must work under all circumstances, i.e. in tunnels, at night, with rain, etcetera [79];

UC-2.7 the system must be detect and recover from hardware faults due to temporary overheating of on-chip components for at least 15 years [40].

Although the use-case is very different, this is again a RT embedded system. The required temperature range is wider than in the first use-case which means that the hardware must be more resilient, but there is more headroom for energy consumption and power. There is uncertainty regarding other applications on the platform, implying dynamism on the system level. The system must be robust against changes in the outside world (light, weather) which means the application must also dynamically respond to input data. Robustness against system faults suggests some form of fault-tolerance is again required.

1.1.3 Common requirements

The requirements of UC-1 and UC-2 can be grouped pairwise according to category, UC-1.1 and UC-2.1 for example both capture the RT requirement. Requirements 2, 3 and 4 of both use-cases capture the resource requirements and can be merged together.

Requirements UC-1.5 and UC-2.5 require special attention. In UC-1.5 multiple encoder applications may be started at run-time, whose characteristics are known at design-time. UC-2.5 the platform is shared with an unknown number of unknown applications that may change at run-time. To handle both it is necessary to follow an approach in which any mix of applications can execute at run-time. We term this a *dynamically changing set* of applications.

Extracting the common denominator of each pair of requirements leads to the following list of common *system requirements* for RT embedded systems:

RQ.1 subject to RT constraints

RQ.2 subject to resource constraints, e.g. power, temperature, energy, memory, processors and interconnect;

RQ.3 shared platforms must concurrently execute either:

- a) a fixed set of applications, or;
- b) a dynamically changing set of applications (in short: dynamic set).

RQ.4 must respond dynamically to data (in short: dynamic application);

RQ.5 must tolerate faults.

These system requirements are valid for many contemporary and future RT embedded systems. Requirements RQ.3a and RQ.3b exclude each other, i.e. a system can

support either one or the other. While each individual requirement is quite straightforward, multiple problems and complexities arise when attempting to combine all of them into one system. These problems will be worked out in Section 1.3.

Requirements RQ.1–RQ.3a are standard in state-of-the-art RT systems, and design of such systems is well understood. We will refer to these as *static* RT systems. As the demand for functionality increases, new and dynamic requirements such as RQ.3b–RQ.5 are added as we have seen in the use-cases. We refer to systems where all requirements except RQ.3a must be met as *dynamic* RT systems.

1.1.4 Fault-tolerance

Requirement RQ.5 states that the systems must be tolerant to hardware faults, we now discuss this requirement in more detail. Faults may have many different causes such as variations in hardware manufacturing, radiation, overheating and aging. Advances in *Very Large Scale Integration (VLSI)* design lead to a decrease of the feature size, which increases the power density of chips and causes higher local temperatures. These hot spots prevent powering on all transistors simultaneously at the nominal voltage, an effect known as dark silicon [53]. During operation hot spots may cause intermittent faults that lead to temporary processor shutdowns [80, 124]. In the long term, hot spots cause increased electromigration and speed up the aging process [102]. These effects make multi-processor platforms more susceptible to faults, and therefore fault-tolerance is an increasingly common requirement for RT embedded systems.

In this work we focus on intermittent faults due to hot spots and permanent faults due to aging and electromigration. As the power density is at its highest in the processors, such faults are most likely to appear there. We therefore focus on processor faults and will not consider the memory and interconnect. Fault-tolerance can be achieved in hardware or software, either through fault masking or by fault detection, containment and recovery [7, 31, 88]. To offer a platform-independent solution we focus on providing fault-tolerance in software.

1.2 Real-time embedded system design

So far we have established the system requirements for static and dynamic RT embedded systems. We consider this the first step in the design of such systems. The rest of the design flow must ensure that these requirements are met. A design flow for RT embedded systems typically consists of the following design steps:

DS.REQ requirements engineering;

DS.MOC description of the application behaviour using a *Model-of-Computation (MoC)*;

DS.M&S mapping and scheduling of applications onto a platform;

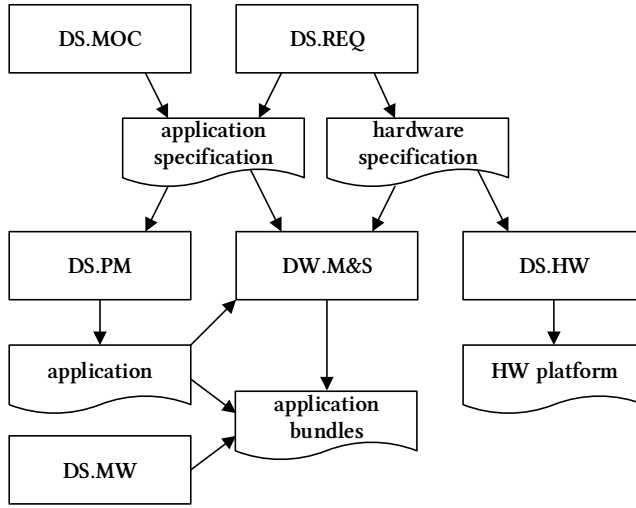


Figure 1.3: An example design flow.

DS.PM application implementation using a *Programming Model (PM)*;

DS.MW design of the middleware (firmware, *Operating System (OS)*, etc.);

DS.HW hardware platform design.

Other classifications are possible, but the above list is sufficiently detailed to explain our contributions [111]. The design steps are ordered top-down from requirements to hardware, but are not necessarily performed in that order. Often applications are mapped to a given hardware platform with given middleware and PM. Figure 1.3 gives an example of a design flow that consists of all the aforementioned steps. The application specification, hardware specification and application are intermediate results. The end result consists of a platform and a *bundle* for each application, i.e. the code and data together with the mapping and scheduling specification for that application [45].

1.2.1 The design steps in detail

A *Model-of-Computation (MoC)* is an abstraction that models how a set of outputs can be computed given a set of inputs, independent of how the computations are performed. Examples are finite-state machines, Turing machines, the time-triggered MoC and *Synchronous Dataflow (SDF)* [35, 68, 74].

Mapping describes the process of allocating resources such as processors, memories and interconnect to *tasks*. The term task is used here to indicate the minimal unit of a MoC that can be executed on its own, e.g. a computation, data transfer or storage operation. Heterogeneous multi-processors contain many components, and

mapping plays a prominent role in the design flow. *Scheduling* describes the process of allocating time to tasks on the resources to which they are mapped. These terms are used for the creation of mappings and schedules, and should not be confused with the *execution* of given mappings and schedules at run-time. Mapping and scheduling are widely researched in the RT community [99, 120].

The *Programming Model (PM)* offers constructs that implement the computation and communication specified by the MoC, still independent of the hardware. Theoretically a PM is also independent of the programming language, but in practice each PM is bound to at most a handful of languages. A PM may provide constructs that are otherwise tedious and error-prone to create, e.g. for parallel execution of tasks. Examples of PMs are PThreads, OpenMP, SDF and Giotto [11, 54].

Middleware offers hardware access through drivers and libraries, hiding implementation details from the PM and the programmer. It may include tasks that execute pre-defined mappings and schedules. Alternatively a *resource manager* may be used that performs mapping and scheduling at run-time, and immediately executes these mappings and schedules [128, 129]. This is what we term *deployment* in this thesis.

The *hardware platform* performs the actual computations and communication operations. We will not address hardware design in this thesis but assume platforms that are designed according to the *correct-by-construction* approach [55]. This means that the hardware is constructed in a way that guarantees certain properties that the MoC requires.

1.2.2 Propagation of real-time requirements through the design flow

Requirements have implications on the design flow steps, which is especially true for RT constraints. The MoC must be able to guarantee a lower bound on the *throughput* or an upper bound on the *latency* of an application given the *Worst-Case Response Time (WCRT)* of the tasks. The throughput determines the minimum number of iterations that must be completed per time unit. The latency captures the maximum amount of time that may elapse before a system responds, i.e. gives an output.

The WCRT depends on the *Worst-Case Execution Time (WCET)* and the worst-case *waiting time* [45]. The execution of task mapping and scheduling is part of the middleware, and contributes to the waiting time. In turn, the WCET depends on the PM constructs that implement computation and communication operations.

Another factor that must be included in the execution time of a task are hardware mechanisms that arbitrate access to shared resources such as processors, memories and the interconnect. The type of hardware component to which a task is mapped determines the WCET of a task [19, 34, 154]. Each processor may have a different clock speed and instruction set, and even an identical instruction may require a different number of clock cycles on each processor type. If it is not possible to give an upper bound on the WCET or the waiting time in any of these design steps, the RT performance cannot be guaranteed in the MoC. We see that the properties of each

design flow step are interdependent, and that the RT requirement can only be met if it is supported in all stages of the design flow.

1.3 Problem statement

We have presented the requirements for two use-cases, extracted the common denominators for static and dynamic RT embedded systems and showed that each system requirement has implications for every step in the design flow. In Section 1.5 we show that existing design flows can combine up to two requirements for dynamic RT systems with the requirements for static RT systems, but no prior work has combined all requirements in one flow. The main problem that we address in this thesis is that no existing design flow can address all the requirements for dynamic RT systems listed in Section 1.1.

	RQ.1: Real-time				RQ.2: Resources	
RQ.3b: Dyn. sets	SP.1				SP.4	
	DS.M&S		DS.MW		DS.M&S	
RQ.4: Dyn. applications	SP.2				SP.5	
	DS.MOC	DS.M&S	DS.PM	DS.MW	DS.M&S	DS.MW
RQ.5: Fault tolerance	SP.3				SP.6	
	DS.M&S	DS.MW	DS.HW		DS.M&S	DS.MW

Table 1.1: A breakdown of the problem in six sub-problems SP.1–SP.6 that are each the product of combining one dynamic requirement with one static requirement, and that each affect one or more design steps (DS).

In this section we dissect the main problem in six sub-problems, each of which translates to a *design flow requirement*. The sub-problems originate from combining each of the dynamic requirements RQ.3b–RQ.5 with each static requirements RQ.1 and RQ.2. Note that RQ.3a is mutually exclusive with RQ.3b and is ignored. Each sub-problem affects one or more design steps. Thus we have a three-dimensional matrix with the static requirements in the first dimension, the dynamic requirements in the second, and the design steps in the third dimension. This problem space is visualised in Table 1.1, where the cells list the sub-problems (SP) and corresponding design steps. We thus identify six sub-problems that impact one or more design steps:

SP.1 For dynamic sets the number of possible combinations of mappings and schedules grows exponentially with the number of applications and the number of events, i.e. asynchronous starts and stops of applications. Large numbers of mappings and schedules cannot be stored in memory-constrained embedded

systems, which rules out static mapping solutions [8]. Mapping and scheduling of dynamic sets must instead be performed at run-time by a resource manager that computes the mapping and schedule in a bounded time. The RT constraints of both the new and existing applications must be met at all times, including during the mapping, scheduling and loading phases.

The resource manager consumes time and resources on the target platform, and is therefore also an application. Its execution time must be predictable, and should be minimised for two reasons: to maximise the available resources for other applications, and to minimise the time required to start other applications. As the number of possible mapping and schedule combinations grows exponentially with each event, the resource manager must explore a vast solution space in little time. Use of heuristics is therefore inevitable, which may be forced to select sub-optimal solutions. This sub-problem must be addressed in the mapping, scheduling and middleware, design flow steps DS.M&S and DS.MW.

DESIGN FLOW REQUIREMENT: *Each application, including the resource manager, must meet its RT constraint independent of other applications, and the execution time of the resource manager must be predictable and minimised.*

SP.2 Applications that respond dynamically to data exhibit different behaviours depending on the input data. These behaviours can be captured in a finite number of application *scenarios*. Each scenario has its own WCET because variations in the input data trigger different computations and control flow paths. The WCET of the whole application also depends on the scenario transitions, which are modeled and analysed in the MoC (DS.MOC) at design-time. A number of dataflow MoCs in which applications are captured as directed graphs are able to do that, namely *Scenario-Aware Dataflow (SADF)*, *Finite-State Machine Scenario-Aware Dataflow (FSM-SADF)* and *Mode-Controlled Dataflow (MCDF)*¹ [83, 136, 138, 139]. Figure 1.4 shows two dataflow scenario graphs. In dataflow the tasks are termed actors, which are the nodes of the graphs. Actor *a* occurs in both scenarios, actors *b* and *c* only in S_1 and actor *d* only in S_2 . We select FSM-SADF because the analysis tools that exist for this MoC can compute tighter throughput bounds than those that are available for the others [3].

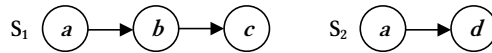


Figure 1.4: Two scenario graphs S_1 and S_2 that describe different behaviours of an application.

However, no PM is available for FSM-SADF. Scenario transitions depend on input data, and the next scenario can only be identified after some of the input

¹ In MCDF, scenarios are termed *modes*.

data is processed. In Figure 1.4 for example actor *a* must be processed before the next scenario is known. The scheduling of scenarios must therefore be performed during execution. This causes a *causality dilemma*, as we cannot know which scenario graph to start with. The causality dilemma must be solved in the PM, and the timing of executing scenario transitions must be correctly captured by the MoC analysis model. This concerns design steps DS.M&S, DS.PM and DS.MW.

DESIGN FLOW REQUIREMENT: *The design flow must contain a PM for FSM-SADF that solves the causality dilemma.*

- SP.3 Fault-tolerance mechanisms to detect and contain faults can be implemented in DS.HW and DS.MW in a way that is transparent to the other design steps [31]. Checkpointing and restart is a suitable technique for fault recovery of dataflow applications as we will see in Chapter 5, and affects DS.M&S and DS.MW. Fault recovery in software affects the timing behaviour, so we must ensure that other running applications are not affected.

DESIGN FLOW REQUIREMENT: *The design flow must implement software fault recovery that may not interfere with other running applications.*

- SP.4 In SP.1 we have seen that a run-time heuristic is required to map and schedule starting applications on a platform. Because the set of running applications is dynamic, the platform utilisation is unpredictable. Therefore it cannot be guaranteed that the resource manager can find sufficient free resources to map and schedule an application that needs to start. To increase the probability of success of future run-time mapping and scheduling actions the application resource requirements must be balanced with respect to the target platform. If the resource requirements of the applications are not balanced the resource manager will soon run out of one type of resources, preventing the start of other applications. Balancing resource requirements must be build into step DS.M&S of the design flow.

DESIGN FLOW REQUIREMENT: *The design flow must balance the resource requirements of applications to increase the probability of success of future run-time mapping and scheduling actions.*

- SP.5 As explained in SP.2 the application behaviour is different in each scenario, and therefore the resource requirements also vary per scenario. The time at which scenario transitions occur is unpredictable. This means that the resource requirements of each application fluctuate unpredictably over time. This moment to moment variation in resource requirements means that a running application may fail because insufficient resources are available for a scenario switch. Such uncertainty is not acceptable for running applications, once an application has started it must always receive the resources that it requires independent of other applications. This must be implemented in steps DS.M&S and DS.MW of the design flow.

DESIGN FLOW REQUIREMENT: *Once an application is started it must receive its required resources at all times, independent of its scenario switches.*

SP.6 Providing fault-tolerance in software comes at the cost of resources. Firstly, techniques for fault detection, containment and recovery require modifications to DS.MW that increase the resource requirements. Secondly, recovering from intermittent and permanent faults using checkpointing and restart in DS.M&S requires free resources to relocate the restarting application to. Similar to SP.4, it is not possible to guarantee that sufficient resources will be available for the re-mapping and re-scheduling, meaning that a fault may result in the failure of one or more applications. To avoid this the design flow must minimise the probability that a running application fails because of a processor fault.

DESIGN FLOW REQUIREMENT: *The design flow must maximise the probability of success of future run-time re-mapping and re-scheduling actions due to processor faults.*

1.4 Contribution

The main goal of this thesis is to propose a design flow to create RT systems that adhere to the dynamic requirements introduced in Section 1.1. In Section 1.3 we refined this goal to six sub-problems SP.1–SP.6 by combining each dynamic requirement with each static requirement. Some of these sub-problems may be solved (partly) in hardware. To achieve a general, platform independent solution however we propose solving them in software, specifically in design steps DS.MOC, DS.M&S and DS.PM. This software approach must be supported in design steps DS.MW and DS.HW. For these latter design steps we use existing implementation of middleware and hardware that we describe next.

1.4.1 Existing platform and design flow

We select an existing design flow named SDF³ [131] and platform named *Composable System-on-Chip (CompSOC)* [45] that feature a two-level mapping and scheduling approach, see Figure 1.5. During the intra-application mapping and scheduling at design-time the actors of each dataflow application are mapped to *Virtual Resources (VRs)* that are to be deployed at run-time. Processors for example are replaced by *Virtual Processors (VPs)*, and a *Static-Order (SO)* schedule is generated for each set of actors that is mapped to the same VP. See the upper half of Figure 1.5, the SO schedules are indicated in the VPs. A VP is a budget that describes a claim to a fraction of the total capacity of a physical processor. Similar budgets are created for the memory, interconnect and other resources. As our contributions focus on processors, we will use VPs as the running example. The collection of budgets combined with the application form a *bundle*. Design-time timing analysis of a mapped dataflow ap-

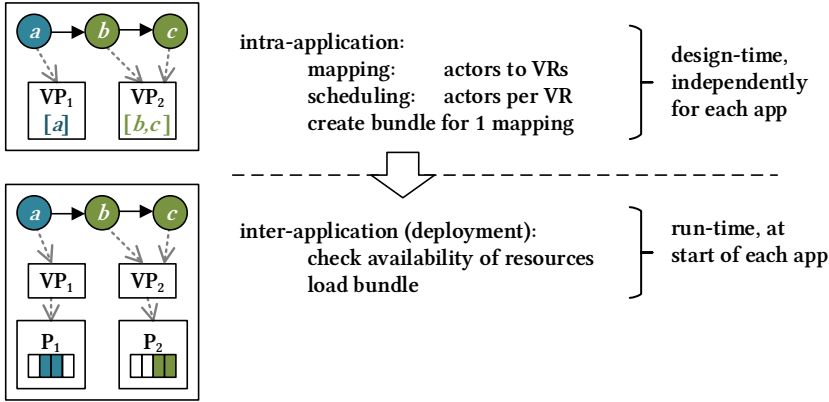


Figure 1.5: The two-layer mapping and scheduling of SDF applications in the existing design flow. Upper layer: the dataflow actors are mapped and scheduled onto *Virtual Resources* (VRs) during design-time, creating one bundle per application that consists of the application and its absolute resource specifications. Lower layer: run-time deployment checks whether the required resources are available and if so, loads the bundle.

plication gives guarantees on the throughput if the budgets are met. The existing platform supports SDF dataflow applications without scenarios.

At run-time the bundles are deployed on the physical resources when an application starts. To ensure that each VP is granted its budget, all VPs on a processor are scheduled using *Time-Division Multiplexing* (TDM) scheduling [45]. See the lower half of Figure 1.5, processors P_1 and P_2 each have four TDM slots of which two are reserved for the application. The mapping and schedule specify which slots on which processors are reserved for the application, i.e. VPs cannot be moved around and the reservation is *absolute*. The resource manager deploys applications by checking whether those specific resources are available, and loads the bundle if this is the case.

The middleware and hardware of the CompSOC platform are *composable*, meaning that the timing behaviour (and therefore the performance) of one application is completely independent of the other applications executing on the same resources. Applications can be verified independently and no system-wide re-verification is necessary after integration.

1.4.2 Solutions to the sub-problems

In the remainder of this section we present a specific solution to each sub-problem, and differentiate between *solutions* (SL) that originate from related work and the *contributions* (CB) made in this thesis. An overview of the solutions and contribu-

	RQ.1: Real-time				RQ.2: Resources	
RQ.3b: Dyn. sets	SL.1				CB.4	
	DS.M&S		DS.MW		DS.M&S	
RQ.4: Dyn. applications	CB.2				SL.5	
	DS.MOC	DS.M&S	DS.PM	DS.MW	DS.M&S	DS.MW
RQ.5: Fault tolerance	SL.3				CB.6	
	DS.M&S	DS.MW	DS.HW		DS.M&S	DS.MW

Table 1.2: A overview of the solutions to the sub-problems. Green cells indicate the contributions of this thesis, yellow cells indicate solutions from related work.

tions is given in Table 1.2, solutions are marked in yellow and contributions in green. Now follows a list of the solutions and contributions. The numbers correspond to the sub-problem that is solved, we repeat each design flow requirement for convenience:

SL.1 DESIGN FLOW REQUIREMENT: *Each application, including the resource manager, must meet its RT constraint independent of other applications, and the execution time of the resource manager must be minimised.*

The CompSOC platform detailed in Subsection 1.4.1 is both predictable and composable. Predictability is a necessary property to compute a WCRT for RT MoCs as described in Section 1.2. Composability ensures that the WCRT calculated for an application cannot be invalidated by the execution or loading of other applications. Together, these properties are sufficient for executing dynamic sets of RT applications.

However, the mapping and scheduling concept of the selected design flow does currently not support dynamic sets as described in RQ.3b because the resource budgets are absolute. To deploy applications on any set of resources whose available capacity meets the budgets in the bundle, the resource budgets must be *relative*. Composability guarantees the absence of interference between applications, regardless to which resources their VPs are mapped. This property can be exploited to realize run-time deployment of dynamic sets with relative resource budgets, where the actual resources to which a VP is mapped are only decided when an application starts. This extension of the mapping and scheduling concept is depicted in the middle layer of Figure 1.6. During deployment the actors can be mapped and scheduled to any physical resource that has sufficient capacity. If more than one resource of the required type fulfills this requirement, the resource manager must select one.

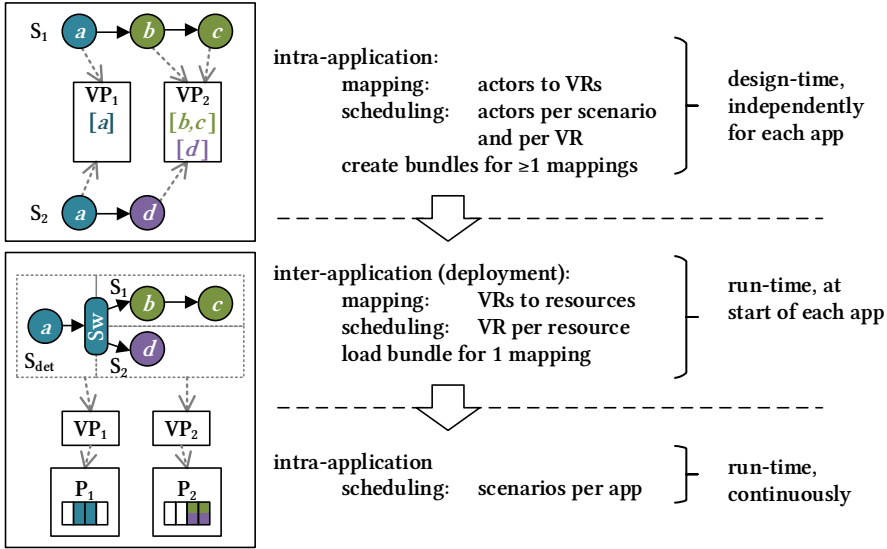


Figure 1.6: The proposed three-layer mapping and scheduling of FSM-SADF applications. In the first layer scenarios S_1 and S_2 are mapped and scheduled to the same set of VRs, the relative resource specifications. In the second layer the VRs are mapped and scheduled to the physical resources, after which the bundle is loaded. In the third layer the scenarios are scheduled inside each application, while executing detector scenario S_{det} the switch actor Sw is configured to forward data to the correct scenario.

This is an instance of the bin-packing problem, which is known to be NP-hard [112]. The resource manager must use a heuristic to minimise its execution time for solving this problem. This mapping and scheduling concept greatly reduces the complexity of the resource manager, as it only needs to ensure that the resource budgets are met and is not concerned with the RT requirements of the applications [85]. The resource manager can be treated as just another application to profit from the predictability and composability [9]. For the contributions presented in this thesis the resource manager concept described in this solution is sufficient, we do not need an implementation. Resource managers of similar design have been implemented in related work [9, 82, 125]. The existing design flow is explained in detail in Chapter 2.

SOLUTION: We select the existing SDF³ design flow and CompSOC platform which ensure that each application meets its RT constraints independently of others. The two-layer mapping and scheduling concept simplifies the run-time deployment heuristic, minimising its execution time and solving SP.1.

CB.2 DESIGN FLOW REQUIREMENT: *The design flow must contain a PM for FSM-SADF that solves the causality dilemma.*

The FSM-SADF MoC provides tight temporal analysis of scenario-based applications. In Chapter 3 we contribute a PM that implements this MoC [147]. Figure 1.6 shows the extended mapping and scheduling concept using an example application with two scenarios. The intra-application mapping and scheduling is now performed for each scenario, actors that occur in multiple scenarios are mapped to the same VRs in each of these. The per-scenario SO schedules are listed inside the VPs in the upper left of the figure. The run-time deployment of VPs has changed to use relative budgets as was explained in SL.1.

The execution graph shown in the lower left of Figure 1.6 is a composition of the *detector scenario* S_{det} and the remainder of the two scenario graphs. The detector graph is identified as the common prefix graph of all scenarios, after whose execution the next scenario is known. A *switch* actor Sw is inserted in between the detector scenario and the remainder of the scenarios. In essence we add an additional layer of intra-application *scenario scheduling* at run-time, thus solving the causality dilemma.

To analyse the exact timing impact of this run-time solution we extend the existing analysis model of the MoC, details are provided in Chapter 3. The PM is both executable and analysable.

CONTRIBUTION: *We contribute a PM for FSM-SADF in Chapter 3 that solves CB.2.*

SL.3 DESIGN FLOW REQUIREMENT: *The design flow must implement software fault recovery that may not interfere with other running applications.*

The design flow and platform introduced in solution SL.1 do not support fault-tolerance. Existing methods for fault detection and containment can be implemented in the platform with little or no timing impact. Fault recovery through checkpointing and restart is a different matter. We select a strategy where the resource manager *re-deploys* all VPs deployed on a faulty processor to the available capacity on other processors.

Re-deployment of an application is similar to deployment of a starting application, except that it must be restored to the state that was captured in its last checkpoint. The fault model that we use assumes that the instructions of an applications are stored in a central, protected memory and are fetched once when the application starts. The data communicated between the actors is also stored in the central memory, even if two actors are mapped on the same processor. This model is further detailed in Chapter 5. Therefore the state will be consistent after a fault, and data memory access times are the same on each processor.² In this fault model the time for re-deployment depends solely on the resource manager and the fetching of data and instructions, which greatly simplifies the analysis. As explained in SL.1, the behaviour of

² In most real systems there will be minor differences, and we must consider the worst-case.

the resource manager does not interfere with other running applications.

SOLUTION: *We select a method for software fault recovery that relies on the resource manager which does not interfere with other running applications, solving SP.3.*

- CB.4 **DESIGN FLOW REQUIREMENT:** *The design flow must balance the resource requirements of applications to increase the probability of success of future run-time mapping and scheduling actions.*

In the selected design flow, SDF³, the *Design Space Exploration (DSE)* trades throughput against the memory footprint during step DS.M&S [132, 135]. In Chapter 4 we introduce a third dimension to this trade-off, namely the size of the VPs, which determines the utilisation of the physical processors. We show that this three-dimensional design space is too large to search exhaustively even for a small example. The throughput changes independently with both the VP size and memory footprint, and these dimensions should therefore be explored simultaneously rather than one after another. We contribute a heuristic to explore the trade-off between VP size and memory footprint under a fixed throughput constraint. The resulting DSE returns points that are on a Pareto-optimal curve and allow a designer to balance the VP size and memory footprint of applications. This can be used as a tool to shape the solution space of the run-time mapping heuristic for dynamically changing sets of applications.

CONTRIBUTION: *We propose an extension of the SDF³ DSE that allows trading VP size against buffer size in Chapter 4, enabling resource balancing and increasing the success of future run-time mapping actions, thereby solving SP.4.*

- SL.5 **DESIGN FLOW REQUIREMENT:** *Once an application is started it must receive its required resources at all times, independent of the scenario switches.*

Applications transition unpredictably between scenarios, causing varying resource requirements. Let us divide the platform resources in two types, stateless and stateful. In a stateless resource, data related to the state of an application may be stored during the *firing* (execution) of actors from that application, but no state is stored in between actor firings or when another application executes on the resource. In the CompSOC platform a number of measures are taken to guarantee that the processor is stateless from the applications view [92, 93]. The memories on the other hand are stateful resources, i.e. they contain state even when no actor of that application is firing.

Both types of resources of a running application can be allocated in two ways: at the application start, or at the start of a scenario. Stateful resources, i.e. the memory, are allocated at the application start which means that the state does not have to be stored and loaded at each scenario switch. While this avoids possible violations of the resource constraints, it means the memory requirements for all scenarios are summed and are allocated as long as the

application is running. Stateless resources, i.e. the processor and interconnect, are allocated at the start of a scenario. To avoid that an application fails at a scenario switch because insufficient stateless resources are available, the maximum over all scenarios and transitions (i.e. the worst-case) is allocated as a budget for the application. In other words, the stateless budgets for each application are sufficiently large to allow all on-demand allocations. Given the constraints we consider this combination of resource allocation the best possible solution.

This strategy for resource allocation is a simplification that guarantees that sufficient resources are available for each scenario transition, but may lead to over-allocation of resources in the average case. The described solution is already implemented in SDF³ and is not a contribution of this thesis.

SOLUTION: The stateful resources for a running applications are allocated at the application start, whereas the stateless resources are allocated at the start of a scenario. This strategy ensures that an application always receives it required resources and prevents application failures at scenario boundaries, solving SP.5.

CB.6 DESIGN FLOW REQUIREMENT: *The design flow must maximise the probability of success of future run-time re-mapping and re-scheduling actions due to processor faults.*

Re-deployment (inter-application mapping and scheduling at run-time) can be considered as a three-dimensional bin-packing problem, where the VPs are the items and the processors, memories and connections the bins. The probability of successful re-deployment of VPs after a fault depends on the space in the bins, of which the processor capacity is especially crucial as one bin was just removed by the fault. Applications have to be dropped if there is insufficient capacity for re-deployment. The balancing of processor capacity and memory footprint presented in contribution CB.4 somewhat increases the probability of success. We propose two methods that focus only on the processor capacity and further maximise the probability of successful re-deployment in Chapter 5 [148].

Firstly, we map each application to more VPs of a smaller size at design-time to increase the probability of successful bin-packing at run-time, at the cost of an increase of the sum of all VP sizes. Secondly we generate multiple mappings at design-time in which VPs are split or resized, and allow switching to such an alternative mapping at run-time.

Contribution: We propose to map the applications to more VPs of a smaller size in Chapter 5, thereby maximising the probability of successful future re-deployment after a fault, thus solving SP.6.

	RQ.1	RQ.2	RQ.3a	RQ.3b	RQ.4	RQ.5
this thesis	✓	✓		✓	✓	✓
existing SDF ³ and CompSOC [2]	✓	✓	✓		*	X
Schor [113]	✓	✓		✓	X	✓
van Stralen and Pimentel [149]	✓	✓		*	*	X
Wildermann et al. [153]	✓	✓		*	*	X
Quan and Pimentel [101]	✓	✓		*	*	X
Weichslgartner et al. [150]	✓	✓		*	X	X
Moreira [86]	✓	✓	✓		✓	X

Table 1.3: A comparison of the design flow described in this thesis with related work. A checkmark (✓) indicates that a requirement can be met, an asterisk (*) means that it can be partly met and a cross (X) means that a requirement cannot be met in that design flow. Note that RQ.3a and RQ.3b are mutually exclusive.

1.5 Related work to overall thesis goals

A number of design flows described in related work implement one or two of the three new requirements RQ.3b–RQ.5 with the first four. In this section we discuss all related work that implements two of the three requirements, and omit work that implements only one. Table 1.3 lists the work discussed in this section and indicates which requirements can be satisfied in each flow. Note that some requirements can be satisfied only partially, indicated by an asterisk (*). We will now discuss each of these works in detail.

The combination of the existing SDF³ design flow and CompSOC platform was introduced in Section 1.4, and will be discussed in detail in Chapter 2 [2, 45, 131]. The SDF³ flow creates absolute resource budgets for fixed sets of applications, RQ.3a. Though SDF³ supports the FSM-SADF MoC there is no PM, hence RQ.4 is only partially met. Finally fault-tolerance is not supported in either SDF³ or CompSOC.

A design flow for mapping dynamically changing sets of applications onto heterogeneous multi-processors is proposed in [113, 114]. It supports RQ.3b and RQ.5, although they use the term *scenario* for what we call *dynamic sets*. Scenarios for different application behaviours as considered in this thesis are not supported. However, they use the *Kahn Process Network (KPN)* MoC in which dynamic application behaviour is not explicit in the model, which complicates design-time analysis [63]. Precise timing analysis of this model is not possible, instead the timing behavior may be analysed statistically which does not result in hard guarantees [89]. Therefore this work strikes a trade-off between RQ.1 and RQ.4, but does not completely fulfill either. A number of optimal mappings is calculated at design-time, which means only a limited number of events can be handled at run-time. Spare processors

are reserved to which the tasks of a faulty processor may be migrated at run-time. Therefore the probability of successfully handling one fault without dropping applications is 100%. Additional faults however cannot be handled, while our approach may handle multiple faults depending on the initial platform utilisation.

The following works all propose a solution to sub-problem SP.1 and to some degree implement RQ.3b and RQ.4, but not RQ.5 [101, 149, 150, 153]. We will now discuss the differences to solution SL.1, which itself is not a contribution of this thesis but the foundation on which we build our contributions.

The work described in [149] proposes a solution to sub-problem SP.1 that addresses RQ.3b and RQ.4 at the same time. They consider intra-application scenarios which are identical to our scenarios, and inter-application scenarios that describe which sets applications may run concurrently. In other words, mappings of different applications sets are fixed at design-time between which resource manager may switch at run-time. This solution is in between RQ.3a and RQ.3b and does not offer the freedom to start and stop applications at will, ruling out online software updates. Requirement RQ.4 is also met only partially, as the intra-application scenario is fixed in each mapping. The design space is explored using a coevolutionary genetic algorithm and results in a representative subset of mappings that combine intra- and inter-application scenarios. Fault-tolerance is not considered but could be implemented using inter-application scenarios.

In [153] the border between intra- and inter-application scenarios is blurred and generalised in the term *multi-mode* systems, where a mode is a fixed set of applications that are each in a specific scenario. Again multiple mappings of fixed application sets are calculated at design-time so that RQ.3b and RQ.4 are both partially met. This solution is comparable to [149].

An improvement on [149] and [153] is presented in [98, 101] by combining design-time mapping with a two-step run-time mapping. In the first run-time step, a mapping is selected that matches the workload scenario. In the second step minor mapping customisations are performed to optimise that mapping. This allows a wider selection of mappings at run-time without having to store them all. While this offers more freedom than the previous two approaches, requirements RQ.3b and RQ.4 are still not fully met.

The work in [150] is similar to the design flow that we selected, as it combines design-time static performance analysis with an isolated execution environment per application at run-time. This guarantees that the design-time guarantees are not violated. After selecting a set of Pareto-optimal operating points (mappings) in the DSE, one operating point is selecting at run-time by solving a constraint satisfaction problem. Complexity at run-time is reduced because the resource manager does not have to consider all possible combinations but only a representative subset. This works for a relatively small set of applications, but as the size of the set increases the number of operating points to be stored still increases exponentially and so does the time to select one. Another limitations is that new applications cannot be added to the system during run-time. Thus requirement RQ.3b is satisfied to a large degree, but not completely. This work does not consider dynamic applications, RQ.4.


A two-layer scheduling approach for dynamic sets on heterogeneous multi-processors similar to the one used in this thesis is presented in [86]. It mixes TDM scheduling of clusters of actors (what we call VPs) with SO scheduling within each cluster. A scheduling flow is presented that finds combined TDM and SO schedules in which the applications meet a minimum throughput and maximum latency. Both types of schedules are modeled in *Homogeneous Synchronous Dataflow (HSDF)* graphs by adding nodes and edges. This flow is very similar to the existing design flow that we selected [2].

1.6 Thesis outline

At the start of this chapter we introduced two use-cases and extracted the common requirements for static and dynamic RT systems. We introduced a general design flow template in Section 1.2. The main goal of this thesis is to create a design flow for dynamic RT systems that adheres to all requirements listed in Section 1.1. Six sub-problems must be addressed to achieve this goal, which are found in Section 1.3 by combining each static with each dynamic requirement.

To solve these sub-problems we select three solutions from related work in Section 1.4, including an existing design flow that is detailed in Chapter 2. The three contributions that solve the other sub-problems are the core of this thesis, and are presented in Chapters 3–5. The first contribution, CB.2, is a PM for FSM-SADF. It is build on a concept for scenario sequencing that solves the causality problem, and is presented in Chapter 3. Chapter 4 contains the second contribution, CB.4, in which we explore the trade-off between the processor utilisation and memory requirements. This leads to a three-dimensional design space that now includes the VP size, we propose a DSE algorithm to explore this space.

Contribution CB.6 is presented in Chapter 5 and consists of two methods that increase the fault-tolerance. We introduce the fault model and zoom in on re-deployment of the VPs from a faulty processor with the aid of mappings calculated at design-time. Each chapter contains an experimental evaluation that shows the validity of the proposed concepts. The conclusions and suggestions for future work can be found in Chapter 6



2 Background and terminology

2.1 Introduction

2

In Section 1.2 we introduced the steps that make up a design flow for RT embedded systems, and in Section 1.4 we selected an existing design flow as solution SL.1 for sub-problems SP.1 and SP.6. To solve the three open sub-problems we extend the selected flow in Chapters 3, 4 and 5 of this thesis. In this chapter we provide the background knowledge and explain the terminology that is necessary to understand these contributions.

We start with an overview of the existing flow in Section 2.2, and highlight where the flow will be extended in the later chapters. All steps of the flow are explored in detail in the remaining sections. The FSM-SADF MoC is explained in Section 2.3 using the example of a video decoder. A concept for the mapping and scheduling has been given in Section 1.4, more terminology and technical details are provided in Section 2.4. The SDF PM is the foundation for the FSM-SADF that we present in Chapter 3, and is discussed in Section 2.5. Similarly, the libraries that implement SDF are discussed in Section 2.6. More details on the CompSOC hardware platform are presented in Section 2.7. The chapter concludes with a summary.

2.2 Design flow

The selected design flow [2, 46] consists of the SDF³ tool for analysis, mapping and scheduling of RT dataflow applications [38, 62, 131, 133, 134, 137], and the CompSOC hardware platform and middleware [44, 45, 48].

A flowchart of the existing design flow is depicted in Figure 2.1. The blocks are color coded to indicate whether the algorithms are part of SDF³ or CompSOC, and whether files are input or output. The matching design step from Section 1.2 is denoted besides each block, and the mapping and scheduling concept corresponds to that in Figure 2.1. This design flow is extended in the context of this thesis, the updated flow is shown in Figure 2.2, whose mapping and scheduling concept corresponds to that in Figure 1.6. The contributions are indicated by a darker shade and white text, some existing algorithms have been updated and a few new ones added. The algorithms and files of Figure 2.1 are explained in the remainder of this chapter. Figure 2.2 will be explained step by step in the remaining chapters of this thesis.

Step DS.REQ, the requirements, is conceptual. Requirements may be expressed textually as has been done in Chapter 1, or graphically, e.g. using the *Unified Modeling Language (UML)* [151]. Requirements form the foundation for writing FSM-SADF

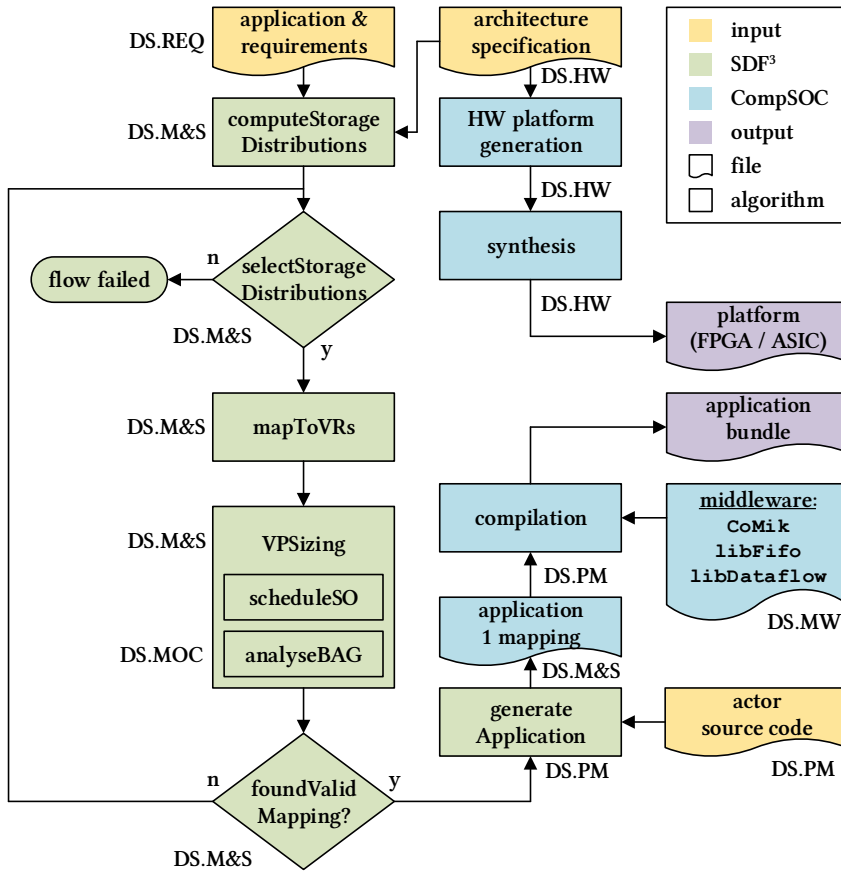


Figure 2.1: A flowchart of the existing design flow. The corresponding design step is denoted besides each block.

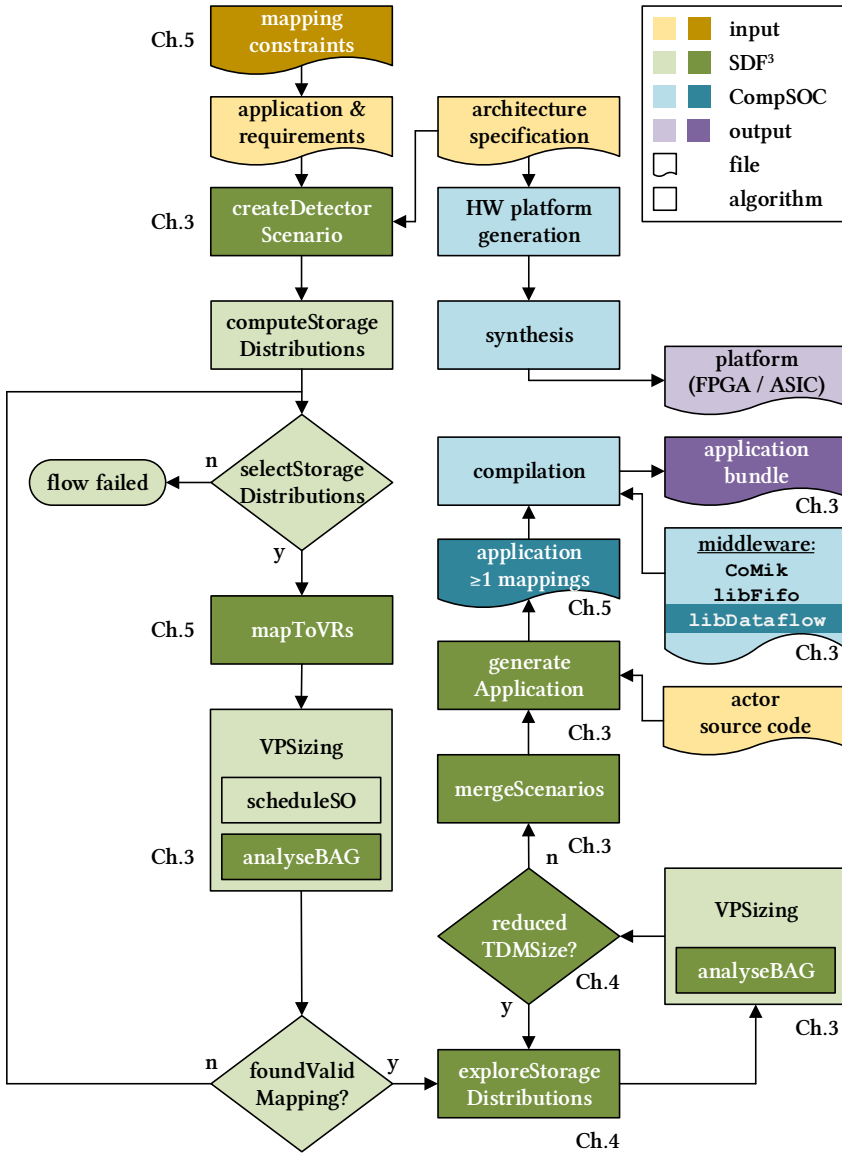


Figure 2.2: The extended design flow. Design steps that are new or updated to implement the contributions are indicated by a darker shade and white text. The corresponding chapter is denoted besides each contribution.

applications using the SDF³ syntax [146]. The only requirement that is directly contained in the input file *applications & requirements* is the throughput constraint.

Step DS.MOC, the MoC, is a mathematical model composed of definitions and formulas and will be presented in Section 2.3. The design flow manipulates and analyses applications that are designed using the MoC using software algorithms contained in the *analyseBAG* algorithm in Figure 2.1. The *Binding-Aware Graph (BAG)* and intra-application SO scheduling will be further explained in Section 2.4.

The mapping and scheduling, step DS.M&S, is implemented in a loop that contains three algorithms. This loop is explained in Section 2.4 and extended for contributions CB.2 and CB.6 in Chapters 3 and 5. Contribution CB.4 requires a second loop for exploring the storage distributions, which is added in Chapter 4.

Step DS.PM, the PM, consists of the *generateApplication* algorithms and a number of files. The *actor source code* is inserted into a framework of empty actors generated by the algorithms, and the `libDataflow` library is compiled into the *system* to execute these actors according to the MoC rules. One mapping is created per application, corresponding to Figure 1.5. More details are provided in Section 2.5.

The CompSOC middleware, step DS.MW, implements the PM. It comprises the `CoMik` μ kernel that implements the inter-application scheduling of VPs, and `libFifo` that provides an implementation for dataflow channels. Together with `libDataflow` these libraries implement the PM for HSDF, SDF and *Cyclo-static Dataflow (CSDF)*, the basics of which will be elaborated in Section 2.6.

Step DS.HW, the hardware, is implemented by the remaining CompSOC blocks and results in a platform bitfile for *Field Programmable Gate Array (FPGA)* or *Application Specific Integrated Circuit (ASIC)*. In case of an FPGA bitfile the application bundles may either be combined into the bitfile if a static mapping is desired, or the bundles can be loaded at run-time as shown in 2.1 [121]. The hardware properties and concepts that are of interest for the contributions of this thesis are discussed in Section 2.7. The existing flow presented thus far can handle HSDF, SDF and CSDF applications. The SDF³ tool can also analyse, map and schedule FSM-SADF applications, but these are not supported by CompSOC PM and middleware.

2.3 Model of Computation

Dataflow is a natural way to describe data-dependent application behaviour [74]. We selected the FSM-SADF MoC in Contribution CB.2 as it supports dynamic responses to data, RQ.4. An FSM-SADF application consists of a collection of scenario graphs that adhere to the SDF rules and could be executed stand-alone as SDF applications. It is possible to switch between scenarios after one full *iteration* (execution) of a scenario graph. The switches that are allowed are captured in a *Finite-State Machine (FSM)*. In this section we motivate our choice for this MoC compared to two other suitable MoCs and go through its properties in detail.

2.3.1 Comparison to other MoCs

The advantage of dataflow MoCs over others such as the *Time-Triggered (TT)* model [68] is that they are work-conserving when combined with the proper scheduling algorithm [64], yet it is possible to find a tight bound on the throughput [41, 132]. In Section 1.3 we saw that there are three dataflow variants that support scenarios, namely SADF, FSM-SADF and MCDF. Two types of throughput analysis are available for SADF. The first of these is based on Markov chains, which allows evaluation of any long-term average or worst-case performance metric [139–142]. In the second method each SADF scenario is converted to an SDF graph, that is suitable for state-space analysis. The overall throughput is the worst-case, i.e. the maximum over all scenarios, which essentially negates the performance benefit achieved by scenarios.

The first method is preferred as it exploits the performance benefit. The probability of each scenario transition is captured in a Markov chain, and scenario transitions within a scenario iteration are allowed. The analysis must implement these operational semantics, which leads to a much larger state-space than that of SDF analysis. FSM-SADF captures scenario transitions in an FSM and MCDF encodes these in a mode controller [83, 136]. Both are a restricted version of SADF, which means that the analysis must implement fewer operational semantics.

Multiple analysis methods are available for FSM-SADF and MCDF. Firstly, they may be converted to HSDF graphs, after which the throughput can be calculated with a *Maximum Cycle Ratio (MCR)* analysis given the actor WCETs [3, 26, 75, 76]. In such a conversion the information on scenarios is lost, which again negates their performance benefits. Secondly, simulation-based analysis may be performed using regular state-space or $(max, +)$ analysis [36]. Regular state-space exploration searches the entire state-space. This may lead to a state-space explosion because the size of the state-space grows exponentially with each additional scenario. Analysis using $(max, +)$ algebra constructs the state-space more efficiently, although state-space explosions may still occur [37, 123]. Other methods exist but are not exact and therefore inferior to $(max, +)$ analysis, though they may be faster [66, 83].

FSM-SADF and MCDF are similar in terms of expressiveness, but $(max, +)$ analysis is not implemented for the latter and other tools are not readily available. We therefore select FSM-SADF as our MoC of choice.

2.3.2 Application scenarios

In requirement UC-1.6 of the first use-case in Section 1.1 we saw that a video decoder application must determine whether an incoming video frame is a full frame or a delta frame, and call the appropriate decoding function. In a sequential language a programmer may solve such a dependency with a straightforward *if-else* construct as shown in Algorithm 1.

Static MoCs such as SDF cannot respond to input-dependent behaviour, resulting in a conservative throughput bound. Consider Algorithm 1, where `decode_full()`

Algorithm 1 Pseudo-code of an abstract video decoder.

```

1: frame = buffer_frame()
2: if detect_frame_type(frame) = full then
3:   x = decode_full(frame)
4:   sub = subtitle_overlay(x)
5: else
6:   x = decode_delta(frame)
7: end if
8: output = construct_frame(x)
9: display(output, sub)

```

has a longer WCET than `decode_delta()`. Yet due to data dependencies invisible at this level of control flow, the WCET of `construct_frame()` might be longer for a delta frame. SDF analysis will consider the WCET of `decode_full()` and longest WCET of `construct_frame()` at the same time, although this situation can never occur. This results in an overly negative throughput bound [42]. FSM-SADF analysis considers a unique WCET for each actor in each scenario, and is therefore able to exclude such situations that cannot occur in reality. The worst worst possible scenario sequence must be accounted for in the FSM [138]. It has been shown that FSM-SADF can provide a tight bound on an applications worst-case throughput [37].

FSM-SADF allows describing different behaviours of the application by capturing these in scenarios. For example, different control flows can be captured by varying the graph topology. Another use for scenarios is to vary the degree of parallelism in an application to deal with varying resource availability or *Quality-of-Service (QoS)* requirements. The designer must capture each behaviour of an application in a scenario graph, possibly based on existing sequential code.

2.3.3 Scenario graphs

The scenario graph for decoding a full video frame (S_{full}) is a directed graph, see Figure 2.3a. The nodes represent the *actors*, each with a known WCET. The actor names in Figure 2.3a are abbreviations of the functions in Algorithm 1, e.g. **bf** for `buffer_frame()`. An actor is the minimal schedulable unit in dataflow terminology, we will from now on use this term instead of task. The edges represent *channels*, the *rates* at their start and end indicate how many data *tokens* are produced and consumed each time the actors *fire* (execute). If the rate is not denoted on a channel, it is one.

A scenario graph may furthermore contain *persistent tokens* that are present at the start and end of each iteration of the graph, these are indicated with labels such as t_i . The graph completes one full iteration when all actors have fired at least once and all persistent tokens have returned to their starting positions. A token is persistent

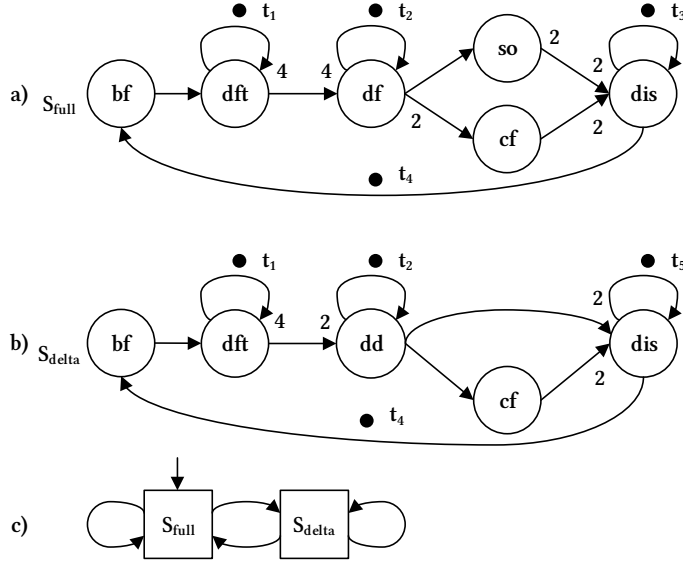


Figure 2.3: **a)** Scenario graph for decoding a full video frame, S_{full} . **b)** scenario graph for decoding a delta video frame, S_{delta} . **c)** The FSM of the video decoder.

if it exists in between the iterations of a graph. Tokens are units of data and can have a different size in each channel, but must always have the same size in one channel.

Actors communicate by exchanging tokens of data that propagate through the channels. Each channel has a production rate at the output port of the source actor, and a consumption rate at the input port of the destination actor. The directed edges resembling channels point from source to destination. The production rate and consumption rates of the channel from actor **dft** to **df** for example are both 4. By convention the rates are omitted when they are 1. Different rates may result in a different *repetition vector entry*, which is the number of times an actor fires each iteration.

An actor may fire if the amount of tokens on each of its incoming channels is equal to or greater than the consumption rate. When an actor fires it consumes a number of tokens equal to the consumption rate from each incoming channel. It then executes the program code embedded in that actor, and concludes by producing a number of tokens on each outgoing channel equal to the production rate of that channel. Actors themselves are stateless, that is, they cannot store data in between subsequent firings.

A channel with the same source and destination actor is termed a *self-edge*, see actors **dft**, **df** and **dis**. Self-edges can be used to model state and to limit multiple simultaneous actor firings (auto-concurrency).

2.3.4 FSM-SADF applications

An FSM-SADF application consists of two or more scenarios and an FSM. The full frame scenario S_{full} described so far only captures one behaviour of the application, namely the decoding of a full video frame. When a delta frame is detected, the application behaviour and thus the scenario graph is different, see S_{delta} in Figure 2.3b. Some actors occur in both scenarios, possibly with different rates: {bf, df, cf, dis}. Other actors occur only in one scenario but might consume a persistent token that occurs in multiple scenarios: {df, dd, so}).

The allowed scenario sequences are specified by the FSM in Figure 2.3c. A video sequence always starts with a full frame scenario, S_{full} , this is indicated in the FSM with an arrow that is not connected on one end. A full frame may be followed by either scenario type, and the same goes for a delta frame. We see that in this application all scenario sequences are allowed.

2.4 Mapping and scheduling

Design step DS.M&S indicated in Figure 2.1 shows the intra-application mapping and scheduling of the existing design flow, the upper layer of Figure 1.5. The mapping and scheduling concept of SDF³ has been described in [131]. We review the relevant parts in this section.

2.4.1 Storage distributions

The first algorithm, *computeStorageDistributions*, takes as an input the application file and the architecture description. An application file contains the following information:

- the throughput constraint;
- the scenario graphs;
- the resource requirements for each actor:
 - the WCET on each processor type;
 - the required amount of instruction memory;
 - the required amount of data memory.
- the size of the tokens on each channel;
- the *Finite-State Machine (FSM)*.

The CompSOC platform features multiple *tiles* that each contain one or more processors, local memories, and interfaces to the interconnect. In this work we assume there is one processor per tile that has separate on-tile instruction and data memories as well as one or more communication memories. An instance of the CompSOC

platform may furthermore contain shared memories, an interconnect and peripheral devices such as an Ethernet interface. We consider heterogeneous platforms that are *Globally Asynchronous, Locally Synchronous (GALS)*, meaning that all elements on one tile have the same clock but different tiles and devices have different clocks [71]. It is not possible to make assumptions about synchronisation. An architecture file contains the following information:

- a per tile description of:
 - interfaces to the interconnect;
 - the processor architecture;
 - the processor arbitration method;
 - the size of the instruction, data and communication memories.
- the interfaces to the interconnect and size of each shared memory;
- the connections within the interconnect;
- the interfaces to the interconnect and type of each peripheral.

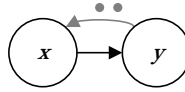


Figure 2.4: The size of the buffer of a channel from actor x to y is modelled by the grey channel and two persistent tokens, indicating a buffer size of two.

In the MoC there is no notion of buffer size, i.e. the number of tokens that can be stored in a channel. In reality there is a limit on the number of tokens because they must be stored in a physical memory. This has to be taken into account during the mapping. A *storage distribution* is a list in which the buffer size of each channel is fixed. The buffer size of a channel may be modeled in a dataflow graph by adding a channel in the direction opposite to the original channel, and inserting a number of persistent tokens equal to the buffer size. See Figure 2.4, the channel from x to y has a buffer size of two tokens.

The storage distribution impacts the throughput of an application. For instance, the graph will surely deadlock if the buffer size is lower than the production or consumption rate of a channel, resulting in a throughput of zero. The maximum possible throughput on the other hand will be reached if the buffer size of each channel is made enormous, which conflicts with the desire to minimize resource usage (RQ.2). The goal of the design flow is to minimize the memory usage while meeting the throughput constraint. Therefore the *computeStorageDistributions* algorithm creates a set of storage distributions from which a subset is selected by *selectStorageDistributions* to serve as the foundation for the mapping and scheduling.

2.4.2 Mapping applications to VRs

In the next algorithm, *mapToVRs*, a number of mappings is created for each storage distribution. Each mapping is an allocation of actors to *Virtual Resources* (VRs), where it is made sure that the memory requirements of the actors fit within the capacity available on each tile. The mapping to processors (VPs) and memory are of special interest in this thesis, other VRs such as the interconnect, shared *Static Random-Access Memory* (SRAM), *Dynamic Random-Access Memory* (DRAM), and peripherals will not be discussed in detail.

The size of the VP cannot be determined yet, as it depends on the intra-application scheduling. An upper limit on the size of a VP is given by the maximum the processor capacity that a VP may consume, which is denoted in the architecture file. If an application is to be executed stand-alone on a platform, the full processor capacity may be given in the architecture file. If an application shares the platform with others, the architecture should give the full capacity minus the size of the VPs required for other applications.

The inter-application deployment of VRs is shown in the lower layer of Figure 1.5. On the CompSOC platform the physical processor arbitration is implemented in the CoMik μ kernel using *Time-Division Multiplexing* (TDM) scheduling [45, 93]. TDM scheduling divides the available capacity in a finite number of TDM *slots*, this is further detailed Section 2.6. The VP size is an absolute budget that describes the exact TDM slots that the VP requires on a physical processor to meet the throughput constraint. As discussed in SL.1 there is no obstacle to using relative VP budgets.

2.4.3 Determining the VP size

In the *VPSizing* algorithm the TDM slots required for each VP are determined using a binary search. The search adjusts the TDM budget of all VPs at the same time in an attempt to find the minimum budget for which the throughput constraint is satisfied. A SO schedule is created for each scenario in each VP using the *scheduleSO* algorithm. A *Static-Order* (SO) schedule is a list that gives the order in which actors will fire inside the VP during run-time.

To determine the throughput of a mapped application one must take into account the timing effects of the scheduling, interconnect and remote memory accesses as well as the interaction between actors mapped to different VPs. This is solved in the *analyseBAG* algorithm, in which the scenario graphs are annotated with models of the middleware and hardware to construct the *Binding-Aware Graph* (BAG) of the application. Dataflow models of the CompSOC hardware platform and middleware are described in [92, 94]. SO schedules can be modeled as described in [20]. The WCET of the additional actors generally decreases the throughput of the application. Once the BAG is constructed according to the rules of the MoC its throughput is analysed as described in Section 2.3.

A mapping is valid if the throughput constraint is satisfied and the size of the VPs does not exceed the maximum capacity that was given in the architecture file. All valid mappings are collected in a set. If the set is empty, the flow attempts to select other storage distributions. When no storage distributions are left, the flow fails. If one or more valid mappings were found, all Pareto-optimal mappings with respect to the usage of memory, interconnect and processors are selected from this set. For each of these mapping the *generateApplication* block creates an actor framework for execution with the *libDataflow* library, see Section 2.6.

2.5 Programming model

A PM is a set of constructs that enables a programmer to write applications that behave according to the rules of the MoC when running on a platform. In literature the term PM is sometimes used interchangeably with *Model-of-Execution (MoE)* [89]. No PM is available for FSM-SADF on CompSOC or any other platform. PMs for HSDF, SDF, KPN and CSDF are available for CompSOC in the form of the *libDataflow* library (formerly *Comp0Se* and *libPose*), see Figure 2.1 [89–91, 94]. In this section we describe the CompSOC SDF PM, which is the foundation for our FSM-SADF PM in Chapter 3.

The SDF PM consists of four main elements: *First-in First-out (FIFO)* buffers, functions, firing rules and SO scheduling. Channels are realised as FIFO style buffers in which no data can be lost and the oldest tokens are the first to be consumed. Actors are implemented as functions with two arguments. The first argument is a list of the input tokens, the second a list of spaces into which the output tokens must be produced

Before an actor may fire it must be checked whether sufficient input tokens are available for consumption in the input FIFOs, and whether sufficient space is available in the output FIFOs to produce the output tokens. These conditions are termed the *firing rules*, and must be checked before an actor is started. If sufficient tokens and spaces are available, the corresponding locations in the FIFO buffers are passed to the actor. Once the computation has finished and the output tokens are copied into the FIFO buffers, input tokens may be released as empty buffer spaces and the actor has finished its firing.

The actors must furthermore be scheduled inside each VP according to the *Static-Order (SO)* schedule created in the *scheduleSO* algorithm. The schedule lists the order in which the actors should be fired. At run-time these steps are executed in the following order. The scheduler selects the next actor to be fired, after which the firing rules are checked for that actor. If the rules are satisfied the actor is called with the corresponding tokens as arguments.

2.6 Middleware

The CompSOC middleware implements the VP deployment described in Section 2.4 and the PM described in Section 2.5. It consists of CoMik, libFifo and libDataflow, see Figure 2.1. Other libraries are available but not relevant in the context of this work [45].

Partitioning can be used for consolidating multiple applications on one platform [52], but in order to meet RT requirements a partition must be temporally isolated [93, 143, 156]. Partitioning of processor resources is of special interest in this thesis, as a dynamically changing set of applications must be executed on a heterogeneous multi-processor (RQ.3b). *Composability* means that the timing behaviour (and therefore the performance) of one application is completely independent of the other applications executing on the same resources [45].

2.6.1 CoMik μ kernel

The CoMik μ kernel implements composable partitioning of processors [50, 92, 93]. TDM processor allocation defines a finite period for each resource, the TDM wheel. This period is divided into a number of evenly spaced time slots of a certain length, measured in clock cycles. The number of slots is the wheel size and must be a natural number. We consider the VPs to be a *requestor* that consumes capacity on a processor. Each requestor that needs to use the resource is designated a number of TDM slots, the inter-application deployment. Within their slots a requestor has exclusive access to the resource, while in all other slots it has no access to the resource and has to wait.

On the CompSOC platform the VPs are deployed statically according to a given table. The TDM slots are cycle-accurately composable, an application that executes in one slot cannot affect an application in another slot even by a single cycle. In between the TDM slots is a CoMik system slot that takes care of the scheduling, interrupts, exceptions, power management and memory management. Within the TDM slots reserved for a VP an application may execute bare-metal or start an OS such as libDataflow. A side effect of this composable partitioning is that a VP may be executed in any set of slots, as long as the total number of slots is equal to the VP size that was calculated by the *constructpVPSchedules* algorithm. We exploit this feature to perform re-deployment of VPs to achieve fault-tolerance in Chapter 5. The technique of deploying applications onto a platform through dynamic loading is described in [121]. However, no resource manager with a VP deployment heuristic such as described in Chapter 1 is available for the CompSOC platform. Therefore the platform has only been used with fixed actor to processor mappings (RQ.3a), and mostly with fixed actor to connection mappings.

2.6.2 libFIFO

The `libFifo` library (formerly C-HEAP) implements the FIFOs as circular buffers [45, 95]. Each FIFO has an administration block that must be initialised when the system is started. The parameters that must be initialised include the size of the tokens, the total number of tokens (i.e. FIFO capacity), the rates and the address where the buffer resides. If the FIFO connects two actors in the same VPs, the buffer may be located in the local data memory or in a remote shared memory. Else if the FIFO connects two actors in different VPs, the buffer can be located in a local communication memory of one of the tiles to which a VP is mapped, or in a remote shared memory. When the actors are on different VPs send and receive buffers are used because the processors typically produce and consume tokens faster than the interconnect can transport them. The administration block contains read and write counters that track the status of the buffer. The library furthermore provides functions to check the current number of tokens and spaces, read and write tokens and release spaces.

2.6.3 libDataflow

The `libDataflow` library (formerly `Comp0Se` [50] and `libPose`) implements the remainder of the dataflow PM, namely the loading of persistent tokens, SO scheduling, firing rule checking, and starting of actors [89, 91, 92]. The scheduling task selects the next actor for execution. Scheduling is *predictable*, i.e. the worst-case performance is guaranteed [45]. Two non-preemptive scheduling methods are available in `libDataflow`, *Round-Robin* (*RR*) and SO scheduling. We use the SO scheduler, which is supplied with a fixed schedule generated by SDF³ that indicates in which order the actors must fire.

The firing rule checking as has been explained in Section 2.5 is an administrative task that loops over all the input and output FIFOs and checks if sufficient tokens and spaces are available. If this is the case, the actor function is called with as arguments the input tokens and spaces for the output tokens. Because an actor cannot start before the required number of input tokens is available, the scheduling is *data driven*. After the actor has finished the FIFO administration is updated to indicate that the former input tokens are now empty buffer spaces, and that the former spaces in the outgoing channels are now data tokens.

2.7 Hardware

The CompSOC hardware platform is a template from which different implementations can be synthesised from an architecture file, see Figure 2.1 [44, 45, 48]. A synthesised platform typically consists of a number of tiles, an interconnect, a shared memory and a number of peripherals. CompSOC is suitable to implement RT systems that adhere to requirements RQ.1–RQ.3a because it is predictable and composable

. These terms were explained for the middleware in Section 2.6, in this section we summarise how they are realised in hardware.

From the hardware point of view we consider an actor to be a *requestor* that consumes capacity on resources such as a processor, memory and interconnect. A request for capacity from a requestor has a certain WCET on each resource. The WCET must be safe, i.e. never underestimate the execution time, and should be tight, i.e. the overestimation should be as small as possible. [155]. A safe and tight WCET can be extracted only if each component is predictable. The WCRT accounts for the WCET and the waiting time that is caused by other requests, possibly from the same requestor. This waiting depends for a large part on the arbitration method that is used. Arbitration is predictable if it can guarantee a minimum performance, and is composable if it is ensured that the assigned capacity (i.e. performance) is independent of the behaviour of other applications. CompSOC implements multiple arbitration methods that are both predictable and composable. It is therefore possible to compute a WCRT which means that the worst-case behaviour of dataflow applications can be statically analysed.

2.8 Summary

This chapter started with a flowchart overview of the existing design flow shown side-by-side with the extended flow that we present in this thesis. We matched the algorithms in the original flow with the design steps from Section 1.2 and the extensions with the remaining chapters. We explained the existing design flow step by step, starting with the FSM-SADF *Model-of-Computation (MoC)* using the example of a video decoder. Each scenario in itself is identical to an SDF application, the allowed scenario sequences are captured in the FSM.

Mapping is performed on the basis of storage distributions that fix the size of each channel. The actors are then mapped to available capacity on the physical processors. Channels are mapped to the memories and interconnect. Each cluster of actors forms a VP for which a *Static-Order (SO)* schedule is created, and which itself is executed on a physical processor using TDM scheduling. Once an application is mapped and scheduled, the minimum size of each VP with which throughput constraint is satisfied can be determined. To do so a *Binding-Aware Graph (BAG)* is build that models the application, SO schedule, middleware and hardware.

As there is no PM for FSM-SADF we reviewed the SDF PM that enables execution of SDF applications according to the rules of that MoC. It defines channels as FIFO buffers and actors as functions. The firing rules check whether sufficient data tokens and space are available in the respective input and output channels of an actor, in which case an actor may be fired. An SO scheduler decides which actor must be fired next according to the given schedule.

Two elements of the CompSOC middleware are relevant in this thesis. Firstly, composable TDM scheduling of VPs is implemented by the CoMik μ kernel. Secondly, the SDF PM is implemented by libDataflow and libFifo. The CompSOC hardware is

predictable and composable. Predictability means that a safe and tight WCRT can be determined, which is necessary for the timing analysis of applications. Composability means that an applications performance is independent of other applications, which is needed to keep the complexity manageable when executing multiple applications on one platform.



3 A scenario-aware dataflow programming model

3.1 Introduction

This chapter presents contribution CB.2, a *Programming Model (PM)* for *Finite-State Machine Scenario-Aware Dataflow (FSM-SADF)*. In Section 3.2 we review the problem in detail and sketch the solution, which is composed of the following elements. A method and implementation for transforming an application into a scenario sequencing model that solves the causality problem is given in Section 3.3. We explain how this model can be converted to a scenario execution model in Section 3.4, and how it can be expanded into a *Binding-Aware Graph (BAG)* that captures the exact timing behaviour in Section 3.5. The PM is evaluated experimentally in Section 3.6. Related work is discussed in Section 3.7, and the chapter concludes with a summary in Section 3.8. Earlier versions of this chapter have been published in [146, 147]. These versions do not cover applications with delayed scenario detection, which is therefore a novel contribution of this thesis.

3

3.2 The causality dilemma

From the two use-cases in Section 1.1 we extracted common requirement RQ.4, dynamic response to input data. The control flow of applications that respond dynamically depends on the received data and can vary in each iteration. These variations can be captured in scenarios, for instance using the FSM-SADF MoC that was introduced in Section 2.3.

While FSM-SADF is an elegant model for tight analysis of dynamically responding applications, there is no corresponding PM to execute such applications. During analysis the allowed scenario sequences are captured in an FSM. During execution the actual scenario sequence must be *detected*. We recognise two different cases. Firstly, the current scenario may have been detected during execution of the previous scenario graph, we term this *delayed* scenario detection. Secondly, the current scenario may be detected during execution of the current scenario graph, we term this *immediate* scenario detection. All scenarios of an application must be detected in either a delayed or immediate fashion, the two types cannot be mixed.

3.2.1 Delayed scenario detection

Two example applications with delayed scenario detection are depicted in Figures 3.1a and 3.1b. Both applications consist of two scenarios and an FSM. The initial scenario that is to be executed during the first iteration after an application starts is fixed.

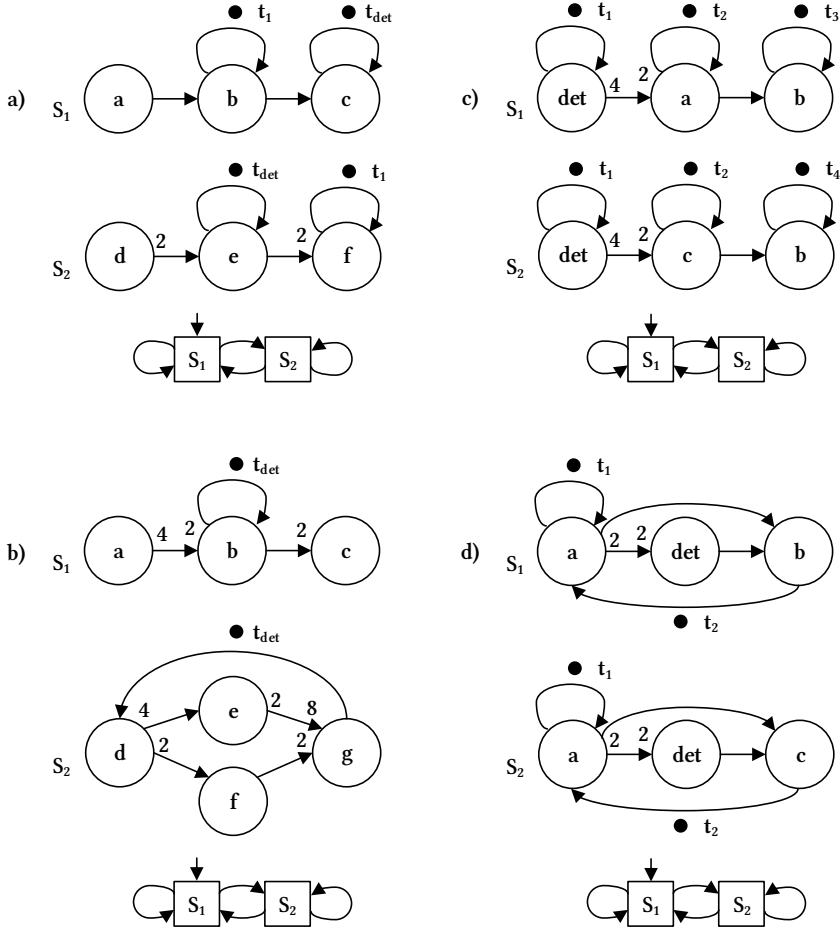


Figure 3.1: Four FSM-SADF applications that are supported by our PM. The applications in **a)** and **b)** have delayed scenario detection and have a detector token t_{det} in each scenario, those in **c)** and **d)** have immediate scenario detection and have a detector actor **det** in each scenario.

This is indicated in the FSM with an arrow that is not connected at the source. In the first iteration of both applications S_j is executed.

In all scenario iterations other than the very first the scenario was detected during the previous iteration. The only possibility to transfer state between scenario iterations is through persistent tokens. For our PM we require that the next scenario identifier is stored in a persistent token t_{det} . This *detector token* must appear exactly once in each scenario graph and may be located on a different channel in each scenario, see Figures 3.1a and 3.1b. Our PM requires the detector token to be identified by the programmer. In the MoC there is no such requirement; to calculate the worst-case scenario sequence it is sufficient to know the allowed scenario transitions indicated by the FSM. Figure 3.2a shows an application that is not supported by our PM because t_{det} does not occur in S_2 .

The benefit of delayed scenario detection is that the only constraint on the scenario graphs is that one detector token must be present in each scenario, its location is not relevant. Furthermore the scenario graphs do not have to be modified to create the scenario sequencing model as we will see in Section 3.3. An example of an application where the next scenario is detected during execution of the current scenario graph is a WLAN receiver [83]. Another example are applications that must either respond to input from a user, e.g. changing the channel in a software-defined radio application, or from another application, e.g. switching to a fail-safe scenario due to an error detected by the resource manager.

However, in any application where the control flow depends on the input data it is not possible to detect the next scenario during execution of the current scenario graph, and immediate scenario detection must be applied. An example of such an application is the MPEG-4 video decoder [137]. Another example is object detection, where the amount and type of processing steps is dependent on the objects that are detected. In the *Smallest Univalued Segment Assimilating Nucleus (SUSAN)* edge detection algorithm presented in Section 3.6 the choice between a sequential or parallelised version of the algorithm is made based on the detected resolution of the input image.

3.2.2 Immediate scenario detection

Two example applications with immediate scenario detection are depicted in Figures 3.1c and 3.1d. The first scenario is fixed and indicated with an arrow.¹ Unlike applications with delayed scenario detection, the current scenario is detected when a certain actor fires. After this *detector actor* has completed the current scenario is known. The idea of detector actors originates from [140].

Note that from the video decoder example in Section 2.3 we know that the first scenario is fixed by the specification, but is still encoded in the input data. Our PM

¹ For applications with immediate scenario detection the first scenario does not have to be fixed, as it is detected during the first scenario iteration anyway. This change would however require a formal change of the FSM-SADF MoC, which is outside of the scope of this thesis.

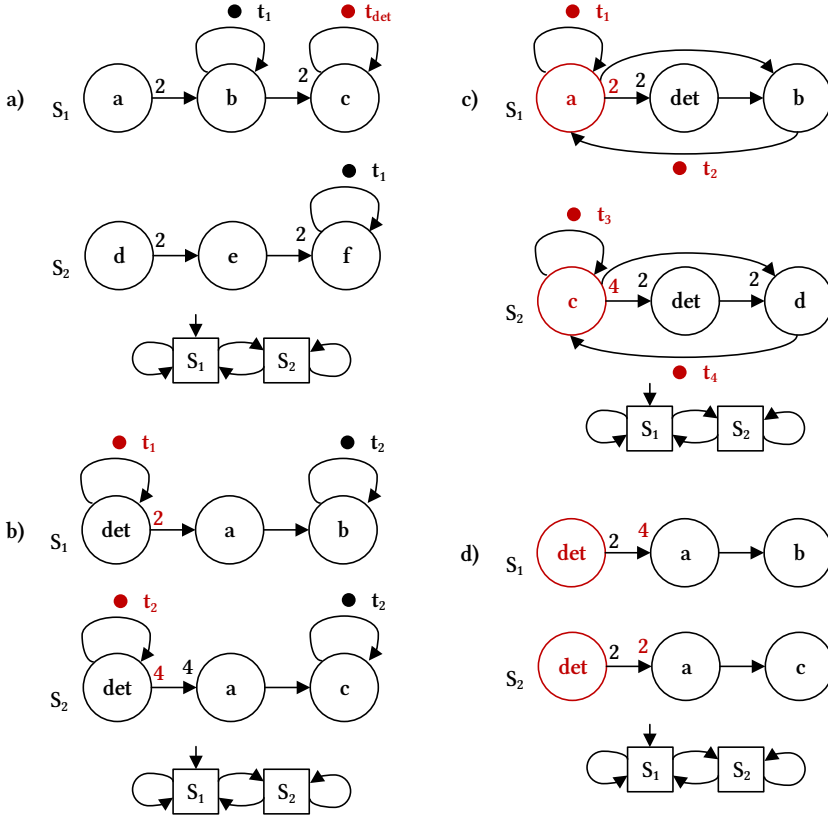


Figure 3.2: FSM-SADF applications that are not supported by our PM, the problems are highlighted in red. **a)** Delayed scenario detection, but there is no detection token t_{det} in S_2 . **b)** Immediate scenario detection, but the persistent token on the self-edge and the production rate of det are different in both scenarios. **c)** Immediate scenario detection, but the prefix graph that is executed before det is different in both scenarios. **d)** Immediate scenario detection, but the repetition vector entry of det is different in both scenarios.

requires the detector actor to be identified by the programmer, it is labeled **det** in Figures 3.1c and 3.1d. It is not required that the detected scenario is explicitly identified in any of the tokens produced on detectors outgoing channels. However, it should be possible to create new channels onto which such an explicit scenario token is produced, see Subsection 3.3.2. At run-time it is impossible to decide whether to start with executing the scenario graph of S_1 or that of S_2 , except in the first iteration. This is the causality dilemma; during execution the current scenario is detected only after the current scenario graph is partially executed.

For our PM we require that the detector actor is the same actor in each scenario. Furthermore it must have the same consumption and production rates at its ports. The consumption rate of a channel into which the detector actor produces tokens may differ in each scenario. Figure 3.2b shows an application that is not supported by our PM because the token consumed by **det** and its production rate is different in both scenarios.

All actors that fire before the detector actor together with the channels connected to those actors form the *prefix graph*. We require this prefix graph to be identical in each scenario, i.e. it must have the same actors, topology, rates and persistent tokens. If we would not introduce this constraint, there would be another causality dilemma before the detector actor even fires. The example in Figure 3.1d has a prefix graph that consists of actor **a**, four channels, and tokens t_1 and t_2 . Figure 3.2c shows an application that is not supported by our PM because the prefix graph is different in both scenarios. Note again that there is no such requirement in the MoC, during analysis it is not relevant in which actor the scenario is detected.

Another requirement that we pose on applications with immediate scenario detection is that the detector actor must have the same repetition vector entry in each scenario. Were the repetition vector entry different, another causality dilemma occurs while executing the SO schedule of the prefix graph. Figure 3.2d shows an application that is not supported by our PM because the detector actor has a different repetition vector entry in both scenarios.

The benefit of immediate scenario detection is that it suits applications that must directly respond to the input data, and it is therefore the solution to RQ.4 and the two use-cases in Section 1.1. The disadvantage is that we must pose a number of constraints on the scenario graphs.

The existing timing analysis in SDF³ uses the operational semantics of applications with immediate scenario detection. The operational semantics of applications with delayed scenario detection are different because the actual scenario lags one iteration behind the detected scenario. However, we argue that the throughput analysis is identical. The reason is that the analysis uses the FSM to determine the allowed scenario transitions, but does not model the actual scenario detection. In other words, it analyses all the possible scenario transitions, not the actual ones. The possible scenario transitions are the same in both cases, and therefore the throughput is the same.

3.2.3 Shared persistent tokens

The consistency of persistent tokens that occur in multiple scenarios must be guaranteed during scenario execution. Consider token t_2 in Figure 3.1c. Once actor **det** has fired and a scenario has been detected, say S_1 , actor **a** is enabled (ready to fire) to fire twice in succession. In the meantime **det** may fire again and detect S_2 , which will then execute in parallel to S_1 . This is an example of *scenario pipelining*. Actors **b** and **c** may attempt to access t_2 in the wrong order or at the same time. This is allowed, but the behaviour is not specified in the MoC. During scenario execution however this behaviour must be specified. The PM must clarify how the access to shared persistent tokens is handled, this is solved in Subsection 3.4.5.

3.2.4 Contributions

The FSM-SADF MoC was not designed to be executed. In this section we have seen that detector tokens or detector actors are necessary to execute scenario graphs, and that the consistency of persistent tokens must be guaranteed. The PM that we propose in this chapter addresses these issues through three contributions to the design flow.

Firstly we present a concept for executing scenarios after one another in Section 3.3, the scenario *sequencing model*. Applications with immediate scenario detection are the most challenging to implement because of the causality problem. We give the *createDetectorScenario* algorithm that splits off the detector token or detector actor together with the prefix graph in a dedicated *detector scenario*. Once the detector scenario has finished, the next scenario is known and can be executed. The scenario sequencing model solves the causality dilemma and is a crucial aspect of the PM proposed in this chapter.

Secondly we present an implementation of the scenario sequencing model for the CompSOC platform, the scenario *execution model* in Section 3.4. The implementation is based on the existing SDF PM and re-uses the implementation of channels, loading persistent tokens, SO scheduling, firing rule checking and actor execution. The intra-application scenario scheduling is added to the middleware `libDataflow` library, see also see Figure 2.2. Consistency of persistent tokens is guaranteed by exploiting the existing `libFIFO` library.

Thirdly, we capture the exact timing impact of the scenario execution model by extending the existing FSM-SADF BAG in Section 3.5. This is necessary to obtain a tight but safe throughput guarantee on the execution of a sequence of scenarios. In short, this chapter presents a generalised concept for execution of FSM-SADF applications whose exact timing behaviour can be analysed. We implement the concept by modifying and extending the existing SDF³ design flow and CompSOC middleware.

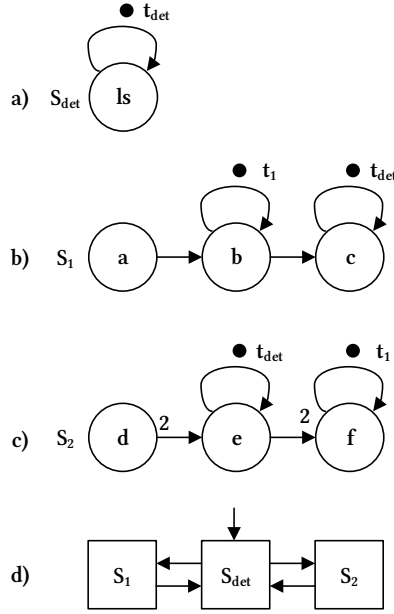


Figure 3.3: Scenario sequencing model of the application with delayed scenario detection in Figure 3.1a. **a)** Detector scenario S_{det} . **b)** Scenario S_1 has not changed. **c)** Scenario S_2 has also not changed. **d)** The transformed FSM places S_{det} before each regular scenario.

3.3 Scenario sequencing model

In this section we present a method to convert FSM-SADF applications to a model that allows to execute a sequence of scenarios. The converted application is also a valid FSM-SADF application which we term the scenario sequencing model. The conversion creates a detector scenario that is always executed before a regular scenario. The next scenario is known when the detector scenario has finished. Creating the detector scenario requires a number of graph transformations on the scenario graphs and the FSM. The conversions for applications with delayed scenario detection are different from those for applications with immediate scenario detection. After the scenario transformations the scenario graphs still adhere to the FSM-SADF rules, and the result of the timing analysis is identical to that of the original graph.

3.3.1 Creating a sequencing model for applications with delayed scenario detection

We explain the algorithm to create a scenario sequencing model for applications with delayed scenario detection using the application depicted in Figure 3.1a. All

elements related to scenario detection will be isolated in the detector scenario. The next scenario is stored in a persistent detector token that is duplicated in the detector scenario. According to the dataflow rules a persistent token cannot hover in thin air but must be attached to a channel. In turn, a channel must have a source and destination actor. The detector token may be attached to a different channel in each scenario. We do not need to replicate these channels, instead we create a new actor ls (load-schedule, see Section 3.4) in the detector scenario with a WCET of zero, and a self-edge to which the detector token is attached. This concludes the first graph transformation for applications with delayed scenario detection. The detector scenario S_{det} of the example application is depicted in Figure 3.3a.

The second and final graph transformation consists of changing the FSM so that the detector scenario is the starting scenario and is always executed before any other scenario. The transformed FSM of the example application is depicted in Figure 3.3d. The two original scenario graphs are not modified. Though the detector scenario is executed before each of the regular scenarios, it does not change the timing analysis at all because it does not contain actors with an execution time other than zero. This scenario sequencing model enables scenario execution, see Section 3.4. To conclude, two scenario transformations are necessary to convert applications with delayed scenario detection to a scenario sequencing model (Delayed Model, DM):

- DM.1 copy the detector token to the detector scenario and attach it onto a self-edge on a newly created actor ls ;
- DM.2 add the detector scenario to the FSM, make it the starting scenario and ensure that it is executed before each regular scenario.

3.3.2 Creating a sequencing model for applications with immediate scenario detection

We now explain the recipe of creating a scenario sequencing model for applications with immediate scenario detection and use the video decoder that was introduced in Section 2.3 as running example. The scenario graphs and FSM are repeated in Figure 3.4 for convenience. Creating the detector scenario requires a number of scenario transformations.

The first transformation isolates all elements required for scenario detection and inserts these in the detector scenario. These elements comprise the detector actor, the prefix graph, and all channels that connect the detector and prefix graph to the other actors, we term these the *synchronisation channels*. These elements are identical in each scenario and are copied once to the detector scenario, which leaves the synchronisation channels connected on only one side. The same elements are then removed from all regular scenarios except the synchronisation channels, which are here now only connected on the other side. Any persistent tokens on the synchronisation channels are copied to the detector scenario and removed from the regular scenarios.

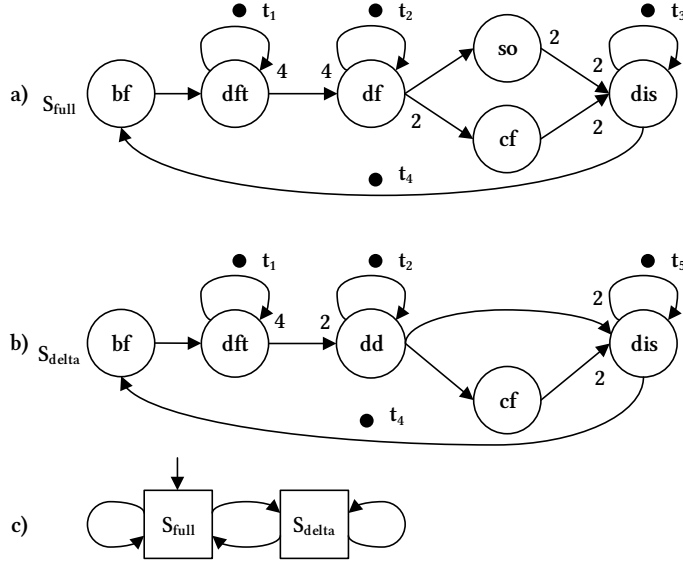


Figure 3.4: a) Scenario graph S_{full} , b) scenario graph S_{delta} and c) the FSM of the video decoder that was introduced in Section 2.3. This figure is identical to Figure 2.3 on page 27.

Detector scenario S_{det} in Figure 3.5a clarifies this graph transformation, we focus for now on the black elements. Actor dft is the scenario detector, it has a self-edge with persistent token t_1 . The prefix graph consists of actor bf and the channel from bf to dft . There are two synchronisation channels, one from dft to df (S_{full}) and dd (S_{delta}), and one from dis to bf . The latter carries persistent token t_4 , which is inserted in S_{det} and removed from the other scenarios.

After this first graph transformation the synchronisation channels are connected on only one side in each of the scenarios, leaving the graphs in an invalid situation. To solve this we perform the second graph transformation, which is to insert *switch* actors at the loose end of each synchronisation channel into which the detector scenario produces tokens, and *select* actors at the loose end of each channel from which the detector scenario consumes tokens. These actors model the transport of tokens from and to the detector scenario. The switch or select actor that closes the same synchronisation channel must have the same name in each scenario. We identify them with a subscript to differentiate between these actors, they are assigned a WCET of zero.

Furthermore a self-edge with a persistent synchronisation token is added to each switch and select actor in each scenario. Actors with the same name must have synchronisation tokens with the same name. This ensures that each synchronisation token is modeled as one physical token during analysis, as proposed in [123]. In our

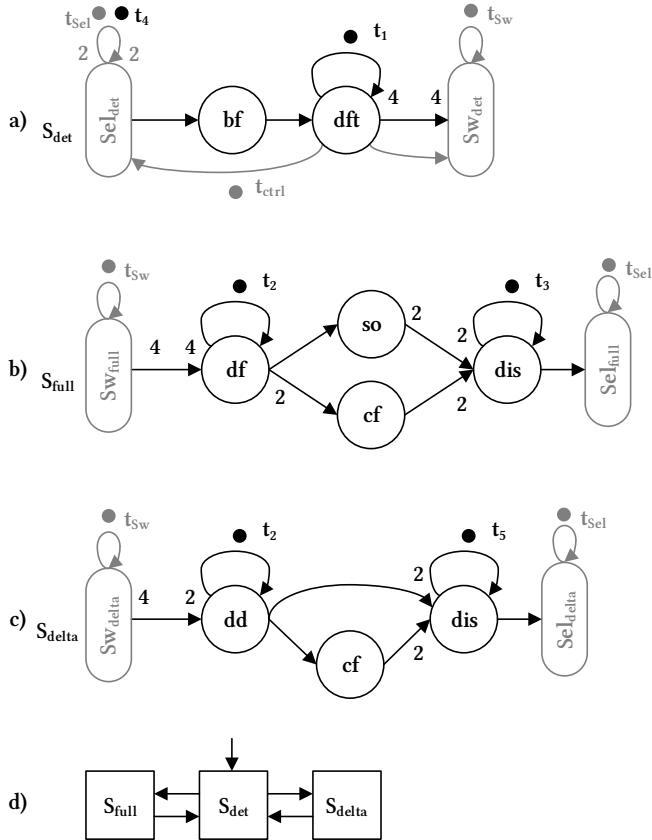


Figure 3.5: Scenario sequencing model of the video decoder application in Figure 3.4 with immediate scenario detection. **a)** Detector scenario S_{det} . **b)** Full frame scenario S_{full} . **c)** Delta frame scenario S_{delta} . **d)** The transformed FSM that now contains S_{det} .

example this second transformation creates actors **Sw** and **Sel** in each scenario as well as persistent tokens t_{sel} and t_{sw} , indicated in grey in Figures 3.5a–c.

Persistent tokens on the synchronisation channels are a special case and require a third transformation. In the detector scenario all select actors must fire first, and all switch actors must fire last. Furthermore the repetition vector entry of existing actors may not change. To accomplish this the persistent tokens on each synchronisation channel are moved onto the self-edge of the switch or select actor to which the channel is connected. The rates of that self-edge are then changed so that all tokens on the self-edge are consumed and produced with each firing of the actor. We show that this not influence the timing behaviour of the graph in Subsection 3.3.3.

In the example there is one such token, t_4 in Figure 3.5a. We see that it was moved onto the self-edge alongside synchronisation token t_{sel} , and the production and consumption rate of that channel are set to 2. If this transformation would have been skipped there would be two tokens on the channel from **Sel_{det}** to **bf** after firing of the former actor, changing the repetition vector entry of **bf** and **dft** to 2. After the transformation there is only one token on the synchronisation channel from **Sel_{det}** to **bf** after **Sel_{det}** fires, and the repetition vector entries remain unchanged.

The fourth graph transformation adds control channels from the detector actor to each switch and select in the detector scenario. The tokens produced into the control channels must identify the detected scenario. Because select actors model the transport of tokens from the regular scenarios to the detector scenario they must fire before any other actor in the detector scenario. Therefore a number of persistent control tokens equal to the repetition vector entry of the actor is attached to all control channels that go to select actors. The control channels and tokens are indicated in grey in Figure 3.5a. Control token t_{ctrl} has been added to the control channel from **dft** to **Sel_{det}**.

Lastly the FSM is changed so that the detector scenario is the starting scenario and is always executed before any other scenario, see Figure 3.5d. In conclusion, five scenario transformations are necessary to convert applications with immediate scenario detection to a scenario sequencing model (Immediate Model, IM):

- IM.1 copy the prefix graph and scenario detector actor to the detector scenario once, and remove these elements from the original scenario graphs except for the synchronisation channels;
- IM.2 insert switch and select actors on the loose ends of the synchronisation channels in each scenario, and add a self-edge with synchronisation token to each of these;
- IM.3 move any persistent token on the synchronisation channels to the self-edge of the switch or select actor to which the channel is connected, and update the rates to consume and produce all tokens at once;
- IM.4 add control channels from the detector actor to each switch and select, adding a persistent control token to all channels that go to a select actor.

IM.5 add the detector scenario to the FSM, make it the starting scenario and make it the initial state.

3.3.3 Timing analysis of applications with immediate scenario detection

The graph transformations described in Subsections 3.3.1 and 3.3.2 result in the scenario sequencing model, which is a collection of valid scenario graphs and an FSM. Timing analysis of these graphs yields exactly the same result as analysis of the original graphs. To show this we will go through the timing analysis of the running example shown in Figure 3.5.

Throughput analysis with SDF³ relies on the timestamps at which tokens are consumed and produced, the actual data value of tokens is not relevant. Temporal analysis of decoding a full video frame with SDF³ visits scenarios S_{det} and S_{full} as indicated by the FSM in Figure 3.5d. Analysis proceeds through the following steps:

AS.1 **Sel_{det}** is the only actor that is enabled initially, it fires and produces a token on the channel to **bf** as well as the new t_{sel} and t_4 at time τ_1 ;

AS.2 **bf** fires, followed by **dft** which detects scenario S_{full} ;

AS.3 **dft** produces its data tokens on the black channel to **Sw_{det}** and a control token onto each grey channel to the switch and select actors;

AS.4 **Sw_{det}** fires and produces the new t_{sw} at time τ_2 ;

This concludes S_{det} . Note that the **Sw** and **Sel** actors did not consume time, i.e. they produce tokens immediately after consumption. Persistent tokens become available in the next scenario from the moment they are produced. We will see that this achieves synchronisation in S_{full} :

AS.5 **Sw_{full}** is blocked until τ_2 , then it fires and produces t_{sw} and the data tokens towards **df**;

AS.6 **df** fires also at τ_2 , i.e. immediately after **dft**;

AS.7 S_{full} continues to execute until **dis** fires;

AS.8 **Sel_{full}** fires as soon as the token from **dis** is available, producing t_{sel} at τ_3 .

This concludes the analysis of S_{full} . Note that S_{det} would be allowed to start again while S_{full} is underway were **Sel_{det}** not blocked until τ_3 . This models the dependency of **bf** on t_4 in Figure 3.4a, effectively preventing scenario pipelining in this particular case. If the channel from **dis** to **bf** would not exist, there would not be a select actor and the scenarios could execute in a pipelined fashion as explained in [36]. We see that the switch and select actors model the transport of tokens between scenarios and ensure synchronisation from the end of one scenario to the start of the next scenario. Note that the timing behaviour has not changed compared to the original graph because the new actors have a WCET of zero. We omit the analysis of a delta frame because it is very similar.

3.3.4 Automatic creation of the scenario sequencing model

We implemented the automatic creation of scenario sequencing models in the existing design flow. Our current implementation can only handle applications with immediate scenario detection and no prefix graphs. We implemented one application with delayed scenario detection manually, see Section 3.6.

As said, identification of the detector actor is a task of the programmer. The subsequent steps have been automated in SDF³. The position of this *createDetectorScenario* algorithm in the flow can be found in Figure 2.2 on page 23. The implementation of *createDetectorScenario* is listed in Algorithm 2.

3

Algorithm 2 Creating a scenario sequencing model.

```

1: function CREATEDETECTORSCENARIO(scenarioGraphs, fsm, detectorActor)
2:   check_detector_actor(scenarioGraphs, detectorActor)
3:   detectorScenGraph = new scenarioGraph
4:   copy_detector_actor(scenarioGraphs, detectorActor, detectorScenGraph)
5:   insert_switch_select_actors(detectorScenGraph)
6:   for sg in scenarioGraphs do
7:     replace_detector_with_switch_and_select_actors(sg)
8:     add_sync_and_control_tokens(sg, detectorScenGraph)
9:   end for
10:  scenarioGraphs.add(detectorScenGraph)
11:  add_detector_scenario(fsm, detectorScenGraph)
12: end function

```

Line 2 in Algorithm 2 checks whether the scenario graphs satisfy the requirements that were posed in Subsection 3.2.2. An empty detector scenario is created in line 3. The detector actor and its incoming and outgoing channels are copied to the detector scenario in line 4, corresponding to IM.1 in Subsection 3.3.2. Switch and select actors with self-edges containing the synchronisation tokens are inserted at the open ends of the synchronisation channels in the detector scenario in line 5. The detector actor is replaced by switch and select actors in each remaining scenario in line 7, synchronisation tokens are added to these actors in line 8. Step IM.2 is implemented in lines 5–8, steps IM.3 and IM.4 in line 5.

The detector scenario graph is added to the list of all scenarios in line 10. The FSM is transformed in line 11, implementing step IM.5.

3.4 Scenario execution model

Section 3.3 showed the transformation of FSM-SADF applications to scenario sequencing models. A sequencing model is a valid FSM-SADF application and its behaviour is identical to that of the original application graph, both logically and in the

timing domain. Therefore the *analyseBAG* timing analysis algorithm in Figure 2.1 on page 22 gives the same result for the original application and the sequencing model.

In this section we present the conversion from the scenario sequencing model to the *scenario execution model* that allows to execute FSM-SADF applications in concert with the middleware. For applications with immediate scenario detection we show that the scenario sequencing model solves the causality dilemma because the next scenario graph is only started after it is detected. We present the scenario execution model in Subsection 3.4.1. The remaining subsections present solutions to the following practical aspects: execution of schedules for applications with delayed scenario detection in Subsection 3.4.2 and immediate scenario detection in 3.4.3, implementation of switch and select actors in Subsection 3.4.4, and sharing persistent tokens in Subsection 3.4.5.

3.4.1 Executing a sequence of scenarios

Executing the scenarios of an application with delayed scenario detection is straightforward because there is no causality dilemma. The scenario graphs may be executed in any order indicated by the FSM, the actual order at run-time is determined by the graph. The scenario execution model depicted in Figure 3.6 is almost identical to the scenario sequencing model in Figure 3.3. The only difference is that additional *load-schedule* (*ls*) actors are added until there are as many as there are VPs. This will be clarified during the explanation of the tailored approach for schedule execution in Subsection 3.4.2.

The causality dilemma that is encountered when attempting to execute an application with immediate scenario detection is solved by splitting off the detector actor and prefix graph into a detector scenario, see Section 3.3. During execution data tokens must be transferred from the detector scenario to the other scenarios and vice versa.

To do so we merge the scenario graphs from the sequencing model into a scenario execution model by merging the switch and select actors. Each switch or select actor in the detector scenario has a matching actor in every other scenario with the same name. As an example see Figure 3.5, the actors that match \mathbf{Sw}_{det} are $\mathbf{Sw}_{\text{full}}$ and $\mathbf{Sw}_{\text{delta}}$. Every set of matching actors is merged. In the example the three matching switch actors are merged into one switch actor \mathbf{Sw} , see Figure 3.7a and b. The three select actors are merged into one select actor \mathbf{Sel} , see Figure 3.7c and d. The figures show that the newly created switch and select actors are connected to the same channels as the actors from which they are created.

The result is one single dataflow graph that contains all scenarios, the scenario execution model in Figure 3.8. As the scenarios are merged into one graph, a number of transformations that were necessary for scenario synchronisation are reverted. Firstly, the synchronisation tokens are no longer necessary and are removed. Secondly, step IM.3 is reversed because enforcing the actor ordering in the detector scenario is no longer necessary. This means that any remaining tokens on the self-

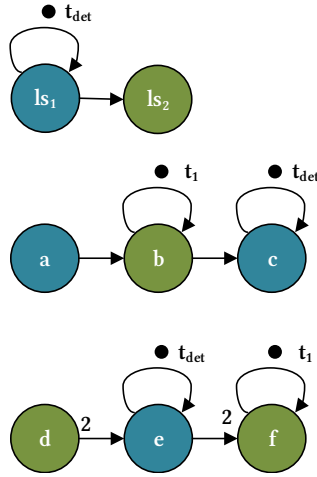


Figure 3.6: Scenario execution model of the application in Figure 3.1a, mapped to two VPs. Blue actors are mapped to VP_1 , green actors to VP_2 .

	VP_1	VP_2
S_{det}	[ls ₁]	[ls ₂]
S_1	[a, c]	[b]
S_2	[e, e]	[d, f]

Table 3.1: The SO schedules of the mapped application in Figure 3.6 of each VP in each scenario.

edges of select actors are moved back to the synchronisation channel from which they originated. As an example see t_4 in Figure 3.7c and d. Thirdly, there are no tokens left on the self-edges of the switch and select actors and these edges are removed. Fourthly, the persistent control tokens added in step IM.4 are removed, see t_{ctrl} in Figure 3.7c that is removed in 3.7d. The other actors do not change. The resulting scenario execution model of the video decoder is shown in Figure 3.8, control channels, load-schedule actors and the switch and select actors are indicated in grey.

The new switch and select actors correspond to the *Boolean Dataflow (BDF)* concept [13]. A switch has one control input, one data input and two data outputs, see Figure 3.7b. It forwards the data token(s) from the data input to one of the data outputs, the correct output is selected depending on the control input. A select has one control input, two data inputs and one data output, see Figure 3.7d. It is the counterpart of a switch and forwards the data token(s) on one of the inputs to the data output based on the control input. A switch actor functions as a multiplexer, a select as de-multiplexer.

A scenario execution model such as shown in Figure 3.8 is a BDF graph. As exact analysis of BDF graphs is not possible, it would seem that this step invalidates the timing analysis required to meet the RT constraint [138]. However, the scenario execution model is not a pure BDF graph but a restricted variant. The graph topology ensures that all switch and select actors always receive the same control token at the same time, ensuring that all actors of one scenario are executed. This essentially implements the FSM, the sequence of control tokens from the detector to all switch and select actors is the actual sequence of FSM states. The timing behaviour of the scenario execution model is identical to that of the scenario sequencing model. The scenario execution model is therefore a restricted BDF graph whose behaviour is identical to that of the original FSM-SADF graph, but it cannot be analysed itself.

Executable dataflow application code is generated from the scenario execution model in the *generateApplication* algorithm, see Figure 2.2 on page 23. The actual execution requires support from the *libDataflow* middleware library in that same figure.

3.4.2 Executing schedules of applications with delayed scenario detection

Mapping and scheduling is performed in algorithms *mapToVRs* and *VPSizing*, see Figure 2.2 on page 23. A mapping is executed by the resource manager as described in Chapter 1. Executing the SO actor schedule of FSM-SADF applications on the other hand requires support from both the PM and middleware, DS.PM and DS.MW.

An SO actor schedule is calculated for each VP in each scenario during the mapping and scheduling. At run-time each application starts with the detector scenario, regardless whether it has delayed or immediate scenario detection. The schedule of the detector scenario of applications with delayed scenario detection consists only of the load-schedule actor, abbreviated [ls] in our examples. The *libDataflow* library is modified to execute a *Rolling Static-Order (RSO)* schedule that is initially set

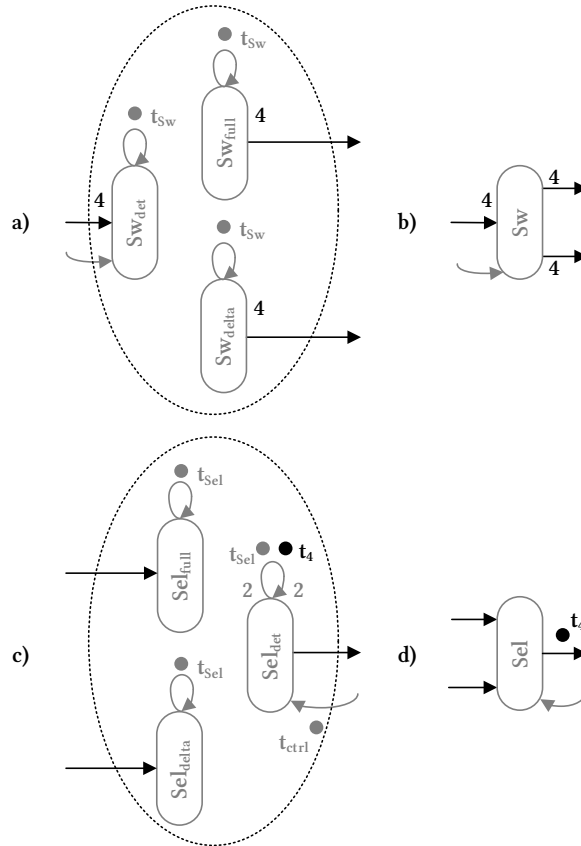


Figure 3.7: **a)** The switch actors and **c)** select actors in the scenario sequencing model are merged into **b)** one BDF switch actor and **d)** one BDF select actor to create the scenario execution model.

to just the schedule of the detector scenario. Inside the load-schedule actor the detector token is inspected, and the schedule that corresponds to the next scenario is concatenated to the current RSO schedule followed by the next detector scenario. In this way it is ensured that `libDataflow` does not run out of actors to schedule.

The example application depicted in Figure 3.3 is mapped to two VPs in Figure 3.6. The SO schedules of each VP in each scenario are listed in Table 3.1. Let us assume that the initial scenario stored in t_{det} is S_1 . Initially the schedule in both VPs consists only of load-schedule actors $[ls_1]$ and $[ls_2]$. Actor ls_1 inspects the token t_{det} and concatenates the actor sequence $[a, c, ls_1]$ to the RSO schedule on VP_1 . This actor also produces a *scenario token* on the channel to ls_2 , which is the first actor to fire in VP_2 . It concatenates the actor sequence $[b, ls_2]$ to the RSO schedule on VP_2 . Then all actors fire in the order dictated by the schedule, actor c updates t_{det} with the next scenario. During the second firing of ls_1 and ls_2 the next scenario actor sequence is concatenated to the RSO schedule, and so on.

3

3.4.3 Executing schedules of applications with immediate scenario detection

When executing applications with immediate scenario detection the detector scenario is also scheduled first. The contrast with applications that have delayed scenario detection is that the detector scenario contains a detector actor, after whose execution the current scenario is known. To concatenate the correct sequence of actors to the RSO schedule, one load-schedule actor is again instantiated in each VP. A control channel through which a scenario token can be transferred is instantiated from the detector actor to each load-schedule actor. Again the `libDataflow` library is called inside each load-schedule actor to extend the RSO schedule with the actor sequence of the detected scenario, followed by that of the detector scenario.

As an example, consider the scenario execution model of the video decoder mapped to two VPs depicted in Figure 3.8. The corresponding schedules are listed in Table 3.2. Detector scenario S_{det} is scheduled first, so the initial schedule of VP_1 is $[dft, ls_1, Sw]$ and that of VP_2 is $[bf, ls_2]$. After firing dft the next scenario is known and its schedule is concatenated to the RSO schedule in actor ls_1 . Assuming S_{full} is detected, the sequence $[df, dft, ls_1, Sw]$ is concatenated to the RSO schedule of VP_1 and $[so, cf, cf, dis, Sel, bf, ls_2]$ to that of VP_2 . We see that different scenarios are scheduled inside each VP, implementing both the intra-application actor scheduling and intra-application scenario scheduling as depicted in Figure 1.6 on page 14

3.4.4 Implementation of switch and select actors

The current `libDataflow` library supports SDF and CSDF, and does not implement switch and select actors. Consider a switch, whose data tokens on the input port are forwarded to one of the output ports. The rate on the other output is effectively

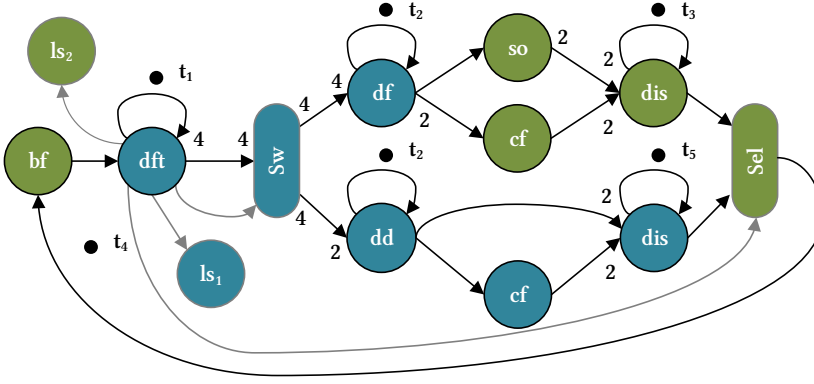


Figure 3.8: Scenario execution model of the video decoder in Figure 3.4, mapped to two VPs. Blue actors are mapped to VP_1 , green actors to VP_2 . This is a valid BDF graph.

	VP_1	VP_2
S_{det}	[dft, ls ₁ , Sw]	[bf, ls ₂]
S_1	[df]	[so, cf, cf, dis, Sel]
S_2	[dd, dd, cf, cf, dis]	[Sel]

Table 3.2: The SO schedules of the mapped application in Figure 3.8 for each scenario in each VP.

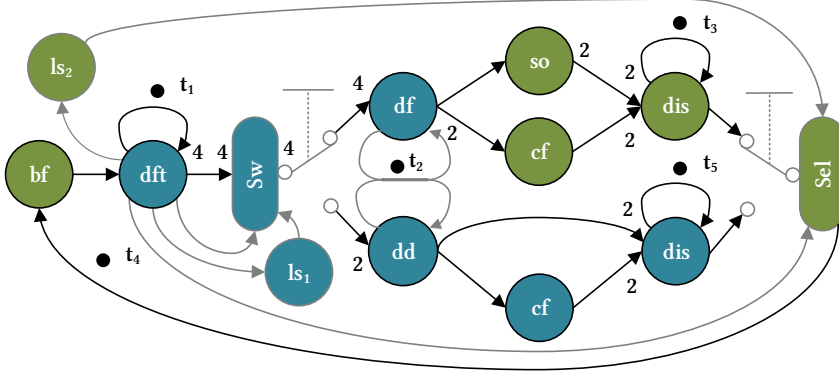


Figure 3.9: Scenario execution model of the video decoder mapped to two VPs, annotated with implementation details. Blue actors are mapped to VP_1 , green actors to VP_2 . The swapping of FIFO channels is indicated with the electrical symbol for a switch, also in grey. Token t_2 is attached to a channel that is connected to multiple ports, see Subsection 3.2.3. This is not a valid dataflow graph but shows how the application graph is executed using the FSM-SADF PM.

zero. Neither changing rates within a scenario nor rates with value zero are currently supported.

Switch and select actors could be implemented by allowing rates to change within a scenario, and allowing rates with a value of zero. This is however not compatible with SDF and CSDF and would break compatibility with these PMs. Instead we propose an implementation that exploits the fact that channels implemented by the `libFIFO` library (see Figure 2.2 on page 23) can be disconnected and reconnected without invalidating data.

For this purpose we added a `set_scenario` function call to `libDataflow` that changes the channel-to-port connections at run-time. We instantiate a switch actor with just a single output port, and connect the correct channel by calling the `set_scenario` function in the load-schedule actor of that VP. Conversely, a select actor has just a single input port to which the correct channel is also connected in the load-schedule actor. A load-schedule actor is always scheduled before any switch or select actors, so the channels are guaranteed to be set up correctly before a switch or select actor fires. The actual function inside switch and select actors is straightforward, it copies the tokens from the input port to the output port.

To visualise this solution we annotated the scenario execution model of Figure 3.8 with implementation details, see Figure 3.9. The swapping of FIFO channels is indicated with the symbol for an electrical switch. The leftmost switch symbol connected to `Sw` connects its single output port to either the channel toward `df` or that toward `dd`. The other channel is left unconnected on one end, effectively giving it rate zero without affecting the firing rules. Swapping the FIFO channels must take place be-

fore the firing rules of **Sw** are checked, i.e. before it fires, which is guaranteed by locating the call in the load-schedule actors. This dependency is visualised with the grey channels from **ls₁** to **Sw** and **ls₂** to **Sel**. The run-time steps (RS) can be matched to the analysis steps in Section 3.3 as follows:

- RS.1 the output port of **Sw** is connected to the FIFO towards **df**;
- RS.2 **Sw** fires and consumes all of its tokens (step AS.4);
- RS.3 **Sw** produces tokens into the connected FIFO (AS.5);
- RS.4 execution of S_{full} continues as usual (step AS.6, AS.7, AS.8 and AS.1).

The select actor **Sel** is similar but de-multiplexes two channels to one.

3.4.5 Implementation of shared persistent tokens

Another difference between FSM-SADF graphs and SDF or CSDF graphs is that persistent tokens may appear in multiple scenarios. To implement this we exploit another useful property of `libFIFO`, which is that multiple ports may be connected to a single FIFO channel. We instantiate each FIFO that contains a shared persistent token only once, and map the port from each relevant source and destination actor in every scenario to that FIFO.

This implementation does still not guarantee consistency. As scenarios may be pipelined, it is possible that before a shared token is produced in the current scenario, it is accessed by an actor from another VP that is already in a later scenario iteration. This may cause a write-before-read or write-during-read, which are both undefined. To avoid this situation we add a mapping constraint: all source actors of the same FIFO must be mapped to the same VP in each scenario. The same goes for the destination actors, which may be in a different VP than the source actors. The RSO scheduling within each VP ensures that actors connected to the same FIFO can never execute simultaneously or overtake each other, avoiding the problem. This mapping constraint partially prevents scenario pipelining at the position of the shared persistent tokens. As scenario pipelining generally improves the throughput the mapping constraint may cause a performance penalty, depending on the application and the mapping.

In the example application with delayed scenario detection depicted in Figure 3.1a there are two shared persistent tokens, t_{det} and t_1 . We visualised the implementation in Figure 3.10, note that this is not a valid dataflow graph. Actors **ls₁**, **c** and **e** are connected to the same FIFO that holds shared persistent token t_{det} . Actors **b** and **f** are both connected to the FIFO that holds t_1 .

In the video decoder application with immediate scenario detection depicted in Figure 3.5 there is one shared persistent token t_2 . The implementation in Figure 3.9 shows how the FIFO containing t_2 is connected to both actors **df** and **dd**. Note that t_1 is not shared because it only appears in the detector scenario, and t_3 and t_5 are

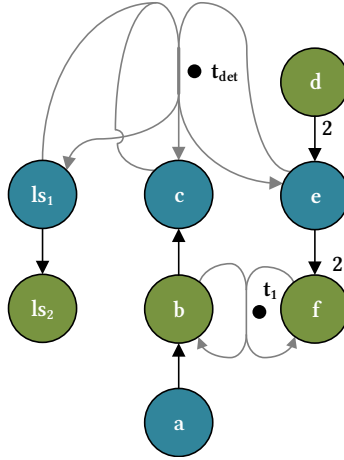


Figure 3.10: The scenario execution model of Figure 3.6 annotated with the implementation of shared persistent tokens. The additional mapping constraints force that actors which access the same shared persistent tokens be mapped to the same VP. This is not a valid dataflow graph.

neither because they occur only in one scenario. The implementation of shared persistent tokens is identical for applications with delayed and with immediate scenario detection.

In this section we showed how to transform a scenario sequencing model to a scenario execution model for applications with delayed and immediate scenario detection. The RSO scheduling is an essential building block of the FSM-SADF PM that we present in this chapter, it allows executing any sequence of scenarios and helps to solve the causality dilemma. Combined with the mapping constraint, the RSO scheduling guarantees the consistency of shared persistent tokens. The load-schedule actors are furthermore exploited for connecting the correct FIFO channels to the switch and select actors before they fire. The scenario execution model is a restricted BDF graph whose timing behaviour is identical to that of the scenario sequencing model. Figures 3.9 and 3.10 show middleware implementation details that are not covered by the original FSM-SADF timing analysis, these are the topic of the next section.

3.5 Extended Binding Aware Graph

Section 3.3 introduced the scenario sequencing model, Section 3.4 the scenario execution model along with the implementation details. The implementation affects the timing and is not modeled in the scenario graphs of an FSM-SADF application. Therefore we re-visit the MoC in this section and extend the BAG so that the exact timing impact of our solution is analysed.

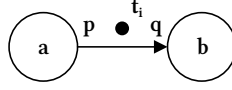


Figure 3.11: Graph with two actors *a* and *b*, connected by a FIFO with production rate *p*, consumption rate *q* and persistent token t_i .

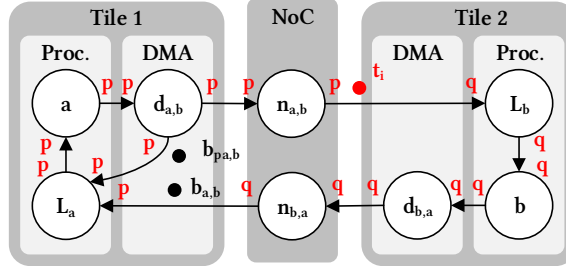


Figure 3.12: CompSOC binding-aware model of the graph in Figure 3.11 with both actors mapped to different processors. **Proc.** denotes processor, $b_{pa,b}$ and $b_{a,b}$ model the source buffer and the FIFO buffer. The model is based on [92, 94], all additions are indicated in red.

The design flow depicted in Figure 2.2 on page 23 shows that multiple mappings are generated from a set of storage distributions. In the *analyseBAG* algorithm the throughput of a mapping is analysed after expanding each mapped scenario graph of the scenario sequencing model to a BAG [134]. During the expansion the scenario graphs are annotated with timed models of the middleware and hardware, e.g. the SO schedule, *Network-on-Chip* (NoC) and memory controller[47, 49]. We will now discuss how the RSO schedule execution, switch and select actors and shared persistent tokens that were described in Section 3.4 are modelled in the BAG.

One load-schedule actor is added to each VP when the BAG is created. The RSO schedule of a VP is extended during execution of the load-schedule actor through a call to the `libDataflow` library. The time required for this call is added to the WCET of the load-schedule actor, modeling the extension of the RSO schedule during runtime.

The switch and select actors copy the input tokens to the output channel. The time required for these copy operations can be measured, but is not accounted for in the WCET of any actor in the original FSM-SADF graph. Instead we add it to the WCET of the switch and select actors which was set to zero in Section 3.4 during creation of the BAG. Connecting the correct FIFO channel to the switch and select actors is done inside the load-schedule actors, therefore the time required for that call to `libDataflow` is also added to those actors.

Shared persistent tokens do not affect the timing behaviour in any way.

In the existing design flow depicted in Figure 2.1 on page 22 SDF and CSDF applications can be mapped onto the CompSOC platform [2, 131]. The existing flow uses

an older platform model that is more conservative and less accurate model than the HSDF platform model described in [92, 94]. We implemented the latter in the context of this thesis, see the *analyseBAG* algorithm in Figure 2.2 on page 23.

The platform model described in [92, 94] is HSDF, meaning that all rates on all channels are one. The scenario graphs of FSM-SADF applications are multi-rate however, which requires an update of the model. We will explain the update using the simple example graph in Figure 3.11 and assume that actors *a* and *b* are mapped to different VPs. To construct the BAG the channel is replaced by a combined model of the *Direct Memory Access (DMA)*, NoC and CoMik μ kernel as explained in [92, 94]. Our proposed extension of that model is shown in Figure 3.12, all changes are indicated in red. We see that all the channels now have rates, and the location of a possible persistent token t_i is also indicated.

Encoding a SO schedule in the BAG of a HSDF graph is straightforward: an edge is inserted from the first actor in the schedule to the second, from the second to the third, and so on. Encoding a schedule in multi-rate graphs is more complicated because actors can have a repetition vector entry larger than one, yet after the first firing the next actor may already be enabled. The encoding must ensure that only the scheduled actor fires, a suitable technique for this is proposed in [20]. We implemented this technique in the design flow, it is part of the *analyseBAG* algorithm.

In this section we showed how the implementation details explained in Section 3.4 are modeled in the BAG. The BAG itself is constructed from the scenario sequencing model introduced in Section 3.3. To select a suitable mapping each mapped scenario sequencing model is extended into a BAG and analysed. The scenario graphs are merged together automatically in the *mergeScenarios* algorithm, resulting in a scenario execution model. This concludes the discussion of the scenario analysis, sequencing model and execution model of the FSM-SADF PM, which concerned the algorithms *createDetectorScenario*, *analyseBAG*, *mergeScenarios*, *generateApplication* and the *libDataflow* library in Figure 2.2 on page 23.

3.6 Experimental evaluation

3.6.1 Delayed scenario detection

As mentioned in Subsection 3.3.4 the design flow was automated only for applications with immediate scenario detection. To demonstrate that the concept for applications with delayed scenario detection also works we implemented one such application manually, namely the scenario execution model in Figure 3.10. The scenario initially stored in t_{det} is S_1 . Actor *c* sets detector token t_{det} to S_2 , while *e* sets detector token t_{det} to S_1 . Inside each actor we print the actor identifier. The output confirms that the actors are executed in the correct order, alternating between scenario S_1 and S_2 .

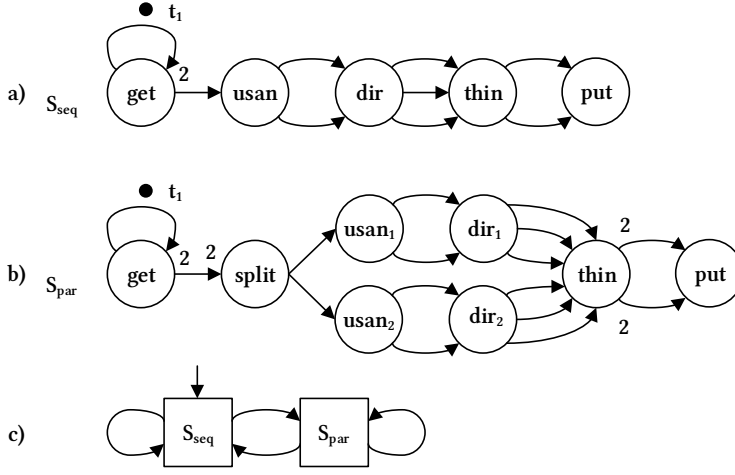


Figure 3.13: a) SUSAN sequential scenario graph, S_{seq} . b) SUSAN parallel scenario graph, S_{par} . c) The FSM.

3.6.2 Immediate scenario detection

We show the PM and automated design flow for applications with immediate scenario detection using a real application, namely the SUSAN edge detection algorithm [126]. It consists of two scenarios, S_{seq} is a sequential graph that represents the original algorithm, see Figure 3.13a. Scenario S_{par} is a parallelised version, see Figure 3.13b. The algorithm reads an image block by block, one of the scenarios is executed for each block. Scenarios are switched based on the image resolution, larger images are decoded by S_{par} . The FSM is shown in Figure 3.13c. We use the design flow to map the application to a two-tile CompSOC platform.

If mapped to two or more VPs scenario S_{par} can achieve a higher throughput than S_{seq} because the computationally intensive actors **usan** and **dir** (direction) can be executed in parallel ($[us_1, dir_1]$ and $[us_2, dir_2]$). To provide both these chains with a block, **split** should have a consumption rate of two. This means the **get** (*get image*) actor has to execute twice. To not violate the constraint concerning the identical repetition vector entries of actors in the detector scenario, the production rate on the channel from **get** to **usan** in S_{seq} must also be two. This implies each image must contain an even number of blocks.

We instantiated a CompSOC platform with two tiles that contain one processor each. Both tiles are clocked with a frequency of 100 MHz. Each processor features instruction and data memories of 256 kB each as well as one DMA with two communication memories of 16kB each. The processors are connected to each other and to an external *Double Data Rate* (DDR) memory via a NoC [43, 45, 47].



Figure 3.14: Original test image (left) and the output of SUSAN computed on the CompSOC platform. The outer band of the image is skipped.

3.6.3 Results

The design flow depicted in Figure 2.2 on page 23 generated executable code for the SUSAN application. The SDF³ tool takes the scenario graphs annotated with execution times and memory requirements as an input, plus an architecture file. The throughput is the number of completed scenario iterations per clock cycle. The analysis gives a maximum throughput of $2.7 \cdot 10^{-7}$ for SUSAN. We measured a throughput of $3.1 \cdot 10^{-7}$ when executing the application on the platform using the implementation described in Section 3.4. The fact that the actual throughput is a little higher than the throughput given by the analysis is consistent with the fact that the model is conservative. The graphic output of the SUSAN edge detection algorithm is shown in Figure 3.14.

A *load-schedule* actor takes 365 cycles to execute including the wrapper, the switch actor takes 1635 cycles. This brings the timing cost of our `libDataflow` modifications to 2000 cycles, which are accounted for in the BAG. We argue that the impact of scenario sequencing on the applications timing is minimal, as the total WCET of SUSAN is 3.3 million cycles. The size of the library is increased by 3 kB to 188 kB.

3.7 Related work

In this section we give an overview of other PMs for RT embedded systems. These related approaches are divided over three subsections: non-dataflow PMs, dataflow PMs that cannot respond dynamically to data, and dataflow PMs that can respond dynamically to data.

3.7.1 Non-dataflow PMs

The first group mainly consists of variants on PMs for the *Time-Triggered Architecture (TTA)* [68, 69]. The TTA is a composable platform on which tasks and communication operations are statically scheduled according to a global clock, i.e. the clocks on all tiles and processors are synchronised. It is possible to construct such heterogeneous multi-processor systems [70], but providing a synchronised clock to all parts of the system is a challenge for scalability.

Building upon the time-triggered concept is the *Precision Timed Machine (PRET)*, which proposes a micro-architecture, compiler and programming language that give complete control over the timing behaviour [12, 32]. It can use Giotto as a platform-independent time-triggered language, which supports *modes* that are similar to scenarios [54]. This high-level language is then converted to the PRET intermediate language, which abstracts hardware details away but exposes the timing primitives. Each action, e.g. starting a task, is triggered at a time that is pre-calculated at design-time, so dynamic application sets (RQ.3b) cannot be implemented. Analysis and certification of such systems is relatively straightforward.

The disadvantage of TTA and PRET is that it is not work-conserving, i.e. a resource may be idle while there is work to do. It is therefore not possible to make an application respond to events at run time (RQ.4), and the system always experiences the worst-case execution which inherently leads to a waste of resources. Dataflow PMs on the other hand are work-conserving because an actor may fire if its firing rules are satisfied, independent of the wall clock time. This simplifies responding to events.

3.7.2 Non-dynamic dataflow PMs

The second group that we consider are dataflow PMs that do not allow dynamic responses to data. The HSDF, SDF and CSDF implementations on the CompSOC platform that have been introduced in Chapter 2 are part of this group [89, 92]. Another dataflow PM is *Cal Actor Language (CAL)*, which is a well-defined, general actor language more versatile than the CompSOC PMs [33]. To implement a PM for a specific dataflow MoC with CAL, the use of the language must be restricted to a subset that matches the MoC. Firing-rule checking is essentially the same as in the CompSOC MoCs, but CAL does not provide middleware to implement scheduling and actor firing on a platform.

We now summarise a number of design flows that construct heterogeneous multi-processors for RT embedded systems and provide a matching PM. The ForSyDe design flow provides top-down, model-driven flow for a platform that supports SDF applications [110]. The flow can generate applications for ForSyDe-generated platforms as well as CompSOC platforms [6, 29]. Daedalus can generate multi-processor platforms and automatically convert sequential code to KPN or *Polyhedral Process Network (PPN)* applications, the latter being a special case of KPN [96, 130]. Both KPN

and PPN are dataflow variants for which a PM is provided for Daedalus platforms. MAMPS is a design flow that supports multiple applications and multiple use-cases, i.e. fixed constellations of actors that pre-defined at design-time [72]. It is based on the SDF MoC and features a PM that implements channels as FIFO buffers, RR scheduling, and shell functions in which actor code may be inserted. The defining property in which all PMs in this second group differ from the FSM-SADF PM presented in this chapter is that they do not support applications that respond dynamically to data. Hence all these methods do not pose a causality dilemma, but are also not suitable to achieve requirement RQ.4.

3.7.3 Dynamic dataflow PMs

The third group that we consider are dataflow PMs that are able to respond dynamically to data. An intermediate representation of the CAL language has been used to generate executable C-code for FSM-SADF applications [122]. However, as this work does not specify how to schedule the actors at the scenario boundaries it does not solve the causality dilemma, and therefore we do not consider this a full PM implementation.

There is no causality dilemma when attempting to execute applications expressed in the SADF MoC discussed in Subsection 2.3.1 [140]. Although no PM has been specified there is no obstacle for executing such applications, the performance analysis for instance simulates such execution [139]. The SADF PM separates control explicitly from data. To implement the control part SADF specifies detector actors that are similar to the ones we introduce for our PM in Section 3.2. There are two key differences between the SADF detectors and our FSM-SADF detectors. Firstly, the SADF detectors are explicitly indicated in the initial scenario graphs and are also required for analysis. FSM-SADF detectors on the other hand are not necessary for analysis and must be indicated only if an application is to be executed. Secondly, SADF detectors are dedicated for scenario detection and cannot process data, this is left to the kernel actors [139]. These detectors receive data tokens, but may only produce control tokens. The FSM-SADF detectors on the other hand are data processing actors that happen to decode the scenario during their execution. They also receive data tokens, but may produce data and control tokens. As explained in Subsection 2.3.1, FSM-SADF is a restricted version of SADF, which benefits the analysis because fewer operational semantics must be implemented. We have therefore opted to implement a PM for FSM-SADF. While we have been inspired by the SADF detector actors, the FSM-SADF detectors are significantly different.

The only full PM that is comparable to the one presented in this chapter is *Mode-Controlled Dataflow (MCDF)*. In MCDF the programmer captures all scenarios (termed *modes*) in one graph using a set of rules and special actors, namely a mode controller and switch and select actors [83, 106]. The first difference is that MCDF currently supports only applications with delayed scenario detection, no examples with immediate scenario can be found in [83]. It may be possible to support applications

with immediate scenario detection through a minor extension of the tools, but to the best of our knowledge no such work currently exists.

MCDF uses the same graph for both temporal analysis and execution on the platform, scenario transitions are implicitly encoded in the mode controller. Though their expressiveness is similar, this reveals another significant difference between FSM-SADF and MCDF. FSM-SADF features separate models for timing analysis and execution. The design of MCDF applications on the other hand contain only one graph for analysis and execution, it both a MoC and PM. This prevents analysis using $(\max, +)$ algebra, the available analysis methods mentioned in [83] are therefore less precise than those of FSM-SADF. To analyse MCDF graphs with $(\max, +)$ algebra, the scenario transitions would have to be made explicit by taking them out of the mode controller.

Our contribution is to enable execution of scenario graphs with minimal effort for the programmer. The scenario sequencing and execution models described in Sections 3.3 and 3.5 are generated automatically. During analysis our switch and select actors behave as normal SDF actors, allowing full re-use of existing analysis techniques. During execution these actors behave as their MCDF counterparts. The execution graph is reminiscent of MCDF, but the mode controller is replaced by the detector scenario [83]. In MCDF partial graph execution is achieved by selecting a different schedule for each scenario, dubbed *Quasi Static-Order Scheduling (QSOS)*. A variant on the MCDF PM using fixed-priority scheduling has also been presented [77]. Unlike QSOS, our RSO schedule is extended immediately after detection of the next scenario. MCDF has been shown to be sufficiently expressive to handle dynamic behaviour of a *Long-Term Evolution (LTE)* receiver, which can also be handled by our PM [106, 108].

A previous version of this chapter splits off detector scenarios that consist of only one actor in a similar way [146]. All actors fire, but execute the encapsulated functions conditionally based on a scenario identifier token. This causes a considerable overhead in large graphs, which we avoid in this chapter. Also, it does not present the BAG extension presented in Section 3.5.

3.8 Summary

There does not exist a PM for the FSM-SADF MoC. In this chapter we show that there are two types of scenario detection: delayed, where the current scenario is detected during the previous iteration, and immediate, where the current scenario is detected during the current iteration. A causality dilemma is encountered when attempting to execute an application with immediate scenario detection, because at run-time it is not possible to know which scenario graph must be started at the start of a scenario iteration.

In this chapter we presented a scenario sequencing model in which all elements related to scenario detection are split off in a separate detector scenario. The detector scenario is always executed before the other scenarios, which solves the causality

dilemma. The timing behaviour of the sequencing model is identical to that of the original application, and it can be expanded into a *Binding-Aware Graph (BAG)* that models the exact timing impact of our PM, middleware and platform.

We furthermore presented a scenario execution model which merges all scenarios of the sequencing model into one graph that is suitable for execution. The detector scenario is always executed first, and the *Rolling Static-Order (RSO)* scheduling ensures that the correct actor sequence is concatenated to the run-time schedule in each VP by the load-schedule actors. The execution model is a restricted BDF graph whose timing behaviour is identical to that of the sequencing model. For the implementation we exploit the flexibility of the `libFIFO` library to implement switch and select actors as well as shared persistent tokens. Furthermore we extended the `libDataflow` library with system calls to extend the RSO schedule and correctly connect the channels to switch and select actors before they fire.

We measured the timing impact of the implementation and added the timing cost in the BAG to the appropriate actors. The PM was automated for applications with immediate scenario detection so that the effort for the programmer is minimal, see Figure 2.2 on page 23. We manually implemented an example application with delayed scenario detection and found that the scenario sequencing and RSO scheduling function correctly. We used the automated flow to map the SUSAN edge detection algorithm with immediate scenario detection onto a two-tile CompSOC platform. This application also functions correctly, and the real throughput proved to be slightly higher than the bound given by the analysis. This shows the analysis model is both conservative and precise, while the cost in terms of timing and memory footprint is marginal.

In conclusion, we presented a PM that consists of a sequencing model, execution model, implementation and extended BAG.



4 Trading Virtual Processor size against buffer size

4.1 Introduction

To execute dynamically changing sets of applications (requirement RQ.3b), the platforms created by the design flow must feature a resource manager that can start and stop applications as required. The sequence of start and stop events during run-time is unknown at design-time. It is not possible to calculate and store all possible deployments (inter-application mapping and schedule combinations) beforehand because their number grows exponentially with each start and stop. Instead the resource manager must use a run-time deployment heuristic. As a result the platform utilisation is unpredictable and it cannot be guaranteed that the resource manager can find sufficient free resources to start an application. To increase the success of future deployment actions, the design flow must balance the resource requirements of each application. This summarizes sub-problem SP.4.

We selected an existing design flow that can trade throughput against memory footprint. It does so by generating several *storage distributions*, i.e. sets that contain a fixed buffer size of each channel, and determining the throughput for each of these by constructing the BAG. This *Design Space Exploration (DSE)* performs the intra-application mapping and scheduling, and comprises the algorithms labelled DS.M&S in Figure 2.1 on page 22.

In this chapter we propose an extension to the DSE that allows to trade the total size of all VPs against the total size of all buffers under a throughput constraint, solution CB.4. The VP size determines the utilisation of the physical processors. The new DSE therefore allows to balance processor usage and memory footprint and offers the designer possibility to increase the success of future inter-application deployment actions. This solves sub-problem SP.4. Figure 1.6 on page 14 gives an overview of the terminology and different layers of mapping and scheduling.

Deployment is reviewed in Section 4.2, we introduce different approaches and motivate the choice for a hybrid method. The remainder of the chapter focuses on the intra-application mapping. The WCRT is explained in detail in Section 4.3, the DSE in Section 4.4. The trade-off between processor usage and memory footprint that is the central contribution of this chapter is presented in Section 4.5. We evaluate this contribution experimentally in Section 4.6. Section 4.7 gives an overview of related work, the chapter ends with a summary in Section 4.8.

4.2 Inter-application deployment

Deployment strategies fall in three categories: design-time, run-time and hybrid [120]. We now give a short evaluation of each strategy and its suitability for dynamic sets of applications. Note that methods with one layer of mapping and scheduling do not have a deployment phase, the applications are directly mapped onto the physical processors. Therefore the term *mapping* instead of deployment is used in most citations in this subsection.

4.2.1 Design-time deployment

Design-time methods are a good fit for deployment fixed sets of RT applications [73, 114]. While computation costs are high, such methods have a global system view and arrive at deployment solutions that are generally superior to those of run-time methods. Most importantly, thorough timing analysis can guarantee that RT requirements are met. The exponential growth of deployments in dynamic sets however poses a problem, as the number of potential deployments quickly runs into the thousands. Even if all of these could be calculated at design-time, storing them in memory-constrained embedded systems is not feasible. There is a possibility to circumvent this, which is to store a certain number of optimal deployments instead of all possible deployments. As the dynamic set changes over time however, the system may end up in a deployment in which it is impossible to start an application without switching to a deployment where existing applications must be migrated, breaking the RT constraints. We conclude that design-time deployment methods are not suitable for dynamic sets.

4.2.2 Run-time deployment

Run-time deployment methods on the other hand are able to respond dynamically to each event [14, 60, 65, 82]. Such deployment algorithms consume processing time on the target platform. The computation time and energy are therefore severely limited, which makes use of heuristics inevitable. Given the same set of optimisation objectives, such heuristics must therefore likely settle for a sub-optimal solution. It is hard to prove that heuristics always produce a solution, even a sub-optimal one, within the available time. If no solution is found, an arriving application may cause a violation of the RT constraints of a running application. Therefore it is difficult to provide hard RT guarantees for run-time deployment methods.

4.2.3 Hybrid deployment and deployment

Hybrid methods attempt to combine the best of both by combining design-time exploration results with a simple resource manager at run-time [117–119]. The method presented in this thesis is a hybrid that combines existing elements from [45, 137].

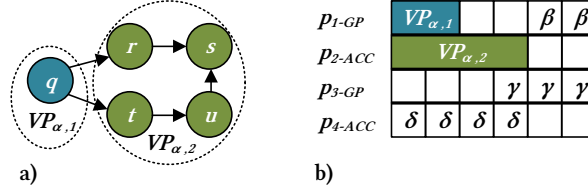


Figure 4.1: **a)** Dataflow application α consisting of five actors is mapped to two VPs. **b)** $VP_{\alpha,1}$ and $VP_{\alpha,2}$ are deployed on general purpose processor p_1 and accelerator p_2 with a budget of 2 and 4 TDM slots respectively. The platform otherwise contains applications β , δ and γ .

The crux of this approach is a separation of concerns. The timing analysis is performed at design-time, after which clusters of dataflow actors are mapped to VPs, see application α in Figure 4.1a. These VPs are containers that guarantee that the real-time constraints of the application are met, irrelevant of which processors the VPs are mapped on.

The run-time response to events is the responsibility of a resource manager which *deploys* the VPs onto physical processors. A VP may be deployed on any physical processor p_i of the correct type with sufficient free capacity, using TDM arbitration. See Figure 4.1b, the two VPs are deployed on a general-purpose processor and accelerator respectively. VPs that belong to applications β , δ and γ are already running and are not affected in any way.

This two-layer design approach has the advantage that the compute intensive timing analysis and mapping are only performed at design-time. At run-time it must be calculated whether sufficient resources are available, and how to fit the VPs into this space. Therefore any application that has been analysed may be started at run-time, even if it was not known at the time at which the system was started. Applications can be started and stopped at any time independent of the current deployment, as long as sufficient capacity is available. This enables online software updates.

4.3 WCRT analysis for intra-application mapping

The goal of the intra-application mapping is to find an allocation of actors to VPs that satisfies the throughput constraint. The existing design flow depicted in Figure 2.1 on page 22 uses heuristics to explore a number of mappings in the *mapToVRs* algorithm [137]. The throughput of each mapping is calculated in the *analyseBAG* algorithm.

To analyse the throughput of a mapping the *Maximum Cycle Ratio (MCR)* of the *Binding-Aware Graph (BAG)* must be calculated [37]. As explained in Section 2.4 and Section 3.5 the BAG consists of the application graph, annotated with models of the hardware, mapping and scheduling. The WCRT is dependent on the interaction between all these models, and must be calculated when generating the BAG. In this

section we explain the WCRT calculation in detail because it is required to understand the evaluation in Section 4.6.

4.3.1 Platform preliminaries

A number of platform properties are relevant for the WCRT analysis. Firstly, CompSOC was designed according to the *Globally Asynchronous, Locally Synchronous (GALS)* paradigm [71]. This means that the processor, memories and other hardware blocks on one tile are supplied with the same physical clock signal. Different tiles however are clocked by different clock signals, and so are the interconnect and shared memories. The VPs on each processor are arbitrated using TDM scheduling. Because we cannot make any assumptions about the TDM wheel alignment between tiles, we must account for the worst-case TDM misalignment during timing analysis. This means that two actors that communicate but are mapped to different tiles may encounter the maximum possible waiting time t_{wait} , see Subsection 4.3.2.

The fault-tolerance concept in Chapter 5 requires re-deployment of VPs. The channels that store the state of an application are implemented with FIFO buffers, which must be accessible from each tile to minimise the timing penalty for re-deployment. To achieve this we store the FIFOs in a central, protected memory. The time to access these shared memory locations is the same from each tile, and must be accounted for in the BAG. The WCRT for accessing shared memory locations can be modeled as presented in [92]. We implemented this model in the updated design flow shown Figure 2.2.

In the existing design flow the size of the VPs is calculated inside the *VPSizing* algorithm, see Figure 2.1. If the number of TDM slots that is required to meet the throughput constraint is larger than the TDM wheel size, the mapping is marked as invalid. Otherwise the mapping is valid, and the VP size is taken as a given. In this thesis we take a new approach and use the size of the VPs as a performance metric. This adds a new dimension to the DSE which allows us to control the processor utilisation, with the end goal of increasing the success of future deployment actions.

4.3.2 WCRT analysis

Analysis of the WCRT is not claimed as a contribution of this thesis, instead we present a method that builds upon related work [4, 75–78, 131]. Let us examine the WCRT calculation of application α depicted in Figure 4.1a. We may deploy $VP_{\alpha,1}$ to any two consecutive TDM slots of a general purpose processor, and $VP_{\alpha,2}$ to any four consecutive slots on an accelerator. An example deployment is shown in Figure 4.1b. Actors $\{r, s, t, u\}$ are mapped to $VP_{\alpha,2}$, and actors r and t have incoming edges from $VP_{\alpha,1}$. Because in a GALS system the TDM wheels of the processors are not synchronised, it is not possible to predict at which time an incoming token will enable actors r or t . We must therefore account for the worst-case and capture it in the WCRT.

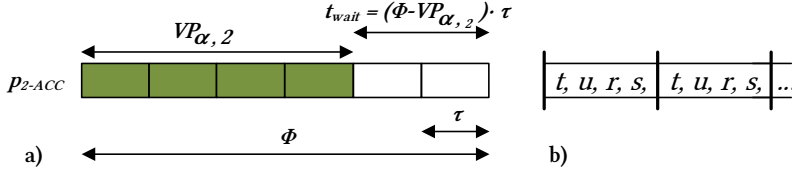


Figure 4.2: **a)** The TDM wheel of p_2 -ACC with Φ slots of length τ cycles. $VP_{\alpha,2}$ is allocated four slots. **b)** The SO schedule that is executed inside $VP_{\alpha,2}$ consists of two atomic blocks $[t, u]$ and $[r, s]$.

A detailed view of the TDM slot allocation is shown in Figure 4.2a. The TDM wheel size Φ of processor p_2 is six slots that each have a length of τ cycles. Because of the TDM wheel misalignment actor r and t may in the worst case be enabled right at the end of $VP_{\alpha,2}$, meaning that $t_{wait} = (\Phi - VP_{\alpha,2}) \cdot \tau$ cycles elapse before the actor fires with an execution time of ET cycles. It may happen that $ET > VP_{\alpha,2} \cdot \tau$ in which case we must also account for each additional wheel rotation until the actor finishes. The total WCRT of these actors thus depends on three addends, see Equation 4.1.

$$WCRT = t_{wait} + ET + \left\lceil \frac{ET}{VP_{i,j} \cdot \tau} - 1 \right\rceil \cdot t_{wait} \quad (4.1)$$

Note that the WCRT is linearly coupled to Φ via t_{wait} . This leads to the effect that the throughput may be increased by increasing the VP size, thereby decreasing t_{wait} . In effect no actual work is performed in these extra TDM slots, they are only necessary to meet the throughput constraint by reducing the waiting time.

In the existing design flow Equation 4.1 is applied to all actors regardless whether they have incoming edges from other VPs. This is overly pessimistic as t_{wait} is only relevant if the actor depends on tokens from other VPs. We implemented an improved WCRT calculation that differentiates between such *worst-case actors*, r and t in the example, and *local actors* such as s and u whose incoming tokens are produced in the same VP. Only the WCRT of the worst-case actors is linearly coupled to Φ in the improved model.

A SO schedule for $VP_{\alpha,2}$ is depicted in Figure 4.2b. The start of the schedule is indicated by the horizontal bars and the schedule is repeated indefinitely. Note that other schedules are possible, r may be scheduled anywhere before s . If a worst-case actor in the SO schedule is followed by one or more local actors, we consider them one *atomic block* of which only the worst-case actor has external dependencies. We can safely assume that the local actors will fire after the worst-case actors in a non-blocking manner. Worst-case actor t in the example is always followed by a firing of u , which is a local actor not dependent on data from other VPs. Similarly, r is always followed by s which is also not dependent on other VPs. We see that there

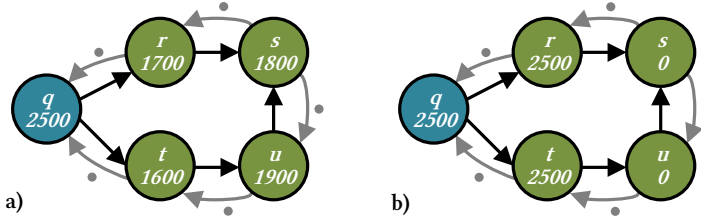


Figure 4.3: **a)** The BAG created using the existing calculation method, the grey channels model the buffer sizes. **b)** The BAG constructed with the improved WCRT calculation proposed in this section.

Actor	q	r	s	t	u
ET	500	700	800	600	900

Table 4.1: The *Execution Times (ETs)* of the actors from Figures 4.1 and 4.3 in cycles.

are two atomic blocks $[t, u]$ and $[r, s]$. The execution times of two actors x and y in each block may be added up, see Equation 4.2.

$$ET_{xy} = ET_x + ET_y \quad (4.2)$$

The *Execution Time (ET)* of the worst-case actor is assigned this value, the ETs of the local actors are set to zero. The WCRT of the atomic block is now dependent on the worst-case actor only. The condition for this optimisation is that the worst-case actor does not appear in another atomic block with a different composition.

Let us clarify the difference between the original and improved WCRT calculation using the mapping in Figure 4.1b. The TDM wheel size is Φ 6 slots, the slot length τ is 500 cycles and the ETs of the actors are given in Table 4.1. The BAG where the WCRT of each actor is calculated with the *existing* method is shown in Figure 4.3a, each actor is annotated with its WCRT. Figure 4.3b shows the same BAG but the WCRT of each actor is calculated with the *updated* method, i.e. Equation 4.2 is applied for each atomic block.

We see that the new WCRT of actor r is 2500 and that of t is 2500, which is lower than the sum of $r + s$ (3500) and $t + u$ (3500) when using the existing method. The throughput is calculated as $Thr = \frac{1}{MCR}$, the critical cycle that must be used to calculate the MCR is in both cases the counter-clockwise cycle through all actors. The MCR found with the existing method is $MCR_{ex} = \frac{9500}{2} = 4750$, that with the new method is $MCR_{new} = \frac{7500}{2} = 3750$. The corresponding throughput numbers are $Thr_{ex} = \frac{1}{4750} = 2.1 \cdot 10^{-4}$ and $Thr_{new} = \frac{1}{3750} = 2.6 \cdot 10^{-4}$, an improvement of 24%. These numbers are an example, other mapped applications might only profit very little or not at all from the improved calculation. We argue that many mapped applications will benefit from the new method, and that the throughput will never be worse than when using the existing calculation method.

Using the WCRT analysis described in this section a VP may be deployed (i.e. mapped and scheduled at run-time) on any processor that has sufficient available slots. This is the result of our choice in selecting this particular combination of worst-case timing analysis with an architecture and middleware that are composable and predictable, together with the two-layer mapping and scheduling approach where actors are mapped VPs which are in turn deployed on physical processors.

4.4 Design Space Exploration

The throughput of an application is dependent on the computation distribution (size of each VP) and on the storage distribution. To capture these in two metrics, we consider size of the computation distribution measured in TDM slots and the size of the storage distribution measured in tokens. This gives a three-dimensional design space spanned by the throughput, total VP size and total buffer size. Note that there can be multiple computation distributions with the same size, and multiple storage distributions with the same size. In Chapter 5 we will see that the distribution of VP sizes play a role in fault-tolerant mapping.

One way to explore the design space is to maximise throughput by enlarging both the VPs and buffers. As we have seen in Chapter 1 however, real-time constraints come in the form of a lower bound on the throughput. At the same time the memory footprint and processor utilisation together with their associated energy usage must be minimised. Therefore we propose to extend the DSE to keep the throughput above but close to the boundary and attempt to minimize the total VP size and total buffer size, resulting in a trade-off between these two dimensions.

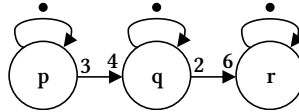


Figure 4.4: A SDF graph with rate differences.

A visualisation of the three-dimensional design space of the dataflow graph in Figure 4.4 is given in Figure 4.5a. The throughput constraint is shown by the red lines, the blue plane shows the design space. The intersection of the red and the blue plane indicates all Pareto-optimal points where the throughput is met. These are indicated in red in the corresponding heat map of Figure 4.5b, which is a visualisation of the throughput plane if we tilt Figure 4.5a so that the throughput axis is parallel to our line of view.

The size of the design space grows exponentially with the number of channels, the number of VPs to which an application must be mapped and the number of scenarios. Consider for example the dataflow application shown in Figure 4.1a. If all five channels may have a buffer size between 1 and 4 tokens, there is a total of 4^5 possible combinations. Mapping the application to two VPs that have a size in

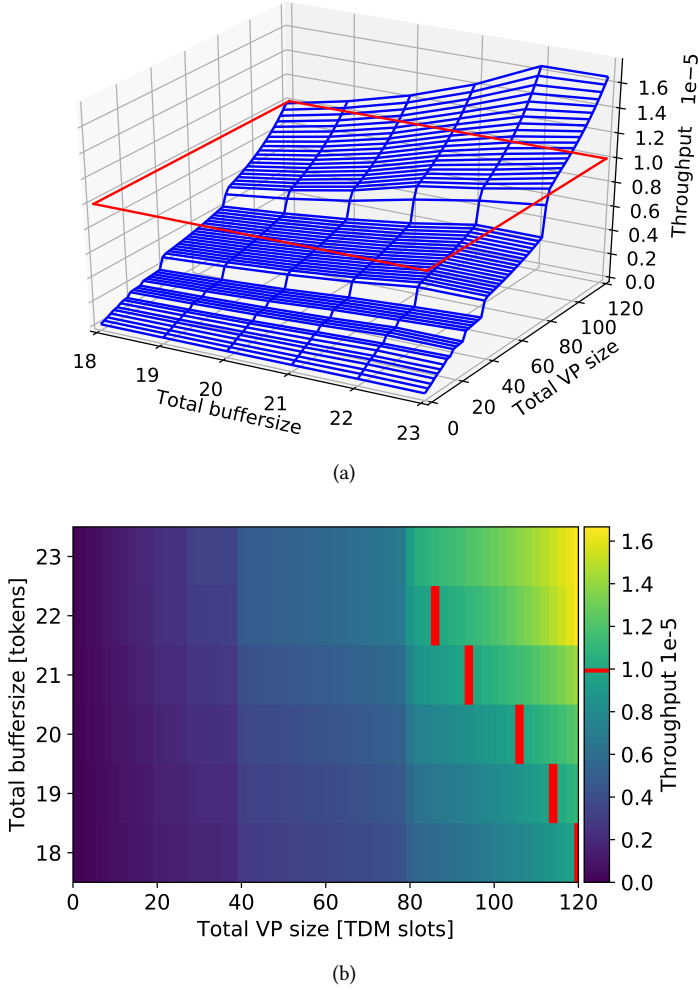


Figure 4.5: **a)** The three-dimensional design space spanned by the application throughput, total buffer size and total VP size in Euclidean space. The throughput constraint is indicated in red. **b)** A heat map of the same design space, the points where the throughput matches the throughput constraint are indicated in red.

between 1 and 60 TDM slots gives 60^2 combinations. Combining these means that each buffer combination must be tried with every VP combination, so we multiply these numbers and arrive at a design space with $3.7 \cdot 10^6$ combinations.

If the application contains a second scenario with different channels, each combination from scenario one may yield a different throughput with each combination

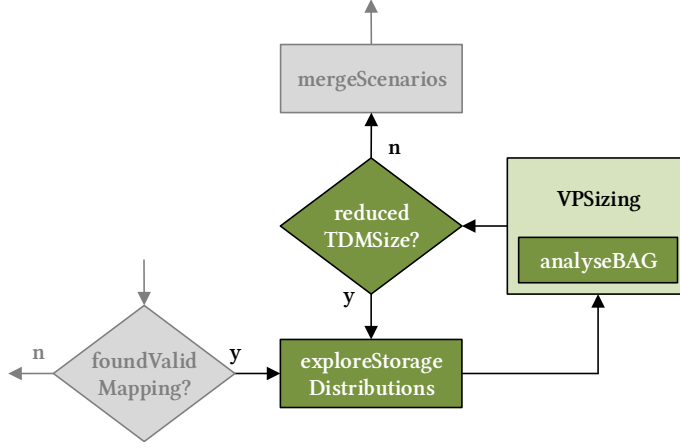


Figure 4.6: Loop of three algorithms that implement the DSE extension. This figure is a detail from the extended design flow depicted in Figure 2.2 on page 23.

from the scenario two. Therefore all combinations must be explored, adding another multiplication factor that gives a design space with a size of $1.4 \cdot 10^{13}$ combinations. If exploring a combination would take only 1 millisecond, the exploration of this search space would take 431 years. We conclude that even the design space of a simple graph is too large to explore exhaustively, and we must resort to heuristics even at design-time.

Existing work on the DSE of dataflow [107, 108, 132, 135] and FSM-SADF applications [4] focuses on the trade-off between total buffer size and throughput. The contribution of this chapter is the exploration of the trade-off between the total VP size required for the application (i.e. the computation capacity) and the total buffer size. Previous DSE approaches such as SDF³ revolve around an exploration of the storage distributions, after which an attempt is made to minimize the VP sizes. We argue that evaluating these dimensions one at a time fails to uncover the trade-off between buffer size and VP size. Because the throughput plane in Figure 4.5a curves in every direction, it cannot be guaranteed that we will arrive at the best possible solution if one dimension is fixed before the other is explored. Instead we propose to extend the DSE to explore both dimensions at the same time in the next section.

4.5 Trade-off: VP size against buffer size

In this section we extend the DSE with an exploration of the trade-off between VP size and buffer size. The extension consists of a loop with three algorithms, see Figure 4.6. The core of the extension is the *exploreStorageDistributions* algorithm, the pseudo code of which is listed in Algorithm 3.

Algorithm 3 The proposed DSE that trades off VP size against buffer size.

```

1: function EXPLORESTORAGE DISTRIBUTIONS(mappings, thrConstraint)
2:   for mapping in mappings do
3:     VPs = selectLargestVPs(mapping)
4:     channels = findInterVPChannels(VPs, mapping)
5:     newMappings = generateChannelCombis(channels, mapping)
6:     for nm in newMappings do
7:       mappings.add(nm)
8:     end for
9:   end for
10: end function

```

4

One or more valid mappings are passed to the *exploreStorageDistributions* algorithm by the *foundValidMapping* algorithm. Instead of accepting the storage distributions of these input mappings as a given, we generate a number of new storage distributions from each valid mapping in which the size of one or more channels is enlarged. To avoid exploring the design space exhaustively, we propose the following heuristic for selecting the candidate channels whose buffer will be enlarged.

We cycle through the list of mappings in line 2 of Algorithm 3. In line 3 all VPs are sorted based on their size, and we select the half of the VPs that has the largest size. From these VPs we determine which channels receive tokens from other VPs in line 4, the intra-tile channels. Next, we enlarge all the selected inter-tile channels by the step size and generate a mapping for each combination of these enlarged channels in line 5. All these new mappings are added to the list of existing mappings in line 7.

We ignore intra-tile channels in line 4 because two actors on the same tile cannot be pipelined anyway. It is possible to construct graphs in which intra-tile channels are critical and constrain the throughput, but this is not common. Actors on different tiles however may be pipelined if the buffer size is sufficiently large. The tokens in such channels are furthermore transported through the NoC and may have a considerable latency. Enlarging the buffers allows deeper pipelining as the producers work in advance, hiding the latency.

Once the *exploreStorageDistributions* algorithm has finished, the VP sizes of the newly generated mappings are calculated in the *VPSizing* algorithm in Figure 4.6. Determining the VP sizes includes calculating the throughput. Mappings for which no valid VP sizes can be found do not meet the throughput constraint and are discarded. The *reducedTDMSize* algorithm then checks if any of the new mappings has a smaller total VP size than the previous minimum. If so, the loop is repeated. If not, the loop is terminated and the algorithm performs a Pareto minimisation of the mappings in the list. This list is forwarded to the *mergeScenarios* algorithm, in which the designer can select a mapping manually. As there is no previous minimum in

the first iteration of the loop, *exploreStorageDistributions* is always executed at least twice.

We will illustrate the DSE extension using the two-VP mapping shown in Figure 4.1b. We assume that all channels initially have a buffer size of 1 token, the initial storage distribution of channels $[qr, rs, qt, tu, us]$ is denoted as $\langle 1, 1, 1, 1, 1 \rangle$. The *exploreStorageDistributions* algorithm will derive several new storage distributions from the given mapping. Firstly, $VP_{\alpha,2}$ is selected in line 3 because it is the largest VP. A list of the inter-tile channels in this VP is created in line 4, resulting in channels $[qr, qt]$. In line 5 all combinations of storage distributions in which channels $[qr, qt]$ are enlarged by the step size are generated, resulting in the distributions $\langle 2, 1, 1, 1, 1 \rangle$, $\langle 1, 1, 2, 1, 1 \rangle$ and $\langle 2, 1, 2, 1, 1 \rangle$. New mappings with these storage distributions are added to the existing mapping in line 7.

The VP sizes of the new mappings are calculated in the *VPSizing* algorithm, the throughput constraint is satisfied for all mappings. The *reducedTDMSize* algorithm determines that all mappings result in the same VPs distribution, namely $[VP_{\alpha,1}, VP_{\alpha,2}] = [2, 4]$ slots, so the minimum total VP size is 6 slots. It decides to do a second iteration of the loop, in which the same channels are enlarged once, which yields the same result. As the minimum VP size has not become smaller after the second iteration, the loop is abandoned.

We see that the DSE starts from one or more initial mappings and selects the inter-tile channels of the VPs with the largest size. It generates all combinations of the selected inter-tile channels enlarged by the step size. If such a new distribution leads to an increase in throughput, the size of the VPs can be reduced in *VPSizing* algorithm. In other words, we “push down” the size of the VPs at the cost of storage space. Looking at Figure 4.5a, this means we start in blue plane above the throughput constraint and move to the maximum on the buffer size-axis, to the minimum on the VP size axis and towards the red plane on the throughput axis.

The DSE terminates when no improvement is found, and results in a list of valid mappings. The mappings in this list are guaranteed to be above the throughput constraint, and represent Pareto-optimal points within the set of configurations that was explored. As an heuristic is used, it is possible that the DSE missed optimal points that would be in the list if they had been found.

4.6 Experimental evaluation

In this section we evaluate the DSE using the application shown in Figure 4.4. This application has one scenario, three actors and five channels with different rates. The application is mapped to two VPs with a wheel size Φ of 60 slots each. To show how the DSE works we perform a nearly-exhaustive exploration of the design space and visualize the result in different graphs. First we evaluate the trade-off between throughput and buffer size by forcing the use of all TDM slots on both VPs, i.e. we explore only the buffer size dimension. This results in the graph plotted in Figure 4.7.

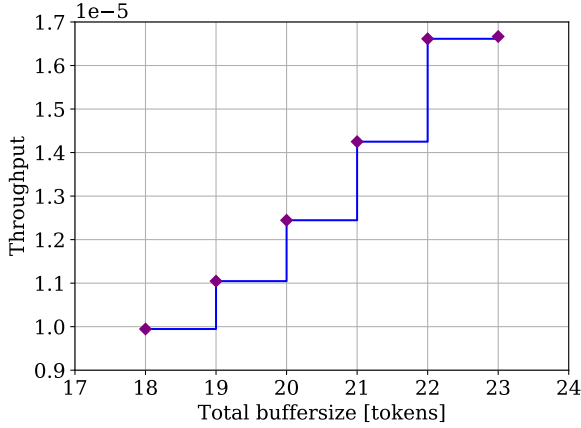


Figure 4.7: The throughput plotted against the total buffer size when mapping the application to two VPs with a size of Φ slots.

The heuristic finds a minimal deadlock-free buffer size of 18 tokens, and the maximal throughput is achieved after five steps at a buffer size of 24 tokens.

Next, we decrease the total VP size with a step size of two (i.e. one TDM slot in both VPs) and perform a throughput analysis in each step. The resulting three-dimensional design space is shown in Figure 4.5a, the corresponding heat map can be found in Figure 4.5b. Note that the figure does not show each combination of buffer sizes and VP sizes, i.e. the full design space, but rather gives us an impression of its shape. The data points from Figure 4.7 can be found in the two-dimensional plane at the 120 TDM slots marker. Starting from 120 slots and going down, we see that the throughput decreases steeply with the total VP size. Note that at every intersection of the plane there is a discrete data point. These points are interconnected to visualise the plane, but in reality the plane is not continuous, i.e. there are no other data points on the lines. This is more accurately shown by the heat map in Figure 4.5b that visualises these data points as discrete blocks. On the VP axis we note two points where the throughput suddenly drops significantly as the number of TDM slots decreases. The heat map in Figure 4.5b shows these drops even more clearly at the 40 and 80 TDM markers.

The explanation for this can be found in the WCRT analysis that was discussed in Section 4.5. If the number of TDM slots assigned to a VP is reduced, the WCRT of the worst-case actors goes up as can be deduced from Equation 4.1, possibly affecting the critical cycle(s) that determine the throughput. The current critical cycle(s) may depend on one or more worst-case actors. The first addend of the equation grows steadily as the VP size decreases, and is responsible for the gradual reduction in throughput between e.g. 120 and 80 slots. The sudden drops in the graph are caused by the ceiling function in the third addend, as the value of t_{wait} is added whenever

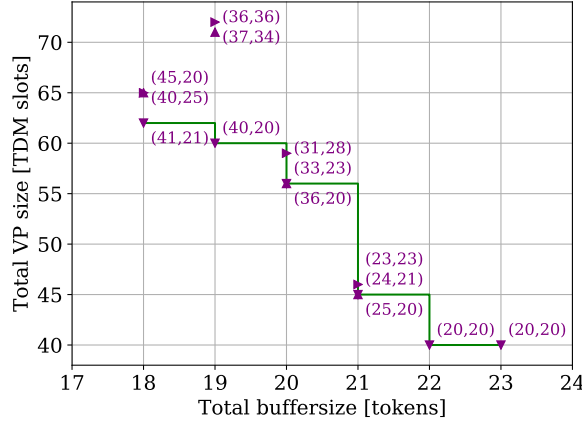


Figure 4.8: The trade-off between total VP size and total buffer size. Data points with the same total buffer size have different markers, the actual sizes of both VPs are reported next to each marker. The green line indicates all Pareto-optimal points.

$\frac{ET}{VP_{i,j} \cdot \tau} - 1$ passes the next largest integer value. For example, if the slot length τ is 500 slots and the ET is 5000 cycles, the value of the addend will jump from t_{wait} at 5 TDM slots to $2 \cdot t_{wait}$ at 4 TDM slots.

From the design space in the heat map of Figure 4.5b we conclude that between 0 and 40 TDM slots the influence of the buffer size is negligible. Between 40 and 120 TDM slots however both dimensions change independently. We conclude that it is indeed necessary to explore both dimensions at the same time.

The VP size can now be plotted against the total buffer size if we pick a throughput constraint of e.g. $1.0 \cdot 10^{-5}$ iterations per time unit. The DSE will pick points above but as close to this constraint as possible, see Figure 4.8. All points in this graph meet the throughput constraint. Next to each data point the sizes of both VPs are denoted. This graph unveils the trade-off between the total VP size and total buffer size, and is the area of the design space that we are interested in. To put this in perspective, both two-dimensional graphs from Figures 4.7 (purple) and 4.8 (green) are positioned in the three-dimensional design space in Figure 4.9. It clearly shows the difference between the existing DSE in which the throughput was maximised at all costs (purple plane), and the new DSE in which the throughput constraint is matched as closely as possible while trading VP size against buffer size (green plane).

In Section 5.7 we select 7 applications for the experiments in Chapter 5. The trade-off between total VP size and total buffer size for these 7 applications is depicted in Figure 4.10, along with a copy of Figure 4.8 named *synth-rates-sdf*. In contrast to that last graph all others show only one step, indicating that there are only two viable options: one that minimises the VP size and one that minimises the buffer size. This

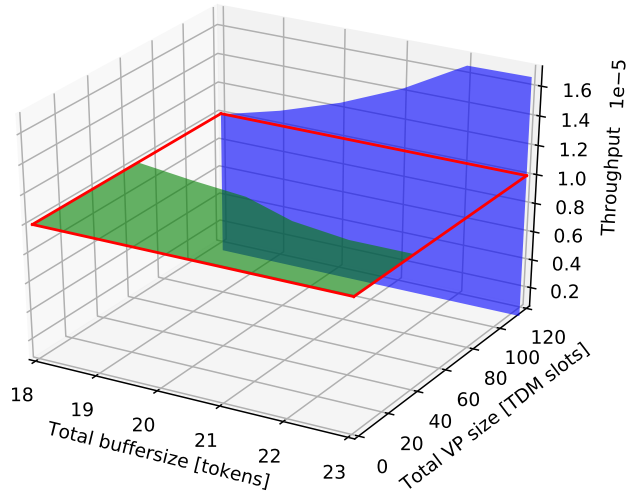


Figure 4.9: The buffer size-throughput trade-off plane of the existing DSE (purple) and buffer size-VP size trade-off plane of the DSE proposed in this chapter (green) positioned in the three-dimensional design space, the throughput constraint is indicated in red.

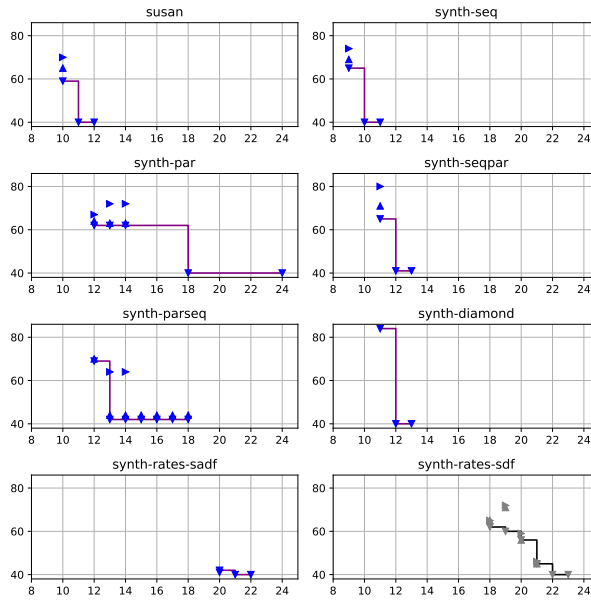


Figure 4.10: The trade-off between total VP size and total buffer size for all the applications depicted in Figure 5.4 (upper six graphs and lower left), and the example graph shown in Figure 4.4 (lower right, copy of Figure 4.8.)

is due to the fact that the synthetic applications are rather straightforward and have no rate differences.

4.7 Related work

Existing dataflow mapping approaches usually offer a trade-off between the throughput and another property such as energy or the size of the storage distribution. A trade-off between throughput and energy is presented in [67]. The trade-off between throughput and the size of the storage distribution has been explored for SDF in [132], CSDF in [135], for MCDF in [109] and for FSM-SADF in [4]. Processor utilisation is not considered in these approaches, which means that the throughput for a certain storage distribution is calculated assuming use of all available processor capacity. This might be a good strategy for memory-constrained embedded systems, as the FIFO buffers that implement dataflow channels are usually located in the on-chip memories. As we have seen in SP.4 however, the processor usage and memory footprint must be balanced to increase the probability of success of future deployment actions. Trading processor utilisation against both throughput and storage distribution size is the unique property that sets our contribution apart from the related work.

The work in [83, 85] is similar because VPs are also computed at design-time and deployed at run-time by a resource manager. While the goal is also to increase the probability of successful deployment, this work focuses on NoC architectures and does not take processor utilisation into account.

4.8 Summary

In this chapter we presented solution CB.4 that solves sub-problem SP.4 and paves the way for meeting requirement RQ.3b, executing dynamically changing sets of applications. We zoomed in on VP deployment and motivated our choice for a hybrid deployment strategy. The goal of this chapter is to allow the designer to increase the success of future run-time deployment actions by balancing the resource requirements of applications during the design-time intra-application mapping and scheduling, the *Design Space Exploration (DSE)*.

The existing DSE trades throughput against total buffer size. The crux of our approach is to add another dimension to the DSE, namely the VP size. This allows to trade the total VP size against the total buffer size under a throughput constraint in the DSE, resulting in a number of Pareto-optimal points. The choice between these points allows to balance the run-time processor utilisation and memory requirements. We show that the throughput changes independently with the VP size and buffer size. This means that the curve of interest, the intersection between the Pareto-plane of VP size and buffer size and the throughput constraint, cannot be found by exploring these dimensions one by one. Instead we propose an extension of the DSE algorithm that explores both dimensions at the same time. The exper-

iments confirm that this yields the curve of the VP size versus the buffer size. A system designer can exploit the trade-off offered by the Pareto-optimal points to increase the success of future run-time deployment actions.



5 Fault-tolerant deployment

5.1 Introduction

In Chapter 1 we argued that the continuous decrease of the feature size in VLSI design leads to hot spots. These cause intermittent faults and processor shutdowns in the short term, and speed up the silicon aging process which leads to permanent faults in the long term [53, 102]. Requirement RQ.5 states that systems must be fault-tolerant, yet it is costly to add circuitry for fault correction. Instead we propose in this chapter a method for, and analysis of, fault-tolerant re-deployment of *Virtual Processors (VPs)* to handle intermittent and permanent processor faults.

Fault recovery may not interfere with running applications. We choose to let the resource manager perform re-deployment of the VPs located on the faulty processor at run-time. However, it cannot be guaranteed that sufficient resources are available to re-deploy the VPs from the faulty processor. This is captured in sub-problem SP.6, which states that the design flow must maximise the probability of success of future run-time re-deployment actions. To do so we propose solution CB.6 in this chapter, which consists of the contributions described in Section 5.2.

The contributions concern both the intra-application mapping phase as explained in Section 5.3 as well as re-deployment by the resource manager, see Section 5.4. A requirement for re-deployment is that actors that were firing when the fault occurred can be safely restarted, this is explained in Section 5.5. The concept for fault-tolerance is presented in Section 5.6 and evaluated in Section 5.7. We discuss related work in Section 5.8, the chapter ends with a summary in Section 5.9. An earlier version of this chapter has been published in [148].

5.2 Recovering from processor faults

As explained in Chapter 2, actors are mapped to VPs at design-time and deployed on physical processors at run-time. Deployment is the term that we use for inter-application mapping and scheduling, see the middle layer of Figure 1.6 on page 14. An example mapping of application α is shown in Figure 5.1a, the deployment in Figure 5.1b. We consider heterogeneous multi-processor platforms with two types of processors, *Accelerators (ACCs)* and *General Purpose (GP)* processors. The platform instance in Figure 5.1b, d and f contains two processors of each type.

As described in Section 4.3 a VP may be deployed on any processor of the correct type at run-time because we selected a particular combination of architecture, middleware, mapping, scheduling, deployment, and timing analysis. In particular, accounting for the worst-case misalignment of the TDM wheels during timing ana-

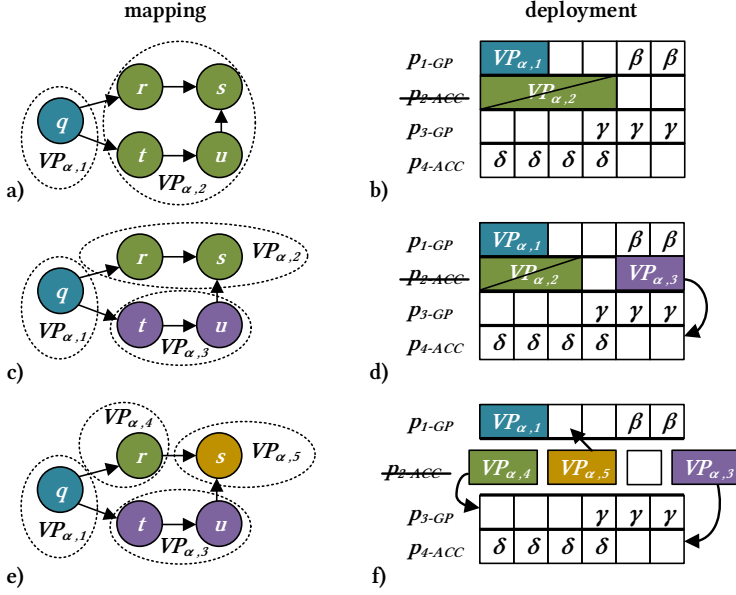


Figure 5.1: Application α with actors $q - u$ is mapped to an increasing number of VPs in figures a), c) and e), which are deployed on processors p_1 and p_2 of different types in figures b), d), and f). When p_2 encounters a fault: b) $VP_{\alpha,2}$ cannot be re-deployed because of its size and type; d) $VP_{\alpha,2}$ cannot be re-deployed because of its type, but $VP_{\alpha,3}$ can be re-deployed on p_4 ; f) we switch to another mapping at run-time in which $VP_{\alpha,2}$ is split and resized for a GP, and then re-deployed on p_1 and p_3 . Note that re-deploying a VP on a processor of a different type may change the ET and WCRT of actors, and thus the size of the VP.

lysis is a requirement for GALS systems. A benefit of this conservative approach is that applications are allowed to be started and stopped at any time independent of the current deployment, as long as sufficient capacity is available. In addition any application that has been analysed offline can be added to the platform at run-time, effectively separating mapping from deployment. This allows online software updates, which are becoming standard practice in many domains such as consumer electronics, medical and automotive.

We use the fact that VPs may be deployed on any processor at no extra cost to achieve fault-tolerance by re-deploying VPs when a processor encounters a fault. Deployment is an instance of the bin-packing problem, and it is impossible to predict what the distribution of VPs will be at a given time [85, 112]. When a fault occurs the free capacity may be too fragmented to allow re-deployment, see Figure 5.1b.

Our first contribution in this chapter is to improve the fault-tolerance of any given mapping through a design-time mapping strategy. By mapping each application to more VPs of a smaller maximum size the probability of successful re-deployment is increased, at the cost of an increase in total size. See the mapping in Figure 5.1c, and consider the improvement of the deployment in Figure 5.1d over that in 5.1b. The total size of application α has increased from 6 to 7 TDM slots, but $VP_{\alpha,3}$ can now be re-deployed on p_4 .

We see in Figure 5.1d that while $VP_{\alpha,2}$ would fit on GP processor p_3 , it is targeted for an ACC and still cannot be re-deployed. Our second contribution solves this by calculating additional mappings in which VPs are *split* and *resized* at design-time, and switching to such an alternative mapping at run-time. As shown in Figure 5.1e, $VP_{\alpha,2}$ is further split into $VP_{\alpha,4}$ and $VP_{\alpha,5}$ which are both targeted for a GP. The deployment in Figure 5.1f shows that the total size increases to 8 slots because the execution time of the actors is longer on a GP. Only when the fault occurs do we switch to the new mapping, and $VP_{\alpha,4}$ and $VP_{\alpha,5}$ can be re-deployed on p_3 and p_1 . The other VPs are not affected by the split and can continue without interruption. The benefits of this contribution are that the cost for splitting a VP are only paid when the fault occurs, and that it is possible to re-deploy VPs on different processor types. This comes at the cost of storing multiple mappings at run-time, increasing the memory requirements.

In Chapter 4 we presented a design-time method that allows to increase the success of future run-time deployment actions. This chapter presents a design-time method to maximise the probability of success of future run-time re-deployment actions. The difference is that when a fault has occurred, the processor capacity has been reduced and the probability for finding sufficient available TDM slots on the processors is lower than in regular deployment actions. Therefore both contributions attempt to reduce the size of individual VPs, at the cost of a larger total size. While the original *Design Space Exploration (DSE)* offered no control over the total VP size of a mapping, the extension in Chapter 4 does. In this chapter we exploit the extension by selecting Pareto-optimal points with a minimal total VP size, which maximises the probability of successful re-deployment by changing only the VP sizes.

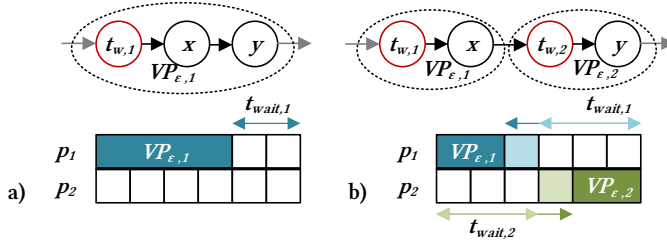


Figure 5.2: **a)** $VP_{\epsilon,1}$ has a waiting time $t_{wait,1}$ of 2 slots, captured by actor $t_{w,1}$. **b)** After splitting naively into $VP_{\epsilon,1}$ and $VP_{\epsilon,2}$ (dark blue and dark green) the waiting time is 4 slots for both VPs. The throughput constraint is not satisfied, so additional slots must be reserved (light blue and light green) to reduce the actor WCRT.

5.3 Mapping dataflow graphs

Our contributions can be applied to any RT MoC that can guarantee a minimum throughput or maximum latency given the actor *Worst-Case Execution Time (WCET)* and a method to compute the resulting *Worst-Case Response Time (WCRT)* of the application. The WCRT can be computed only if the platform is predictable. To illustrate this we use FSM-SADF, which can guarantee a minimum bound on the throughput given the actor WCET by means of computing the resulting WCRT of the application as explained in Section 4.3 [3]. The contributions of this chapter trade fault-tolerance against the size of the applications, measured in TDM slots. To provide insight in this cost trade-off we now take a detailed look at the design-time, intra-application mapping.

Dataflow actors are mapped to VPs. The size of each VP is determined by calculating the WCRT of each path through the dataflow subgraph that is mapped to the VP. Because we consider GALS systems, the TDM wheels of the physical processors may be misaligned. As each VP may be deployed on a different physical processor, the WCRT must account for the worst-case TDM wheel misalignment between VPs.

Consider for example application ϵ in Figure 5.2a. Actors x and y are mapped to $VP_{\epsilon,1}$, which uses 4 TDM slots out of a total wheel size of 6. The waiting time $t_{wait,1}$ is two slots, which is captured by the red actor $t_{w,1}$ that is inserted in front of the VP during timing analysis. The WCRT for $VP_{\epsilon,1}$ thus consists of the sum of $t_{wait,1}$ and the actor execution times.

The essence of the first contribution is to increase the number of VPs an application is mapped to. We split $VP_{\epsilon,1}$ into $VP_{\epsilon,1}$ and $VP_{\epsilon,2}$ as shown in Figure 5.2b. If the slot allocation of the new VPs would solely be based on the execution time, both receive two slots as indicated by the dark blue and dark green slots on p_1 and p_2 respectively. The waiting time is now 4 slots for both VPs. This may lead to a violation of the throughput requirement. To counter this, we can reduce the WCRT by reserving additional TDM slots, as indicated by the light blue and light green slots in the figure. Note that no actual work will be performed in those slots, they are necessary

only because of the assumed worst-case TDM wheel misalignment. In other words, the extra slots reduce the latency and increase the throughput. We see that the application now requires 6 slots in total, and may still be executed on one physical processor.

5.4 Resource manager

Deployment is an instance of the bin-packing problem, which is known to be NP-hard [112]. Use of heuristics is inevitable to find a solution at run-time. It is intuitive that the probability of success increases if the heuristic is provided with more but smaller items to fill the bins, *if* the sum of item sizes is the same. The latter is not true, as we have seen in Section 5.3. Instead the sum of the items (VPs) size grows if an application is mapped to more VPs. We can only expect a benefit if the increase in probability of success caused by the smaller VPs outweighs the decrease caused by the growth in total size.

Deployment of VPs on physical processors at run-time must be performed by a resource manager that is capable of dynamic loading [121]. In this chapter we will assume that such a manager is all-knowing, i.e. that it considers all possible solutions to the bin-packing problem if it must (re-)deploy an application. In reality it will only have limited time to find a subset of the solutions using a heuristic. Both the selection of such a heuristic and the practical implementation are outside of the scope of this thesis. The results that we find are therefore an upper bound on the fault-tolerance that a real resource manager will be able to achieve.

5.5 Fault model

In this thesis we focus on dealing with intermittent and permanent processor faults through checkpointing and restart [31]. To re-deploy a running application in case of a fault, its state must be consistent. Therefore we will now investigate the storage of state in the dataflow MoC. Remember that actors are stateless, and communicate by producing and consuming data tokens into and from the channels. All state is therefore stored in the channels.

An actor firing consists of three parts: consumption of the input tokens, execution, and production of the output tokens. During firing an actor may temporarily have internal state that is not in the tokens. To ensure that state cannot be lost, we impose the requirement that the input tokens of an actor may only be discarded after the firing is complete and the output tokens are updated. This ensures that an actor can always be re-fired, which guarantees that any fault can be corrected.

The channels must be accessible and in a consistent condition after a fault. To simplify our fault model we use the conservative assumption that the channels as well as the instruction memory are stored in a central, protected memory and are fetched each time when required by a processor. In reality this will have a major impact on the performance, and there are schemes to avoid this and still guarantee consistent

memory, e.g. error-correcting codes. Discussion of such schemes is outside of the scope of this thesis.

Techniques exist for detection of faults on a processor, e.g. through acceptance tests based on timing, coding, reasonability and structure, and may be implemented in hardware, software or both [31, 51, 104]. Upon detection we assume that an exception handler halts all VPs deployed on that processor. Because all input tokens of an actor that was firing are still valid, it can be restarted on another processor as long as the position in the schedule is known. The current position of the schedule can be stored in a (self-) edge.

There are many types of dataflow, e.g. SDF, CSDF and SADF. While each has its own specific rules and restrictions, our work is sufficiently general to be applied to all these types.

5.6 Fault-tolerance concept

5

Heterogeneous multi-processor platforms feature multiple types of processors that differ in clock speed, architecture and instruction set. The contributions that we present in this section are not limited by the number or variety of processor types available on a platform. The minimum number of processor types necessary to explain our contributions is two. For the sake of brevity we will continue with such a platform that only consists of *General Purpose (GP)* and *Accelerator (ACC)* processors throughout this chapter.

We assume, without loss of generality, that all actors in a dataflow graph can be executed on a GP. A subset actors can benefit from execution on an ACC because they have a shorter WCET on that type. All others cannot be executed on an ACC at all.

5.6.1 Re-deployment

A fault on a processor affects all applications of which one or more VPs are deployed on that processor. There are different strategies to deal with such faults. Those applications may for example be dropped, switched to a safe mode, or the VPs may be re-deployed on different processors [65]. We argue that re-deployment is the preferred strategy because the functionality is maintained and the interruption of service minimised. Whether sufficient free capacity is available for re-deployment can be checked by a resource manager. If this is not the case, it may still choose another strategy to deal with the fault. From the system perspective every increase of the probability of successful re-deployment is significant, because it reduces the risk of having to drop applications.

Re-deployment of VPs as introduced in Section 5.2 is enabled by two principles. Firstly, the MoC provides a method to compute the WCRT on a predictable platform given the actor WCET and a platform model, from which the VP size can be calculated. Secondly, the WCRT accounts for the worst-case TDM wheel misalignment

between VPs so that a VP may be deployed on any processor of the correct type (e.g. general purpose) that has sufficient TDM slots available.

The first contribution of this chapter is to improve fault-tolerance on a platform by increasing the probability on successful re-deployment. To do this we calculate alternative mappings for each application in which it is mapped to more VPs than in the *baseline* mapping that is explained next.

We consider two baseline mappings for each application. Firstly an application may be mapped to general-purpose VPs only, in which case it will not profit from the accelerators. We vary the amount of VPs and, considering all Pareto-optimal points of each of the resulting mappings, select the mapping with the smallest overall size in terms of TDM slots. This is the *homogeneous* baseline mapping. We will see in Section 5.7 that this always results in a mapping to one VP, denoted with $[gp]$.¹ Secondly an application may be mapped to at least one GP and one ACC processor. Again we vary the amount of GPs and ACCs, consider all Pareto-optimal points, and find that the $[gp, acc]$ mapping (i.e. one GP and one ACC) always results in the smallest overall size. We refer to this mapping as the *heterogeneous* baseline mapping.

5

5.6.2 Resize and split

Re-deployment of a gp is straightforward, it may only be deployed on a GP processor with sufficient free capacity. For an acc on the other hand there are two options. It may be deployed on an ACC processor with sufficient free capacity, or we may generate a second mapping in which the acc is resized (i.e. enlarged) to allow for a GP processor mapping. This expands the solution space and increases the probability of successful re-deployment when an ACC processor fails.

Consider for example a failure of the ACC processor to which the second VP of mapping $[gp, acc]$ is deployed. At design-time a second mapping $[gp, gp]$ is generated in which the first gp is forced to be identical, and the second VP accommodates the acc actors from the first mapping. During run-time the resource manager can switch to this second mapping if it cannot re-deploy the acc . The disadvantage is that the second mapping must be stored during run-time, and that the resource manager needs the capacity to switch between mappings.

The resizing strategy can potentially also be applied to a homogeneous mapping. As explained in Section 5.3, some TDM slots may be reserved solely to reduce the WCRT. Such slots may be removed from one VP and transferred to another without violating the throughput constraint, i.e. the VPs act like communicating vessels. This could be used to manipulate the location of available slots over the platform. As it does not change the total size of an application however, we expect only a minor benefit and will not explore this strategy further.

Instead we propose as second contribution of this chapter to *split* and *resize* VPs only when the ACC processor to which the acc is deployed encounters a fault at run-

¹ We use the abbreviations GP and ACC to refer to physical processors of type general-purpose and accelerator, and gp and acc for VPs that may be deployed on these respective processor types.

time. Consider again the mapping $[gp, acc]$, for which we generate a second mapping $[gp, gp, gp]$ to which we switch when a fault occurs on the ACC. The first gp is again identical, but acc VP is now spread over the two gp VPs. A split can also be applied to a homogeneous mapping.

The advantage is that the additional cost for splitting a VP is only paid when the fault actually occurs, which lowers the cost and platform utilisation in normal operation. This comes again at the price of storing the additional mapping and extending the resource manager with the capacity to switch between mappings. Switching between mappings is more complicated in comparison to the resize strategy, as each VP has an internal schedule that must be split in two in a consistent manner. This split operation may be costly depending on the size and type of the schedule, and will cause additional switching overhead at run-time.

5.7 Experimental evaluation

5.7.1 Preliminary

We use eight different applications for our experiments. Two of these are real-world streaming algorithms, namely a JPEG decoder and an one-scenario version of the SUSAN edge detection algorithm introduced in Chapter 3. Their dataflow graphs are depicted in Figure 5.3.

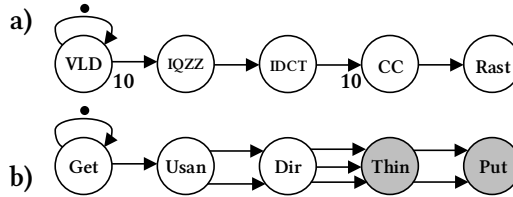


Figure 5.3: Two dataflow graphs of **a)** the JPEG decoder, and **b)** the SUSAN edge detection algorithm. Actors *Thin* and *Put* can benefit from an accelerator.

We furthermore use seven synthetic applications to explore the effect of typical graph structures, see Figure 5.4. Applications *seq* and *sota* model sequential applications where the result of a previous computation is required to continue with the next. The opposite is modeled in *par*, almost all computations are parallelised. Combinations of the two former types of applications are modeled in *seqpar* and *parseq*. An application where graph first diverges, i.e. the result of one computation is required by multiple others, and then converges, i.e. the results of two computations are required by one other, is modeled in *diamond*. An application with different rates and two scenarios is modeled in *two-scen*.

The worst-case execution times of the JPEG and SUSAN actors are measured on an instance of the CompSOC platform. Each actor in the synthetic applications has

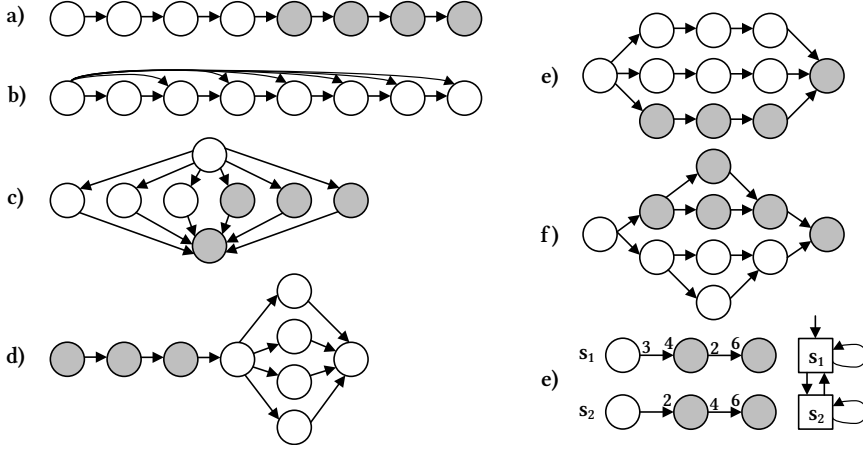


Figure 5.4: Six dataflow applications with different topologies: **a)** sequential (seq), **b)** sequential one-to-all (sota), **c)** parallel (par), **d)** sequential-parallel (seqpar), **e)** parallel-sequential (parseq), **f)** diamond, **g)** two scenarios (two-scen). Actors that benefit from an accelerator are indicated in grey.

an execution time of 10k cycles on a GP. Actors that can benefit from running on an accelerator are indicated in grey in the figures, their execution time is 1k cycles on an ACC, representing a speedup of 10x. All other actors cannot be mapped to an ACC at all.

We employ two different platforms to evaluate our contributions. A homogeneous platform with four GP processors will be used to investigate re-deployment of VPs to processors of the same type. A heterogeneous platform with two GPs and one ACC processor is applied for re-deployment of VPs to processors of a different type. Both platforms have a TDM wheel size of 60 slots on all processors, which allows for easy comparison. On many platforms the wheel size and slot length of GP and ACC processors will differ. It is important to note that the cost expressed in TDM slots cannot be compared between processors if this is the case.

Subsection 5.7.2 describes how we generate different mappings for both platforms. In Subsection 5.7.3 we create sets of applications for each mapping, and generate all possible deployments. In each deployment we simulate a failure of each processor and calculate the average probability of finding a valid re-deployment. The resize and split strategies are evaluated similarly in Subsection 5.7.4

The platform utilisation is obtained by dividing the number of used TDM slots used in a certain deployment by the total number of TDM slots available on the platform. We explore the trade-offs for different utilisations in 5.7.5.

5.7.2 Mapping

In the context of this chapter we modified the *mapToVRs* algorithm in the existing design flow, see Figure 2.2 in Chapter 2. This algorithm generates a number of Pareto-optimised actor-to-VP mappings for each application. We modified the mapping algorithm to always select the Pareto-optimal point that features the lowest total number of TDM slots, ignoring memory usage and bandwidth that dominate other Pareto points. This isolates the VP size from other mapping parameters and maximises the probability of successful re-deployment.

We create four mappings per application for the homogeneous platform by varying the number of VPs to which an application is mapped between one and four, i.e. $[gp]$, ..., $[gp, gp, gp, gp]$. The setpoint for the throughput is determined by mapping each application to a single VP with a size of exactly 40 TDM slots, the homogeneous baseline mapping. The two, three and four VP mappings are generated with the same throughput setting. The goal is to deploy four applications to four processors, so 160 out of 240 TDM slots or two-thirds of the platform will be used if the baseline mapping is selected for all four applications. We consider this a reasonable utilisation for a fault-tolerant platform.

	$[gp]$	$[gp, gp]$	$[gp, gp, gp]$	$[gp, gp, gp, gp]$
JPEG	[40]	[33, 10]	[52, 10, 53]	[1, 9, 53, 52]
SUSAN	[40]	[30, 10]	[29, 10, 2]	[1, 3, 1, 36]
seq	[40]	[35, 5]	[30, 5, 5]	[25, 5, 5, 5]
sota	[40]	[37, 40]	[38, 40, 40]	[25, 5, 5, 31]
par	[40]	[10, 30]	[5, 25, 10]	[5, 20, 10, 5]
seqpar	[40]	[36, 5]	[33, 5, 5]	[29, 5, 5, 5]
parseq	[40]	[22, 20]	[14, 27, 4]	[10, 27, 4, 4]
diamond	[40]	[20, 20]	[16, 4, 20]	[4, 4, 8, 27]
two-scen	[40]	[20, 20]	[7, 20, 20]	[−]

Table 5.1: Mappings for the homogeneous platform for all eight applications. The number of VPs varies from 1–4. An empty set indicates that no mapping can be found.

Table 5.1 shows the mapping of all applications to 1–4 VPs. The numbers in each cell give the size of the VPs, the total cost of a mapping is their sum. We observe that the total cost increases as the number of VPs increases. The reason for the increase is that the WCRT increases for each additional VP, as explained in Section 5.3. In case the cost does not increase (e.g. SUSAN between 1 VP and 2 VPs), the number of required cycles increases but is absorbed by the unused part of the TDM slots

already allocated. Note that no mapping can be found for two-scen using 4 VPs, as it contains only three actors.

The results of the JPEG and sota applications in Table 5.1 stand out. When either is mapped to 2 (sota only), 3 or 4 VPs the total number of required TDM slots is larger than the TDM wheel size. In case of the JPEG this is caused by the fact that the execution time is concentrated in the IQZZ and IDCT actors. For the sota application it is caused by the fact that every actor receives data from the first actor, therefore the algorithm cannot find a mapping of actors that reduces the WCRT. As they cannot be deployed on one processor when mapped to 3 or 4 VPs, we conclude that our concept does not work for these types of applications. We exclude these two from the re-deployment experiments and continue with the remaining seven applications in Table 5.1.

	$[gp, gp]$	$[gp, \mathbf{gp}, \mathbf{gp}]$	$[\mathbf{gp}, gp, \mathbf{gp}]$	$[\mathbf{gp}, \mathbf{gp}, \mathbf{gp}, \mathbf{gp}]$
SUSAN	[30, 10]	[-]	[29, 10, 2]	[-]
seq	[35, 5]	[-]	[5, 5, 30]	[-]
par	[10, 30]	[-]	[5, 30, 5]	[-]
seqpar	[36, 5]	[-]	[5, 5, 33]	[-]
parseq	[22, 20]	[-]	[-]	[-]
diamond	[20, 20]	[-]	[-]	[-]
two-scen	[20, 20]	[-]	[10, 20, 13]	[-]

Table 5.2: The split strategy applied to the $[gp, gp]$ mapping. The new VPs are indicated in bold. An empty set indicates that no mapping can be found.

Additional mappings must be generated to evaluate the split strategy proposed in Subsection 5.6.2. A fault may affect any combination of VPs of one application, so we generate a mapping for each possible permutation of faults. Each faulty VP must be split, while the others keep their size. In the $[gp]$ mapping there is only one VP to be split, resulting in the $[gp, gp]$ mapping in the third column of Table 5.1. Applied to the $[gp, gp]$ mapping for the homogeneous platform, we obtain the result shown in Table 5.2. The new VPs are indicated in bold type. We see that no mappings can be found if a fault is encountered in either the second VP or in both VPs at the same time. This is because the drastic increase of the WCRT in these two cases violates the throughput constraint.

We will deploy three applications to the heterogeneous platform using three different mappings, the 2 VP mappings can be found in Table 5.3 and the three VP mappings in Table 5.4. The setpoint for the throughput is chosen so that the sum of both VPs in the $[gp, acc]$ heterogeneous baseline mapping is 26 slots. Table 5.3 includes the results of the resize and split strategies, VPs that are resized or added are indicated in bold. It is not possible to find a solution for all applications when using

	$[gp, acc]$	$[gp, \mathbf{gp}]$	$[gp, \mathbf{gp}, \mathbf{gp}]$
SUSAN	[25, 1]	[25, 6]	[25, 2, 5]
seq	[23, 3]	[23, 23]	[23, 18, 7]
par	[22, 4]	[22, 21]	[22, 12, 12]
seqpar	[24, 2]	[24, 12]	[24, 8, 4]
parseq	[24, 2]	[24, 15]	[24, 7, 7]
diamond	[23, 3]	[23, 22]	[23, 10, 15]
two-scen	[22, 4]	[22, 24]	[22, 20, 10]

Table 5.3: 2-VP mappings for the heterogeneous platform with *resize* and *split* strategies. VPs that are resized or added are indicated in bold.

	$[gp, acc, acc]$	$[gp, \mathbf{gp}, \mathbf{gp}]$	$[gp, gp, acc]$	$[gp, gp, \mathbf{gp}]$
SUSAN	[25, 1, 1]	[25, 2, 5]	[4, 22, 1]	[4, 22, 6]
seq	[23, 2, 1]	[23, 18, 7]	[7, 18, 3]	[7, 18, 23]
par	[22, 2, 1]	[22, 20, 7]	[17, 7, 4]	[17, 7, 21]
seqpar	[24, 1, 1]	[24, 8, 4]	[10, 16, 2]	[10, 16, 12]
parseq	[24, 1, 1]	[24, 10, 4]	[10, 14, 2]	[10, 14, 15]
diamond	[23, 1, 2]	[23, 10, 15]	[19, 5, 3]	[19, 5, 22]
two-scen	[22, 2, 1]	[22, 20, 10]	[10, 22, 2]	[10, 22, 20]

Table 5.4: 3-VP mappings for the heterogeneous platform with the *resize* strategy. VPs that are resized or added are indicated in bold.

the split strategy for mappings $[gp, acc, acc]$ and $[gp, gp, acc]$ in Table 5.4, as the VPs are too small to further split up. As expected, the required number of slots increases steeply when resizing an *acc* for a GP processor because the actor execution times increase tenfold. We note that splitting the VP comes at a cost of one or two slots, but the maximum size of the two new VPs is always smaller than that achieved with the *resize* strategy.

5.7.3 Re-deployment

The probability on successful re-deployment is analysed using Algorithm 4. The mapping results for homogeneous and heterogeneous platforms from Subsection 5.7.2 are stored in *Comma-Separated Values (CSV)* files, and so is the architecture description. These are the input to the algorithm and are parsed into a data structure, see

line 2 and 3. The desired size of the application sets is also indicated in the mappings file, all permutations of that size are created in line 4. In line 5 all possible deployments for each set are generated. We consider unique deployments only, i.e. permutations achieved by swapping processors are excluded. A failure of each processor in each deployment is simulated in line 6. In this function we calculate the average probability of finding a valid re-deployment for each VP mapping. This results in a list of probabilities, which is returned as the end result. Algorithm 4 is boiled down to its bare essence, the pseudo-code of the different functions that are called are listed in Appendix B.

Algorithm 4 Analysis of the probability on successful re-deployment.

```

1: function ANALYSE_RE-DEPLOYMENTS(mappings.csv, architecture.csv)
2:   processors = parse_architecture(architecture.csv)
3:   mappings, vp_archs = parse_mappings(mappings.csv)
4:   appSets = create_application_sets(mappings)
5:   deployments = generate_deployments(processors, vp_archs, mappings,
   appSets)
6:   return generate_re-deployments(processors, vp_archs, deployments,
   mappings)
7: end function

```

5

The same algorithm is used for both the homogeneous and heterogeneous case, we will now focus on the former. For each of the four mappings on the homogeneous platform all permutations of four out of seven applications are created, which amounts to thirty-five unique sets of four applications. The total number of deployments returned by the *generate_deployments* function for a 2 VP mapping onto 4 processors is in the order of 10^2 , for a 3 VP mapping it is 10^5 and for a 4 VP mapping 10^7 .

The results of re-deployment on homogeneous platforms are summarised in the upper row of black boxes in Figure 5.5. On the arrows percentages represent the probability of successful re-deployment averaged over the processors, sets and deployments. Inside the boxes the average cost of the re-deployment is denoted, which is in this case equal to the initial mapping. The baseline $[gp]$ mapping has a cost of 40 slots. There is exactly one possible deployment, namely one VP per processor, which leaves 20 free slots per processor. The probability that a VP can be re-deployed when a processor fails is then 0%.

Next, the probability of successful re-deployment of the $[gp, gp]$ mapping is 74.1% at an average cost increase of 1.7 slots or 0.71% of the overall capacity. The $[gp, gp, gp]$ mapping results in a probability on successful re-deployment of 60.6% at a cost of 9.1 slots. This is lower than for the $[gp, gp]$ mapping, which can be attributed to the fact that the utilisation of the platform is higher even before the fault occurs, which reduces the solution space.

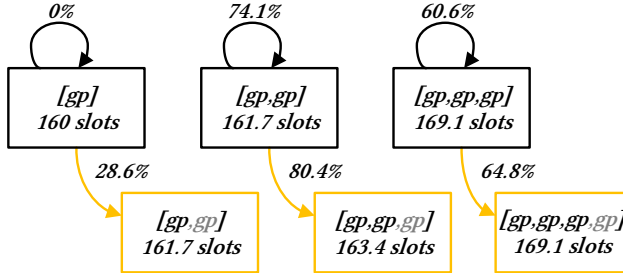


Figure 5.5: Re-deployment results for the homogeneous platform. Arrows indicate the probability of successful re-deployment, the boxes contain the number of VPs and cost of the solution. Black boxes and arrows indicate the default strategy, yellow the split strategy.

5

The results confirm the validity of our first contribution, i.e. the fault-tolerance increases when an application is mapped to more VPs. The success of the approach will however depend on each unique combination of TDM wheel size, number of applications and processors, throughput constraints, initial VP sizes, and dataflow graph topologies. The possible gains from this strategy should therefore be analysed for each system independently.

5.7.4 Resize and split

We repeat the re-deployment experiment on the homogeneous platform for the split strategy. As shown in Table 5.2 it is not possible to split every VP, those VPs are re-deployed without any changes. The results are shown in the lower row of yellow boxes in Figure 5.5. Note that after re-deployment the platform contains applications mapped to different numbers of VPs, indicated by the grey font in the figure. When the split strategy is applied to the baseline mapping the probability on successful re-deployment is 28.6%, for the $[gp, gp]$ and $[gp, gp, gp]$ mappings the probabilities are 80.4% and 64.8% respectively. We see that the split strategy offers an improvement over the default strategy in all three cases. The cost increases marginally in the 1 and 2 VP case, and not at all for the 3 VP case because all VPs can be split without additional cost. We conclude that the split strategy further increases the fault-tolerance on the homogeneous platform.

Algorithm 4 is also applied to calculate the probability of successful deployment for all three mappings of the heterogeneous platform. For each of these mappings we create thirty-five unique sets of four applications in line 3, which are all permutations of three out of seven applications. In line 5 we simulate a failure of the ACC processor only to zoom in on re-deployment between processors of different types. The initial deployments are depicted in the upper row of black boxes in Figure 5.6. An *acc* cannot be re-deployed without either resizing or splitting it. The

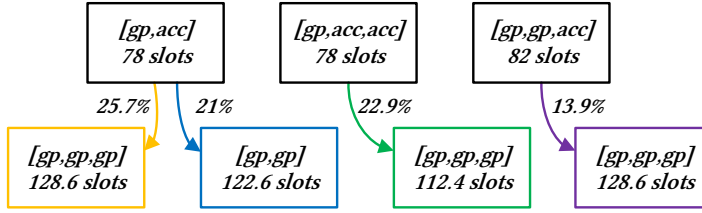


Figure 5.6: Re-deployment on the heterogeneous platform. Arrows indicate the probability of successful re-deployment, the boxes contain the VP size and cost in slots. Black boxes indicates the initial deployments, yellow the split strategy and blue, green and purple the resize strategy for the different mappings.

resizing strategy for the $[gp, acc]$ mapping is depicted in the blue box (second from left) in the second row. We see that the probability on successful re-deployment is 21%, while the total size increases with 44.6 slots, or 24.8% of the total platform capacity. The split strategy for the same deployment is shown in the leftmost yellow box, and yields a 25.7% probability of success at a cost of 46 slots. Again we see that the fault-tolerance increases at the VP size, confirming the second contribution. The higher costs are caused by the conversion of *acc* VPs to *gp*.

The probability on successful re-deployment of the $[gp, acc, acc]$ mappings with the resize strategy is 22.9%, which is a minor improvement over the $[gp, acc]$ mapping but still not as good as the split strategy. The resource usage for the $[gp, acc, acc]$ mapping however is significantly lower than for both $[gp, acc]$ solutions. This is because the actors are distributed over 3 VPs from the beginning, providing a better spread of actors than the $[gp, acc]$ split strategy. The $[gp, gp, acc]$ mapping does not perform well compared to the others because the utilisation of the GP processors is higher from the beginning, leaving less available capacity for re-deployment.

5.7.5 Trade-offs

The experiments on both the homogeneous and heterogeneous platforms showed a trade-off between fault-tolerance and TDM slots for a fixed platform utilisation at the time of the fault. Because in reality the utilisation is unpredictable, we vary the utilisation on the heterogeneous platform on the x-axis of Figure 5.7. The graph furthermore shows the solution cost for a selected number of data points. The data from Figure 5.6 is indicated in the graph by the red box.

If the utilisation is 54 slots or lower, all strategies result in successful re-deployment because there is always sufficient free capacity to re-deploy each VP. Likewise, if the utilisation is 90 or higher, no re-deployment can be found for any of the strategies because there is insufficient free capacity. These lower and higher thresholds will exist on every combination of platform and application set, but their value cannot be predicted.

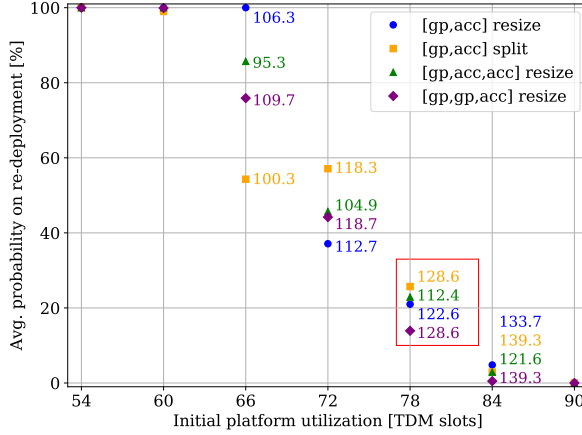


Figure 5.7: The probability of re-deployment for a varying initial platform utilisation and different strategies. The cost of the solution in TDM slots is printed in color for selected data points.

5

In between these thresholds each strategy has a curve that trades off fault-tolerance with cost. We see that the $[gp, acc]$ resize strategy provides better fault-tolerance at the edges of this curve, while $[gp, acc]$ split performs well in the middle. The results of the latter are however somewhat unpredictable because it often happens that a VP cannot be split. Hence the anomaly at an initial platform utilisation of 72 slots, where the strategy performs better than at 66 slots.

The $[gp, acc, acc]$ strategy consistently outperforms the others in terms of the solution cost and always has a (shared) second place in fault-tolerance. This may well make it the strategy of choice in this particular setup, unless the fault-tolerance must be maximised at all cost. In that case the $[gp, acc]$ split strategy provides the highest probability of successful re-deployment, averaged over all utilisations.

5.8 Related work

An overview of concepts and terminology for fault-tolerant systems can be found in [7, 31]. Surveys of fault-tolerance targeted specifically for multi-processors can be found in [88, 97]. In this section we discuss existing work related to re-deployment and to resource managers.

5.8.1 Re-deployment

Deployment strategies fall in three categories: design-time, run-time and hybrid [120]. The hybrid that we use combines elements proposed in [45, 137] and allows

any deployment of VPs at run-time at the cost of rigorous timing analysis at design-time.

An alternative method is to calculate and analyse a fixed number of VP deployments at design-time [73, 114]. In [21, 73] tasks from faulty processors are re-deployed (migrated) with the goal of minimising migration cost and the degradation of application throughput at the same time. In the former work all possible fault-mappings are generated at design-time. This allows more precise timing analysis, but the number of deployments grows exponentially with the number of VPs, processors and re-deployment events. These deployments must also be stored during run-time, and unlike our solution no new applications can be accepted at run-time.

Spare processors are reserved to cope with faults in [15, 114]. While this method is safe and easy to verify, it is not affordable in low-cost, high-performance systems, for which our method is targeted. Furthermore it can only deal with as many faulty processors as there are spare processors, while our method allows to keep on handling faults.

Re-deployment strategies focused on performance and the reduction of the communication energy are presented in [15, 28]. The former formulates the problem as an integer linear programming problem to generate all deployments for their KPN applications at design-time, and proposes an online heuristic for use at run-time. The latter proposes a spare processor placement technique and assesses its impact on the fault-tolerant properties, and consider the effect of system fragmentation but do not try to remedy this by splitting up applications as we do in this chapter.

Kahn Process Network (KPN) as used in [28, 114] is a dataflow MoC that cannot be statically analysed. Therefore there is no method to compute the WCRT, and our method for fault-tolerance cannot be applied for such applications but can still be of use if occasional deadline misses are acceptable.

There exist several works that attempt to avoid faults due to aging and thermal effects. An interesting concept is presented in [124], where computations are migrated away from hot spots preventively based on a temperature model. The lifetime of multi-processors is extended by accounting for aging in the mapping algorithm in [10, 24], and for the aging of NoC links in particular in [22]. Similarly, the lifetime is improved by accounting for inter- and intra-application thermal variations in [25] and by accounting for the voltage and frequency in [23]. A method for runtime re-deployment that uses the intermittent fault rate as an indicator of wear-out to improve the lifetime of multi-processors is presented in [105]. The advantage over our method is that a fault may be prevented completely, yet this cannot be guaranteed. Therefore fault-tolerance is still required, and the works mentioned in this paragraph are complementary rather than a replacement for our method.

Another combination of design-time mapping and run-time deployment with support for scenarios is proposed in [100]. This work differentiates between intra- and inter-application scenarios. The latter describe combinations of applications that may run at the same time, meaning that applications do not have the flexibility to start and stop at any time. This is compensated for by a run-time mapping algorithm that attempts to improve system performance by performing mapping

customisations This work uses KPN dataflow and targets homogeneous platforms, but does not specify the type of scheduling.

5.8.2 Resource manager

Resource managers that suit our requirements have been described in existing work, therefore we do not implement a resource manager in the context of this thesis. A comparison of different resource manager algorithms is given in [125]. Heuristics that minimise the communication overhead are given in [115, 116].

A resource manager with support for run-time deployment of dynamic sets is described in [84]. They show that a modified first-fit vector bin-packing is a good solution to the NP-complete problem as it can allocate 95% of the resources. The described approach assumes composability and two-layer mapping and scheduling using VPs. The resource manager described in this work would be a suitable solution for the platform that we use in this thesis.

A resource manager that combines a greedy heuristic with actor clustering is presented in [85]. Clustering increases the probability of successful deployment because it reduces the required bandwidth, except when processors are almost full. This is in line with our findings. Fault-tolerance however is not considered. Design-time mapping for mixed-criticality and run-time deployment is proposed in [65]. Low criticality tasks can be dropped to guarantee the WCRT of highly critical tasks in case of faults, but not re-deployed.

5.9 Summary

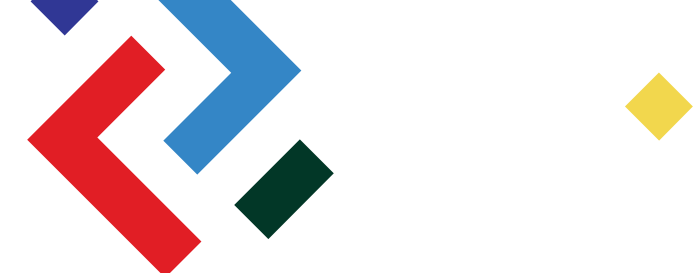
In this chapter we presented solution CB.6 that solves sub-problem SP.6 and enhances the fault-tolerance of systems, thus helping to meet requirement RQ.5 The design flow introduced in the previous chapters can analyse dataflow applications and map them to a set of VPs at design-time. At run-time a VP can be deployed by the resource manager on any physical processor of the target type that has sufficient available TDM slots. This gives the flexibility to add new applications during run-time, but makes it impossible to predict what the deployment will be at any given time. When a processor fails, the available slots on other processors may be too fragmented to re-deploy the VPs from the faulty processor.

In the first contribution of this chapter we show that the probability of successful re-deployment increases if applications are mapped to more VPs of a smaller size, at the cost of additional TDM slots. This maximises the probability of successful future re-deployment. To exploit this feature for fault-tolerance each application must be started with such a multi-VP mapping, meaning that extra slots are also used when no fault has occurred. This is overcome with the second contribution, which is to split a VP only when a fault occurs. Because of the lower overall resource usage this further increases the probability for successful re-deployment at the cost of storing multiple mappings at run-time and the time required for splitting the schedule. In

both contributions we exploit the DSE extension from Chapter 4 and select those mappings that have the lowest cost in TDM slots.

We evaluate the proposed strategies experimentally and show that mapping an application to more VPs indeed increases the probability of successful re-deployment, both for processors of the same type and processors of different types using resizing. This works up to a certain limit, after which the probability decreases because of the increased total cost. Furthermore we find that splitting VPs only when a fault occurs increases the probability more than splitting VPs beforehand. This is due to the smaller total VP size when no fault occurred, leaving more slots available.

When running the experiments for sample deployments with different utilisations, a lower threshold is revealed below which a re-deployment can be found with all strategies, as well a higher threshold above which no re-deployment can be found with any strategy. In between the thresholds we provide insight in the trade-off between maximising fault-tolerance and minimising cost that is offered by the different strategies. The numbers reported in this chapter cannot be generalised for other platforms and applications. In low-cost, high-performance systems however we argue any increase of the fault-tolerance that is achieved without spare processors is an improvement of significance.



6 Conclusions and future work

In Chapter 1 of this thesis we introduced two use-cases for *Real-Time (RT)* embedded systems, namely a point-to-point video surveillance system and an adaptive cruise control system. It is possible to design static versions of such systems using contemporary design flows. However, advances in functionality and technology lead to new, dynamic requirements that cause new challenges in the design flow. We re-visit the three dynamic requirements from Chapter 1 in Sections 6.1–6.3. Possible directions for future work that build upon the contributions of this thesis are presented in Section 6.4

6.1 Dynamic response to input data

Applications are increasingly dynamic because they must respond to input data. Capturing such dynamic behaviour in a static *Model-of-Computation (MoC)* leads to overly negative analysis results and over-reservation of resources. Instead we select the *Finite-State Machine Scenario-Aware Dataflow (FSM-SADF)* MoC that captures different behaviours in separate scenarios and can provide a bound on the minimum throughput of an application. The analysis with available tools that use $(max, +)$ algebra is precise and elegant, but a causality dilemma is encountered when attempting to implement this MoC in a *Programming Model (PM)*.

In Chapter 3 we proposed a PM for FSM-SADF that allows dynamic responses to input data because the scenario scheduling is performed at run-time. The application designer must create the FSM, scenario graphs and actor functions. We recognise two types of applications. Firstly, in applications with delayed scenario detection the current scenario is detected in the previous iteration and stored in a detector token. Secondly, in applications with immediate scenario detection the current scenario is detected during the current iteration in a detector actor. A causality dilemma is encountered when attempting to implement the PM for the latter.

We describe how the scenario graphs of both application types can be transformed to a scenario sequencing model in which the detector token or actor is split off in a detector scenario, which is always executed before the other scenarios. The timing behaviour of the sequencing model is identical to that of the original application, and can be expanded into a *Binding-Aware Graph (BAG)* that models the exact timing of our PM, middleware and hardware.

The sequencing model can also be merged into a scenario execution model, resulting in one graph that contains all scenarios and is suitable for execution. The detector scenario is always executed first, it contains a load-schedule actor that extends the *Rolling Static-Order (RSO)* schedule with the correct actor sequence of the

next scenario. The sequencing model and execution model together solve the causality dilemma.

We modified the existing design flow to automatically generate the sequencing model and execution model for applications with immediate scenario detection. We also provide the middleware that implements the PM on the CompSOC platform. The exact timing behaviour of the implementation was analysed and added to the BAG models. The resulting FSM-SADF PM consists of a method, design flow, middleware and analysis model that can design applications which can dynamically respond to input data.

6.2 Dynamic response to the user and environment

The set of active applications is dynamic because the system must respond to the user and changes in the environment. To deal with events at run-time it must be possible to start or stop an application at any moment in time. In this thesis we select a combination of architecture, middleware, deployment, mapping, scheduling, and timing analysis that allows to deploy the *Virtual Processors (VPs)* of an application to any processor of the correct type, provided that sufficient capacity is available.

The computation intensive timing analysis is performed at design-time for mapping actors to VPs. If the TDM budget of each VP is granted at run-time, an application is guaranteed to meet its RT constraints. This separates design-time analysis from run-time deployment and limits the size of the solution space of the resource manager. Deployment is thus reduced to a three-dimensional bin-packing problem where the applications (items) must fit on the available processors, memories and interconnect (bins). In this work we assume that the interconnect is fully connected and has sufficient bandwidth to support all possible deployments, which removes the interconnect from the equation.

Resource managers that can deploy VPs at run-time have been proposed in existing work. It is not feasible to calculate and store all possible deployments at design-time, so the resource manager must use a heuristic. The platform utilisation at the moment at which an application must be deployed cannot be predicted. In Chapter 4 we extend the *Design Space Exploration (DSE)* algorithm to trade the total VP size against the total FIFO buffer size while meeting the throughput constraint. The VP size translates to run-time processor utilisation, and the buffer size to memory footprint. We show that the throughput changes independently with the VP size and buffer size, therefore we propose an algorithm that explores both dimensions at the same time. The extended DSE allows the designer to balance the processor and memory requirements by selecting one from multiple Pareto-optimal points, allowing to increase the success of future deployment actions in dynamic sets. Systems that run a dynamic set of applications can dynamically respond to the user and environment.

6.3 Dynamic response to processor faults

Because of processor faults, there is dynamism in the set of available processors. The decreasing feature size of VLSI designs increases the power density and leads to hot spots in the processors. These cause intermittent faults on the short term and permanent faults on the long term. We propose to employ a resource manager to re-deploy all VPs from a faulty processor to unused capacity on other processors. This is similar to the regular deployment problem, except that processor capacity is more scarce after a fault.

Reserving spare processors to handle faults only allows to handle as many faults as there are spares, and is too expensive for the type of systems we consider. Using re-deployment however, it is not possible to guarantee that sufficient resources are available which means that applications can fail. In Chapter 5 we maximise the probability of successful re-deployment by mapping applications to more VPs of a smaller size at design-time, which comes at the cost of a larger total VP size even if no fault has occurred. The bin-packing heuristic has a higher probability to fit more smaller items into the bins (processors), but the higher total cost may offset this benefit. As a second contribution we propose to split and resize VPs at run-time, and possibly target them for a different processor type. Though these must be stored at run-time, the advantage is that the total VP size increases only when a fault occurs. Both contributions exploit the Pareto-optimal points that result from our DSE extension in Chapter 4, we selected the mappings with the lowest cost in terms of TDM slots.

Experiments confirm that both strategies succeed in increasing the probability of successful re-deployments. The contributions maximise the probability of successfully responding to dynamic processor faults given the VPs that must be re-deployed and the available processor capacity.

6.4 Future work

In this section we present three possible directions for future work. Firstly, the DSE may be extended to trade-off power against throughput, VP size and memory footprint to prevent temporary faults due to hot spots. We have seen that the ever-decreasing feature size of VLSI designs leads to an increased power density. With use of *Dynamic Voltage and Frequency Scaling (DVFS)* the voltage and frequency of processors may be manipulated to change the power density during run-time [23, 92].

We will explain this concept by assuming that each processor has two power settings, **low** and **high**, though in reality there will be more. At the **low** setting the average power is sufficiently low to prevent overheating of processors, but the WCET of actors is high. At the **high** setting the WCET of actors is lowered but this setting can only be sustained for a certain maximum time t_{high} before the processor overheats and faults start to occur. After scaling back to the **low** setting it takes a certain time

t_{cool} for the processor to cool down and reach its regular operating temperature again.

If the DSE is supplied with a different WCET for each power setting, it may be extended to generate a schedule with DVFS settings that is executed in sync with the actor SO schedule. In the DVFS schedule it must be assured that a processor cannot be at the **high** setting for longer than t_{high} , and that a minimum time t_{cool} elapses after switching from **low** to **high** before switching to **high** again. This will probably prevent most faults due to hot spots, but it cannot be guaranteed that no faults occur at all.

Secondly, the resource manager may perform trade-offs at run-time. In this thesis we have assumed use of a straightforward resource manager that attempts to deploy a starting application using only one mapping, and has no alternative course of action if insufficient resources are available. Instead the resource manager may select one of several mappings that were calculated during the DSE and trade throughput, processor utilisation and memory footprint depending on the available resources. Even if many deployments are possible, it may attempt to increase the success on future deployment actions by minimising the use of scarce resources.

Thirdly, we may increase performance by storing the instructions and FIFO buffers in the local memory of a processor. In this thesis we assume that each FIFO is stored in a protected central memory, and is in a valid state after a processor fault. The time required for accessing a remote FIFO is modeled in the BAG and is identical for each processor. Instead the FIFO buffers may be located in the local memory of the processor to which the VP that contains the destination actor of the channel is deployed, reducing the time required for reading the FIFO. In the case of an intra-tile channel, the source actor is mapped to that same VP and the time required for writing the FIFO is also reduced. These reduced access times will result in a higher throughput, which in turn may be translated to smaller VPs as explained in Chapter 4.

After a fault the VPs are re-deployed and require access to the same FIFO buffers, that are now located in the local memory of the faulty processor. There are two options to deal with this. The first option is to keep the buffers in that same memory, and choose a new mapping that accounts for the access times that have now increased. While this does not require migration of data it leads to larger VPs that increase the processor utilisation. On top of that, a mapping for each possible combination of buffer locations must be stored during run-time. The second option is to migrate the buffers to the local memory of the processor on which the VP is re-deployed. This does not require different mappings, but rare events such as the time required for migration cannot be modeled in the current timing analysis, so data migration may lead to a deadline violation. Further research is necessary to decide which of these two options is the most viable.

Appendices

A Use-case requirements

This appendix lists the relevant passages of the sources that back up the requirements of the use-cases introduced in Section 1.4, using the same numbering. The passages are either cited verbatim, or a screenshot is provided. The choice of corporate sources and the claims made in these (e.g. “The Best ...”) do not reflect the preferences or opinions of the author in any way.

UC-1.1 L. Snidaro et al. *Fusing Multiple Video Sensors for Surveillance* [127]:

“Real-time detection, tracking, recognition, and activity understanding of moving objects from multiple sensors represent fundamental issues to be solved in order to develop surveillance systems that are able to autonomously monitor wide and complex environments.”

“In the experiments we employed cameras able to acquire between 25 and 30 frames per second. In order to synchronize the input frames to the processing flow we used a centralized capture routine that was polling the sensors 25 times per second to retrieve new frames.”

Cisco Systems, Inc. *Video Quality of Service (QoS) Tutorial* [17]:

“Transport network SLAs for video quality


Recommended network SLA for video[4] is as follows:

- Latency $\leq 150 - 300\ ms$
- Jitter $\leq 10 - 50\ ms$
- Loss $\leq 0.5\%$

Incidentally the recommended network SLA for transporting audio are:

- Latency $\leq 150 - 300\ ms$
- Jitter $\leq 20 - 50\ ms$
- Loss $\leq 1\%$

UC-1.2 Intel Corporation. *Temperature Grades and Associated Temperature Ranges* [61]:



Overview

Device Support

RoHS Compliant

Temperature Grades and Associated Temperature Ranges

Note: For MAX® 7000AE devices, extended temperature range is defined as -40°C to 130°C. For MAX® 7000AE devices, automotive temperature range is defined as -40°C to 130°C. For Intel® Arria® 10 E devices, extended temperature range is defined as 0°C to 100°C. For more information, refer to the Intel® Arria® 10 device overview and the Intel® Arria® 10 device datasheet.

Temperature Grade	Temperature Range
Commercial	0°C to 85°C
Industrial	-40°C to 100°C
Extended	-40°C to 125°C
Military	-55°C to 125°C
Automotive	-40°C to 125°C

UC-1.3 J. R. Delaney. *The Best Medical Alert Systems of 2018* [27]:

PC

REVIEWS · BEST PICKS · HOW-TO · NEWS · SMART HOME · BUSINESS · SHOP

What are you looking for?

LGV40ThinQ · FacebookHack · 1CoolThing · Amazon · MicrosoftSurface

Subscribe

The Best Medical Alert Systems of 2018

A personal emergency response system can give you peace of mind knowing help is only a button-press away. Here's everything you need to know to find the best medical alert system for yourself or an elderly family member.

By John R. Delaney September 11, 2018 11:23AM EST

14 SHARES

PCMag reviews products independently, but we may earn affiliate commissions from buying links on this page. Terms of use.

Product	Bay Alarm Medical In-Home Medical Alert	Bay Alarm Medical Spillscend In-Car Medical Alert	GreatCall Lively Mobile	MobileHelp Classic	LifeFone At Home Landline Medical Alert System	Philips Lifeline GoSafe 2	Bay Alarm Medical Mobile GPS Help Button	LifeStation At Home Medical Alert	LifeStation Premium Mobile	Medical Guardian Classic Guardian
										
	\$19.95	\$39.95	\$24.99	\$29.95	\$24.95	\$44.95	\$27.95	\$25.95	\$25.95	\$29.95
Battery	32 hours	15 minutes of talk; 12 hours idle	36 hours	30 hours	32 hours	36 hours	72 hours	32 hours	5 days	32 hours

UC-1.4 Amazon.com, Inc. *Best Sellers in Remote Home Monitoring Systems* [5]:

Amazon Best Sellers

Our most popular products based on sales. Updated hourly.

Best Sellers in Remote Home Monitoring Systems

<p>#1</p>  <p>Zmodo Wireless Security Camera System (2 Pack) , Smart Home HD Indoor... ★★★★☆ 6,176 \$64.23 ✓prime</p>	<p>#2</p>  <p>Yi 4pc Home Camera, Wi-Fi IP Security Surveillance System with Night Vision... ★★★★☆ 1,059 \$95.99 ✓prime</p>	<p>#3</p>  <p>Yi 4pc Home Camera, 1080p Wireless IP Security Surveillance System with Night... ★★★★☆ 155 \$119.99 ✓prime</p>
<p>#4</p>  <p>Nest Security Camera, Keep An Eye On What Matters to You, From Anywhere... ★★★★☆ 5,926 \$165.00 ✓prime</p>	<p>#5</p>  <p>Arlo by NETGEAR Security System - 4 Wire-Free HD Cameras Indoor/Outdoor... ★★★★☆ 415 \$299.99 ✓prime</p>	<p>#6</p>  <p>Zmodo EZCam Wireless Two-Way Audio Smart HD IP Home Security Camera... ★★★★☆ 6,176 \$29.99 ✓prime</p>

A. Colon and W. Greenwald. *The Best Home Security Cameras of 2018* [18]:

PC REVIEWS - BEST PICKS - HOW-TO - NEWS - SMART HOME - BUSINESS - SHOP -

What are you looking for?

LG40ThinQ FacebookHack 1CoolThing Amazon MicrosoftSurface

Subscribe:

Reviews Consumer Electronics Smart Home Home Security **Home Security Cameras**

The Best Home Security Cameras of 2018

If you want to keep an eye on what's happening at home when you're not there, but you don't want to invest in a full-fledged home security system, a Wi-Fi-connected camera is worth a look. These are the best home security cameras in our testing.

By Alex Colon and Will Greenwald September 11, 2018 11:23AM EST

289 SHARES

PCMag reviews products independently, but we may earn affiliate commissions from buying links on this page. [Terms of use.](#)

Product	Netgear Arlo Pro 2	D-Link Full HD Wi-Fi Camera DCS-8300LH	iSmartAlarm Camera Keep Pro	Tend Secure Lynx Indoor	Wyze Cam V2	Nest Cam IQ	Nest Cam IQ Outdoor	Netgear Arlo Go	Reolink Argus 2	Ring Spotlight Cam Battery
										
	\$429.99	\$89.94	Best Price	\$48.94	\$19.99	\$299.00	\$349.00	\$429.99	\$129.99	\$199.00

- UC-1.5 Cisco Systems, Inc. *Stream Manager Video Surveillance Solutions Reference Network Design* [16]:

“By using multicast protocols, the hosts that want to receive traffic from a multicast group can join and leave the group dynamically. Hosts can be members of more than one group and must explicitly join a group before receiving the desired content.”

“Dynamic streams can be created in a number of ways. For example, an operator with a CCTV monitor and keyboard that are attached to a decoder can cause dynamic streams by frequently switching the monitor to various cameras across the network, some of which could be located over low-bandwidth WAN connections.”

“The most challenging scenario for which to engineer is the Cisco Stream Manager Client Viewing Module installed on a laptop computer that can rove anywhere in a WAN environment with a wide-open multicast policy. This device could call up to 10 concurrent streams to any location in the network using a standard screen layout.”

- UC-1.6 Wikipedia, *Video compression picture types* [152] “In the field of video compression a video frame is compressed using different algorithms with different advantages and disadvantages, centered mainly around amount of data compression. These different algorithms for video frames are called picture types or frame types.”

- UC-1.7 A. Doblander et al. *Improving fault-tolerance in intelligent video surveillance by monitoring, diagnosis and dynamic reconfiguration* [30]:

“In this paper, we present an approach for improving fault-tolerance and service availability in intelligent video surveillance (IVS) systems. A typical IVS system consists of various intelligent video sensors that combine image sensing with video analysis and network streaming. System monitoring and fault diagnosis followed by appropriate dynamic system reconfiguration mitigate effects of faults and therefore enhance the system’s fault-tolerance.” “A key requirement for dynamically reacting to faults and failures is to detect abnormal behavior and isolate affected system components.”

- UC-2.1 R.A.P.M. van den Bleek. *Design of a Hybrid Adaptive Cruise Control Stop-&-Go System* [103]:

“This controller can now be applied off-line enabling fast-sampling MPC for the adaptive cruise control Stop-&-Go application. For this application, the sample frequency is 1000 Hz.”

- UC-2.2 Intel Corporation. *Temperature Grades and Associated Temperature Ranges* [61]: See UC-1.2.

UC-2.3 B. Motruk et al. *Power Monitoring for Mixed-Criticality on a Many-Core Platform* [87]:

“Mixed-critical applications on a many-core platform have to be sufficiently independent to be certified separately. This does not only include independence in terms of time and space, but also in terms of power consumption as the available energy for a many-core system has to be shared by all running applications. Increased power consumption of one application may reduce the available energy for other applications or the reliability and lifetime of the complete chip.”

“A chip’s overall power budget is defined at design time based on packaging and cooling, battery capacity (if applicable), and environmental conditions. Meeting this budget is especially important for battery-powered devices as increased power consumption of one application would reduce the available energy for all other running applications. Moreover, high local power consumption could lead to hotspots influencing neighboring components, or may reduce the entire chip’s lifetime and reliability.”

UC-2.4 D. Mohr et al. *The road to 2020 and beyond: What’s driving the global automotive industry?* [81]:

“Key challenge 1: Complexity and cost pressure. There will be more platform sharing and more modular systems. At the same time, regulatory pressures will tighten, and prices in established markets are likely to be flat.”

UC-2.5 B. Akesson et al. *Composability and predictability for independent application development, verification, and execution* [1]:

“To reduce cost, platform resources, such as processors, hardware accelerators, interconnect, and memories, are shared between applications. [...] The second problem is that verification of a use-case cannot begin until all applications it comprises are available. Timely completion of the verification process hence depends on the availability of all applications, which may be developed by different teams inside the company, or by independent software vendors. The last problem is that use-case verification becomes a circular process that must be repeated if an application is added, removed, or modified.”

UC-2.6 N. Louw. *TECH: We test three adaptive cruise control systems* [79]:

“Continental manufactures not only tyres, but is one of the main Adaptive Cruise Control (ACC) system suppliers, with more than 30 million of its radar sensors sold to automotive OEMs. We recently chatted to Norbert Hammerschmidt, Continental’s head of programme management for radar, about the technology. According to him, because it is unaffected by adverse weather conditions (including thick fog), the radar sensor is the most common component used in vehicles fitted with ACC.”

UC-2.7 G. Georgakos et al. *Reliability Challenges for Electric Vehicles: From Devices to Architecture and Systems Software* [40]:

“To make matters worse, the electronics in a car are exposed to harsh conditions, extreme temperature variations, and often, strong electromagnetic fields, which further aggravates the reliability problem. Further, the electronics in subsystems like battery monitoring and management in electric vehicles are always “on” for the entire lifetime of the car, which is in the range of 10-15 years and sometimes even more. This makes issues like aging an important concern.”

“In automotive, IC manufacturers need to guarantee specified functionality for 2-5 years operating time depending on application and temperature range and up to 15 years in standby mode, and desire them for even longer time to avoid reputational risks. At the same time, their ICs are sometimes used in very harsh conditions (e.g. temperatures up to 150°C and for special purposes also up to 175°C at reduced life times), and almost continuous (e.g. taxis being used in multiple shifts; battery management electronics in electrical vehicles) which amplifies the aging.”

B Algorithms for fault-tolerant deployment

Algorithm 4 Analysis of the probability on successful re-deployment.
This is an exact copy of Algorithm 4 on page 97, repeated here for convenience.

```
1: function ANALYSE_RE-DEPLOYMENTS(mappings.csv, architecture.csv)
2:   processors = parse_architecture(architecture.csv)
3:   mappings, vp_archs = parse_mappings(mappings.csv)
4:   appSets = create_application_sets(mappings)
5:   deployments = generate_deployments(processors, vp_archs, mappings,
   appSets)
6:   return generate_re-deployments(processors, vp_archs, deployments,
   mappings)
7: end function
```

Algorithm 5 Generate a list of all possible deployments for each type of VP architecture. This function is called in Algorithm 4.

```
1: function GENERATE_DEPLOYMENTS(processors, vp_archs, mappings, appSet)
2:   deployments = []
3:   for vp_arch in vp_archs do
4:     deployments.append(vp_arch)
5:     for app_set in app_sets do
6:       vp_list = []
7:       for app in app_set do
8:         vp_list.append(app.vps)
9:       end for
10:      deployment = [processors.length][]
11:      deploy_vps(processors, deployments[vp_arch], vp_list, deployment)
12:    end for
13:  end for
14:  return deployments
15: end function
```

Algorithm 6 Calculate the probability of successful re-deployment for each processor failure, for each type of VP architecture. This function is called in Algorithm 4.

```

1: function GENERATE_RE-DEPLOYMENTS(processors, vp_archs, deployments,
   mappings)
2:   probabilities = [vp_archs.length]
3:   for vp_arch in vp_archs do
4:     success = 0
5:     for deployment in deployments[vp_arch] do
6:       for faulty_proc in processors do
7:         replacement_vps = get_replacement_vps(deployment[faulty_proc],
   mappings)
8:         delete deployment[faulty_proc]
9:         re-deployments = []
10:        deploy_vps(processors - faulty_proc, re-deployments,
   replacement_vps, deployment)
11:        if not re-deployments.empty then
12:          success += 1
13:        end if
14:      end for
15:    end for
16:    probabilities[vp_arch] = success / (deployments[vp_arch].length *
   processors.length)
17:  end for
18:  return probabilities
19: end function

```

Algorithm 7 Recursive function to generate all deployment permutations, avoiding equivalents. This function is called in Algorithms 5 and 6.

```

1: function DEPLOY_VPS(processors, deployments, vp_list, deployment)
2:   if not vp_list.empty then
3:     current_type = vp_list[0].procType
4:     total_procs = total_procs_of_type(processors, current_type)
5:     free_procs = free_procs_of_type(processors, current_type)
6:     used_procs = total_procs - free_procs
7:     for proc in used_procs + 1 do
8:       if sufficient_space_on_proc(proc, vp_list[0]) then
9:         new_deployment = deployment.copy()
10:        new_vp_list = vp_list.copy
11:        new_deployment[proc].append(new_vp_list[0])
12:        delete new_vp_list[0]
13:        deploy_vps(deployments, new_vp_list, new_deployment)
14:      end if
15:    end for
16:  else
17:    deployments.append(deployment)
18:  end if
19: end function

```

Bibliography

- [1] B. Akesson, A. Molnos, A. Hansson, J. A. Angelo, and K. Goossens. Composability and predictability for independent application development, verification, and execution. *Multi-processor System-on-Chip - Hardware Design and Tool Integration, Circuits and Systems*, pages chapter 2, pages 25–56, November 2010. (Cited on pages 3 and 115.)
- [2] B. Akesson, S. Stuijk, A. Molnos, M. Koedam, R. Stefan, A. Nelson, A. B. Nejad, and K. Goossens. Virtual Platform for Mixed-Time Criticality Applications: the CoMPSoC Architecture and SDF3 Design Flow. In *QVVP*, 2012. (Cited on pages 18, 20, 21, and 59.)
- [3] H. Alizadeh Ara, M. Geilen, T. Basten, A. Behrouzian, M. Hendriks, and D. Goswami. Tight temporal bounds for dataflow applications mapped onto shared resources. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–8, May 2016. doi: 10.1109/SIES.2016.7509444. (Cited on pages 9, 25, and 88.)
- [4] H. Alizadeh Ara, M. Geilen, A. Behrouzian, and T. Basten. Throughput-buffering trade-off analysis for scenario-aware dataflow models. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems, RTNS '18*, pages 265–275, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6463-8. doi: 10.1145/3273905.3273921. URL <http://doi.acm.org/10.1145/3273905.3273921>. (Cited on pages 70, 75, and 82.)
- [5] Amazon.com, Inc. Best sellers in remote home monitoring systems, 2018. URL <https://www.amazon.com/Best-Sellers-Electronics-Remote-Home-Monitoring-Systems/zgbs/electronics/7161093011>. (Cited on pages 1 and 113.)
- [6] S.-H. Attarzadeh-Niaki, E. Altinel, M. Koedam, A. Molnos, I. Sander, and K. Goossens. *A Composable and Predictable MPSoC Design Flow for Multiple Real-Time Applications*, pages 157–174. Springer International Publishing, Cham, 2017. ISBN 978-3-319-47307-9. doi: 10.1007/978-3-319-47307-9_6. URL https://doi.org/10.1007/978-3-319-47307-9_6. (Cited on page 63.)
- [7] A. Avizienis, J. C. Laprie, and B. Randell. Fundamental concepts of dependability. Rapport LAAS 01145, LAAS, Apr 2000. (Cited on pages 5 and 100.)
- [8] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling Real-Time Mixed-Criticality Jobs. *Com-*

- puters, *IEEE Transactions on*, 61(8):1140 –1152, aug. 2012. ISSN 0018-9340. doi: 10.1109/TC.2011.142. (Cited on page 9.)
- [9] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K. Arzen, V. Romero, and C. Scordino. Resource management on multicore systems: The ACTORS approach. *Micro, IEEE*, 31(3):72–81, May 2011. ISSN 0272-1732. doi: 10.1109/MM.2011.1. (Cited on page 14.)
- [10] C. Bolchini, M. Carminati, A. Miele, A. Das, A. Kumar, and B. Veeravalli. Run-time mapping for reliable many-cores based on energy/performance trade-offs. In *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pages 58–64, Oct 2013. doi: 10.1109/DFT.2013.6653583. (Cited on page 101.)
- [11] C. Breshears. *The Art of Concurrency: A Thread Monkey’s Guide to Writing Parallel Applications*. O’Reilly Media, Inc., 1st edition, 2009. ISBN 0596521537. (Cited on page 7.)
- [12] D. Broman, M. Zimmer, Y. Kim, H. Kim, J. Cai, A. Shrivastava, S. A. Edwards, and E. A. Lee. Precision timed infrastructure: Design challenges. In *In the Proceedings of the Electronic System Level Synthesis Conference (ESLsyn)*, Austin, Texas, USA, May 2013. URL <http://chess.eecs.berkeley.edu/pubs/993.html>. (Cited on page 63.)
- [13] J. T. Buck. *Scheduling Dynamic Dataflow Graphs With Bounded Memory Using The Token Flow Model*. PhD thesis, University of California at Berkeley, 1993. (Cited on page 52.)
- [14] E. Carvalho, N. Calazans, and F. Moraes. Heuristics for Dynamic Task Mapping in NoC-based Heterogeneous MPSoCs. pages 34 –40, May 2007. doi: 10.1109/RSP.2007.26. (Cited on page 68.)
- [15] C. L. Chou and R. Marculescu. FARM: Fault-aware resource management in NoC-based multiprocessor platforms. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011. doi: 10.1109/DATE.2011.5763113. (Cited on page 101.)
- [16] Cisco Systems, Inc. Stream manager video surveillance solutions reference network design. Technical report, 2007. URL https://www.cisco.com/c/dam/en/us/td/docs/solutions/Enterprise/Video/Stream_Manager_Video_Surveillance_SRND.pdf. (Cited on pages 1 and 114.)
- [17] Cisco Systems, Inc. Video quality of service (QOS) tutorial, 2017. URL <https://www.cisco.com/c/en/us/support/docs/quality-of-service-qos/qos-video/212134-Video-Quality-of-Service-QOS-Tutorial.html>. (Cited on pages 1 and 111.)

-
- [18] A. Colon and W. Greenwald. The best home security cameras of 2018, September 2018. URL <https://www.pcmag.com/article2/0,2817,2475954,00.asp>. (Cited on pages 1 and 113.)
- [19] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. M. Burguière, J. Reineke, B. Triquet, and R. Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of ERTSS 2010*, 2010. (Cited on page 7.)
- [20] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Modeling static-order schedules in synchronous dataflow graphs. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pages 775–780, March 2012. doi: 10.1109/DATE.2012.6176588. (Cited on pages 30 and 60.)
- [21] A. Das and A. Kumar. Fault-aware task re-mapping for throughput constrained multimedia applications on NoC-based MPSoCs. In *2012 23rd IEEE International Symposium on Rapid System Prototyping (RSP)*, pages 149–155, Oct 2012. doi: 10.1109/RSP.2012.6380704. (Cited on page 101.)
- [22] A. Das, A. Kumar, and B. Veeravalli. Reliability-driven task mapping for lifetime extension of Networks-on-Chip based multiprocessor systems. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 689–694, March 2013. doi: 10.7873/DATE.2013.149. (Cited on page 101.)
- [23] A. Das, A. Kumar, and B. Veeravalli. Temperature aware energy-reliability trade-offs for mapping of throughput-constrained applications on multimedia mpsoCs. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014. doi: 10.7873/DATE.2014.115. (Cited on pages 101 and 107.)
- [24] A. Das, A. Kumar, B. Veeravalli, C. Bolchini, and A. Miele. Combined DVFS and mapping exploration for lifetime and soft-error susceptibility improvement in MPSoCs. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014. doi: 10.7873/DATE.2014.074. (Cited on page 101.)
- [25] A. Das, R. A. Shafik, G. V. Merrett, B. M. Al-Hashimi, A. Kumar, and B. Veeravalli. Reinforcement learning-based inter- and intra-application thermal optimization for lifetime improvement of multicore systems. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014. doi: 10.1145/2593069.2593199. (Cited on page 101.)
- [26] A. Dasdan and R. K. Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, Oct 1998. ISSN 0278-0070. doi: 10.1109/43.728912. (Cited on page 25.)

- [27] J. R. Delaney. The best medical alert systems of 2018, September 2018. URL <https://www.pcmag.com/article/356981/the-best-medical-alert-systems>. (Cited on pages 1 and 112.)
- [28] O. Derin, D. Kabakci, and L. Fiorin. Online task remapping strategies for fault-tolerant Network-on-Chip multiprocessors. In *Proceedings of the Fifth ACM/IEEE International Symposium*, pages 129–136, May 2011. doi: 10.1145/1999946.1999967. (Cited on page 101.)
- [29] P. I. Diallo, S. Attarzadeh-Niaki, F. Robino, I. Sander, J. Champeau, and J. Oberg. A formal, model-driven design flow for system simulation and multi-core implementation. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, June 2015. doi: 10.1109/SIES.2015.7185067. (Cited on page 63.)
- [30] A. Doblander, A. Maier, B. Rinner, and H. Schwabach. Improving fault-tolerance in intelligent video surveillance by monitoring, diagnosis and dynamic reconfiguration. In *Third International Workshop on Intelligent Solutions in Embedded Systems, 2005.*, pages 194–201, May 2005. doi: 10.1109/WISES.2005.1438728. (Cited on pages 2 and 114.)
- [31] E. Dubrova. *Fault Tolerant Design*. Springer, 2013. (Cited on pages 5, 10, 89, 90, and 100.)
- [32] S. Edwards and E. Lee. The case for the precision timed (PRET) machine. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 264–265, June 2007. (Cited on page 63.)
- [33] J. Eker and J. W. Janneck. CAL language report: Specification of the CAL actor language. Technical report, University of California at Berkeley, 2003. (Cited on page 63.)
- [34] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002. (Cited on page 7.)
- [35] M. Fernández. *Models of Computation*. Springer, 2009. (Cited on page 6.)
- [36] M. Geilen. Synchronous dataflow scenarios. *ACM Trans. Embed. Comput. Syst.*, 10(2):16:1–16:31, Jan. 2011. ISSN 1539-9087. doi: 10.1145/1880050.1880052. URL <http://doi.acm.org/10.1145/1880050.1880052>. (Cited on pages 25 and 48.)
- [37] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES/ISSS '10*, pages 125–134, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-905-3. doi: 10.1145/1878961.1878985. URL <http://doi.acm.org/10.1145/1878961.1878985>. (Cited on pages 25, 26, and 69.)

-
- [38] M. Geilen, S. Stuijk, and T. Basten. Predictable dynamic embedded data processing. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 320–327, July 2012. doi: 10.1109/SAMOS.2012.6404194. (Cited on page 21.)
- [39] M. Geilen, M. Skelin, R. van Kampenhout, H. Alizadeh Ara, T. Basten, S. Stuijk, and K. Goossens. *System Scenario-based Design Principles and Applications*, chapter Scenarios in Dataflow Modelling and Analysis. Springer, 2019. ISBN 978-3-030-20342-9.
- [40] G. Georgakos, U. Schlichtmann, R. Schneider, and S. Chakraborty. Reliability challenges for electric vehicles: From devices to architecture and systems software. In *Proceedings of the 50th Annual Design Automation Conference, DAC '13*, pages 98:1–98:9, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2071-9. doi: 10.1145/2463209.2488855. URL <http://doi.acm.org/10.1145/2463209.2488855>. (Cited on pages 4 and 116.)
- [41] 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 *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 25–36, June 2006. doi: 10.1109/ACSD.2006.33. (Cited on page 25.)
- [42] S. V. Gheorghita, S. Stuijk, T. Basten, and H. Corporaal. Automatic scenario detection for improved WCET estimation. In *Proceedings of the 42Nd Annual Design Automation Conference, DAC '05*, pages 101–104, New York, NY, USA, 2005. ACM. ISBN 1-59593-058-2. doi: 10.1145/1065579.1065610. URL <http://doi.acm.org/10.1145/1065579.1065610>. (Cited on page 26.)
- [43] M. D. Gomony. *Scalable and bandwidth-efficient memory subsystem design for real-time systems*. PhD thesis, Eindhoven University of Technology, 2015. (Cited on page 61.)
- [44] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad, A. Nelson, and S. Sinha. Virtual Execution Platforms for Mixed-time-criticality Systems: The CompSOC Architecture and Design Flow. *SIGBED Rev.*, 10(3):23–34, Oct. 2013. ISSN 1551-3688. doi: 10.1145/2544350.2544353. URL <http://doi.acm.org/10.1145/2544350.2544353>. (Cited on pages 21 and 33.)
- [45] K. Goossens, M. Koedam, A. Nelson, S. Sinha, S. Goossens, Y. Li, G. Breaban, R. van Kampenhout, R. Tavakoli Najafabadi, J. Valencia, H. Ahmadi Balef, B. Akesson, S. Stuijk, M. Geilen, D. Goswami, and M. Nabi Najafabadi. *Handbook of hardware/software codesign*, chapter NoC-based multiprocessor architecture for mixed-time-criticality applications, pages 491–530. Springer, 11 2017. ISBN 978-94-017-7266-2. (Cited on pages 6, 7, 11, 12, 18, 21, 30, 32, 33, 61, 68, and 100.)

-
- [46] S. Goossens, B. Akesson, M. Koedam, A. B. Nejad, A. Nelson, and K. Goossens. The CompSOC design flow for virtual execution platforms. In *Proceedings of the 10th FPGAWorld Conference, FPGAWorld '13*, pages 7:1–7:6, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2496-0. doi: 10.1145/2513683.2513690. URL <http://doi.acm.org/10.1145/2513683.2513690>. (Cited on page 21.)
 - [47] S. Goossens, K. Chandrasekar, B. Akesson, and K. Goossens. *Memory Controllers for Mixed-Time-Criticality Systems*. Springer, 2016. (Cited on pages 59 and 61.)
 - [48] A. Hansson, K. Goossens, M. Bekooij, and J. Huiskens. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):2:1–2:24, Jan. 2009. ISSN 1084-4309. doi: 10.1145/1455229.1455231. URL <http://doi.acm.org/10.1145/1455229.1455231>. (Cited on pages 21 and 33.)
 - [49] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *IET Computers Digital Techniques*, 3(5):398–412, Sep. 2009. ISSN 1751-8601. doi: 10.1049/iet-cdt.2008.0093. (Cited on page 59.)
 - [50] A. Hansson, M. Ekerhult, A. Molnos, A. Milutinovic, A. Nelson, J. Ambrose, and K. Goossens. Design and implementation of an operating system for composable processor sharing. *Microprocessors and Microsystems*, 35(2):246 – 260, 2011. ISSN 0141-9331. doi: <http://dx.doi.org/10.1016/j.micpro.2010.08.008>. URL <http://www.sciencedirect.com/science/article/pii/S0141933110000608>. Special issue on Network-on-Chip Architectures and Design Methodologies. (Cited on pages 32 and 33.)
 - [51] S. K. S. Hari, M. L. Li, P. Ramachandran, B. Choi, and S. V. Adve. mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 122–132, Dec 2009. (Cited on page 90.)
 - [52] G. Heiser. The role of virtualization in embedded systems. In *IIES '08: Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pages 11–16, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-126-2. doi: <http://doi.acm.org/10.1145/1435458.1435461>. (Cited on page 32.)
 - [53] J. Henkel, S. Pagani, H. Khdr, F. Kriebel, S. Rehman, and M. Shafique. Towards performance and reliability-efficient computing in the dark silicon era. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2016. (Cited on pages 5 and 85.)
 - [54] 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. ISSN 0018-9219. doi: 10.1109/JPROC.2002.805825. (Cited on pages 7 and 63.)

-
- [55] T. A. Henzinger and J. Sifakis. *M 2006: Formal Methods*, chapter The Embedded Systems Design Challenge, pages 1–15. Springer Berlin Heidelberg, 2006. (Cited on page 7.)
- [56] R. Hilbrich and J. R. van Kampenhout. Dynamic reconfiguration in NoC-based MPSoCs in the avionics domain. In *IWMSE '10: Proceedings of the 3rd International Workshop on Multicore Software Engineering*, pages 56–57, New York, NY, USA, 2010. ACM.
- [57] R. Hilbrich and J. R. van Kampenhout. Partitioning and Task Transfer on NoC-based Many-Core Processors in the Avionics Domain. In *4. Workshop: Entwicklung zuverlässiger Software-Systeme (Stuttgart, Deutschland) and Journal "Softwaretechnikrends"*, 2011.
- [58] R. Hilbrich, J. R. van Kampenhout, M. Daun, D. T. Weyer, and D. Sojer. *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology*, chapter Modeling Quality Aspects: Real-Time, pages 119–128. Springer, 2012. ISBN 978-3-642-34614-9.
- [59] R. Hilbrich, J. R. van Kampenhout, and H.-J. Goltz. Modellbasierte generierung statischer schedules fuer sicherheitskritische, eingebettete systeme mit multicore-prozessoren und harten echtzeitanforderungen. In *Herausforderungen durch Echtzeitbetrieb*, Informatik aktuell, pages 29–38. Springer, 2012.
- [60] P. K. F. Hölzenspies, J. L. Hurink, J. Kuper, and G. J. M. Smit. Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (MPSoC). In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 212–217, New York, NY, USA, 2008. ACM. ISBN 978-3-9810801-3-1. doi: <http://doi.acm.org/10.1145/1403375.1403427>. (Cited on page 68.)
- [61] Intel Corporation. Temperature grades and associated temperature ranges, 2018. URL <https://www.intel.com/content/www/us/en/products/programmable/temperature.html>. (Cited on pages 1, 3, 112, and 114.)
- [62] R. Jordans, F. Siyoum, S. Stuijk, A. Kumar, and H. Corporaal. An automated flow to map throughput constrained applications to a MPSoC. In P. Lucas, L. Thiele, B. Triquet, T. Ungerer, and R. Wilhelm, editors, *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *OpenAccess Series in Informatics (OASICS)*, pages 47–58, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-28-6. doi: <http://dx.doi.org/10.4230/OASICS.PPES.2011.47>. URL <http://drops.dagstuhl.de/opus/volltexte/2011/3081>. (Cited on page 21.)
- [63] G. Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974. (Cited on page 18.)

- [64] U. Kanade. Performance of work conserving schedulers and scheduling of some synchronous dataflow graphs. In *Proceedings. Tenth International Conference on Parallel and Distributed Systems, 2004. ICPADS 2004.*, pages 521–529, July 2004. doi: 10.1109/ICPADS.2004.1316134. (Cited on page 25.)
- [65] S.-h. Kang, H. Yang, S. Kim, I. Bacivarov, S. Ha, and L. Thiele. Static mapping of mixed-critical applications for fault-tolerant MPSoCs. In *Proceedings of the 51st Annual Design Automation Conference, DAC '14*, pages 31:1–31:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2730-5. doi: 10.1145/2593069.2593221. URL <http://doi.acm.org/10.1145/2593069.2593221>. (Cited on pages 68, 90, and 102.)
- [66] J.-P. Katoen and H. Wu. Probabilistic model checking for uncertain scenario-aware data flow. *ACM Trans. Des. Autom. Electron. Syst.*, 22(1):15:1–15:27, Sept. 2016. ISSN 1084-4309. doi: 10.1145/2914788. URL <http://doi.acm.org/10.1145/2914788>. (Cited on page 25.)
- [67] P. N. Khanh, A. K. Singh, A. Kumar, and K. M. M. Aung. Incorporating energy and throughput awareness in design space exploration and run-time mapping for heterogeneous MPSoCs. In *2013 Euromicro Conference on Digital System Design*, pages 513–521, Sep. 2013. doi: 10.1109/DSD.2013.61. (Cited on page 82.)
- [68] H. Kopetz. *Real-Time Systems*. Springer US, 2011. (Cited on pages 6, 25, and 63.)
- [69] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, Jan 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2002.805821. (Cited on page 63.)
- [70] H. Kopetz, C. El Salloum, B. Huber, R. Obermaisser, and C. Paukovits. Composability in the time-triggered System-on-Chip architecture. In *SOC Conference, 2008 IEEE International*, pages 87–90, Sept 2008. doi: 10.1109/SOCC.2008.4641485. (Cited on page 63.)
- [71] M. Krstic, E. Grass, F. K. Gürkaynak, and P. Vivet. Globally asynchronous, locally synchronous circuits: Overview and outlook. *IEEE Design Test of Computers*, 24(5):430–441, Sept 2007. ISSN 0740-7475. doi: 10.1109/MDT.2007.164. (Cited on pages 29 and 70.)
- [72] A. Kumar, S. Fernando, Y. Ha, B. Mesman, and H. Corporaal. Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA. *ACM Trans. Des. Autom. Electron. Syst.*, 13(3):40:1–40:27, July 2008. ISSN 1084-4309. doi: 10.1145/1367045.1367049. URL <http://doi.acm.org/10.1145/1367045.1367049>. (Cited on page 64.)

-
- [73] C. Lee, H. Kim, H.-w. Park, S. Kim, H. Oh, and S. Ha. A task remapping technique for reliable multi-core embedded systems. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Code-sign and System Synthesis*, CODES/ISSS '10, pages 307–316, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-905-3. doi: 10.1145/1878961.1879014. URL <http://doi.acm.org/10.1145/1878961.1879014>. (Cited on pages 68 and 101.)
- [74] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on*, C-36(1): 24–35, Jan 1987. ISSN 0018-9340. doi: 10.1109/TC.1987.5009446. (Cited on pages 6 and 24.)
- [75] A. Lele, O. Moreira, and P. J. Cuijpers. A new data flow analysis model for TDM. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 237–246, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1425-1. doi: 10.1145/2380356.2380399. URL <http://doi.acm.org/10.1145/2380356.2380399>. (Cited on pages 25 and 70.)
- [76] A. Lele, O. Moreira, K. Butala, P. J. L. Cuijpers, and K. v. Berkel. Cyclo-static data flow model for TDM. In *2014 14th International Conference on Application of Concurrency to System Design*, pages 82–91, June 2014. doi: 10.1109/ACSD.2014.17. (Cited on page 25.)
- [77] A. Lele, O. Moreira, and K. van Berkel. FP-scheduling for mode-controlled dataflow: A case study. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1257–1260, March 2015. doi: 10.7873/DATE.2015.0938. (Cited on page 65.)
- [78] A. Lele, O. Moreira, P. J. Cuijpers, and K. van Berkel. Response modeling runtime schedulers for timing analysis of self-timed dataflow graphs. *Journal of Systems Architecture*, 65:15 – 29, 2016. ISSN 1383-7621. doi: <https://doi.org/10.1016/j.sysarc.2016.02.003>. URL <http://www.sciencedirect.com/science/article/pii/S1383762116000242>. (Cited on page 70.)
- [79] N. Louw. TECH: We test three adaptive cruise control systems, May 2018. URL <http://www.carmag.co.za/technical/tech-we-test-three-adaptive-cruise-control-systems/>. (Cited on pages 4 and 115.)
- [80] A. Merkel and F. Bellosa. Balancing power consumption in multiprocessor systems. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 403–414, New York, NY, USA, 2006. ACM. ISBN 1-59593-322-0. doi: <http://doi.acm.org/10.1145/1217935.1217974>. (Cited on page 5.)
- [81] D. Mohr, N. Mueller, A. Krieg, P. Gao, H.-W. Kaas, A. Krieger, and R. Hensley. The road to 2020 and beyond: What's driving the global automotive industry? Technical report, 2013. (Cited on pages 3 and 115.)

- [82] O. Moreira and M. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007(1), 2007. ISSN 1687-6180. doi: 10.1155/2007/83710. URL <http://asp.eurasipjournals.com/content/2007/1/083710>. (Cited on pages 14 and 68.)
- [83] O. Moreira and H. Corporaal. *Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor*. Springer, 2014. (Cited on pages 9, 25, 39, 64, 65, and 82.)
- [84] O. Moreira, J. D. Mol, M. Bekooij, and J. van Meerbergen. Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *11th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 332–341, March 2005. doi: 10.1109/RTAS.2005.33. (Cited on page 102.)
- [85] O. Moreira, J. J.-D. Mol, and M. Bekooij. Online resource management in a multiprocessor with a Network-on-Chip. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1557–1564, New York, NY, USA, 2007. ACM. ISBN 1-59593-480-4. doi: <http://doi.acm.org/10.1145/1244002.1244335>. (Cited on pages 14, 82, 87, and 102.)
- [86] O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software, EMSOFT '07*, pages 57–66, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-825-1. doi: 10.1145/1289927.1289941. URL <http://doi.acm.org/10.1145/1289927.1289941>. (Cited on pages 18 and 20.)
- [87] B. Motruk, J. Diemer, R. Buchty, and M. Berekovic. Power monitoring for mixed-criticality on a many-core platform. In H. Kubátová, C. Hochberger, M. Daněk, and B. Sick, editors, *Architecture of Computing Systems ARCS*, pages 13–24, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-36424-2. (Cited on pages 3 and 115.)
- [88] H. Mushtaq, Z. Al-Ars, and K. Bertels. Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. In *IDT 2011*, 2011. (Cited on pages 5 and 100.)
- [89] A. B. Nejad. *Composable Virtual Platforms for Mixed-Criticality Embedded Systems*. PhD thesis, Delft University of Technology, 2014. (Cited on pages 18, 31, 33, and 63.)
- [90] A. B. Nejad, A. Molnos, and K. Goossens. A unified execution model for data-driven applications on a composable MPSoC. In *Proceedings of the 2011 14th Euromicro Conference on Digital System Design, DSD '11*, pages 818–822, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4494-6. doi: 10.1109/DSD.2011.110. URL <http://dx.doi.org/10.1109/DSD.2011.110>.

-
- [91] A. B. Nejad, A. Molnos, and K. Goossens. A unified execution model for multiple computation models of streaming applications on a composable MPSoC. In *Elsevier Journal of Systems Architecture*, 2013. (Cited on pages 31 and 33.)
- [92] A. Nelson. *Composable and Predictable Power Management*. PhD thesis, Delft University of Technology, 2014. (Cited on pages 16, 30, 32, 33, 59, 60, 63, 70, and 107.)
- [93] A. Nelson, A. B. Nejad, A. Molnos, M. Koedam, and K. Goossens. CoMik: A Predictable and Cycle-Accurately Composable Real-Time Microkernel. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2014. (Cited on pages 16, 30, and 32.)
- [94] A. Nelson, K. Goossens, and B. Akesson. Dataflow formalisation of real-time streaming applications on a composable and predictable multi-processor SOC. *J. Syst. Archit.*, 61(9):435–448, Oct. 2015. ISSN 1383-7621. doi: 10.1016/j.sysarc.2015.04.001. URL <http://dx.doi.org/10.1016/j.sysarc.2015.04.001>. (Cited on pages 30, 31, 59, and 60.)
- [95] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. Peset Llopis, and P. Lippens. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems*, 7(3):233–270, Oct 2002. ISSN 1572-8080. doi: 10.1023/A:1019782306621. URL <https://doi.org/10.1023/A:1019782306621>. (Cited on page 33.)
- [96] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zisulescu, and E. Deprettere. Daedalus: Toward composable multimedia MP-SoC design. In *2008 45th ACM/IEEE Design Automation Conference*, pages 574–579, June 2008. doi: 10.1145/1391469.1391615. (Cited on page 63.)
- [97] L. Osinski, T. Langer, and J. Mottok. A survey of fault tolerance approaches on different architecture levels. In *ARCS 2017; 30th International Conference on Architecture of Computing Systems*, pages 1–9, April 2017. (Cited on page 100.)
- [98] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112, Feb 2006. ISSN 0018-9340. doi: 10.1109/TC.2006.16. (Cited on page 19.)
- [99] R. Pop and S. Kumar. A Survey of Techniques for Mapping and Scheduling Applications to Network on Chip Systems. Technical Report ISSN 1404 – 0018, Embedded Systems Group, Department of Electronics and Computer Engineering, Jönköping University, 2004. (Cited on page 7.)
- [100] W. Quan and A. D. Pimentel. A scenario-based run-time task mapping algorithm for MPSoCs. In *DAC. ACM*, 2013. ISBN 978-1-4503-2071-9. (Cited on page 101.)

- [101] W. Quan and A. D. Pimentel. A hybrid task mapping algorithm for heterogeneous MPSoCs. *ACM Trans. Embed. Comput. Syst.*, 14(1):14:1–14:25, Jan. 2015. ISSN 1539-9087. doi: 10.1145/2680542. URL <http://doi.acm.org/10.1145/2680542>. (Cited on pages 18 and 19.)
- [102] A. M. Rahmani, M. H. Haghighayan, A. Miele, P. Liljeberg, A. Jantsch, and H. Tenhunen. Reliability-aware runtime power management for many-core systems in the dark silicon era. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(2):427–440, Feb 2017. ISSN 1063-8210. doi: 10.1109/TVLSI.2016.2591798. (Cited on pages 5 and 85.)
- [103] R.A.P.M. van den Bleek. Design of a hybrid adaptive cruise control Stop-&-Go system. Master’s thesis, Eindhoven University of Technology, 2007. (Cited on pages 3 and 114.)
- [104] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. August. Design and evaluation of hybrid fault-detection systems. In *32nd International Symposium on Computer Architecture (ISCA’05)*, pages 148–159, June 2005. doi: 10.1109/ISCA.2005.21. (Cited on page 90.)
- [105] S. S. Sahoo, A. Kumar, and B. Veeravalli. Design and evaluation of reliability-oriented task re-mapping in MPSoCs using time-series analysis of intermittent faults. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 798–803, March 2016. (Cited on page 101.)
- [106] H. Salunkhe, O. Moreira, and K. van Berkel. Mode-controlled dataflow based modeling and analysis of a 4G-LTE receiver. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, March 2014. doi: 10.7873/DATE.2014.225. (Cited on pages 64 and 65.)
- [107] H. Salunkhe, O. Moreira, and K. van Berkel. Buffer allocation for real-time streaming on a multi-processor without back-pressure. In *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pages 20–29, Oct 2014. doi: 10.1109/ESTIMedia.2014.6962342. (Cited on page 75.)
- [108] H. Salunkhe, O. Moreira, and K. van Berkel. Modeling & analysis of an LTE-advanced receiver using mode-controlled dataflow. *Microprocess. Microsyst.*, 47(PA):216–230, Nov. 2016. ISSN 0141-9331. doi: 10.1016/j.micpro.2016.09.013. URL <https://doi.org/10.1016/j.micpro.2016.09.013>. (Cited on pages 65 and 75.)
- [109] H. L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. PhD thesis, Eindhoven University of Technology, 2017. (Cited on page 82.)
- [110] I. Sander. *System Modeling and Design Refinement in ForSyDe*. PhD thesis, KTH Royal Institute of Technology, 2003. (Cited on page 63.)

-
- [111] I. Sander, A. Jantsch, and S.-H. Attarzadeh-Niaki. *ForSyDe: System Design Using a Functional Language and Models of Computation*, pages 99–140. Springer Netherlands, Dordrecht, 2017. ISBN 978-94-017-7267-9. doi: 10.1007/978-94-017-7267-9_5. URL https://doi.org/10.1007/978-94-017-7267-9_5. (Cited on page 6.)
- [112] G. Scheithauer. *Introduction to Cutting and Packing Optimization*. Springer, 2017. (Cited on pages 14, 87, and 89.)
- [113] L. Schor. *Programming Framework for Reliable and Efficient Embedded Many-Core Systems*. PhD thesis, ETH Zurich, 2014. (Cited on page 18.)
- [114] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES '12*, pages 71–80, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1424-4. doi: 10.1145/2380403.2380422. URL <http://doi.acm.org/10.1145/2380403.2380422>. (Cited on pages 18, 68, and 101.)
- [115] A. K. Singh, W. Jigang, A. Kumar, and T. Srikanthan. Run-time mapping of multiple communicating tasks on MPSoC platforms. *Procedia Computer Science*, 1(1):1019 – 1026, 2010. ISSN 1877-0509. doi: <https://doi.org/10.1016/j.procs.2010.04.113>. URL <http://www.sciencedirect.com/science/article/pii/S1877050910001146>. ICCS 2010. (Cited on page 102.)
- [116] A. K. Singh, T. Srikanthan, A. Kumar, and W. Jigang. Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms. *Journal of Systems Architecture*, 56(7):242 – 255, 2010. ISSN 1383-7621. doi: <https://doi.org/10.1016/j.sysarc.2010.04.007>. URL <http://www.sciencedirect.com/science/article/pii/S1383762110000330>. Special Issue on HW/SW Co-Design: Systems and Networks on Chip. (Cited on page 102.)
- [117] A. K. Singh, A. Kumar, and T. Srikanthan. A hybrid strategy for mapping multiple throughput-constrained applications on MPSoCs. In *2011 Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 175–184, Oct 2011. doi: 10.1145/2038698.2038726. (Cited on page 68.)
- [118] A. K. Singh, A. Kumar, and T. Srikanthan. A hybrid strategy for mapping multiple throughput-constrained applications on MPSoCs. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES '11*, pages 175–184, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0713-0. doi: 10.1145/2038698.2038726. URL <http://doi.acm.org/10.1145/2038698.2038726>.

- [119] A. K. Singh, A. Kumar, and T. Srikanthan. Accelerating throughput-aware runtime mapping for heterogeneous MPSoCs. *ACM Trans. Des. Autom. Electron. Syst.*, 18(1):9:1–9:29, Jan. 2013. ISSN 1084-4309. doi: 10.1145/2390191.2390200. URL <http://doi.acm.org/10.1145/2390191.2390200>. (Cited on page 68.)
- [120] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: Survey of current and emerging trends. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10, May 2013. doi: 10.1145/2463209.2488734. (Cited on pages 7, 68, and 100.)
- [121] S. Sinha, M. Koedam, R. Van Wijk, A. Nelson, A. Nejad, M. Geilen, and K. Goossens. Composable and predictable dynamic loading for time-critical partitioned systems. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 285–292, Aug 2014. doi: 10.1109/DSD.2014.40. (Cited on pages 24, 32, and 89.)
- [122] F. Siyoun. *Worst-case Temporal Analysis of Dynamic Streaming Applications*. PhD thesis, Eindhoven University of Technology, 2014. (Cited on page 64.)
- [123] F. Siyoun, M. Geilen, O. Moreira, and H. Corporaal. Worst-case throughput analysis of real-time dynamic streaming applications. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Code-sign and System Synthesis, CODES+ISSS '12*, pages 463–472, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1426-8. doi: 10.1145/2380445.2380517. URL <http://doi.acm.org/10.1145/2380445.2380517>. (Cited on pages 25 and 45.)
- [124] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, Mar. 2004. ISSN 1544-3566. doi: 10.1145/980152.980157. URL <http://doi.acm.org/10.1145/980152.980157>. (Cited on pages 5 and 101.)
- [125] L. T. Smit, G. J. Smit, J. L. Hurink, H. Broersma, D. Paulusma, and P. T. Wolkotte. Run-time assignment of tasks to multiple heterogeneous processors. In *5TH PROGRESS Symposium on Embedded Systems*, pages 185–192. STW Technology Foundation, 2004. URL <http://doc.utwente.nl/49442/>. (Cited on pages 14 and 102.)
- [126] S. M. Smith and J. M. Brady. SUSAN—a new approach to low level image processing. *International Journal of Computer Vision*, 23(1):45–78, 1997. ISSN 1573-1405. doi: 10.1023/A:1007963824710. URL <http://dx.doi.org/10.1023/A:1007963824710>. (Cited on page 61.)
- [127] L. Snidaro, I. Visentini, and G. L. Foresti. Fusing multiple video sensors for surveillance. *ACM Trans. Multimedia Comput. Commun. Appl.*, 8(1):7:1–7:18, Feb. 2012. ISSN 1551-6857. doi: 10.1145/2071396.2071403. URL <http://doi.acm.org/10.1145/2071396.2071403>. (Cited on pages 1 and 111.)

-
- [128] R. Stefan. *Resource allocation in time-division-multiplexed Networks-on-Chip*. PhD thesis, Delft University of Technology, 2012. (Cited on page 7.)
- [129] R. Stefan, A. B. Nejad, and K. Goossens. Online allocation for contention-free-routing NoCs. In *Proceedings of the 2012 Interconnection Network Architecture: On-Chip, Multi-Chip Workshop*, INA-OCMC '12, pages 13–16, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1010-9. doi: 10.1145/2107763.2107767. URL <http://doi.acm.org/10.1145/2107763.2107767>. (Cited on page 7.)
- [130] T. Stefanov, A. Pimentel, and H. Nikolov. *Daedalus: System-Level Design Methodology for Streaming Multiprocessor Embedded Systems on Chips*. Springer International Publishing, 2017. (Cited on page 63.)
- [131] S. Stuijk. *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, Eindhoven University of Technology, 2007. (Cited on pages 11, 18, 21, 28, 59, and 70.)
- [132] 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, New York, NY, USA, 2006. ACM. ISBN 1-59593-381-6. doi: 10.1145/1146909.1147138. URL <http://doi.acm.org/10.1145/1146909.1147138>. (Cited on pages 16, 25, 75, and 82.)
- [133] S. Stuijk, M. Geilen, and T. Basten. SDF³: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006. doi: 10.1109/ACSD.2006.23. URL <http://www.es.ele.tue.nl/sdf3>. (Cited on page 21.)
- [134] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 777–782, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1. doi: 10.1145/1278480.1278674. URL <http://doi.acm.org/10.1145/1278480.1278674>. (Cited on pages 21 and 59.)
- [135] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, Oct 2008. ISSN 0018-9340. doi: 10.1109/TC.2008.58. (Cited on pages 16, 75, and 82.)
- [136] S. Stuijk, A. Ghamarian, B. Theelen, M. Geilen, and T. Basten. FSM-based SADF. Technical report, Eindhoven University of Technology, 2008. (Cited on pages 9 and 25.)

- [137] S. Stuijk, M. Geilen, and T. Basten. A predictable multiprocessor design flow for streaming applications with dynamic behaviour. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pages 548–555, Sept 2010. doi: 10.1109/DSD.2010.31. (Cited on pages 21, 39, 68, 69, and 100.)
- [138] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 404–411, July 2011. doi: 10.1109/SAMOS.2011.6045491. (Cited on pages 9, 26, and 52.)
- [139] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *MEMOCODE*, pages 185–194, July 2006. doi: 10.1109/MEMCOD.2006.1695924. (Cited on pages 9, 25, and 64.)
- [140] B. Theelen, M. Geilen, S. Stuijk, S. Gheorghita, T. Basten, J. Voeten, and A. Ghamarian. Scenario-aware dataflow. Technical report, Eindhoven University of Technology, Department of Electrical Engineering, Electronic Systems group, 2008. (Cited on pages 39 and 64.)
- [141] B. Theelen, J.-P. Katoen, and H. Wu. Model checking of scenario-aware dataflow with cadp. In *DAC Europe, DATE '12*, pages 653–658, San Jose, CA, USA, 2012. EDA Consortium. ISBN 978-3-9810801-8-6. URL <http://dl.acm.org/citation.cfm?id=2492708.2492873>.
- [142] B. D. Theelen. A performance analysis tool for scenario-aware streaming applications. In *Fourth International Conference on the Quantitative Evaluation of Systems (QEST 2007)*, pages 269–270, Sep. 2007. doi: 10.1109/QEST.2007.7. (Cited on page 25.)
- [143] S. Trujillo, A. Crespo, A. Alonso, and J. Pérez. MultiPARTES: Multi-core partitioning and virtualization for easing the certification of mixed-criticality systems. *Microprocessors and Microsystems*, 38(8, Part B):921 – 932, 2014. ISSN 0141-9331. doi: <http://dx.doi.org/10.1016/j.micpro.2014.09.004>. URL <http://www.sciencedirect.com/science/article/pii/S0141933114001380>. (Cited on page 32.)
- [144] J. R. van Kampenhout. Deterministic task transfer in Network-on-Chip based multi-core processors. Master’s thesis, Delft University of Technology, 2011.
- [145] J. R. van Kampenhout and R. Hilbrich. Model-Based Deployment of Mission-Critical Spacecraft Applications on Multicore Processors. In *Reliable Software Technologies, Ada-Europe 2013*, volume 7896 of *Lecture Notes in Computer Science*, pages 35–50. Springer Berlin Heidelberg, 2013.

-
- [146] R. van Kampenhout, S. Stuijk, and K. Goossens. A scenario-aware dataflow programming model. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 25–32, Aug 2015. (Cited on pages 24, 37, and 65.)
- [147] R. van Kampenhout, S. Stuijk, and K. Goossens. Programming and analysing scenario-aware dataflow on a multi-processor platform. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 876–881, March 2017. (Cited on pages 15 and 37.)
- [148] R. van Kampenhout, S. Stuijk, and K. Goossens. Fault-tolerant deployment of dataflow applications using virtual processors. In *Digital System Design (DSD), 2018 Euromicro Conference on*, pages 77–84, Aug 2018. (Cited on pages 17 and 85.)
- [149] P. van Stralen and A. Pimentel. Scenario-based design space exploration of MPSoCs. In *2010 IEEE International Conference on Computer Design*, pages 305–312, Oct 2010. doi: 10.1109/ICCD.2010.5647727. (Cited on pages 18 and 19.)
- [150] A. Weichslgartner, S. Wildermann, D. Gangadharan, M. Glaß, and J. Teich. A design-time/run-time application mapping methodology for predictable execution time in MPSoCs. Technical report, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2017. (Cited on pages 18 and 19.)
- [151] Wikipedia. Unified Modeling Language, November 2018. URL https://en.wikipedia.org/wiki/Unified_Modeling_Language. (Cited on page 21.)
- [152] Wikipedia. Video compression picture types, October 2018. URL https://en.wikipedia.org/wiki/Video_compression_picture_types. (Cited on pages 1 and 114.)
- [153] S. Wildermann, F. Reimann, D. Ziener, and J. Teich. Symbolic design space exploration for multi-mode reconfigurable systems. In *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 129–138, Oct 2011. doi: 10.1145/2039370.2039393. (Cited on pages 18 and 19.)
- [154] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. ISSN 1539-9087. doi: 10.1145/1347375.1347389. URL <http://doi.acm.org/10.1145/1347375.1347389>. (Cited on page 7.)
- [155] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. *IEEE Transactions on Computer-Aided Design of*

- Integrated Circuits and Systems*, 28(7):966 –978, july 2009. ISSN 0278-0070. doi: 10.1109/TCAD.2009.2013287. (Cited on page 34.)
- [156] J. Windsor and K. Hjortnaes. Time and Space Partitioning in Spacecraft Avionics. In *Space Mission Challenges for Information Technology, 2009. SMC-IT 2009. Third IEEE International Conference on*, pages 13 –20, july 2009. doi: 10.1109/SMC-IT.2009.11. (Cited on page 32.)

List of Acronyms

ACC	Accelerator	85
ASIC	Application Specific Integrated Circuit	24
BAG	Binding-Aware Graph	24
BDF	Boolean Dataflow	52
CAL	Cal Actor Language	63
CompSOC	Composable System-on-Chip	11
CSDF	Cyclo-static Dataflow	24
CSV	Comma-Separated Values	96
DDR	Double Data Rate	61
DSE	Design Space Exploration	16
DMA	Direct Memory Access	60
DRAM	Dynamic Random-Access Memory	30
DVFS	Dynamic Voltage and Frequency Scaling	107
ET	Execution Time	72
FIFO	First-in First-out	31
FPGA	Field Programmable Gate Array	24
FPS	Frames Per Second	1
FSM	Finite-State Machine	24
FSM-SADF	Finite-State Machine Scenario-Aware Dataflow	9
GALS	Globally Asynchronous, Locally Synchronous	29
GP	General Purpose	85
HSDF	Homogeneous Synchronous Dataflow	20
JPEG	Joint Photographic Experts Group	
KPN	Kahn Process Network	18
LTE	Long-Term Evolution	65
MCR	Maximum Cycle Ratio	25
MoC	Model-of-Computation	5
MoE	Model-of-Execution	31
MCDF	Mode-Controlled Dataflow	9
MPEG	Moving Picture Expert Group	
NoC	Network-on-Chip	59
OS	Operating System	6
PM	Programming Model	6

PRET	Precision Timed Machine	63
PPN	Polyhedral Process Network	63
RT	Real-Time	2
RR	Round-Robin	33
RSO	Rolling Static-Order	52
SADF	Scenario-Aware Dataflow	9
SDF	Synchronous Dataflow	6
SRAM	Static Random-Access Memory	30
SO	Static-Order	11
SUSAN	Smallest Univalve Segment Assimilating Nucleus	39
TDM	Time-Division Multiplexing	12
TT	Time-Triggered	25
TTA	Time-Triggered Architecture	63
UML	Unified Modeling Language	21
VLSI	Very Large Scale Integration	5
VP	Virtual Processor	11
VR	Virtual Resource	11
WCET	Worst-Case Execution Time	7
WCRT	Worst-Case Response Time	7
WLAN	Wireless Local Area Network	
QoS	Quality-of-Service	26
QSOS	Quasi Static-Order Scheduling	65

Index

- actor, 9, 24, 26
- actor, detector, 39, 64
- actor, select, 45, 54
- actor, switch, 45, 54
- algebra, (*max*, +), 25, 65
- analysis, (*max*, +), 25
- analysis, state-space, 25
- analysis, throughput, 25
- ASIC, 24

- bin-packing, 14, 17, 87, 89
- bitfile, 24
- budget, 11
- buffer, FIFO, 31, 33, 70
- bundle, 6, 11

- CAL, 63
- causality dilemma, 10, 41
- channel, 26, 89
- channel, synchronisation, 44
- CoMik, 24, 32
- composable, 12, 32, 33
- CompSOC, 11, 12, 16, 21, 28, 31, 33, 70

- dataflow, 9, 24, 26, 63
- dataflow, cyclo-static, 24
- dataflow, finite-state machine scenario-aware, 9, 24, 25
- dataflow, homogeneous synchronous, 24, 25
- dataflow, mode-controlled, 9, 25, 64
- dataflow, scenario-aware, 9, 25, 64
- dataflow, synchronous, 6, 7, 24, 31
- deadline, 2
- deployment, 7, 12, 13, 30, 32, 68, 85
- design flow, 5
- detection, scenario, 37
- DSE, 16, 67, 73

- dynamic set, 4, 8, 13

- embedded systems, 1
- events, 8, 63, 67
- execution model, scenario, 42, 49
- execution time, worst-case, 7, 9, 34, 88

- fault-tolerance, 5, 11, 85, 87, 90
- fire, 16, 26, 89
- firing rules, 31
- FPGA, 24
- FSM, 6, 24, 26, 39

- GALS, 29, 70, 87, 88
- Giotto, 7, 63
- graph, binding-aware, 30, 58
- graph, prefix, 41
- graph, scenario, 24

- hot spots, 5

- iteration, 24, 26

- Kahn process network, 18, 31

- latency, 7, 76
- libDataflow, 24, 31, 33, 52, 56
- libFifo, 24, 33

- mapping, 6, 28
- Markov chains, 25
- MCR, 25, 69
- middleware, 7, 24, 32
- MoC, 6, 24
- MoC, time-triggered, 6, 25

- OpenMP, 7

- Pareto-optimal, 16, 19, 31, 73, 77
- partition, 32

- pipelining, scenario, 42
- platform, hardware, 7, 24, 33
- predictable, 13, 33, 88
- PRET, 63
- processor, 5, 7, 11, 28, 67
- processor, spare, 18, 101
- processor, virtual, 11, 13
- programming model, 7, 31
- PThreads, 7

- rate, 26
- rate, consumption, 27
- rate, production, 27
- re-deployment, 15, 17, 32
- real-time, 2
- repetition vector, 27
- requestor, 32, 34
- requirements, 4, 21
- resize, 87
- resource manager, 7, 12, 14, 89
- resource, stateless, 16
- resource, virtual, 11, 15, 30
- response time, worst-case, 7, 13, 34, 69, 70, 88
- Round-Robin, 33

- scenario, 9, 26
- scenario detection, delayed, 37
- scenario detection, immediate, 39
- scenario, detector, 15, 42
- schedule, rolling static-order, 52, 59
- schedule, static-order, 11, 15, 30
- scheduling, 7, 28
- SDF³, 11, 16, 18, 21, 75
- self-edge, 27
- sequencing model, scenario, 42, 43
- split, 87
- storage distribution, 29, 67

- task, 6
- TDM, 12, 30, 32, 70
- throughput, 7, 29, 30, 73, 88
- tile, 28
- token, 26, 89
- token, detector, 39
- token, persistent, 26
- token, scenario, 54
- token, shared persistent, 42, 57
- transition, scenario, 25, 41, 65
- TTA, 63
- Turing machines, 6

- UML, 21

- VLSI, 5, 85

- waiting time, 7, 70
- work-conserving, 25, 63

Acknowledgements

First and foremost I would like to express my gratitude to my promotor Prof. Kees Goossens and co-promotor Dr. Sander Stuijk for their excellent supervision and guidance during my Ph.D. research. Besides an extraordinary knowledge in their respective academic fields, they both combine an informal, practical approach to management with a great enthusiasm for solving scientific and technical problems. My steady stream of notes taken during our weekly meetings testifies to the many ideas that were generated during these years, and also reveal how these have been refined and developed upon to arrive at tangible results. I would never have imagined that we could talk for so long about arrows and circles! Especially when my motivation waned Sander and Kees helped me getting back on track and to focus on the important bits. Sander, thanks for the practical help and quick problem-solving that saved me many of hours pioneering on my own. Kees, thank you for creating the warm and collegial atmosphere.

Furthermore, I would like to thank the doctorate committee for reading my thesis, providing constructive feedback, and agreeing to oppose me at the defence. A hearty thanks goes to Marja for creating a great ambiance in the Electronic Systems (ES) group, for her help with administrative issues, and for our pleasurable chats about life. Another warm thanks goes to Margot, Rian and Feyza for their support. I also thank all current and former CompSOC colleagues for their collaboration and input during our meetings and discussions, as well as the fun we had during lunch, coffee breaks and bar visits: Alessandro, Andrew, Gabriela, Sven, Hadi the second, Hamide, Juan, Manil, Martijn, Mojtaba, Rasool, Shayan, Shubendu, Yonghui and Zhan. Thanks to all other ES colleagues for the coffee-machine chats and general good atmosphere: Amir, Andreia, Bram, Edwin, Firew, Francesco, Gert-Jan, Hadi the first, Kamlesh, João, Joost, Luc, Mark, Maurice, Mladen, Paul, Roel, Sajid, Sayandip, Shreya, Umar, Victor and everyone I forgot to mention. A special thanks goes to Martijn Koedam for his invaluable help with tools, debugging and the 5LIB0 course, and to Dr. Marc Geilen for his ad-hoc advice and editing work on the scenarios book chapter.

Thanks to my parents for their belief and support in every step of my career. More family thanks to Cornelie, Sjors, Lotje, Ruben and Benjamin for the good times and giving me a hard time each sjoel-competition. Many thanks to the brullers: Beejee, Edu, Leo, Mart'n, Slaze and Zomer. I could not have done it without you guys. A big shoutout also to the Berlin party gang: Anneli, Dora, Job, Joon, Klabbie, Rebekka, Sjoerd and Skippy. Another big thanks to my dutch party friends: Bram, Greetje, Irene, Lemke, Leonie, Lotte, Mijke, Natalja, Niek, and many others.

A special thanks to Lieu for recent medical support, recipes and hospitality. Finally an extra special thanks to Leo and Toon for their persisting moral support.

About the Author

Reinier van Kampenhout was born on the 5th of May 1983 in Beilen, Drenthe. He attended MAVO secondary school and then completed an education in telematics at polytechnical college (MBO) in 2003. Subsequently he pursued his bachelor's degree in Electronic Product Design and Engineering at the Hanze University of Applied Sciences in Groningen, which he obtained in 2006. After that he completed a bridging program and obtained his master's degree in Computer Engineering from Delft University of Technology in 2011 with a focus on embedded systems. Next, he worked for four years as a project scientist at the Fraunhofer FOKUS institute in Berlin. In 2014, he started the pursue of his Ph.D. degree in the Electronic Systems group at Eindhoven University of Technology, culminating in this dissertation. His research interests include real-time embedded systems, programming models, multi-processors, dataflow and fault-tolerance.

List of Publications Related to this Thesis

R. van Kampenhout, S. Stuijk, and K. Goossens. A scenario-aware dataflow programming model. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 25–32, Aug 2015

R. van Kampenhout, S. Stuijk, and K. Goossens. Programming and analysing scenario-aware dataflow on a multi-processor platform. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 876–881, March 2017

K. Goossens, M. Koedam, A. Nelson, S. Sinha, S. Goossens, Y. Li, G. Breaban, R. van Kampenhout, R. Tavakoli Najafabadi, J. Valencia, H. Ahmadi Balef, B. Akesson, S. Stuijk, M. Geilen, D. Goswami, and M. Nabi Najafabadi. *Handbook of hardware/software codesign*, chapter NoC-based multiprocessor architecture for mixed-time-criticality applications, pages 491– 530. Springer, 11 2017. ISBN 978-94-017-7266-2

R. van Kampenhout, S. Stuijk, and K. Goossens. Fault-tolerant deployment of dataflow applications using virtual processors. In *Digital System Design (DSD), 2018 Euromicro Conference on*, pages 77–84, Aug 2018

M. Geilen, M. Skelin, R. van Kampenhout, H. Alizadeh Ara, T. Basten, S. Stuijk, and K. Goossens. *System Scenario-based Design Principles and Applications*, chapter Scenarios in Dataflow Modelling and Analysis. Springer, 2019. ISBN 978-3-030-20342-9

List of Publications Unrelated to this Thesis

R. Hilbrich and J. R. van Kampenhout. Dynamic reconfiguration in NoC-based MPSoCs in the avionics domain. In *IWMSE '10: Proceedings of the 3rd International Workshop on Multicore Software Engineering*, pages 56–57, New York, NY, USA, 2010. ACM

R. Hilbrich and J. R. van Kampenhout. Partitioning and Task Transfer on NoC-based Many-Core Processors in the Avionics Domain. In *4. Workshop: Entwicklung zuverlässiger Software-Systeme (Stuttgart, Deutschland) and Journal "Software-waretechartrends"*, 2011

J. R. van Kampenhout. Deterministic task transfer in Network-on-Chip based multi-core processors. Master's thesis, Delft University of Technology, 2011

R. Hilbrich, J. R. van Kampenhout, and H.-J. Goltz. Modellbasierte generierung statischer schedules fuer sicherheitskritische, eingebettete systeme mit multicore-prozessoren und harten echtzeitanforderungen. In *Herausforderungen durch Echtzeitbetrieb*, Informatik aktuell, pages 29–38. Springer, 2012

R. Hilbrich, J. R. van Kampenhout, M. Daun, D. T. Weyer, and D. Sojer. *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology*, chapter Modeling Quality Aspects: Real-Time, pages 119–128. Springer, 2012. ISBN 978-3-642-34614-9

J. R. van Kampenhout and R. Hilbrich. Model-Based Deployment of Mission-Critical Spacecraft Applications on Multicore Processors. In *Reliable Software Technologies, Ada-Europe 2013*, volume 7896 of *Lecture Notes in Computer Science*, pages 35–50. Springer Berlin Heidelberg, 2013

