Simulation Based Execution Time Analysis of SDF Applications on Heterogeneous MPSoCs Using Measured Delay Distributions

Von der Fakultät für Informatik, Wirtschafts- und Rechtswissenschaften der Carl von Ossietzky Universität Oldenburg zur Erlangung des Grades und Titels

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

angenommene Dissertation

von Herrn Ralf Stemmer
geboren am 23. Mai 1987 in Meschede

Gutachter: Prof. Dr.-Ing. Wolfgang Nebel

Weiterer Gutachter: Prof. Dr. Sébastien Le Nours

Tag der Disputation: 22. Januar 2024

Summary

Embedded systems become more and more part of the everyday life. With the boom of the Internet-of-Things the amount of those small computer systems increased tremendously. Also the complexity of tasks they have to do changed from simple control algorithms to image-processing, video-processing or even Artificial Intelligence. Therefore Heterogeneous Multi-Processor Systems-on-a-Chip (HMPSoCs) are commonly used to execute those software systems. To increase power and cost efficiency, a trade-off between the resources of the hardware and the needs of the software must be found. While optimizing the mapping of the software on the hardware components the performance impact must be evaluated to guarantee a certain quality of service. Classical formal analysis methods are often too compute intensive for MPSoCs. Simulation based approaches can improve the scalability.

In my work, I present a simulation based approach to analyze timing behavior of software executed on HMPSoCs. Therefore I use the distribution of execution times for the performance model. This method provides a more realistic estimation of execution times of an application. Goal of this work is, to provide a suitable method for design space exploration of HMPSoCs. For my approach strict models are used for modeling hardware and software. This reduces the complexity of the performance analysis. The hardware is a set of composable tiles with private memory for instructions and local data. The software model differentiates between computation and communication phases. This allows to determine when and where interference can happen. To characterize the system, the execution time of computation phases of the software components get measured. By keeping the distribution of execution times instead of just abstracting the data to best-case and worst-case delay, the analysis results are more representative to the actual behavior of the real system. Shared resources like communication interface and shared memory get modeled in detail.

For evaluation, a heterogeneous MPSoC with up to 7 processing elements is used. This HMPSoC is used to run different applications like a Sobel-Filter (simple example) and a JPEG-Decoder (computation intensive). For the evaluation the analyzed execution time gets compared to measured execution times of the same mapping. Therefore the execution time from the beginning to the end of an iteration of sample applications gets analyzed and measured. This is done for various applications with different mappings for several hardware configurations.

The proposed analysis method allows an insight into the distribution of execution times of possible mappings for an application on an HMPSoC. Kernel Density Estimation as inference technique showed good results and allowed to not just get a reasonable average execution time but also a representative distribution function of possible execution times as simulation result. To get a well suited distribution function for an actor, it is important to characterize this actor on all different instruction set architectures with all different combinations of available hardware accelerators like floating point units or hardware multiplier. While measuring a certain mapping and configuration usually took more than 2 h, the simulation finished within less than 45 min in most cases. The modeling and analysis approach presented in this thesis allowed a prediction of the average execution time of a certain mapping and configuration with an error less than 5 %.

Zusammenfassung

Eingebettete Systeme werden mehr und mehr Teil des täglichen Lebens. Mit dem Boom der Internet-of-Things ist die Anzahl kleiner Computersysteme enorm gestiegen. Auch die Komplexität der Aufgaben, die sie erfüllen müssen, hat sich von einfachen Steuerungsalgorithmen bis hin zu Bildverarbeitung, Videoverarbeitung oder gar künstlicher Intelligenz gesteigert. Daher werden heterogene Mehrprozessorsysteme auf einem Chip (HMPSoCs) häufig zur Ausführung dieser Softwaresysteme verwendet. Um die Energie- und Kosteneffizienz zu erhöhen, muss ein Kompromiss zwischen den Ressourcen der Hardware und den Anforderungen der Software gefunden werden. Bei der Optimierung des Mappings der Softwarekomponenten auf die Hardwarekomponenten müssen die Auswirkungen auf die Leistung bewertet werden, um eine gewisse Qualität der von der Software zu erfüllenden Leistung zu gewährleisten. Klassische formale Analysemethoden sind oft selbst für kleine Mehrprozessorsysteme zu rechenintensiv. Simulationsbasierte Ansätze können die Skalierbarkeit verbessern.

In meiner Arbeit präsentiere ich einen simulationsbasierten Ansatz um das Timing-Verhalten von Software zu analysieren, die auf HMPSoCs ausgeführt wird. Dazu verwende ich die Verteilung der Ausführungszeiten für das Ausführungszeitmodell. Diese Methode bietet eine realistischere Abschätzung der Ausführungszeiten einer Anwendung. Ziel dieser Arbeit ist es, eine geeignete Methode für die Design Space Exploration von HMPSoCs bereitzustellen. Für meinen Ansatz werden strikte Modelle für die Modellierung von Hardware und Software verwendet. Dadurch wird die Komplexität der Ausführungszeitanalyse reduziert. Die Hardware ist ein Menge komponierbarer Tiles mit privatem Speicher für Instruktionen und lokale Daten. Das Softwaremodell unterscheidet zwischen Berechnungs- und Kommunikationsphasen. Dadurch lässt sich bestimmen, wann es zu Engpässen bei dem Zugriff auf gemeinsame Ressource, wie zum Beispiel ein Kommunikationsbus, auftreten können. Um das System zu charakterisieren, wird die Ausführungszeit der Berechnungsphasen der Softwarekomponenten gemessen. Indem man die Verteilung der Ausführungszeiten verwendet, anstatt die Daten nur auf Best-Case- und Worst-Case-Verzögerungen zu abstrahieren, sind die Analyseergebnisse repräsentativer für das tatsächliche Verhalten des realen Systems. Gemeinsam genutzte Ressourcen wie Kommunikationsschnittstellen und gemeinsam genutzter Speicher werden im Detail modelliert.

Für die Evaluierung wird ein heterogener MPSoC mit bis zu 7 Prozessoren verwendet. Auf diesem HMPSoC laufen verschiedene Anwendungen wie ein Sobel-Filter (einfaches Beispiel) und einen JPEG-Decoder (rechenintensiv). Für die Auswertung wird die analysierte Ausführungszeit mit gemessenen Ausführungszeiten desselben Mappings verglichen. Dazu wird die Ausführungszeit vom Beginn bis zum Ende einer Iteration der Beispielanwendungen analysiert und gemessen. Dies geschieht für verschiedene Anwendungen mit unterschiedlichen Mappings für verschiedene Hardware-Konfigurationen.

Die vorgeschlagene Analysemethode erlaubt einen Einblick in die Verteilung der Ausführungszeiten möglicher Mappings für eine Anwendung auf einem HMPSoC. Kernel Density Estimation (KDE) als Abstraktion der charakterisierten Ausführungszeiten zeigte gute Ergebnisse in der Evaluation. KDE ermöglichte es nicht nur eine relativ präzise durchschnittliche Ausführungszeit als Simulationsergebnis zu erhalten sondern auch eine repräsentative Verteilungsfunktion der möglichen Ausführungszeiten. Um eine gute und repräsentative Verteilungsfunktion für eine Softwarecomponente zu erhalten, ist es wichtig, diese Komponente auf allen verschiedenen Befehlssatzarchitekturen mit allen verschiedenen Kombinationen von verfügbaren Hardware-Beschleunigern, wie eine Gleitkommaeinheiten (FPU) oder einen Multiplizierer in Hardware, zu charakterisieren. Während die Messung eines bestimmten Mappings und einer bestimmten Konfiguration normalerweise mehr als 2 Stunden dauerte, wurde die Simulation in den meisten Fällen in weniger als 45 Minuten abgeschlossen. Der in dieser Arbeit vorgestellte Modellierungs- und Analyseansatz ermöglichte eine Vorhersage der durchschnittlichen Ausführungszeit für eine bestimmtes Mapping und eine bestimmte Hardwarekonfiguration mit einem Fehler von weniger als 5 %.

Contents

1 Introduction . . . 1
1.1 Motivation . . . 1
1.2 Research Questions . . . 3
1.3 Contributions . . . 4
1.4 Prior Publications . . . 4
1.5 Work Flow . . . 5
1.6 Structure . . . 7

2 Related Work . . . 8
2.1 Overview of Analysis Approaches . . . 8
2.2 Simulation Based Approaches . . . 8
2.3 Formal Approaches . . . 9
2.4 Hybrid Probabilistic Methods . . . 10
2.5 Execution Time Estimation . . . 13

3 Software, Hardware and Delay Models . . . 15
3.1 Model of Computation . . . 15
3.1.1 Buffers . . . 15
3.1.2 Actors . . . 16
3.1.3 Ports . . . 17
3.1.4 Channels . . . 18
3.2 Model of Architecture . . . 18
3.2.1 Execution Platform . . . 19
3.2.2 Tile . . . 20
3.2.3 Processing Element . . . 20
3.2.4 Memory . . . 21
3.2.5 Interconnect . . . 21
3.3 Delay Model . . . 22
3.3.1 Delay . . . 22
3.3.2 Modeling Delay Occurrence Probability . . . 22
3.3.3 Delay Distribution Inference . . . 23
3.3.4 Computation Delay Set . . . 25
3.4 Communication Delay Models . . . 26
3.4.1 Cycle Accurate Model . . . 29
3.4.2 Transaction Level Model . . . 32
3.4.3 Message Level Model . . . 33
3.5 System Model . . . 34
3.5.1 Mapping . . . 35
3.5.2 Scheduling . . . 35

4 Characterization and Analysis Approach . . . 37
4.1 Delay Measurement Infrastructure . . . 37
4.1.1 Measuring Pipelined Execution . . . 38
4.1.2 Proposed Measurement Infrastructure . . . 40
4.1.3 Verification of Delay Measurement Infrastructure . . . 42
4.2 Characterization . . . 43
4.2.1 Computation Delay Distribution . . . 44
4.2.2 Abstract Communication Delay . . . 47
4.2.3 Elementary Communication Delays . . . 50
4.3 Simulation . . . 51
4.3.1 SystemC TLM Interface . . . 52
4.3.2 SDF Application . . . 52
4.3.3 Delay Vector . . . 53
4.3.4 Monitor . . . 53
4.3.5 MoC Implementation . . . 53
4.3.6 MoA Implementation . . . 54
4.3.7 Simulation Configuration . . . 55
4.4 Work Flow . . . 56

5 Models and Simulation Evaluation . . . 58
5.1 Evaluation Setup . . . 58
5.1.1 Sobel Filter . . . 59
5.1.2 JPEG Decoder . . . 60
5.1.3 Evaluation Platform . . . 61
5.2 Evaluation Concept . . . 62
5.3 Experiment Execution and Functional Verification . . . 64
5.4 Computation Model Evaluation . . . 65
5.4.1 Experiment Setup . . . 66
5.4.2 Experiment Results . . . 66
5.4.3 Discussion . . . 68
5.5 Communication Model Evaluation . . . 70
5.5.1 Experiment Setup . . . 70
5.5.2 Experiment Results . . . 70
5.5.3 Discussion . . . 71
5.6 Simulation Time Evaluation . . . 72
5.6.1 Experiment Setup . . . 72
5.6.2 Experiment Results . . . 73
5.6.3 Discussion . . . 75

6 Discussion . . . 77

7 Conclusion . . . 79

8 Future Work 82

9 References 83

References 84

Chapter 1
Introduction

1.1 Motivation

This thesis proposes a simulation based execution time analysis of synchronous dataflow applications executed on Heterogeneous Multi-Processor Systems-on-Chips (HMP-SoCs). For execution time characterization, a measurement based approach is followed.

Execution time analysis of software can be important in many domains with many different goals. This work focuses on data flow applications like sensor data processing as it can be found in automotive systems. In automotive context, for example automated vehicles, the safety of the system is crucial. The safety of such a system is not just based on functional correctness, but also on its timing behavior. A system that works functional correct, but provides its computation result too late, may still be unsafe. Examples for such time critical systems are emergency braking systems or air-bag controllers. Other factors like the development cost of the design of such systems are also an important consideration. To achieve a good trade-off for a safe but cost efficient system, early design space exploration for engineering suitable system components as well as early performance assessment is important for design decisions.

For this performance assessment, models of the system under analysis are created and analyzed. There are different approaches for execution time analysis that can be categorized as formal approaches, simulation based approaches and hybrid approaches that combine both [Gaj+09]. A detailed examination of advantages and disadvantages of the different analysis approaches has been done in Sec 2.1.

Formal approaches can scale well for larger systems under analysis, but lack of the possibility to consider different states of components which may be necessary to model shared resources access. State based methods can apply model checking techniques to fully explore the state space of a model, which does not scale well. Formal approaches can give certain guarantees like a upper and lower execution time bounds, but these guarantees come with the downside of a high analysis time and overestimated execution time bounds as analysis result. In conclusion, state based formal approaches tend to not scale well with large systems regarding analysis time [Fak+15]. Furthermore, formal methods tend to overestimate execution times [Per+09] which can lead to design decisions for too cost intensive hardware that is capable to run the software fast enough.

Simulation based execution time analysis can be faster than state based formal methods and can provide a tighter execution time bound [Ste+19c]. Drawback is a lack of guarantee that the simulated system never violates the analyzed timing bounds. With a simulation it cannot be guaranteed that the whole state space got fully covered. There can be cases after a certain amount of simulation runs that have not yet been simulated.

In this thesis, focus is on data processing algorithms as they could be used in feedback control systems (FCS). FCS get regularly input data to adapt a system to follow a certain set value. This input might be preprocessed sensor data. An example for preprocessing data that is used by a FCS can be a JPEG encoded image that gets sent from an image sensor (camera) once every second. The image then needs to be decoded and additional image filters may need to be applied to provide input to a feedback control algorithm that the current state of the system controls. If, in rare cases, one input sample does not come in time, the feedback controller may still be in a safe and functional state. A feedback controller that works on video data may still function if one frame once in a while gets dropped because the decoding algorithm misses its deadline. Thus, tight execution time bounds allow more cost efficient hardware selection without the risk of violating safety requirements in case a deadline is missed from time to time.

The execution time of data processing algorithms like a JPEG-decoder can vary for several reasons. There are data dependent execution paths inside the algorithm leading to different execution times. Further execution paths can be introduced by the compiler to work around missing hardware support for example for floating point operations. In context of Multi-Processor System-on-a-Chip shared hardware resources are another cause of varying execution times.

Providing a distribution of possible execution times as analysis result can support the designer to decide if a hardware fits certain requirements. If the average execution time is close to the worst simulated execution time the system may more likely violate a deadline than a system where the average execution time is closer to the best simulated time.

To perform an execution time analysis, models for the software application and the hardware platform, on which the software gets executed, are required. The level of detail of theses models have a huge impact on accuracy and analysis time. Furthermore, delays annotated to the model to perform execution time analysis need to be characterized. The proposed methods and models are developed to be used in context of design space exploration, not for safety analysis. This allows focus on an abstract but fast modeling and analysis approach to allow the system designers to narrow down the design space during early development. For a more mature product more detailed and time consuming approaches may be needed to guarantee a safe operation.

For delay characterization, a measurement based approach is proposed. This approach can provide cycle accurate delays while being independent from the hardware platform, while static code analysis would be tight to specific instruction set architectures. Furthermore, a measurement based approach allows to model the probability distribution function of delays of the characterized system component. Giving this distribution function as input into the simulation, allows the simulation to predict the delay distribution of the overall system. This gives the designers further information about how likely deadlines may be missed.

1.2 Research Questions

This thesis addresses the research questions formulated in the following list. These questions originated from discussion with my colleague Maher Fakih about his PhD thesis [Fak16]. Especially questioning the formal analysis approach he followed and the scalability issues he was facing with this approach. In early research presented in [Ste+17; Ste+19c] these questions have been refined and extended by addressing a measurement based approach for characterization and a simulation based analysis approach. In related work (See Chap 2) several analysis approaches have been presented and compared with most of them addressing similar challenges, each with focus on different aspects. One focus of my thesis is following a practical analysis and characterization approach that could be used for design space exploration or early product design decision making. This leads to the question on how well the proposed techniques and models work on realistic systems and how well their scalability is.

1. Are measured delays a suitable abstraction for timing of application functions?
2. What is necessary to predict execution time distributions of an application on an MPSoC in short time?
3. How can I achieve good accuracy for analyzing realistic systems?
4. How can I achieve good scalability for analyzing realistic systems?
5. What are the limitations of my approach when applying to consumer of-the-shelf platforms?

The first question addresses the suitability of abstraction for software. In context of execution time analysis, a suitable abstraction of a software application allows representing its execution delay as accurate as needed for an accurate overall system analysis. Since the characterization of software function is done following a measurement based approach, a simulation based execution time analysis makes only sense when the simulation is faster than using the characterization measurement infrastructure to simply measure an instance of the final product configuration. Furthermore the followed characterization and analysis strategy should be capable to scale up with realistic systems. A realistic system to the understanding of the author of this thesis, is a software application that fulfills a reasonable task that cannot be analyzed by a human looking at its instructions. This software then should be executed on a Multi-Processor System-on-a-Chip with more than two processing elements. Since use-case of the presented analysis approach is not safety analysis but design space exploration, an error less than 10 % is the desired goal for accuracy.

In the conclusion, Chap 7, the answers to the research questions are presented and discussed.

1.3 Contributions

The following contributions are presented in this thesis:

1. A simulation based real-time analysis approach for heterogeneous tile-based MPSoCs and applications following the Synchronous Dataflow (SDF) constraints (Sec 4.3).
2. An interference free execution time measurement infrastructure for characterizing SDF actor computation delay and for shared interconnect characterization. (Sec 4.1, Sec 4.2)
3. A probabilistic execution time analysis approach based on measurement based characterization (Sec 4.1) and probability density functions (Sec 3.3).
4. A workflow for simulation based real-time analysis, modeling and characterization using measured delay information (Sec 4.4).
5. A comparison of different abstraction levels for execution time, using accuracy and scalability of the analysis as metric. Compared have been a single average delay with an uniform distribution between worst and best observed execution time, Gauss distribution and a Kernel Density Estimation approach (Sec 3.3, Sec 5.4)
6. A comparison of different abstraction levels for state based communication models. A cycle accurate model (Sec 3.4.1) and transaction level model (Sec 3.4.2) have been created and evaluated together with a third model, a message level model (Sec 3.4.3) by Vu et al ([Vu+21]). For evaluation accuracy and scalability was used as metric. (Sec 3.4, Sec 5.5).

1.4 Prior Publications

Most of the concepts I present in this thesis have already been published by me in peer reviewed scientific journals, conferences and workshops. This section provides an overview of these publications and their contributions. A list of my publications in context of this thesis is provided in Chap 9.

My first publication leading me towards the research and research questions I present in this thesis was 2017 in [Ste+17]. For that publication I applied some concepts and models from the PhD thesis [Fak16] of my former colleague Maher Fakih. I adopted his model checking based real time analysis approach for MPSoCs to finite state machine based scenario aware dataflow graphs (FSM-SADFG) [Stu+08].

In [Ste+19c] I used a measurement infrastructure that has been developed and published ([SFS17]) by Christof Schlaak for his master thesis under guidance and support by me and Maher Fakih. An improved and specialized adaptation of this measurement infrastructure (Sec 4.1.2) is used to characterize the execution time of synchronous dataflow actors (Sec 3.1). These data have been used for execution time analysis using statistical model checking.

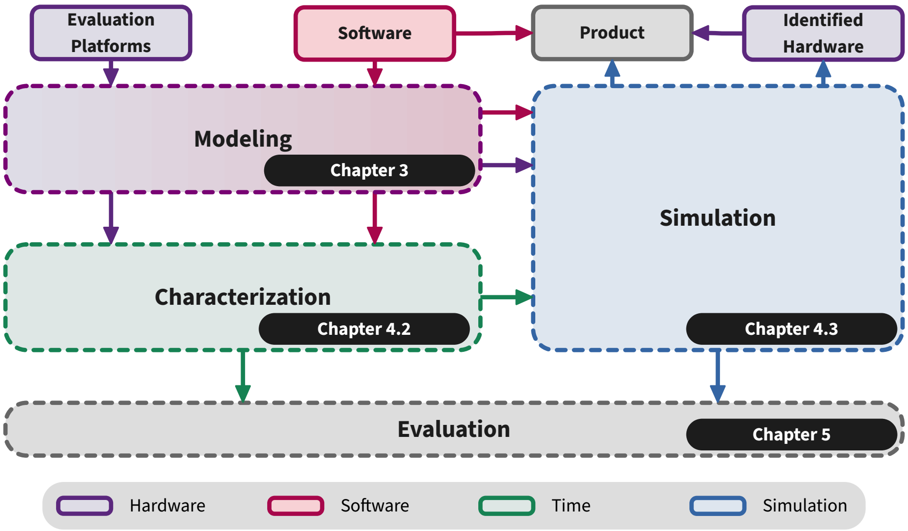


Fig. 1.1 Overview of the modeling, characterization and simulation workflow presented in this thesis. In the evaluation, simulated timing characteristics get compared to actually observed timings.

In [Ste+19a] I first presented the use of a SystemC simulation for real time analysis. There I also introduced the “Cycle Accurate” communication model that is described in detail in Sec 3.4.1. This was also the first publication with my colleague Hai-Dang Vu from the Université de Nantes. We did a feasibility study published as technical report [Ste+19b] where we investigated in statistical model checking techniques that got further investigated and applied in the PhD thesis of Hai-Dang Vu [Vu21].

The models and simulation have been improved and further evaluated in [Ste+20]. Focus was accuracy and scalability of the simulation compared against a simple static analytical model. Therefore a seven tiles MPSoC (Sec 5.1.3) and a JPEG decoder (Sec 5.1.2) have been developed and introduced that are used by me in this thesis as well as by Hai-Dang Vu.

In [Ste+21] I first introduced Kernel Density Estimation to derive actor delays from measured data. I compared different computation delay distribution functions and different communication models including the “Message Level” (Sec 3.4.3) model introduced by Hai-Dang Vu in [Vu+21]. Furthermore I introduced the “Transaction Level” model (Sec 3.4.2) as hybrid between the cycle accurate and message level model.

1.5 Work Flow

This section describes the work flow followed in this thesis for the modeling, characterization and execution time analysis. An abstract view on this work flow is shown in Fig. 1.1. The purpose of this flow is to help embedded systems designers to answer the question, which hardware fits best to execute a given software within certain time constraints with as low costs as possible. This solution, fast and cheapest hardware that runs a certain software, is the product (Fig. 1.1 Gray) that the designer who uses the proposed approach aims for. For evaluation of the models and the process the simulation results get also compared against the actual observed behavior.

The rectangles with solid lines represent system components like software or hardware. Rounded rectangles with dashed lines describe processes. The arrows represent the flow of artifacts of the process which can be a system component or information gained by one process required by another one. The black ellipses annotate the chapter or section of this document in which the labeled processes are described in detail.

Through the whole document, a unique color scheme is used to represent certain components and artifacts. Hardware, hardware components and hardware models are colored in purple. Software, software components and software models are colored in magenta. Delay information for both, hardware or software as well as the overall system execution time are colored in green. The simulation and simulation processes or simulation components are colored in blue. Data and data dependency related aspects are colored in brown. Other artifacts or processes are colored in gray.

The workflow (Fig. 1.1) starts with the software which shall be deployed on a product and a set of evaluation platforms that provide execution platforms with features and instruction set architectures that are candidates for the final product hardware. The MPSoCs on the evaluation platforms can be less powerful (Lower clock frequency, less processing elements) and therefore less expensive than the MPSoC that is later required for the product, because the evaluation platforms are only used for characterizing small parts of the software that shall be deployed later. Anyway all features like a floating point unit and instruction set architectures that shall be considered for design space exploration must be available and supported for the characterization process.

The given software and all MPSoCs of all evaluation platforms will be modeled. These models are an abstract representation of the software and the execution platforms of the evaluation hardware with the purpose of fast execution time analysis for a design space exploration. For the software, the Synchronous Data Flow model (SDF) [LM87] is used which allows to isolated independent parts of the software, and models the data flow between those independent parts. For the hardware, a tile based model as proposed by [Fak+15] is used, which allows to represent multi processor system on chips (MPSoCs) as a set of independent tiles. These models allow a composable representation of sub components of hardware and software. The modeling process is described in detail in Chap 3. The models are used in the simulation for configuring the simulated platform.

To be able to do an execution time analysis, delay information need to be annotated to the models. These delays are collected during the characterization process. Therefore each part of the software, corresponding to its model, gets executed on each kind of execution platform. This is done by extending the hardware and software with measurement infrastructure. This measurement infrastructure then provides observed execution times. To characterize the communication between software components, and therefore the inter-processor communication of the considered MPSoCs, the system bus behavior gets modeled in detail. Details of the characterization are described in Sec 4.2.

The models of the software and execution platforms, together with the characterized delay information are then given into a simulation process. This process starts with mapping software components on a possible execution platform that will be simulated. The execution platform can be configured in any way, as long as only features and instruction set architectures are used that have been modeled and characterized. Then the overall execution time of the software executed on such a hardware is simulated. The result of the simulation is a distribution of simulated execution times. This process can be repeated until a suitable hardware configuration and mapping has been found. The simulation is described in detail in Sec 4.3. Beside using the simulation for execution time analysis, functional simulation is also possible for testing the application code. Functional tests are not focus of this work. Still this feature is used to verify experiment setups.

The output of the simulations of potential execution platforms are information of platform configurations and mappings of the software components on those platforms. This information supports the designer of the hardware/software system to decide which hardware to build or buy, and how to map the software in an efficient way on that hardware.

For evaluation (Chap 5) the measurement infrastructure is used for measuring the overall execution time of the software executed on a certain hardware configuration. This actual observed execution time execution time distribution is then used to compare the execution time distribution predicted by the simulation.

1.6 Structure

The document continues with presenting and discussing related work in Chap 2. In Chap 3 the software, hardware and delay models used in this thesis are presented. Chap 4 presents the characterization and analysis methods and processes for annotating delay information to the models and how they are used within the simulation. In Chap 5 the presented models and simulation are evaluated and discussed. The overall concept is discussed in Chap 6. Chap 7 concludes this thesis and provides answers to the research questions from Sec 1.2. Chap 8 presents future work and potential applications.

Chapter 2
Related Work

2.1 Overview of Analysis Approaches

Timing analysis approaches are commonly (e.g. [Gaj+09]) classified as:

1. Simulation-based approaches, which partially test system properties based on a limited set of stimuli,
2. Formal approaches, which statically check system properties in an exhaustive way, and
3. Hybrid approaches, which combine simulation-based and formal approaches.

In the following sections related work following these approaches are presented and discussed. The chapter ends with discussing different execution time estimation techniques.

2.2 Simulation Based Approaches

Different simulation-based approaches exist to evaluate multi-processor architecture performance early in the design process. In the proposed approaches, models of hardware-software systems are formed by combining an application model and a platform model. In the early design phase, a full description of application functionalities is not mandatory and workload models of the application are used. A workload model expresses the computation and communication loads like time or power consumption that an application causes when executed on platform resources. Such performance models are then generated as executable descriptions and simulated. The execution time of each load primitive is approximated as a delay, which is typically estimated from measurements on real prototypes or analysis of low level simulations. SystemCoDesigner [Kei+09], Daedalus [Erb+07], SCE [Döm+08], Palladio [BKR07], and Koski [Kan+06] are good examples of academic approaches. They are compared in [Ger+09]. Other existing academic approaches are presented by Kreku et al. in [Kre+08] and by Arpinen et al. in [Arp+09]. Furthermore, some industrial frameworks such as Intel CoFluent Studio [Int], Timing Architect [Arc], ChronSIM [Chr], TraceAnalyzer [Tra], Space Codesign [Spa] and Visualsim from Mirabilis Design [Mir] have emerged as well.

In [LMS13] Lu et al. address resource conflict based timing accuracy problem in temporally decoupled transaction level simulation by an analytical approach. Delay formulas are proposed according to shared resource usage and availability. They are then used during simulation to adapt the end of each synchronization request. The presented communication models in this thesis are sensitive to execution delay of the communication participants. Thus my contention delay model can dynamically adapt to communication changes based on the computation delay of the communication participants.

Simulation-based approaches require extensive architecture analysis under various possible working scenarios. Furthermore, created system models can hardly be exhaustively simulated. Due to insufficient corner case coverage, simulation-based approaches are thus limited to determine guaranteed limits about system properties. One other important issue concerns the accuracy of created models. As architecture components are modeled as abstractions of low level details, there is no guarantee that the created architecture model reflects with good accuracy the whole system performance. Finally, with the rising complexity of many-core platforms, execution of simulation models also increase simulation time.

In my work I follow a hybrid approach by using an analytical communication model and a probabilistic computation model within a simulation. For my approach, lack of guarantees is not a concern because the simulation is used for design space exploration, not for safety analysis. Scalability regarding simulation time and accuracy have been investigated in the evaluation chapter (See Sec 5.6 of this thesis).

2.3 Formal Approaches

Due to insufficient corner case coverage, simulation-based approaches are insufficient to determine guaranteed limits about system properties. Different formal approaches have thus been proposed to analyze MPSoCs and provide hard real-time and performance bounds. These formal approaches are commonly classified as state-based approaches and analytical approaches.

Most of the available static real-time methods are of analytical nature (c.f. [Per+09] for an overview). Since analytical methods depend on solving closed-form equations (characterizing the system temporal behavior), they have the advantage of being scalable to analyze large-scale systems. MPA-RTC [Hua+12] and SymTA/S [Hen+05] are two representatives of compositional analytical methods for multi-core systems.

Despite their advantages of being scalable, analytical methods abstract from state-based operation mode of the analyzed system such as complex state-based arbitration protocols or inter-processor communication task dependencies. This leads to pessimistic over-approximated results compared to state-based methods [Per+09]. Many recent approaches for the software timing analysis on many- and multi-core architectures are built on state-based analysis techniques. The two main considered application classes are data flow based applications, for example modeled as synchronous data flow graphs (SDFGs), [KW16; Ske+15; Zhu+15; TS15; Ahm+14; MG13; Yan+10] and generic real-time task-based applications [NWY99; HV06; Lv+10; Gus+10; Gia+12; BHM08; Zha11; Bük13]. State-based real-time methods are based on the fact of representing the analyzed system as a transition system (states and transitions). Because the real operation states of the hardware and software behavior are reflected, tighter results can be obtained compared to analytical methods. However, state-based approaches allow exhaustive analysis of system properties at the expense of time-consuming modeling and analysis effort.

In [Fak+15], the adoption of model-checking for real-time analysis of SDFGs running on MPSoCs with shared communication resources is presented. Especially, it is highlighted in [Fak+15] that state-based approaches can hardly address systems made of high number of heterogeneous components. This approach utilizes timed automata to represent tasks and arbitration protocols for buses and memories. It uses Best- (BCET) and Worst-Case Execution Time (WCET) intervals as lower and upper bounds of the estimated execution time of a specific platform.

Since the approach I propose in this thesis requires models of multiple MPSoCs, high modeling effort is a downside. Furthermore the trade-off between analysis time, modeling effort (abstraction) and accuracy make formal approaches more suitable for safety analysis of single hardware-software systems, and less interesting for design space estimation. The approach presented in this work is a hybrid approach with a probabilistic computation model combined with a formal state based communication model. However, the pure formal approach of Fakih [Fak+15] was a starting point for the simulation based hybrid approach presented in this thesis. A hybrid approach using a statistical model checker has been used to evaluate the idea of an analysis considering distribution functions of measured delays [Ste+19c].

2.4 Hybrid Probabilistic Methods

Probabilistic models are frequently adopted to model systems where uncertainty and variability need to be considered. In the context of embedded systems, probabilistic models represent a means of capturing system variability coming from system sensitivity to environment and low level effects of hardware platforms. Probabilistic models like discrete time Markov chains or Markov automata can be used to appropriately capture this variability. Probabilistic models are extensions of labeled transition system and allow variations about execution times and state transitions to be considered. Quantitative analysis of probabilistic models can be used to quantify to what extent a given time property is satisfied. Numerical approaches exist that compute the exact measure of the probability at the expense of time-consuming analysis effort. As an illustration, the adoption of probabilistic model checking for evaluation of dynamic data-flow behaviors is presented in [KW16]. Markov automata are used as the fundamental probabilistic model to capture and analyze architectures. Characteristics as application buffer occupancy, timing performance and platform energy consumption are estimated. However, this approach is restricted to fully predictable platforms with low influence of platform resources on timing variations.

Another approach to analyze probabilistic models is to simulate the model for many runs and monitor simulations to approximate the probability that time properties are met. Statistical Model Checking (SMC) has been proposed as an alternative to numerical approaches to avoid an exhaustive exploration of the state-space model. SMC refers to a series of techniques that are used to explore a sub-part of the state-space and provides an estimation about the probability that a given property is satisfied. SMC designates a set of statistical techniques that present the following advantages:

- As classical model checking approach, SMC is based on a formal semantic of systems that allows to reason on behavioral properties. SMC is used to answer qualitative questions (*Is the probability for a model to satisfy a given property greater or equal to a certain threshold?*) and quantitative questions (*What is the probability for a model to satisfy a given property?*).
- It simply requires an executable model of the system that can be simulated and checked against state-based properties expressed in temporal logics. The observed executions are processed to decide with some confidence whether the system satisfies a given property.
- As a simulation-based approach, it is less memory and time intensive than exhaustive approaches.

Various probabilistic model-checkers support statistical model-checking, as for example UPPAAL-SMC [Dav+11], Prism [KNP11], and Plasma-Lab [JLS12]. This approach has been considered in various application domains [Bul+12].

The usage of UPPAAL-SMC to optimize task allocation and scheduling on multi-processor platform is presented in [Che+15]. Application tasks and processing elements are captured in a Network of Price Timed Automata (NPTA). It is assumed that each task execution time follows a Gaussian distribution.

Authors in [Caz+13a; Caz+16] propose a measurement-based approach in combination with hardware and/or software randomization techniques. Their goal is to conduct a probabilistic worst-case execution time (pWCET) through the application of Extreme Value Theory (EVT). While I in contrast aim for predicting the distribution of possible execution times aiming towards a high accurate prediction of the average execution time.

An iterative probabilistic approach has been presented by Kumar [Kum09] to model the resource contention together with stochastic task execution times to provide estimates for the throughput of SDF applications on multiprocessor systems.

In [Nou+14b] the integration of SMC methods in a system-level verification approach is presented. It corresponds to a stochastic extension of the BIP formalism and associated tool set [Bas+11]. An SMC engine is presented to sample and control simulation execution in order to decide if the system model satisfies a given property. The preparation process of time annotations in system model is presented in [Nou+14a] where a statistical inference process is proposed to capture low-level platform effects on application execution. A many-core platform running an image recognition application is considered and stochastic extension of BIP is then used to evaluate the application execution time.

A solution is presented in [NLQ16] to apply SMC analysis methods for systems modeled in SystemC. The execution traces of the analyzed model are monitored and a statistical model checker is used to verify temporal properties. The monitor is automatically generated based on a given set of variables to be observed. The statistical model-checker is implemented as a plugin of the Plasma-Lab [Boy+13].

As I pointed out in [Ste+21], [Nou+14a] et al. propose a statistical inference process to capture low level platform effects on computation execution time. Based on multiple runs of a studied application on the targeted true platform, the Distribution Fitting method [Le 10] is adopted to statistically learn the best distribution that fits the observed data. Created probabilistic models are then simulated in conjunction with statistical model checking methods to estimate probability that timing properties are met. However, the execution times of the multiple paths in software are not identified. Moreover, influence of shared communication resources is not explicitly separated from computation execution time. In [BPT10], Bobrek et al. propose an approach to raise the abstraction level of simulation while still estimating contention at shared resources such as memories and buses. It interleaves an event-driven simulation to coarsely capture processor execution overhead with an analytical stochastic model used to estimate contention for shared resources.

In my work I follow a formal state based approach for the communication model. The communication is modeled on instruction level which allows to consider computation time (execution of instruction that do not interfere with shared resourced) within the communication process that does interfere with shared resources. For the computation model I follow an abstract probabilistic approach by describing the execution time of functional units as probability density function. I combine simulation with the analytical expression of shared resource effects on communication. This approach allows time-consuming simulation events to be significantly reduced while maintaining a good level of accuracy.

Tab. 2.1 compares related simulation based method that follow a hybrid and probabilistic approach. It shows the related work that use hybrid probabilistic methods in the first column. The other columns compare these approaches against the approach I follow in this thesis for computation and communication models separately. For the computation model, related work is compared regarding using a probabilistic model while for communication it is compared regarding using a state based analytical model. Empty circles (○) indicate that a different approach has been used. Filled circles (●) indicate a similar approach and half filled circles (◓) an approach that follows a similar concept in a different direction.

The approches [Che+15; Caz+16; Kum09; KW16] use no [Caz+16], or very abstract distribution functions to like Gass distribution [Che+15] model computation

Table 2.1 Classification of related simulation based hybrid probabilistic methods.

Related work	Probabilistic Computation Model	Low Level State Based Communication Model
Bobrek et al. [BPT10]	○	○
Cazorla et al. [Caz+16]	◓	○
Kumar et al. [Kum09]	◓	◓
Chen et al. [Che+15]	◓	◓
Katoen et al. [KW16]	◓	◓
Nouri et al. [Nou+14a]	●	○
This work	●	●

delays. Or they focus on predicting the probability of a certain worst case execution time (pWCET)[KW16; Caz+16]. The approaches [BPT10; Nou+14a] focusing on a probabilistic communication model. While [BPT10] only comes with a state based approach to model computation delay. In contrast to my work, all use very high level communication models while I propose an instruction level state based model. The communication models of [Che+15; Kum09; KW16] in contrast use abstract high level communication models.

2.5 Execution Time Estimation

All of the above mentioned analysis approaches and the approach presented in this thesis, rely on the estimation of execution times. For the simulation-based and some probabilistic approaches, the estimation can be performed through a detailed instruction set architecture simulation model at instruction- or even cycle-accurate level. This involves very time consuming simulations and belongs to the class of measurement-based execution time analysis techniques, that have to deal with the rare-event problem[1]. Another technique is timing back-annotation to functional or analytical system models, applying timing measurement or static timing analysis approaches. For most real-time systems, the Worst-Case Execution Time (WCET), which is a safe upper bound of all observable execution times, is the most relevant metric. Sometimes also the Best-Case Execution Time (BCET) is considered in combination with WCET in a [BCET, WCET] interval. Static timing analysis [Wil+10] is the state-of-the-art technique to determine the WCET and BCET respectively. With the adoption of today's MPSoCs, limitations of static timing analysis are becoming apparent [Cul+10].

Source-level simulation of software has been proposed to preserve simulation accuracy [Bri+15]. The source code of the software is annotated with timing extracted from machine code analysis.

---

[1] Rare events are events that occur with low frequency but which have potentially widespread impact, e.g. on the execution time.

For modern multi-processor architectures, measurement-based approaches can overcome some problems of static timing analysis [Kir+05]. But this comes at a price, since measurement-based timing analysis is facing the rare-event problem, its analysis results are either largely over-approximated (using a large safety margin) or untrustworthy (due to a missing probability analysis). To overcome these challenges for measurement-based timing analysis for modern processors and multi-core systems, the work in the context of the projects PROARTIS[2] [Caz+13b] and PROXIMA[3] [Caz+16] propose a new way for timing analysis. Instead of applying improved static analysis, a measurement based approach in combination with hardware and/or software randomization techniques to conduct a probabilistic worst-case execution time (pWCET) through the application of extreme value theory (EVT) has been proposed. Both projects have successfully assessed that measurement based execution time analysis with the combination of EVT can be used to construct a worst-case probability distribution.

In my work I apply measurement based delay analysis of restricted and well analyzable Models of Computation (MoCs). I use measured execution time and probabilistic inference methods to estimate execution time variability in data-dependent software. The Kernel Density Estimation (KDE) [Ros56; Par62] method is adopted to infer distribution laws based on a limited set of evaluated execution paths. The proposed approach allows getting a distribution of possible execution times as result, not only a best, average or worst case execution time. For fast simulation, computation is abstracted to a distribution of delays instead of relying on instruction accurate simulation.

---

[2] Probabilistically Analysable Real-Time Systems (`http://www.proartis-project.eu`)

[3] Probabilistic real-time control of mixed-criticality multicore and manycore systems (`http://www.proxima-project.eu`)

Chapter 3

Software, Hardware and Delay Models

3.1 Model of Computation

I use Synchronous Data Flow (SDF) [LM87] as computation model (MoC). SDF is used to describe the data flow between *actors* via communication *channels*. SDF allows a strict separation of computation (*compute*) and communication phases (*read*, *write*) of actors. A simple example of an SDF graph is shown in Fig. 3.1. The formal syntax of an SDF graph (SDFG) as it is used in this thesis gets defined in this section.

Definition 3.1 An *SDF graph* (SDFG) is a tuple $G = (\mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{B}, \gamma)$ where $\mathcal{A}$ is a finite non-empty set of *Actors* A that represent the computational part of an SDF based application. $\mathcal{C}$ is a finite set of *Channels* C which represent the communication between actors. $\mathcal{P}$ is a finite set of ports that connects actors with channels and define the data rate for communication. $\mathcal{B}$ is a finite set of buffers that store data which is transferred between actors. The repetition vector γ describes the execution order of the actors to execute a full iteration of the SDF graph.

The strict separation of computation and communication phases of SDF applications allow to distinguish between computation delay models and communication delay models later presented in this thesis in Sec 3.3.

3.1.1 Buffers

To maintain distinction of computation and communication when implementing and mapping a SDF application on hardware, it is important to define the ownership of buffers.

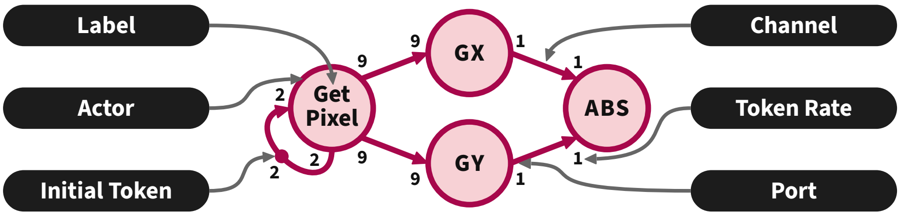


Fig. 3.1 Example of a synchronous data flow graph

During computation, Actors may only access private memory that is accessible without interference with other Actors. During communication, data that got produced by one Actor needs to be accessible by a consuming Actor. Therefore shared memory is required.

Definition 3.2 A *Buffer* $B^X \in \mathcal{B}$ is defined as a tuple $B^X = (Size, Address, State)$. The size $Size \in \mathbb{N}_{>0}$ defines how many tokens t can be stored in the buffer. The position of the buffer in memory (Sec 3.2) is given by the attribute $Address \in \mathbb{N}_0$. The $State \in \{\text{full}, \text{empty}\}$ says if tokens can be written into the buffer ($State = \text{empty}$) or read from the buffer ($State = \text{full}$). $X \in \{\text{P}, \text{S}\}$ defines if a buffer is considered a private buffer B^P so that it can only be accessed by one actor A, or if it is a shared buffer B^S that can be accessed by two actors. One that writes to it, and one that reads.

Definition 3.3 A *Token* $t \in \vec{t}$ is an abstract representation of data. A token vector $\vec{t}$ describes an ordered series of tokens. In context of software code also called *data array*.

Assumption 1. *All tokens requires the same amount of memory to store the data it represents. The number of bits of a toke is the same amount of bits of a data word of the hardware architecture the modeled software is executed on.*

Buffer access follows the First In First Out (FIFO) principle.

Assumption 2. *The FIFO access is blocking. An actor can only switch to its compute phase after it read all tokens from all incoming channels.*

3.1.2 Actors

Definition 3.4 An *Actor* $A_f \in \mathcal{A}$ represents an inference free functional part f of a software. An actor is a tuple $A_f = (\vec{P}_{\text{In}}, \vec{P}_{\text{Out}}, f)$ consisting of a finite set of input and output ports $\vec{P}_{\text{In}}, \vec{P}_{\text{Out}} \subseteq \{\mathcal{P}, \emptyset\}$, and f as label representing the functionality of the actor.

An actor A_f represents an atomic behavior (software function f) of a software application that gets input data, processes it without interfering with other actors and provides the result as output. Data is abstracted as *token* $t \in \vec{t}$. Actors have no hidden internal state. If the execution of an actor depends on its previous execution, it communicates this state to itself via a feedback communication channel as it is the case for the example actor $A_{GetPixel}$ in Fig. 3.1.

An actor consists of up to three different execution phases: A computation phase (*compute*) and two communication phases (*read*, *write*).

During the read phase of an actor, each input port of the input port vector $\vec{P}_{\text{in}}$ of an actor reads the input data $\vec{t}_x$ from a shared buffer B^S_x of a channel C_x into the private buffer $B^\text{P}_{p_i}$ of port i. This is done sequentially for each input port.

The input data stored in the private buffers of the input ports will then be processed by the function f of the actor during the compute phase. The results get stored in the private buffers of the output ports.

During the write phase, these output port buffers are then written into the shared buffers of the channels that are associated to the output port similar to the read phase

An actor can only switch to its compute phase after all required tokens have been read from all incoming channels. After the compute phase, the actor switches into write phase to write all tokens to the outgoing channels.

While each actor has an execution phase, the read and write phases are optional. Actors that only generate data without having any input are called *Producer Actor*. Actors that only read and process input data without having any output are called *Consumer Actor*.

For example, a producer actor can be a software function that gets data from a hardware sensor. An example for a consumer actor can be a software function that sets an actuator. In both cases, the behavior of the sensor or the actuator are part of the actor model with respect to functionality as well as execution time.

Assumption 3. *During the compute phase, no interference with any other actor can occur.*

The hardware used to execute an actor must support Assumption 3. For MPSoCs this separation only works when the instructions are placed on separated memories, connected with separate interconnects to the processing element as it is realized in tile-based hardware platforms. Details are described in the Model of Architecture section (Sec 3.2),

3.1.3 Ports

A *Port* is the interface between an actor A_x and a channel C_y. Ports are used to transfer tokens from a local buffer B_i^{P} of a port p_i of an actor into a shared buffer B_y^{S} of a channel C_y. The amount of tokens that get transferred is defined by the port rate $Rate$. Ports have a direction Dir and can either read (input ports) or write (output ports) to a channel.

Definition 3.5 A *Port* $p \in \mathcal{P}$ is a tuple $p = (Dir, Rate, B^{\mathrm{P}})$ where $Dir \in \{\mathrm{In}, \mathrm{Out}\}$ defines if it is an input or output port. The port rate $Rate \in \mathbb{N}_{>0}$ defines how many tokens are transferred by the port during one read or write phase. Each port has a local buffer $B^{\mathrm{P}} \in \mathcal{B}$ to store tokens that have been read from a corresponding channel or tokens that are ready to be written to a channel.

Following sections will refer to the implementation of the port behavior as `ReadTokens` function for input ports and `WriteTokens` function for output ports.

These functions implement the behavior or copying data between local buffers abstracted as ports and shared buffers abstracted as channels.

During the read phase of an actor, all input ports of that actor read tokens from there associated channels. The token rate defines how many tokens are read. During the write phase, all output ports write their tokens into their associated channels.

Reading and writing is a blocking process. Ports are processed in order. The reading process is blocked until all tokens, depending on the token rate, have been read from a port. Then tokens from the next port of an actor are read. The writing process is done in a similar manner.

3.1.4 Channels

Channels are used to communicate data between two actors. Channels represent shared FIFO buffers (B^{S}) that can be mapped to any memory that is accessible by the output port of a producer (an actor that writes on that channel) and the input port of a consumer (an actor that reads from that channel). During the read phase of an actor, tokens get read from the channels buffer. During the write phase, tokens get written into the channel. Each port of all actors is connected to exactly one channel, and all channels are connected to one input port and one output port.

Definition 3.6 $\mathcal{C} \subseteq \mathcal{P} \times \mathcal{P}$ is a set of channels (represented as edges in the visual syntax of SDF). A *Channel* $C \in \mathcal{C}$ is a tuple $C = (B^{\mathrm{S}}, \vec{t}_{\mathrm{Init}})$ where $B^{\mathrm{S}} \in \mathcal{B}$ is a buffer to store tokens that get written into the channel C. Channels connect actors via ports. A channel may be initialized with initial tokens $\vec{t}_{\mathrm{Init}}$. A shared buffer B_x^{S} referrers to the token buffer of the channel $C_x \in \mathcal{C}$.

Initial tokens $\vec{t}_{\mathrm{Init}}$ are written into shared memory of a channel once before executing an SDF application. They are used to solve deadlocks by feedback loops between one or multiple actors.

3.2 Model of Architecture

The Model of Architecture (MoA) is based on the definition from [Fak+15] which I have updated and generalized in [Ste+17; Ste+19c; Ste+19a]. This section presents the tile based MPSoC architecture model used in this thesis with a detailed view on processing element interfaces and considering heterogeneous instruction set architectures (ISA). A simple example of such a platform is shown in Fig. 3.2.

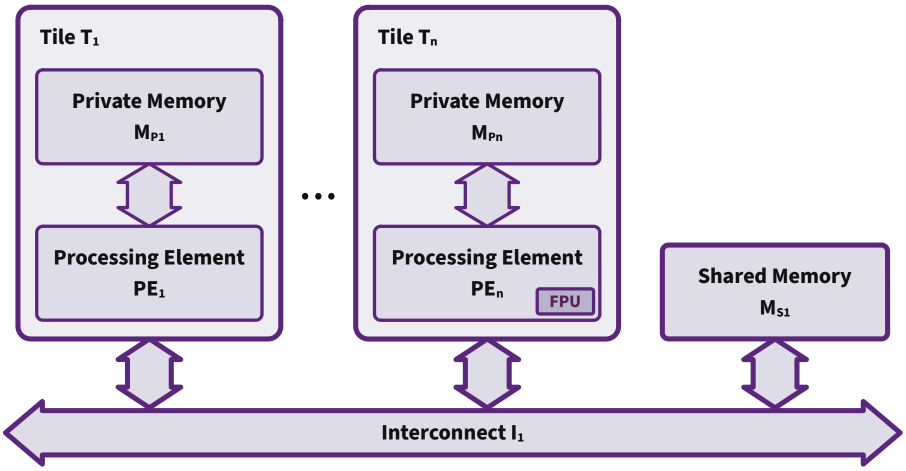


Fig. 3.2 Example of an execution platform with n tiles. All tiles are connected to a shared memory via an interconnect.

3.2.1 Execution Platform

On a very abstract view on an embedded multi processor system, these execution platforms (EP) consists of following hardware components:

- A set of processing elements ($\mathcal{PE}$) that execute the instructions of a software.
- A set of memory ($\mathcal{M}$) that stores data and instructions.
- A set of interconnects ($\mathcal{I}$) that connects the processing elements with memory.

To be able to execude software components without interfering with other software components (See Assumption 3), software needs to be executed on isolated tiles ($\mathcal{T}$) with instructions and local data stored in private memory ($\mathcal{M}_P$).

Definition 3.7 An *Execution Platform* EP contains all hardware components needed to execute software. It is defined as $EP \subseteq \mathcal{T} \times \mathcal{I} \times \mathcal{M}_S$, where $\mathcal{T}$ is a finite non-empty set of Tiles on that code gets executed in isolation, $\mathcal{M}_S$ is a finite set of shared memory and $\mathcal{I}$ is a finite set of interconnects that connect tiles ($T \in \mathcal{T}$) with each other and with shared memory ($M_S \in \mathcal{M}_S$).

There are some basic assumptions that apply to each component inside the execution platform:

A word processed on a processing element has the same size as a word transferred by an interconnect and the same size a word inside a memory has. Overhead like start or stop bits is neglected by the model.

Assumption 4. *The data word size in bits is the same for each component.*

Beside the word length, also the clock frequency needs to be the same for each component. So for each component, one clock cycle takes the same amount of time.

Considering dynamic voltage and frequency scaling as well as different clock domains are not subject of this work[1].

Assumption 5. *The clock frequency of each component is the same. The clock frequency does not change over time.*

3.2.2 Tile

To make the execution platform composable with respect to time (Assumption 3) an environment needs to be created that allows multiple processing elements executing software without interfering with each other. A tile ensures this composability by isolating a processing element with its own private memory from interaction with shared resources during the execution of instructions. This is limited to code that only works on local data. Interference with other tiles can happen when the code accesses shared resources.

Definition 3.8 A *Tile* is a single processing element $PE \in \mathcal{PE}$ with a private memory $M_P \in \mathcal{M}_P$. It is defined as a tuple $T = (PE, M_P)$. Where $PE \in \mathcal{PE}$ is a processing element with a specific instruction set architecture and feature set (like a Floating Point Unit) and M_P is private memory connected to the processing element via dedicated peer-to-peer connection. Tiles have only a single processing element and one private memory.

Assumption 6. *A tile represents an isolated processing environment that does not interfere in any way with other tiles or execution platform components, as long as no shared resources are accessed. Instructions and local data are stored on private memory only.*

3.2.3 Processing Element

A processing element $PE \in \mathcal{PE}$ implements an instruction set architecture (ISA). Software instructions mapped to a processing element will be executed by it. Different processing elements can implement different ISAs or come with different ISA features like a floating point unit (FPU) or hardware multiplier (MUL).

Definition 3.9 A *Processing Element* $PE \in \mathcal{PE}$ is a tuple $PE = (ISA, \mathcal{F})$ Where ISA references to the instruction set architecture implemented by the processing element. $\mathcal{F}$ is a set of optional ISA features and accelerators enabled for this processing element.

---

[1] Applying clock gating to adapted models has been evaluated by Oliver Klemp within his master thesis supervised by Maher Fakih and me. Published in [Kle+19]

An example of a processing element implementing the *MicroBlaze* architecture [Xil21] with enabled FPU and hardware multiplier is $PE = (\text{MicroBlaze}, \{\text{FPU}, \text{MUL}\})$.

3.2.4 Memory

In context of this thesis, memory is used for two different purposes. There is private memory $\mathcal{M}_P$ and shared memory $\mathcal{M}_S$. For each processing element $PE_i \in \mathcal{PE}$ there exists a dedicated memory $M_i \in \mathcal{M}_P$ called private memory. This private memory can only be accessed by its associated processing element.

A shared memory $M_k \in \mathcal{M}_S$ is accessible by multiple processing elements $\mathcal{PE}_k \subseteq \mathcal{PE}$ This memory is used for data exchange between multiple tiles.

Definition 3.10 The set of memories $\mathcal{M} = \mathcal{M}_P \cup \mathcal{M}_S$ represents all memories of an execution platform EP. A *Memory* $M \in \mathcal{M}$ is defined as tuple $M = (S, d_r, d_w)$ where $S \in \mathbb{N}_{>0}$ is the memory size in bytes. d_r the delay (or access time) to read from the memory and d_w the delay for write access.
Memory delay $d_{r,w}$ is defined in Sec 3.3

Assumption 7. *The memory access delay d_r and d_w are assumed being constant. This assumption is valid for static memory (SRAM). This implies the absence of additional caches and in-memory-logic which may lead to variance in memory access time.*

Assumption 8. *For data exchange between different tiles shared memory is used. Shared memory is connected via a shared interconnect to the processing elements.*

3.2.5 Interconnect

Interconnects are used to connect tiles with shared memory. This allows processing elements of tiles to exchange data with processing elements of other tiles.

Definition 3.11 An *Interconnect* $I \in \mathcal{I}$ connects Tiles $\mathcal{T}$ with shared memory $\mathcal{M}_S$. Where $\mathcal{I} \subseteq \mathcal{T} \times \mathcal{M}_S$ is a shared interconnect.

Assumption 9. *It is assumed that for each word that gets transferred the interconnect needs to be arbitrated.*

Assumption 10. *A First Come First Serve arbitration protocol is assumed for all interconnects.*

3.3 Delay Model

In context of this thesis, time is the primary metric used to characterize the performance of hardware and software. Relative time is denoted as *delay* that represents the amount of clock cycles an activity takes between start and finish. Using clock cycles instead of time in seconds makes the model independent from the actual clock frequency.

3.3.1 Delay

Definition 3.12 A *Delay* $d_x \in \mathbb{N}_0$ represents the amount of clock cycles required to perform an action x.

Assumption 11. *All clock cycles used inside a model are derived from the same global clock. Each clock cycle represents the same amount of time. There is no jitter between clock cycles. The clock runs with a steady and constant frequency.*

Delay can be represented as single value or as a vector of multiple delays. Depending on the complexity of hardware or software components the same functional behavior can lead to different delays. In this case the probability for a certain delay to occur needs to be modeled. This is described in Sec 3.3.2. An example where a single delay is enough to describe timing behavior is access time of local memory:

Definition 3.13 As defined in Def. 3.10, each memory has a read access delay d_r and a write access delay d_w.
$M.d_{r,w} \in \mathbb{N}_0 \, \forall \, M \in \mathcal{M}_P$ and $M.d_{r,w} \in \mathbb{N}_{>0} \, \forall \, M \in \mathcal{M}_S$.
The delay is given in clock cycles per data word.

As defined in definition 3.13 private memory can have, but is not limited to, an access time of $d_{r,w} = 0 \, \forall \, M_P \in \mathcal{M}_P$ clock cycles. A common use case for a memory access time are instruction memories where the actual access time is part of the instruction execution time (Usually called *instruction fetch phase*). To avoid having the instruction fetch delay twice, in the memory access model and in the execution delay model, setting the memory access time to $d_r = 0$ is necessary. Following Assumption 7, for all memories static RAM and no caches have been used.

3.3.2 Modeling Delay Occurrence Probability

Delay inside an embedded system can vary for many reasons. For example low level effects like Pipelining of code execution in a processing element or resource contention on shared hardware components. On higher level, the execution of software can vary in time because of different data dependent execution paths. Therefore a single delay may not always be sufficiently represent the time behavior of hardware or software component.

To be able to consider different possible delays d_x for the same action x, a vector of delays $\vec{d}_x$ is used.

Definition 3.14 A *Delay Vector* $\vec{d}_x$ contains multiple possible delays $[d_{x,0}, \ldots, d_{x,n}]$ for an action x. The number of times the same delay $d_{x,i}$ occurs inside the delay vector determines its probability of occurrence [Mar63].

The distribution of delays in a delay vector $\vec{d}_x$ represent the observed distribution of delay performing an activity x took. Either as record of observed delay or as set of delays following a distribution function after applying inference techniques on observed delays.

Delays are stored in a delay vector in a way that their occurrence in the delay vector represent their observed or calculated probability. The probability of a delay d_i to occur is represented by the occurrence of that delay d_i in the delay vector $\vec{d}_x$. So when one delay exists only once in the delay vector, it occurs with the probability of $1/|\vec{d}_x|$. If the same value occurs twice, it comes with the probability of $2/|\vec{d}_x|$. A delay d_a that occur twice as often as different delay d_b is also stored twice as often in the delay vector: $\vec{d} = [d_a, d_a, d_b]$. Its probability to occur is $P(d_a) = 2/3$.

Definition 3.15 The probability $P(d_i)$ of a delay $d_i \in \vec{d}_x$ follows the probability density function of the observed delays of an action x:
$P_{\text{Distribution}}(d_i) \approx P_{\text{Observed}}(d_i)\ \forall d_i \in \vec{d}_x$
Where Distribution represents the distribution function used to abstract the observed distribution of delays.

3.3.3 Delay Distribution Inference

There are several ways to address the variation of delays of an observed system. The most easy way is to calculate the average of all observed delays. Often the whole range of possible delays must be considered. In this case a uniform distribution of delays between the *Best Observed Execution Time* (BOET) and the *Worst Observed Execution Time* (WOET) can be a valid abstraction. Distribution functions can also be used to extrapolate the observed delays to get a safety margin to the BOET and WOET. For a more accurate delay model more sophisticated distribution functions can be used to fit the observed distribution of delays. For a general way to abstract the distribution of delays based on observation, Kernel Density Estimation (KDE) [Ros56; Par62] can be applied to the observed delays.

A measurement infrastructure like the one used for this work which is described in Sec 4.1.2 provides delay samples of an observed actor. The series of samples are called Delay Vector $\vec{d}_S$. These samples can easily be used to obtain best and worst observed

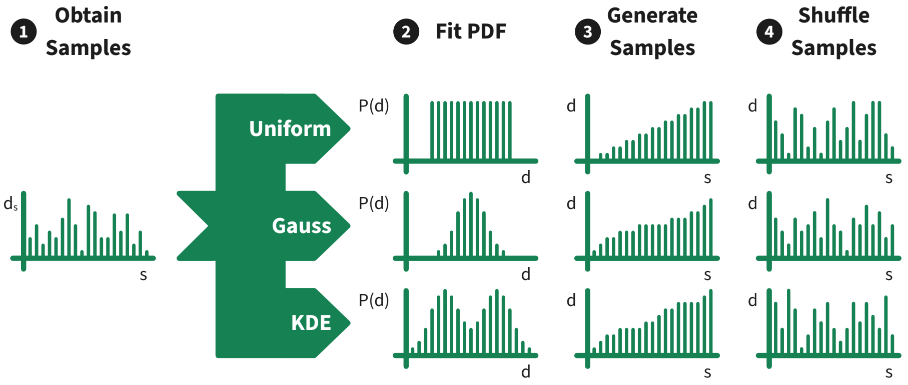


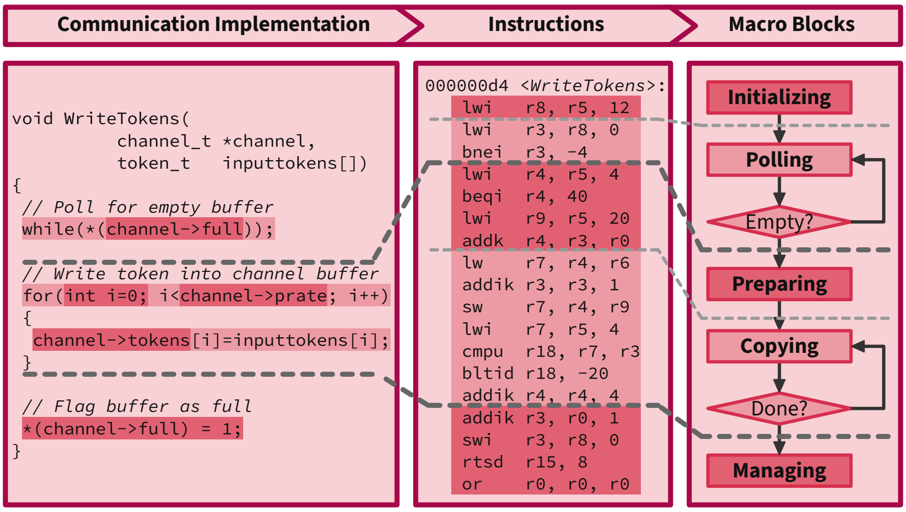
Fig. 3.3 Workflow of generating a vector of random delays following a specific distribution function.

execution time as well as the average observed execution time of that actor. Because the observed behavior does not cover all possible delays, different post-processing steps can be applied to the samples to get a more general description of the timing characteristics of an actor. To abstract how the actor execution time can be in general, a distribution function needs to be fit to the observed occurrence of measured delays.

In Fig. 3.3 this process is shown with an example where from observed samples ① a more generalized delay vector gets derived. This is done by fitting a probability distribution function to the observed delay vector (②). Simple distribution functions can be an uniform distribution interpolating between worst and best observed delay. Or a Gauss distribution that focus more on the average observed delay but also covers rare delays lower than the best, and higher than the worst observed delay. For software this can be a very error prone abstraction process especially with the goal to analyze the distribution of delays of the overall system. The challenge is to represent data dependency. Different branches of a software function can lead to very different delays. A more sophisticated method to obtain a distribution function based on measured data can be Kernel Density Estimation [Par62; Ros56].

The Kernel Density Estimation assumes for each sample $d_i \in \vec{d}_S$ a weighted distribution function $P_{\text{kernel}}(d_i)$ - the *kernel*. In this thesis a Gauss distribution is used as kernel. The sum of these kernels results in a smooth distribution function representing the whole data set.

From the distribution function, a new delay vector gets generated Fig. 3.3 ③. This new delay vector provides delays following the desired distribution function. To use the derived delay vector for analysis, the elements of the vector get shuffled to achieve a random order ④. This is important to not just use all best case delays of all actors at the begin of a simulation and all worst case delays at the end. A combination of a better case and a worse case for a partial actor delay could lead to a worst case for the overall execution time.



```
void WriteTokens(
          channel_t *channel,
          token_t   inputtokens[])
{
 // Poll for empty buffer
 while(*(channel->full));

 // Write token into channel buffer
 for(int i=0; i<channel->prate; i++)
 {
  channel->tokens[i]=inputtokens[i];
 }

 // Flag buffer as full
 *(channel->full) = 1;
}
```

```
000000d4 <WriteTokens>:
   lwi   r8, r5, 12
   lwi   r3, r8, 0
   bnei  r3, -4
   lwi   r4, r5, 4
   beqi  r4, 40
   lwi   r9, r5, 20
   addk  r4, r3, r0
   lw    r7, r4, r6
   addik r3, r3, 1
   sw    r7, r4, r9
   lwi   r7, r5, 4
   cmpu  r18, r7, r3
   bltid r18, -20
   addik r4, r4, 4
   addik r3, r0, 1
   swi   r3, r8, 0
   rtsd  r15, 8
   or    r0, r0, r0
```

Fig. 3.4 The communication source code for writing tokens into a channel buffer has been compiled and its instructions analyzed to identify Macro Blocks for modeling communication. Based on a figure I published in [Ste+19a].

In the computation model evaluation section Sec 5.4 I compare the different distribution functions (Uniform, Gauss, KDE-Based) regarding accuracy.

3.3.4 Computation Delay Set

The computation time of functional behavior highly depends on the processing element the code implementing the function got executed on. This leads to individual delay distributions for each actor on each specific type of Tile. Different types of Tiles may have different accelerators like a floating point unit or even a totally different instruction set architecture. To be able to perform a design space exploration, for each combination of Actor and Tile, the corresponding delay vector needs to be available for analysis of certain Actor-Tile mappings. So each combination of SDF-Application G and execution platform EP a set of delays $\mathcal{D}$ needs to exist:

Definition 3.16 A delay set $\mathcal{D} = \mathcal{T} \times \mathcal{A}$ is a set of delay vectors $\vec{d}_{A,T} \in \mathcal{D}$ where $\vec{d}_{A,T}$ represents the computation delay of an actor A on a tile T.

3.4 Communication Delay Models

The communication delay model requires more effort compared to the computation model. To cover interference on the shared resources, there are four aspects to consider:

- The communication between actors includes computational parts for accessing the private memory data and calculating addresses. This violates the assumption that computation and communication are separated (Assumption 3).
- The amount of traffic generated on the interconnect depends on the number of tokens that get communicated. Since the token rate of all ports are static and defined before run-time, this information can be used for simplification of a model.
- The communication function needs to wait and check repetitively the state of the channels buffer to have tokens or space available (aka polling) (Assumption 2). Polling on shared resources add interference points to the model.
- To model bus contention as precisely as possible, the moment and delay of shared memory access must be as accurate as possible. This requires a sophisticated communication model with the risk of significantly increase analysis time.

Conceptually communication between two SDF actors is realized via channels. During the write phase of an actor, data generated by one of the actor gets copied from the local buffer of the writing actor into a FIFO buffer that is associated with the channel that connects those actors. During the read phase of the other actor, data gets copied from the channel's FIFO buffer into the local buffer of the reading actor. Before copying, the writing actor needs to ensure the FIFO buffer has enough space and the reading actor needs to ensure the FIFO buffer has enough tokens to read. This leads to a very similar pattern for the reading and writing functions, listed in Fig. 3.4 (left). Looking at the instructions after compiling the functions that implement the communication between actors, macro blocks can be identified Fig. 3.4 (middle, and right). These macro block pattern is identical for reading and writing. The instructions of each macro block are different. Because the pattern of macro blocks is strong related to the source code, it is most likely identical with any instruction set architecture. In the example given in Fig. 3.4 the Xilinx MicroBlaze [Xil21] instruction set architecture has been used.

The communication function consists of five macro blocks. The *Initializing* macro block that prepares the *Polling* block. During polling, the state of the FIFO buffer of a channel gets checked. After that, a *Preparing* macro block prepared the *Copying* loop in that tokens get copied from private memory to shared memory or from shared memory to private memory. The last macro block, *Managing*, updates the FIFO buffer state. Beside the instructions directly related to the communication algorithm, also instructions introduced by the compiler for implementing the application binary interface (ABI), like setting up the stack pointer for local variables, is included.

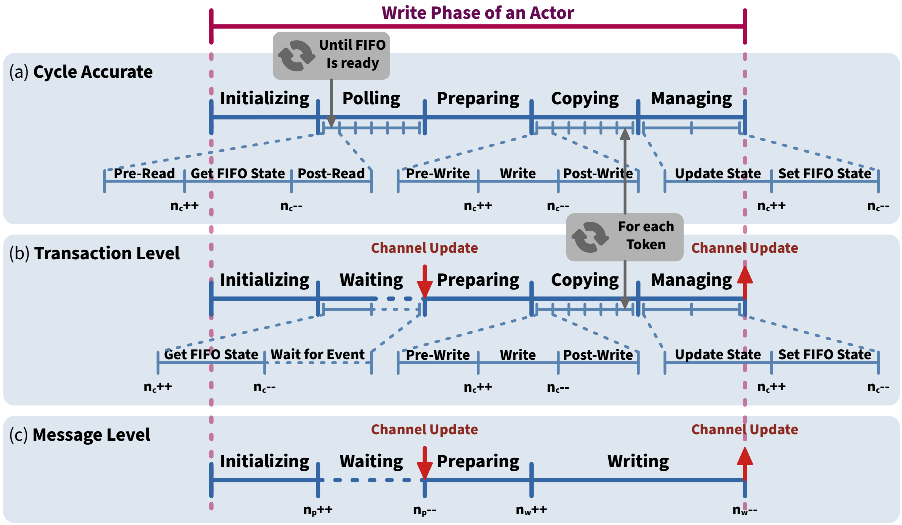


Fig. 3.5 Simulation steps of the communication model of an actor write phase using the following three approaches:
(a) Cycle Accurate [Ste+19a], (b) Transaction Level [Ste+21], (c) Message Level [Vu+21].
This figure has been published by me first in [Ste+21].

During the work on this thesis I followed two different approaches, published in [Ste+19a; Ste+21] and a third approach by Hai-Dang Vu et al. [Vu+21]. Each of the approaches are visualized in Fig. 3.5 first published by me in [Ste+21].

Fig. 3.5 shows the process of writing tokens from an actor into a channels memory during the *Write Phase* of the execution of an actor. The write process has been split into four to five macro blocks, depending on the abstraction level of the model communication model. Two of the presented models use events to improve simulation speed. Waiting for an event, and raising an event are denoted by red arrows. Some parts of the communication process are split into sub-parts to present further details that are mentioned in the model descriptions in Sec 3.4.1 and Sec 3.4.2. The gray boxes annotate conditions how often a sub-part gets executed.

All models consider a First Come First Serve communication policy (Assumption 10) for communicating tokens between actors using a shared memory. The different variables n in Fig. 3.5 denotes the amount of tiles that are accessing a shared resource simultaneously. The proposed models use this number as parameter to consider increasing contention on the interconnect with increasing number of communication participants. For the Cycle Accurate (CA) and Transaction Level (TL) models, all communicating tiles n_c are considered equally. The Message Level (ML) model distinguishes between writing tiles n_w, reading tiles n_r and polling tiles n_p that are waiting for a free or filled FIFO buffer. So $n_c = n_w + n_r + n_p$. While the CA and TL model consider each token individually, the ML models formulas for calculating certain time spans and considers all consumed or produced tokens at once, denoted by T. The suffix ++ denote increasing the value of a variable $n \in \{n_c, n_w, n_r, n_p\}$ when switching from one communication part to another. The suffix `--` is used to decreasing the value of a variable.

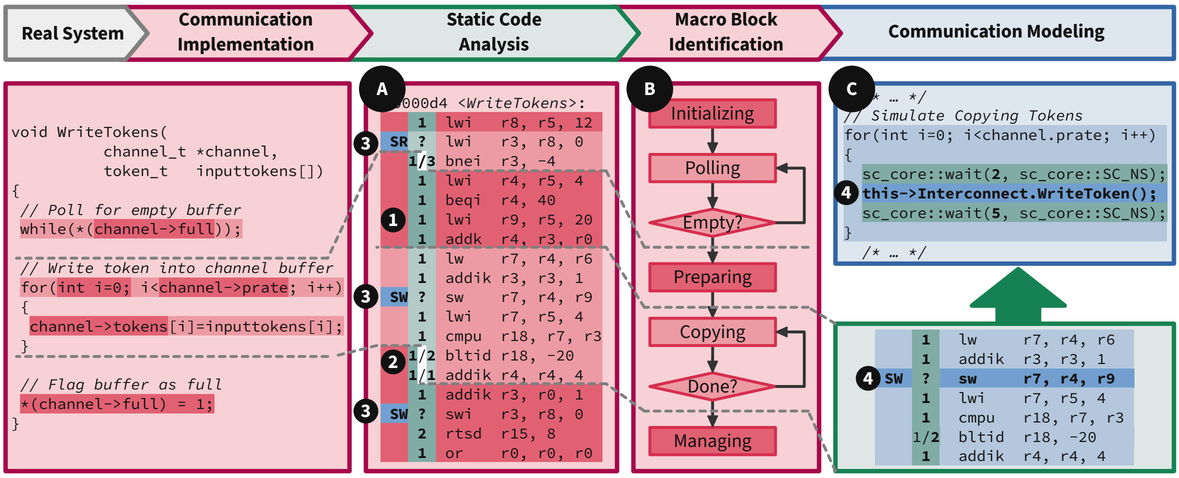
The SDF model allows strict separation of communicational and computational parts of an application (See Sec. 3.1). When implementing an application following SDF semantics this separation vanishes, because the communication requires executing instructions on the processing element. So the communication includes computational parts as well. During these computational parts (like incrementing an index or jumping in a loop), no interconnect access takes part. Remember that the instructions and local data are stored in private memory so that instruction execution does not interfere with data communication in the context of SDF application execution. The individual parts of the individual communication models are described in the following sub sections.

Three different approaches have been used to address the communication modeling challenge. All models represent a specific implementation of the communication algorithm on a certain hardware platform. Each of of the models considers a different abstraction level. In the evaluation (Sec 5.5) these models are compared regarding accuracy and simulation speed.

The *Cycle Accurate (CA) model* (Sec 3.4.1), first presented in [Ste+19a], comes with a low abstraction of the communication driver. This model considers the instructions executed on the processing element to perform communication, as well as the delays during communicating single tokens over the interconnect. This model is visualized at the top of Fig. 3.5.

The *Message Level (ML) model* (Sec 3.4.3), first presented in [Vu+21], is an abstract model focussing on communicating a whole set of Tokens at once. Individual communication and instruction execution steps are abstracted to simple messages. Therefore the interconnect usage is pre-calculated based on active communicating processing elements. This model is visualized at the bottom of Fig. 3.5.

The *Transaction Level (TL) model* (Sec 3.4.2), first presented in [Ste+21], abstracts the polling phases of the communication to reduce simulation overhead. The cycle accurate model requires a lot of simulation time compared to the message level model, as a comparison of simulation time in [Vu+21] showed. For simulation time optimization the transaction level model abstracts polling on a channel FIFO buffer inside the simulation by using events instead of modeling and simulating the polling process cycle accurate. This model is visualized in the middle of Fig. 3.5. The TL model is an enhancement of the CA model with the knowledge gained by the ML model construction.



```
void WriteTokens(
          channel_t *channel,
          token_t   inputtokens[])
{
 // Poll for empty buffer
 while(*(channel->full));

 // Write token into channel buffer
 for(int i=0; i<channel->prate; i++)
 {
  channel->tokens[i]=inputtokens[i];
 }

 // Flag buffer as full
 *(channel->full) = 1;
}
```

```
000d4 <WriteTokens>:
      1   lwi    r8, r5, 12
 SR   ?   lwi    r3, r8, 0
      1/3 bnei   r3, -4
      1   lwi    r4, r5, 4
      1   beqi   r4, 40
      1   lwi    r9, r5, 20
      1   addk   r4, r3, r0
      1   lw     r7, r4, r6
      1   addik  r3, r3, 1
 SW   ?   sw     r7, r4, r9
      1   lwi    r7, r5, 4
      1   cmpu   r18, r7, r3
      1/2 bltid  r18, -20
      1/1 addik  r4, r4, 4
      1   addik  r3, r0, 1
 SW   ?   swi    r3, r8, 0
      2   rtsd   r15, 8
      1   or     r0, r0, r0
```

```
/* … */
// Simulate Copying Tokens
for(int i=0; i<channel.prate; i++)
{
  sc_core::wait(2, sc_core::SC_NS);
  this->Interconnect.WriteToken();
  sc_core::wait(5, sc_core::SC_NS);
}
  /* … */
```

```
     1   lw     r7, r4, r6
     1   addik  r3, r3, 1
SW   ?   sw     r7, r4, r9
     1   lwi    r7, r5, 4
     1   cmpu   r18, r7, r3
     1/2 bltid  r18, -20
     1   addik  r4, r4, 4
```

Fig. 3.6 Workflow of creating a communication model for a SystemC simulation based on static code analysis. (Based on [Ste+19a]).

3.4.1 Cycle Accurate Model

This section recapitulates my work on the Cycle Accurate communication model first published in [Ste+19a] and recapitulated in [Ste+21].

The Cycle Accurate communication model is cycle accurate from perspective of executed instructions of the communication drivers running on the connected processing element. It is not signal accurate from perspective of the interconnect, meaning that not each bit change on the signal lines of the communication interface is modeled. The detailed arbitration process, in this work considered being a First-Come-First-Serve (FCFS) arbitration, is abstracted as a look up table that defines observed penalty delays depending on the amount of contender n_c (Fig. 3.5). Building up this look up table is part of the characterization process described in Sec 4.2.

The Cycle Accurate communication model represents the communication process between an actor and shared memory as accurate as possible. The model focuses on the software communication drivers and can be easily applied to different interconnects. The modeling steps are visualized in Fig. 3.6 using the function to write tokens from an Actor into the FIFO buffer of a Channel (`WriteTokens`) as example. The process for modeling the reading function `ReadTokens` is the same. The modeling process is demonstrated using the MicroBlaze instruction set architecture.

To identify the delay spent on the local tile, and the delay spent for transferring data between a tile and shared memory, the source code of the communication functions needs to be compiled for the target platform. The compiled functions can then be disassembled (Fig. 3.6 Ⓐ). Creating assembly code from the source code directly may lead to inconsistency between the assembly code and the binary code because some compiler/assembler optimization may have not been applied yet. Compiling the source code first to the binary that shall be executed and then using a disassembler to generate the mnemonic representation of this code mitigates this effect.

After disassembling the communication functions, the amount of clock cycles to execute each instruction needs to be annotated to the assembly code. This process requires knowledge about the Instruction Set Architecture. The execution time of each instruction must be derived from the datasheet of the execution platform. In general the instructions can be grouped into three different types of instructions.

- Instructions with a single execution time.
- Instructions with multiple conditional execution times.
- Instructions that access shared resources leading to contention based execution times.

The example code in Fig. 3.6 Ⓐ shows assembly code for Xilinx MicroBlaze [Xil21] processors. This architecture comes with pipelined instruction execution which has to be considered. According to the datasheet *"one instruction is completed on every cycle"*[Xil21]. Exceptions, also documented in the datasheet, have been considered as well. These exceptions correspond to the different types of instructions and have been annotated in Fig. 3.6 as well (②,③).

The simplest case are instructions that have a single fixed amount of clock cycles it takes to execute them. The `lwi` ① for example loads data from private memory which takes always a delay of $d_{lwi} = 1$ clock cycle. This proposition is valid because the model of architecture guarantees interference free access to private memory (Assumption 6).

Then there are instructions that have a variable execution time. For example branching instructions execution time ② which is based whether a jump to a new address is performed or not. There may be corner cases where it can be assumed that such an instruction has a fixed execution time based on knowledge of the algorithm. For example a check for a variable to be zero can be considered being always false when the developer knows that the algorithm does not allow the variable being zero. In the example code this situation occurs one line above ① (`beqi`) where the amount of tokens to transfer is checked. The communication function never transfers 0 Tokens.

The third type of instructions access shared resources. In 3.6 Ⓐ these lines are marked with ③. Instruction that access shared memory are annotated with `SR` (Shared, Read) and `SW` (Shared, Write). Such an instruction performs a read or write access to a shared memory via a shared interconnect. The execution time of these instructions highly depends on the resource contention. This situation needs to be considered in the model. The actual execution time of these instructions will be determined during simulation of the communication.

The delay annotated code to transfer tokens from an actor to a FIFO buffer of a channel, or from such a FIFO buffer to an actor, can be split into different macro blocks. The macro blocks in Fig. 3.6 Ⓑ are equivalent to the phases shown in the communication model overview in 3.5.

Each macro block can consist of instructions that only access data from local memory or registers, instructions that write a word into memory that is connected to a shared interconnect or instructions that read a word from such a shared memory. Further more macro blocks may be executed in loops. In this case the loop-instructions are part of the computational overhead of a block. The execution time of macro blocks that only access local resources can be determined by static code analysis and later summed up to a single delay. The analysis of macro blocks that contain instructions that access shared resources can only provide a computation delay offset. The communication time with the shared resources will be determined by the simulation and require further characterization processes of the interconnection (See Sec 4.2.2).

There are five macro blocks for reading or writing tokens: *Initializing*, *Polling* or *Waiting*, *Preparing*, *Copying* and *Managing*. Both, reading and writing, have the same structure and only varies in the amount of clock cycles for computation, interconnect accesses and loop cycles. Combining several instructions to a single macro block reduces the amount of simulation steps compared to considering and simulating each instruction delay individually.

The *Initializing*-Block represents necessary instruction execution to prepare for initiating the communication process with a FIFO buffer on a shared memory. This also includes programming language specific setup of the context of a function but also algorithmic behavior like initializing temporary variables used inside the `WriteTokens` function.

The *Polling*-Block abstracts the process of polling for a valid channel buffer state. For reading, the FIFO buffer of the channel must contain enough tokens to read, for writing, the FIFO must contain enough space for tokens to write. Reading and Writing is blocking. So the loops stays active until the required buffer state is given. The polling process is modeled in detail considering the computational overhead for the polling loop and each interconnect access for checking the FIFO state.

The *Preparing*-Block represents setting up the copy-loop to transfer tokens between private local memory of an actor and the shared memory of the channel. Part of this preparation is to initialize a counter variable and to get the token rate that represents how may tokens will be copied into the channel buffer.

The *Copying*-Block models the token transfer of tokens. When modeling a write process, this is the transfer of tokens from a producing actor into the FIFO buffer of a channel. Modeling a reading process, this is the transfer from the FIFO buffer of a channel into the private local memory of a consuming actor. This is usually a loop that gets executed as many times as tokens need to be transferred. The amount of loop iterations is strictly related to the consume or produce rate of the port of an actor for a specific channel. Each token is handled individually considering the computation overhead of the loop and the communication part of the token transfer.

The *Managing*-Block updates the meta data of the communication channel like the fill-state of the FIFO buffer. This phase also includes the `return` instruction to leave the called communication function.

After identifying the macro blocks they can be transferred into a structure that can be simulated. Fig. 3.6 ⓒ shows this process on the example of simulating the copying process of data in SystemC. Instructions with static delays can be consolidated as a single SystemC `wait` statement. Instructions annotated with SW or RW to highlight shared resource access are represented as individual interconnect accesses ④. The interconnect model considers a fixed delay offset for the shared memory access, and an additional contention penalty depending on how many processing elements trying to access the shared interconnect (See Fig. 3.5 n_c). Details about the interconnect contention penalty delay are described in Sec 4.2.2.

3.4.2 Transaction Level Model

This section recapitulates my work on the Transaction Level communication model first published in [Ste+21].

The Transaction Level model shown in the middle of Fig. 3.5 is based on the Cycle Accurate model described in Sec. 3.4.1. To improve simulation speed, the polling process is modeled using events.

The Cycle Accurate model describes each iteration of polling on the FIFO buffer state. On a real system with a configuration where most of the actors are executed in parallel, most of the time these actors are polling on the FIFO buffer waiting for data that can be processed. Because each polling iteration consists of computational and communicational parts, multiple simulation steps are required. This decreases the simulation speed tremendously. To avoid these simulation steps, the polling process is simulated using events instead of explicitly simulating the polling communication. When the FIFO buffer is not in the state that it must be to perform the read or write process, the simulated actor waits for an event that signals a state change inside the FIFO buffer. For each channel exists an exclusive event signal. The state update event is triggered after another simulated actor updated the FIFO buffer state. An actor writing into a buffer triggers that event after it wrote all its tokens into the FIFO buffer. An actor reading from a buffer triggers that event after it read all its tokens from the FIFO buffer. So the simulation kernel can easily skip the polling activity without simulating each individual polling attempt. Simulated actors that are in a waiting situation during their read or write phase do not require simulation time and therefore CPU load.

Similar to the Cycle Accurate model, the Transaction Level model also considers a fixed delay offset for the shared memory access, and an additional contention penalty depending on how many processing elements trying to access the shared interconnect (See Fig. 3.5 n_c). In contrast to the Cycle Accurate model, the Transaction Level model assumes continuous bus access by the polling actors. Any computational overhead where the bus is not accessed is ignored.

3.4.3 Message Level Model

This section summarizes our work first published in [Vu+21] and recapitulated in [Ste+21]. The key idea of the Message Level model comes from my colleague Hai-Dang Vu and has been fully documented in his PhD Thesis [Vu21].

Of the three communication models presented in this thesis (Shown in Fig. 3.5), the Message Level model is the most abstract one. This makes it also the fastest one when it comes to simulation time as shown in the communication model evaluation Sec 5.5.

The goal behind the Message Level model is, similar to the Transaction Level model, to reduce the amount of simulation steps and so to increase the simulation speed. To still achieve a good level of accuracy a hybrid approach is used that combines simulation and analytical models.

In the Message Level part of Fig. 3.5, four steps for writing tokens are shown: Initializing, Waiting, Preparing and Writing. The same structure can be considered for reading tokens.

The analytical part of model is used to formulate the duration of each communication phase shown for the Message Level model in Fig. 3.5. The simulation is used for taking into account potential penalty delays due to contention at shared resources. For calculating the contention penalty delays, the amount of communicating actors is tracked in the simulation. Same to the other communication models presented in this chapter, the number of communicating actors is represented by n in the model as well. In contrast to the other models, this model differentiates between polling (n_p), reading (n_r) and writing (n_w) actors.

Similar to the Cycle Accurate and Transaction Level models the *Initializing-Step* represents the overhead of initializing the transfer of tokens into a FIFO buffer or out of it. Same for the *Preparing-Block* that represents the overhead of setting up the loop for the bulk transfer of tokens.

The *Waiting-Step* is modeled by using synchronization events between reading and writing tokens on a same FIFO buffer. These events are shown in Fig. 3.5 by the red arrows. When the FIFO buffer in that the tokens should be written is full, the Waiting phase is entered. Once the reading tokens function finishes accessing the buffer and frees buffer space to write to, it sends an event to trigger the writing tokens function to start writing. This synchronization method reduces the number of polling states considered by the simulation kernel, similar as it has been done in the Transaction Level model (See Sec 3.4.2).

The *Writing-Step* models the transfer of one or more tokens into the channel buffer. The writing duration is computed using an analytical model built from the knowledge of elementary communication phases and the bus arbitration policy. This analytical model is explained in detail in [Vu21]. In our approach, the timed Petri net (TPN) formalism is adopted to formulate the relationships between elementary communication steps. It represents a timed extension of Petri nets for which time is expressed as minimal durations on the sojourn of tokens on places [Bac+92]. The adoption of the TPN formalism allows to describe different communication situations with different numbers of tiles and simultaneous processing of reading and writing tokens into different channel buffers.

The variables n_p, n_w, n_r shown in Fig. 3.5 (Message Level diagram) are used to trace the communication situation by counting the number of on-going *Waiting-Step* (n_p) that represents polling activity, *Writing-Step* (n_w) and *Reading-Step* (n_r). These are the steps an actor can be in while it is in its communication phase. At the beginning of each communication phase, the variables are incremented. They get decremented at the end of each phase. The communication duration for reading or writing tokens is calculated based on those variables.

After the time for communicating the tokens passed, an event gets triggered notifying that the communication process has finished. The combination of using events and pre-calculating the estimated communication delays increases the simulation speed compared to the other communication models

The analytical part of the Message Level model require some *elementary delays* [Vu21]. While all delays used in the Cycle Accurate and Transaction Level model are from the perspective of the instruction execution on a specific processing element, the elementary delays used in the ML model are from the perspective of the communication bus. So these delays represent the amount of clock cycles between different active and idle state changes of the bus used for communicating with a shared resource. This also included for example the delay between the last word sent for checking a buffer state during the polling phase, and the first token read or written into the buffer during the reading or writing phase. From this delay, the computation overhead between the polling and reading/writing phase can be derived.

3.5 System Model

For characterization, simulation and evaluation, the system under analysis needs to be built from its software and hardware components. This requires mapping of the software (actors and channels) to hardware components (tiles and memory), explained in Sec 3.5.1. Actors mapped on tiles need to be scheduled (Sec 3.5.2).

In Fig. 3.7 an example of a mapped and scheduled system (right) is shown. Inputs of the mapping and scheduling process are an application following following SDF semantic as defined in Sec 3.1 and an execution platform following the tile based architecture presented in Sec 3.2.

The actors of the SDF application are mapped to the tiles on that they will be executed. The channels used by the actors for communication are mapped to local or shared memory. The example SDF application shown in Fig. 3.7 is the Sobel-Filter use-case that is described in detail in Sec 5.1.1.

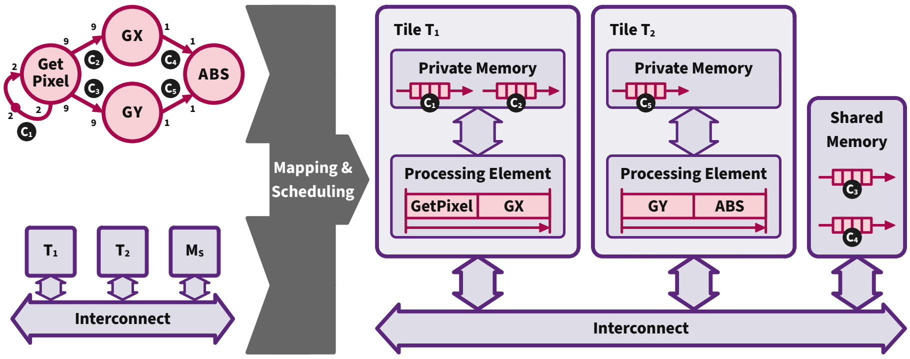


Fig. 3.7 Example mapping of a four-actor SDF application on a two-tile hardware. Some channels are mapped to private memory, some to shared memory. Actors on a tile are scheduled in static order.

3.5.1 Mapping

Definition 3.17 The actor mapping function $\theta : \mathcal{A} \times \mathcal{T}$ maps each actor $A_i \in \mathcal{A}$ to a single tile $T_j \in \mathcal{T}$.

Definition 3.18 The channel mapping function $\mu : C \times \mathcal{M}_P \cup \mathcal{M}_S$ maps each channel $C_i \in C$ to a single private or shared memory $M_j \in \mathcal{M}_P \cup \mathcal{M}_S$.

If the producing actor and consuming actor of a channel is mapped to the same tile, the channel can be mapped to private memory. Otherwise the channel needs to be mapped to shared memory, so that both tiles can access the memory.

In Fig. 3.7, the channels C_1 and C_2 are mapped to the private memory of tile T_1 because both actors accessing those channels are mapped to the processing element of the same tile. So there is no need to use shared resources for communication. However, this design decision is optional. All channels can be mapped to shared memory, but only some channels can be mapped to some private memory.

3.5.2 Scheduling

Since multiple actors can be mapped to one tile, these actors need to be scheduled. A static order scheduling strategy is used to make sure all actors are scheduled in a predictable order and without run-time overhead by more sophisticated scheduling algorithms. An algorithm to determine the local scheduling order of the actors on a single processing element of a multi-processor system has been described by[Dam+12]. It is based on the repetition vector of the SDFG as it is presented in [Stu07]. In context of this thesis, one execution of the whole SDF application from the first actor until the last actor in the scheduling has been executed is called *iteration*. The time it takes is called *iteration execution time* or *iteration duration*.

Assumption 12. *The SDF application is self scheduled with static order. There is no execution time overhead for scheduling actors. After one actor finished its writing phase, the next actor on the tile starts with its read phase. In case any depending actor (that gets executed on a different tile) did not write its token on the incoming channel of the scheduled actor, the scheduled actor polls on the buffers channel until the data is available, which is part of its read phase.*

Chapter 4
Characterization and Analysis Approach

4.1 Delay Measurement Infrastructure

For an execution time analysis delays are required. These delays can be obtained in several ways. One example can be static code analysis as it has been done for the Cycle Accurate communication model (Sec 3.4.1). Another approach which has been followed in this work is measurement based execution time observation.

In Fig. 4.1 two extensions (green) to a tile based execution platform (purple) are shown. One extension is a communication measurement component that is connected to the main communication interconnect of the tile based system. Via this connection, ongoing communication can be observed from the interconnect on signal level. The observed information can then be sent via a dedicated bus to a host system, without interfering communication on the main interconnect. This communication observation approach has been used by [Vu21] to characterize the Message Level communication model described in Sec 3.4.3. In context of his and my work we used the *Xilinx System Integrated Logic Analyzer* (SystemILA[1])

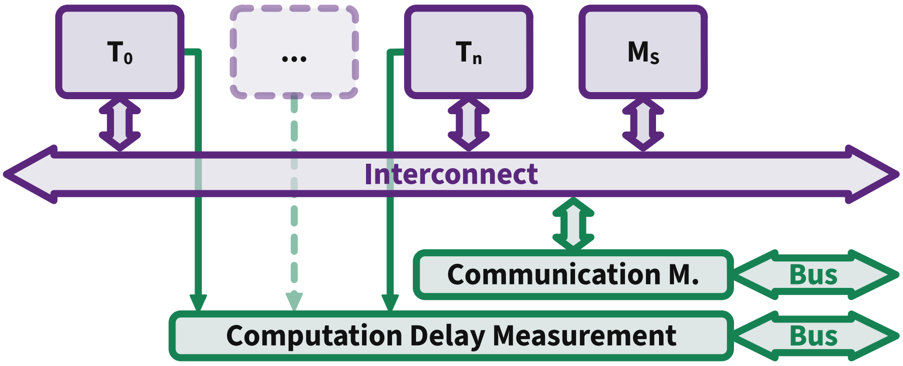


Fig. 4.1 Concept for measuring computation and communication delay. The execution platform (purple) is equipped with different measurement infrastructures (green) for computation and communication characterization. The computation measurement infrastructure is connected to all n tiles of the evaluation platform, the communication measurement infrastructure is connected to the communication interconnect.

---

[1] Xilinx System Integrated Logic Analyzer, `https://www.xilinx.com/products/intellectual-property/system-ila.html`, Visited 06.01.2023

For observing computation delay, a computation delay measurement infrastructure needs to be connected to the processing units that execute the code that needs to be characterized. Therefore a measurement infrastructure has been created based on concepts first published by Schlaak et al [SFS17]. The measurement infrastructure concept from [SFS17] has been enhanced by supporting measuring pipelined SDF-application iterations as described in Sec 4.1.1 In Sec 4.1.2 the implementation of the measurement infrastructure gets described and in Sec 4.1.3 the verification of this implementation is presented.

The core idea behind the measurement concept is, that a counter clocked with the same frequency of the processing elements is used to count the amount of clock cycles between a *Start* and *Stop* signal. These signals are sent from the processing elements via a peripheral bus to a dedicated measurement component. After a measurement has been performed the results, amount of clock cycles passed between Start and Stop signal, will be transferred to a host computer via a dedicated Universal Asynchronous Receiver and Transmitter (UART) interface.

4.1.1 Measuring Pipelined Execution

Fig. 4.2 (a), shows the execution of the example application from Fig. 3.7 of Sec 3.5 over time. Three iterations, highlighted by the indices of the actor names, of the Sobel-Filter use-case (Sec 5.1.1) are shown. The 4 actors of the Sobel-Filter are mapped to 2 tiles in a way that pipelined execution is possible. While the ABS actor of the first iteration is

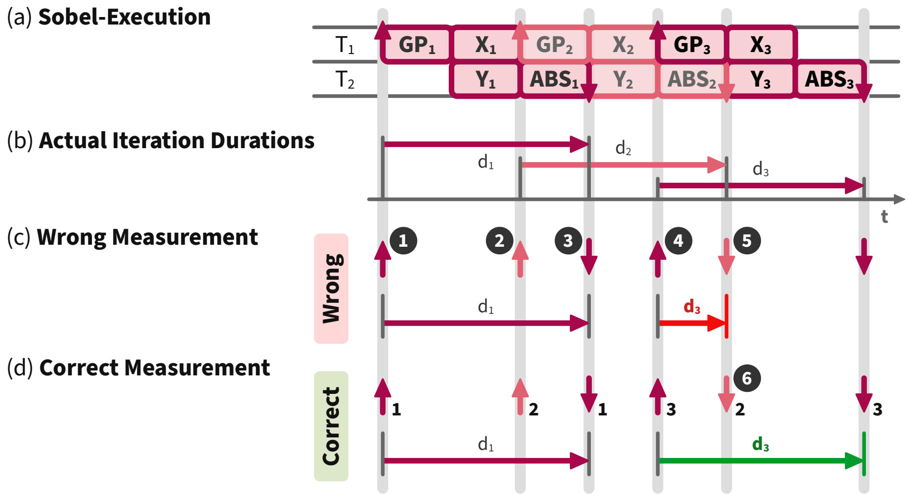


Fig. 4.2 Pipelining can lead to invalid measured iteration durations. To address this, counters are used to match Start events to their corresponding Stop events.

still being executed on T_2, the GetPixel (GP) actor for the next iteration already started being executed on T_1.

To measure the iteration duration of executing the Sobel-Filer application, with each begin of the execution of the GetPixel actor, a *Start* signal gets sent to the measurement infrastructure. At the end of the ABS execution a *Stop* signal gets sent. The actual iteration durations are shown in Fig. 4.2 (b). Because of the pipelining behavior, measuring each duration exceeds the capability of the measurement infrastructure that can only perform one measurement at a time.

Applying the measurement technique presented in [SFS17] leads to wrong measurements as shown in Fig. 4.2 (c). For better explanation, Start and Stop events mentioned in this text are enumerated by numbers inside black circles. Start events are visualized by an upwards directed arrow, Stop events by a downwards directed arrow.

The first shown iteration is measured correct. The Start signal ① from the first iteration starts the measurement process. A second start signal ② triggered by the start of the second iteration gets correctly ignored, because the measurement infrastructure is already in measurement state. Then the Stop signal ③ from the first iteration occurs and stops the measurement. A valid delay d_1 of the first iteration has been measured.

When the third iterations starts, a new Start signal ④ gets send to the measurement infrastructure and starts a new measurement. Now, the Stop signal ⑤ triggered by the still executed second iteration stops the measurement again. When in measurement state, the infrastructure finished a measurement as soon as a Stop signal arrived. Instead of measuring the actual iteration duration of the third iteration from the Start and Stop signals of the third iteration, only the time difference between the start of the third iteration and the end of the second iteration is measured. This leads to a wrong duration d_3 for the third iteration.

To address this issue, the measurement concept from [SFS17] needs to be improved. The issue that leads to the wrong measured delays of pipelined execution is the lack of knowledge which Start and Stop signal belongs to which iteration. To solve this issue, each Start and each Stop needs to get an ID that connects the individual signal to an iteration which triggered that signal. To accomplish this, the Start and Stop signals are used to increment two counters. With each Start signal a Starts-Counter gets incremented, and with each Stop signal a Stops-Counter gets incremented. In Fig. 4.2 (d) the individual counter values are annotated at the bottom right of the arrows of the Start and Stop signals.

If now a Start signal occurs, the Starts-Counter value gets incremented and the measurement started. When a Stop signal comes, the Stops-Counter value gets incremented and compared to the Starts-Counter value. If the values of the two counters do not match, the Stop signal gets ignored (Fig. 4.2 (d) ⑥). Only a Stop signal that leads to a Stops-Counter value that matches the Starts-Counter value stops the measurement. This leads to a valid iteration duration d_3.

The amounts of bits for the counters can be low since an overflow would not affect the concept of having two counters with identical value to finish a measurement. Anyway,

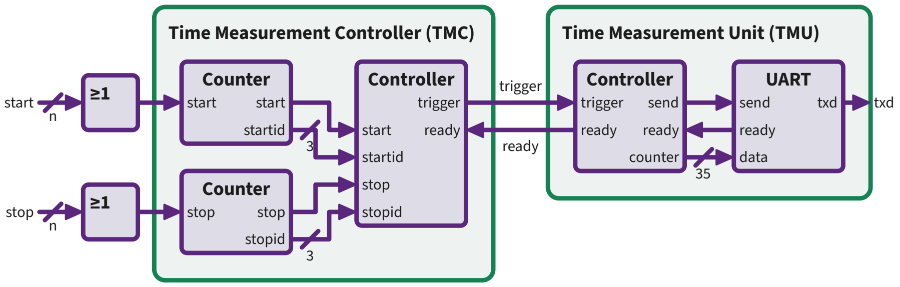


Fig. 4.3 Hardware components (purple) of the Time Measurement Infrastructure (green). The Time Measurement Controller (TMC) coordinates starting and stopping a measurement that gets performed by the Time Measurement Unit (TMU). The TMU sends measured delays via UART to a host computer. For simplicity clock and reset signals are not shown.

both counters need to have the same bit width and must represent values larger than the maximum of parallel running iterations.

4.1.2 Proposed Measurement Infrastructure

The *TMx* measurement infrastructure was first presented by Schlaak et al. in [SFS17]. The now refined version was first presented in [Ste+20]. The infrastructure is split into multiple components. Details of the implementation in hardware (purple) of two of them (*TMC* and *TMU*) are shown in Fig. 4.3 (green). For simplicity reason, clock and reset signals are not explicit show. Still, all shown components are connected to the system clock and system reset signal. For this measurement method, it is important that all components are clocked with the same frequency (Assumption 5 - All components are connected to the same clock).

To connect each tile with the time measurement infrastructure (TMC and TMU) a third component, the *Timing Measurement infrastructure Bridge (TMB)* is used. This bridge is not explicit shown in Fig. 4.3. Purpose of this bridge is to translate a data package from a processing element's communication interface (for example a peripheral bus) into the individual control signals (Start/Stop) expected by the Timing Measurement Controller (TMC). This bridge may need to be adapted to the peripheral interface of the processing element that shall be connected to the Time Measurement infrastructure. For each tile that gets connected to the TMC, one TMB is used that provides one Start and one Stop signal. When connecting n tiles to the TMC, n Start and n Stop signals needs to be combined into a single Start and a single Stop signals as it is required for the Time Measurement Controller (TMC). This is done with a n input OR-gate as shown in Fig. 4.3.

Assumption 13. *To avoid any interferences with other tiles it is assumed that the measurement infrastructure is connected to an individual dedicated interface. This can be a General Purpose Input/Output (GPIO) interface or a peripheral bus.*

The management of the individual Start/Stop signals from the tiles are managed by the *Timing Measurement Controller (TMC)*. Whenever a start or stop signal comes from any of the tiles, a start signal is sent to the TMU. For the measurement it is not important to distinguish between tiles.

When characterizing only the compute phase of a single actor, Start and Stop will come from the same tile the actor is mapped to. When, for example, the whole iteration of SDF application as shown in Fig. 4.2, the Start signal will come from a tile T_1 before the actor A_{GetPixel} gets executed. The Stop signal may come from tile T_2 after execution the actor A_{ABS}.

In case of pipelined execution of an SDF application, it is important to keep track which Start/Stop signal correlated to which iteration of the execution as described in Sec 4.1.1. Therefore the Start and Stop signal are routed through a counter component (See Fig. 4.3). These counters extend the Start/Stop signals with an iteration ID. A 3 bit counter has been used to be sure to have more IDs than required for the evaluation platform (Described in Sec 5.1.3) that could allow pipelined execution of up to seven stages, depending on the mapping and scheduling of the JPEG use-case application Sec 5.1.2. The Start/Stop signals as well as their IDs are inputs to a state machine. This state machine controls the TMU component, which is responsible for measuring and communicating the delay between a Start and Stop signal with the same ID.

The *Time Measurement Unit (TMU)* is basically a counter clocked with the same cycle rate the processors have. The counter increments its counter value with each clock cycle, as long as the *trigger* signal, which is controlled by the TMC, is high. The counter, part of the TMU Controller, can be started and stopped from any tile individually via the TMC without interference. So that Assumption 6 and Assumption 3 are not violated. When the counter got stopped, the TMU sends its counter value via UART to a host computer that collects all measured data. After the data has been sent, the counter gets reset and the *ready* signal gets set to inform the TMC that a new measurement can start.

The physical UART protocol allows transmission of single bytes (8 bit). When transferring multiple bytes, a logical protocol needs to ensure, that the begin and end of a series of bytes (a package) can be identified by the receiver within the stream of data it receives. To allow sending counter values larger than one byte, the most significant bit (MSB) is used to mark a byte as the first byte of a multi-byte counter value. If the MSB is set, a new series of bytes starts. If the MSB is not set, it is a byte that is within a series. When the host computer starts listening to the byte stream, it needs to skip all incoming bytes until the first one gets received with its MSB set. Because one bit of a transferred byte is reserved, the number of bits used for the counter should be a multiple of seven to make use of all available bit in a data package. The implementation of a protocol for identifying the begin of a counter value within a data stream is also an extension to the implementation and concept presented in [SFS17].

For measuring the delay between Start and Stop, a 35 bit counter is used. The execution platform that is used for evaluation is clocked with a frequency of 100 MHz. A 28 bit counter could measure delay up to approximate two seconds. For most cases this should be enough. To be on the safe side for applications that come with long iteration delays a counter with 35 bit has been used which requires only one byte more in the data package sent to the host computer. This allows measuring delays up to approximate 343 seconds. The Time Measurement Infrastructure has been implemented with the hardware description language VHDL and gets synthesised together with the evaluation hardware (Sec 5.1.3) on an Field Programmable Gate Array (FPGA). The source code is of the measurement infrastructure if part of the evaluation setup that is available for download[2].

4.1.3 Verification of Delay Measurement Infrastructure

To ensure the Time Measurement Infrastructure is working correct, it got tested on different abstraction levels. Each hardware sub-component shown in Fig. 4.3 (purple) has been individually tested using signal and cycle accurate simulations inside the Xilinx Vivado Design Suite. Furthermore, also within the simulation, the TMC and TMU components have been tested. The Time Measurement infrastructure Bridge (TMB) has been generated by a code generator inside the Xilinx Vivado Design Suite. The Measurement Infrastructure components and sub-components have been tested regarding functional correctness and timing behavior.

Beside the component tests, the measurement infrastructure has been instantiated and tested on a real FPGA, together with Xilinx MicroBlaze processors. On those processors, test code got executed. The measurement infrastructure has been used to measure the delay of these test cases. Additionally, the execution times of the test cases have been calculated manually, by summing up the execution times as documented in the datasheet for the instantiated processor.

In context of these tests, the influence of the measurement infrastructure on the execution of the code which gets observed has been characterized. That the presented measurement concept is intrusive has already been evaluated in [SFS17]. To send a Start or Stop signal from a processing element to the TMB periphery component, code needs to be executed. Executing code takes some clock cycles which impacts the execution time characteristic of the whole system. For example the delay introduced by the Start or Stop instruction can lead to a small shift in the moment an actor communicates data over a shared interconnect. This can lead to cause or avoid contention on the interconnect. To guarantee that the system equipped with Start and Stop instructions

---

[2] Evaluation code and data: `https://zenodo.org/record/7976829`, visited 28.05.2023.

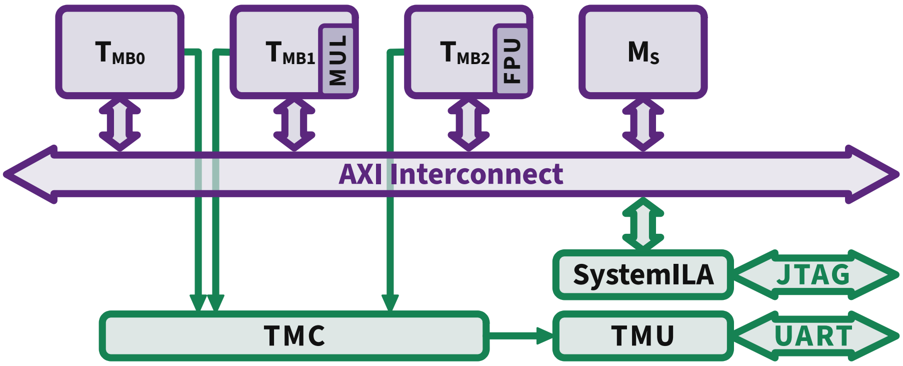


Fig. 4.4 Evaluation platform (purple) for characterization providing three tile types. One without hardware accelerator, one with hardware multiplication (MUL) and one with floating point unit (FPU). The platform is equipped with different measurement infrastructure (green) for computation and communication characterization.

does not change its behavior in production, the Start and Stop instructions need to be replaced by No Operation (NOP) instructions that take the exact same clock cycles for execution as proposed in [SFS17].

4.2 Characterization

For the delay models of the execution of an actor, the compute phase needs to be characterized depending on the hardware features of the tile the code is executed on. Additionally the timing characteristics of the interconnection between tiles and shared memory need to be characterized.

The characterization is based on several techniques. Most of the characterization is done measurement based. Only one exception has been made for the computational part of the Cycle Accurate communication model (Sec 3.4.1) which uses static code analysis. Still the shared memory access is measurement based.

For the compute phase the amount of clock cycles for executing the instructions is measured. This is done for different input data (input tokens) to cover as many execution paths as possible. Since the code of an actor changes on different processors (e.g. with and without FPU), for each processor flavor this characterization needs to be done.

Fig. 4.4 shows an instance of an execution platform conforming to the model of architecture from Sec 3.2. This instance consists of three tiles with a MicroBlaze processor as processing element. Each tile comes with a slightly different configuration of the processing element. Tile T_{MB0} comes with a MicroBlaze without any accelerators. Tile T_{MB1} has a processing element extended with hardware multiplication. Tile T_{MB2} provides a processing element that is equipped with a floating point unit. The tiles are connected to a shared memory M_S with an AXI Interconnect. Additional to the execution platform a measurement infrastructure has been added (Green).

One part of the measurement infrastructure (Fig. 4.4(green)) is used to characterize the interconnect on signal level. This is required for the elementary delays used by the Message Level communication model (Sec 3.4.3) by [Vu+21]. Details of the Message Level characterization can be found in Sec 4.2.3. The observed data is provided by a JTAG interface. Another part is the *TMx* measurement infrastructure described in Sec 4.1 that consists of a Time Measurement Controller (TMC) and a Time Measurement Unit (TMU). The TMx infrastructure focus on observing execution delays at instruction level and is used for characterizing computation delays (Sec 4.2.1) as well as the communication delay (Sec 4.2.2) for the Cycle Accurate and Transaction Level models (Sec 3.4.1, Sec 3.4.2). The observed delays are provided by an UART interface.

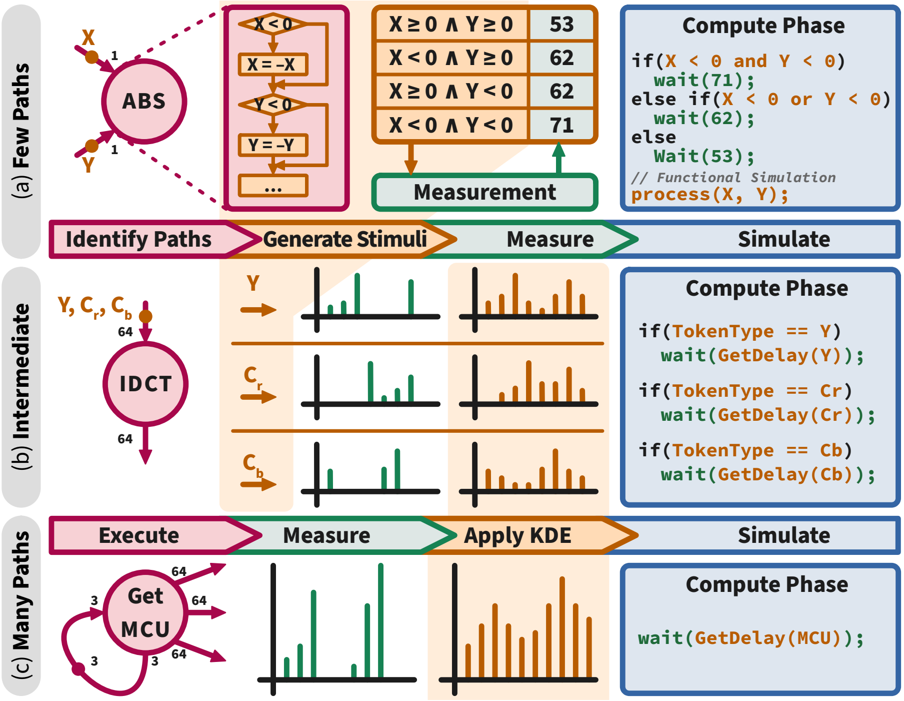
For the MicroBlaze based system shown in Fig. 4.4, the AXI Stream Interfaces is used as peripheral bus to not interfere with the AXI Interconnect system bus.

4.2.1 Computation Delay Distribution

Annotating delays to the compute phase of an actor can be challenging depending on how many execution paths the algorithm, implemented in the actor, provides. In Fig. 4.5 three different actors from the use-cases later described in Sec 5.1 are shown. The figure shows three different abstraction levels of how data dependent computation delay can be obtained and simulated. I published this concept of addressing data dependency first in [Ste+21].

The *ABS* actor (Fig. 4.5a) comes with only four different execution paths, depending on its two input tokens X and Y. In this case, static code analysis can be applied to get the amount of clock cycles required for executing each individual path. Alternatively precise stimuli data can be generated to measure all known execution paths. Inside the simulation, each execution path can be identified and the corresponding delay to that execution path can be used to proceed simulated time. To do so, functional simulation is mandatory since the values of the input token define the execution path that needs to be simulated. There can be many situation where just looking at the source code of an actor may not be enough to identify its execution paths during runtime.

The *IDCT* actor for the JPEG-Decode use-case (Sec 5.1.2) shown in Fig. 4.5b comes with many execution paths that are not easy to cover. It implements an inverse discrete cosine transformation. The execution path depends on the 64 incoming tokens that represent an 8×8 matrix with 12 bit precision per element [TT93]. Additional to the many executions paths on source level, it uses lots of floating point operations. Compiling such an actor for a processing element without floating point unit adds many more execution paths to the binary. Still, expert knowledge about the data characteristics can be used to create three different sets of stimuli data. The *IDCT* actor gets executed



$X \geq 0 \land Y \geq 0$	53
$X < 0 \land Y \geq 0$	62
$X \geq 0 \land Y < 0$	62
$X < 0 \land Y < 0$	71

```
if(X < 0 and Y < 0)
  wait(71);
else if(X < 0 or Y < 0)
  wait(62);
else
  Wait(53);
// Functional Simulation
process(X, Y);
```

```
if(TokenType == Y)
  wait(GetDelay(Y));
if(TokenType == Cr)
  wait(GetDelay(Cr));
if(TokenType == Cb)
  wait(GetDelay(Cb));
```

```
wait(GetDelay(MCU));
```

Fig. 4.5 Data dependency considerations for computation delay simulation shown on three different abstraction levels. Differentiating between each individual execution path (a), several sets of stimuli data (b) or a purely black box approach (c).
This figure has been published by me first in [Ste+21].

for each of three different color channels used for JPEG encoded images. Each of these color channels uses a different level of compression. This leads to different computation delay characteristics. As shown in Fig. 4.5b for each color channel a separate set of measured data can be collected. These different characteristics are shown in Fig. 4.6. The plot also shows, that each color channel leads to a different average execution time (dashed line) of an actor. On the measured delays inference techniques like Kernel Density Estimation can be applied to inter and extrapolate the distribution of observed delays to increase the coverage of possible execution paths. For the simulation, only the type of data - the color channel in this example - needs to be differentiated. In the proposed JPEG-Decoder use-case, instead of a single *IDCT* actor, three types of *IDCT* actors are modeled. All three actors use the exact same implementation of the inverse discrete cosine transformation but are used for processing different color channels. So each of the three types comes with a different computation delay representation.

The *GetMCU* actor in Fig. 4.5c implements several de-compression algorithms like run length decoding and Huffman decoding. It provides 8 matrices (Minimum Coded Unit block, MCU) for each of the three color channels from a bit stream. Similar to the *IDCT* actor, static code analysis is not reasonable. In this particular case, expert

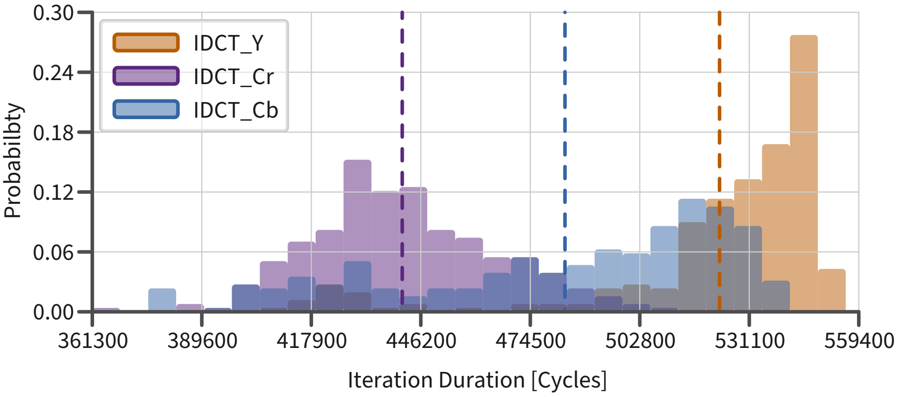


Fig. 4.6 Distribution of computation delays of the *IDCT* actor of a JPEG-Decoder. Using the same implementation of the algorithm leads to different delay characteristics depending on the color channel on that the algorithm gets applied to.

knowledge of the algorithm inside the actor cannot help to differentiate different sets of stimuli data. In this case the actor can only be considered as black box. Anyway, the delay characteristics of the actor can be measured. Applying Kernel Density Estimation further smoothes the data. Inside the simulation, the actor is then represented by a single "random" delay following the distribution function derived from the measured delays.

With the measurement infrastructure presented in Sec 4.1, the execution time of actors can be characterized without knowledge of the algorithm and its implementation of the actor. The distribution of observed execution times of an actor can be derived from the measured samples so that they can be used in a probabilistic models. The inference process is described in Sec 3.3.

To measure the delay of the execution of an actor, Start and Stop instructions to trigger the Timing Measurement Unit need to be placed around the function call of the compute phase of an actor. So after reading all incoming tokens, before the compute phase starts, the start of the measurement gets triggered. After the compute phase ends and before the write phase starts, the stop of the measurement gets triggered.

Because the execution time varies on different execution platform, this process needs to be repeated on all different types of tiles. Beside hardware differences, also differences in the data that gets processed by an actor can lead to different execution times. Fig. 4.4 shows an execution platform with three different types of tiles. Each actor of an SDF application needs to be characterized for each of those tiles.

There can be different execution paths that are taken depending on input data. Some data dependency can be obvious like `if` statements in the source code or loops that have an iteration count depending on input data. Other data dependency can be introduced by the compiler and may not be obvious from the source code. The compiler can introduce additional execution paths by applying code optimization techniques or by replaying operations, that cannot be performed by the target platform, with software solutions.

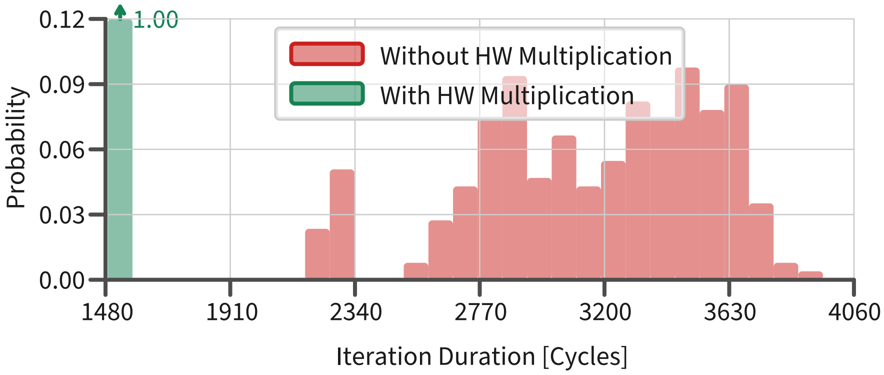


Fig. 4.7 Distribution of computation delays of the *Inverse Quantization* actor for the Y-channel of a JPEG-Decoder. Using the same implementation of the algorithm leads to different delay characteristics depending is the processing element is equipped with hardware multiplication or not.

Fig. 4.7 shows the distribution of the same inverse quantization source code applied to the Y-channel of a JPEG-Decoder. The inverse quantization is a loop with a fixed length of 64 iterations. In each iteration a single multiplication is performed. In green, the code got compiled and executed on a processing element that got extended by a hardware multiplier so that multiplication takes always the same amount of clock cycles. In red the same code got compiled for the same processing element just with deactivated hardware multiplication, so that the compiler had to use soft-multiplication. Another example are floating point operation in the source code that may need to be executed on a processing element without a floating point unit.

Obtaining just one sample may not be enough. To get a realistic characteristic, many samples need to be taken with representative input data given to the characterized actor if possible. For all actors modeled in this thesis 1 000 000 samples have been measured. In case of the *GetMCU*, each observed execution time occurs at least 7812 times[3] in the set of measured samples.

4.2.2 Abstract Communication Delay

For the Cycle Accurate and Transaction Level communication model (Sec 3.4) the delay of a read and write access to a shared memory is required. This sub section describes the characterization of these read and write accesses to a shared memory for the evaluation platform used and described in Chap 5.

On a MicroBlaze processor that is used as processing element on the evaluation platform, memory accesses are performed by the `lw` and `lwi` instructions for reading.

---

[3] Identifying the lowest repetition of an observed delay for the GetEncodedImageBlock actor:
`sort -n GetEncodedImageBlock.txt | uniq -c | sort -n | head`

Table 4.1 Observed shared memory access delay with 0 to 6 bus contender. The access has been differentiated between read and write access. For each setup, 1000 samples have been taken. There was no variance between the samples.

Access		**Reading Contender**						**Writing Contender**					
	0	**1**	**2**	**3**	**4**	**5**	**6**	**1**	**2**	**3**	**4**	**5**	**6**
Read	12	17	15	23	31	39	47	12	19	14	19	24	29
Write	9	15	12	20	28	36	44	9	16	11	16	21	26

For writing the instructions `sw` and `swi` can be used [Xil21]. Each instructions transfers a single token at a time.

The read and write access to the private memory can be determined from the MicroBlaze datasheet [Xil21]. Private memory access always takes 1 Cycle for reading or writing [Xil21]. Furthermore it can be guaranteed that there will be no contention on the interface between the processing element and its private memory. So the access time will always be the same.

In contrast to private memory access, shared memory access delays are highly dependent of the activity on the shared interconnect between the processing element and the shared memory.

To obtain the read and write access delays to shared memory via an shared interconnect, the execution time of the read and write instructions has been measured. To ensure no pipelining effect on processor level for executing instructions influence the result of the measurement, the code needs to be prepared to cause pipeline stalls whenever this can happen. For example, after executing a load instruction, the next instruction should be an instruction that processes the loaded data. If a Stop signal gets send right after executing a load instruction, the data transfer may not yet finished. Depending on the processor architecture, the additional execution time of the processing instruction may need to be subtracted.

The evaluation platform (Sec 5.1.3) has 7 tiles. One tile has been used to measure the shared memory access delay. Several situations have been observed to characterize the communication delay of the interconnect of the evaluation platform. Reading or writing a token into the shared memory from one tile without any other processing elements accessing the same memory. And two times six other characterization processed to observe changes of the delay depending on the contention of the interconnect. Therefore the six other tiles have been used to continuously read or write tokens to or from the same shared memory. For each measurement, 1000 Samples have been taken. The results of the characterization is shown in Tab. 4.1. All delays are given in clock cycles.

Tab. 4.1 shows the access delay for reading or writing a single token to a shared memory while 0 to 6 contender are reading or writing tokens to the same shared memory. For the used memory and interconnect, the delay only depends on the access type (read or write) and the amount of contenders. If these parameters are fixed, no variance in communication delay have be observed.

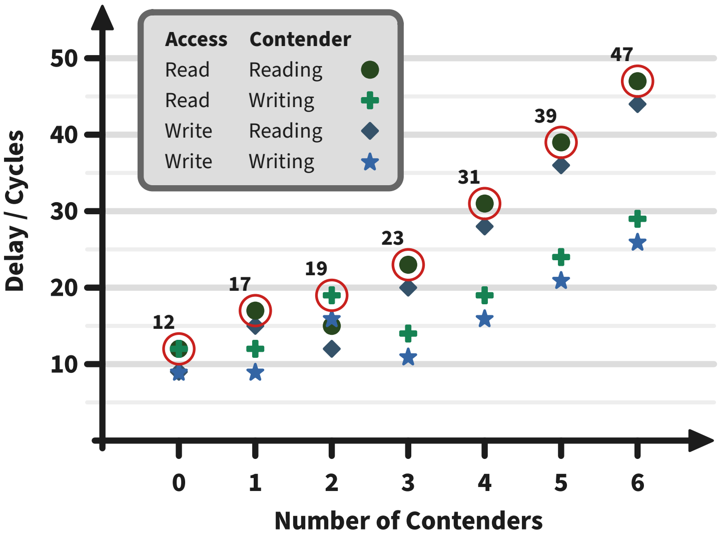


Fig. 4.8 Plot of observed shared memory access delay with 0 to 6 bus contender. The access has been differentiated between read and write access. The worst observed delay are marked by a red circles.

The shared memory access delay data from Tab. 4.1 is visualized in Fig. 4.8. In most cases writing is faster because the write-instructions do not need to wait for the shared memory to response with data. Also writing bus contenders can release the bus access earlier. For reading data, the observed processing element as well as the contenders need to wait, and therefore hold the bus access, until the shared memory responses with the requested data.

Fig. 4.8 clearly shows an anomaly for one and two contenders. For example, reading a token from shared memory is faster with two other reading processing elements (15 cycles) instead of only one (17 cycles). This anomaly can be reproduced but has not been investigated further. It may be an optimization artifact from the interconnect or memory IP component that has been used.

The Cycle Accurate and Transaction Level communication models (Sec 3.4) require a single memory access delay based on the amount n_c of contender. There is no distinction between reading and writing contender. So for each amount n_c of contender, the worst observed shared memory access delay is used as highlighted by the red circles in Fig. 4.8. The Cycle Accurate and Transaction Level communication models do not differentiate between different types of contention. Therefore always the worst observed delay is used. To refine the models with regard to consider different types of bus accesses remains future work.

Table 4.2 Elementary delays from [Vu21]. The delays are given in clock cycles. They are used for the Message Level communication model of the evaluation platform used in my and [Vu21] thesis.

	Mem. Access	**Init.**	**Waiting**	**Pre**	**Post**
Reading	$d_r = 8$	$d_{init_r} = 15$	$d_{rl} = 14$	$d_{pr_r} = 15$	$d_{po_r} = 11$
Writing	$d_w = 5$	$d_{init_w} = 16$	$d_{wl} = 13$	$d_{pr_w} = 15$	$d_{po_w} = 9$
Polling	$d_p = 8$		$d_{pl} = 7$		

4.2.3 Elementary Communication Delays

The Message Level communication model (See Sec 3.4.3), required a different approach of characterization. This section provides a summary of how these elementary communication delays have been extracted from the communication interface of the execution platform used in this thesis. All details of the model and the characterization process can be read in the PhD Thesis of my colleague Hai-Dang Vu [Vu21].

In contrast to the Cycle Accurate and Transaction Level models, the delays of the Message Level model (Sec 3.4.3) are not modeled from the perspective of the execution of specific communication related instructions. The Message Level model delays, here called *elementary delays*, are obtained from the signals activity on the communication bus.

To get the elementary delay, the signals of the communication interface need to be observed. Therefore the evaluation platform got equipped with the Xilinx System Integrated Logic Analyzer (SystemILA[4]) as shown in Fig. 4.4. The SystemILA IP component can be used to monitor the bus activity of the Xilinx AXI4LITE interconnect used as communication interface between the processing elements and shared memory of the evaluation platform used for this thesis.

The integrated logic analyzer has then been used to observe the valid-state signals during reading and writing tokens to shared memory. From these observation the elementary delays have been derived. The actual data transfer has been characterized between shared memory and processing element as well as the time between transfers that represent local computation time on the communicating processing element.

The results of this characterization process for the execution platform used in this thesis can be found in Tab. 4.2. The table lists all parameters of the Message Level Model as described in Sec 3.4.3 and [Vu21]. The rows of the table address the communication driver macro blocks described in Sec 3.4. The values are given in clock cycles.

---

[4] Xilinx System Integrated Logic Analyzer, `https://www.xilinx.com/products/intellectual-property/system-ila.html`, Visited 06.01.2023

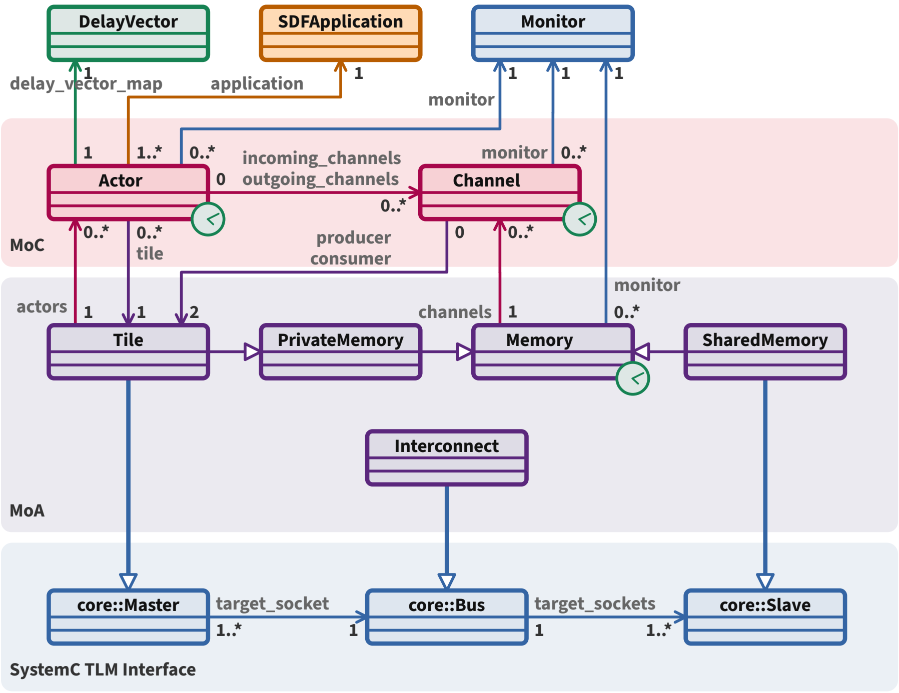


Fig. 4.9 Architecture of the simulation code as UML class diagram. The clock symbols show in which classes delay is simulated.

4.3 Simulation

This section describes the simulation developed for execution time analysis. The SystemC simulation is a representation of the whole system: Hardware, software and their temporal and (optional) their functional behavior. It models the mapped and scheduled SDF application on a specific hardware platform. Each simulation is also configured to use a specific communication model introduced in Sec 3.4 and delay representation for the computational part of the simulated application.

Beside timing behavior, the simulation is also capable by simulating the functional behavior of the software system. This is an optional feature that can be used to further investigate data dependent delays. It is also used to verify functional correct behavior of the simulation and the system under analysis (See Sec 5.3). The functional simulation is focusing on the algorithmic level and is not signal accurate.

In Fig. 4.9 an overview of the architecture of the simulation is shown. It follows the UML standard for class diagrams [OMG17]. Classes are colored corresponding to the models and concepts they implement. Magenta for model of computation, purple for model of architecture related classes. Blue classes implement simulation infrastructure. The class that handles the application code that shall be executed when running a functional simulation is colored in brown. Delay related parts are colored in green. All classes that simulate timing behavior are annotated by a green clock symbol. The following sub sections provide further information about the concepts and implementation of the simulation.

The source code is of the simulation if part of the evaluation setup that is available for download[5]. It has been published[6] first in context of my journal article in 2021 [Ste+21].

4.3.1 SystemC TLM Interface

To simulate timing characteristics of a hardware-software system, the simulation kernel *SystemC* [AS12] is used. SystemC is a `C++` library to describe the timing behavior of hardware and software components and simulate them in an event based simulation. There are several different abstraction level supported. In context of this thesis, the Transaction-level Modeling (SystemC-TLM) layer is used.

Three base classes `Master`, `Bus` and `Slave` provide an abstract interface to the SystemC simulation kernel. They use the SystemC Transaction-level Modeling interface of SystemC. Multiple slave components and multiple master components can be connected to a bus. When data from a `Master` instance shall be transferred to a `Slave` instance or vice versa, an instance of the `Bus` class coordinates the activation of the master and slave components.

To simulate delay, the SystemC `wait` function is used. Since the wait function requires a time unit, the unit `SC_NS` for nanoseconds is used. Anyway instead of nanosecond, all delay numbers are given in clock cycles. And so, also the simulation result represents clock cycles. So for simulation, one clock cycle is defined to take 1 ns.

4.3.2 SDF Application

Beside the execution time characteristics the simulation can also simulate functional behavior of the application. The `SDFApplication` class implements dynamic loading of application code and data to simulate functional behavior as well as different sets of input data. Instructions and data need to be provided as shared object file. The instructions needs to be compatible with the instruction set architecture of the processor that runs the simulation. An `Actor` instance can use this class to execute a specific function represented by this actor.

---

[5] Evaluation code and data: `https://zenodo.org/record/7976829`, visited 28.05.2023.

[6] Simulation source code: `https://zenodo.org/record/4876805`, visited 05.05.2023. Published in context of [Ste+21].

4.3.3 Delay Vector

The `DelayVector` class implements the inference techniques described in Sec 3.3. Each instance of this class reads and processes the characterization data measured for a single hardware setup and actor. When starting the simulation, the measured samples of the actor-tile combination that will be simulated gets loaded. Then the delay vector following a specified distribution function gets derived from that samples. The resulting delay vector gets cached and provided the delays used for each simulation run.

For the calculation of the distribution functions, the Python module *SciPy* [Vir+20] is used within helper functions called within an embedded Python environment that is part of the simulation. The randomization of delay returned by `GetDelay` is done by the *GNU Scientific Library* (GSL) [Gal+07].

The delay vector class provides a single method `GetDelay` that returns a different, random delay d_x each time called. The value ranges of the delays and their distribution corresponds to the desired distribution function that shall be used.

For drawing a random delay d_x from a delay vector $\vec{d}_y$ with a probability $p = P_z(d_x)$ where z represents an arbitrary probability distribution function a method demonstrated in [Mar63] is used. A fixed amount of memory is allocated and filled with values so that the probability of a value to occur within the fixed amount of memory is of the same probability of the value to occur in the distribution to represent [Mar63].

4.3.4 Monitor

The `Monitor` class provides methods to trace simulated delay. In general it is used to trace how long an iteration of an SDF application for a given setup takes. This is then one sample of the results of the simulation. After one iteration of the simulated SDF application is done, the delta delay in simulated clock cycles between the start and end of that iteration is stored in a file. It also allows to output results from the functional simulation to verify that the application works correct.

For debugging and verification the monitor can also trace state changes in execution of SDF actors by tracing the Read, Compute and Write phases. Furthermore the communication steps of the communication models as well as memory access (Read/Write) can be traced.

4.3.5 MoC Implementation

The model of computation described in Sec 3.1 is implemented by the two classes `Actor` and `Channel`.

The `Actor` class provides a general view of the implementation and delay characteristics of an actor. It implements the execution semantic of SDF actors and provides interfaces to other components of the simulation. This class is a pure virtual class and requires a derived class representing a specific actor and that provides at least an implementation of the *Compute* phase. Optionally the derived class can implement the *Read* or *Write* phase, when actual data instead of abstract tokens shall be transferred for functional simulation.

For the compute phase, actual application code can be loaded and executed with the help of the `SDFApplication` class. For the read an write phases, references to instances of the `Channel` class can be used to transfer tokens to other actors.

For simulating the execution time of the compute phase of an actor, delays from an instance of the `DelayVector` class are used. Because a delay vector only describes the execution time distribution of an actor on a specific configuration of a tile, a set of delay vectors is given to each instance of the actor class. Depending on which type of tile the actor gets mapped, the related delay vector is used.

To simulate an actor, a new class must be derived from the abstract base class `Actor`. The derived class can optionally provide functional code and actually transfer data through channels. Anyway it must implement the `Compute` method with at least drawing a random delay via the `GetDelay` method from its delay vector and pass that delay to the SystemC `wait` function.

Individual actors can be marked as starting actor or ending actor. These actor trigger the monitor that observes the simulated execution time of a single iteration of an SDF application. A starting actor is the first actor that gets execution in an iteration. The ending actor is the last actor of an iteration.

The `Channel` class implements the communication between actors. This includes the implementation of the communication delay models from Sec 3.4. The channel class provides two methods: `ReadTokens` and `WriteTokens` that implement the functional behavior of their counter parts in the real system as well as the timing behavior corresponding the selected communication model. Because only instances of the Tiles class can communicate tokens between other tiles via shared memory, each channel needs to know the tile the producer actor is mapped to as well as the tile the consumer actor is mapped to. These can be the same tile for local communication.

When mapping an actor instance to a tile, a reference to that tile instance is given to the actor. This reference is used to configure all channels that are connected to the actor.

4.3.6 MoA Implementation

The model of architecture described in Sec 3.2 is implemented with the classes `Tile`, `Memory`, and `Interconnect`. While `Memory` comes with two derived classes `PrivateMemory` and `SharedMemory`.

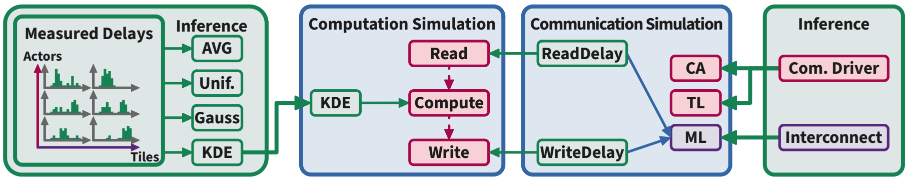


Fig. 4.10 Example of a simulation setup. Based on [Ste+21]

The `Memory` class implements the FIFO access semantics and simulates the read and write delay to access the memory. Only the actual read or write delay of accessing a token in the memory is simulated. The communication delay simulation between memory and a tile is implemented by the `Channel` class.

The two classes `PrivateMemory` and `SharedMemory` are derived from the `Memory` class. Shared memory will be handled by SystemC as a slave device. Therefore it also inherits from the `Slave` class. From perspective of the simulation, private memory is not a dedicated hardware component. So the `Tile` class is derived from the private memory class and inherits the behavior of the memory, while the private memory class does not have any dependencies to the SystemC simulation.

The tile class is a SystemC module with a thread (a SystemC `SC_THREAD`) that gets started with the start of the simulation. Inside the thread of each tile all actors assigned to a tile get initialized first. Then for a configured amount of iterations, each assigned actor gets executed. The `Tile` class implements the static order scheduling of `Actor` instances.

The `Interconnect` class implements the functional part of the First Come First Serve arbitration protocol. Arbitration penalty delays are simulated by the communication model implementation inside the `Channel` class.

4.3.7 Simulation Configuration

Fig. 4.10 presents an abstract view on the execution of a simulation run with focus on the setup of the simulation. The simulation allows to choose between different computation delay models and communication models.

Before being able to run simulations, the simulated actors on the simulated tiles need to be characterized. Therefore, for each combination of actors and tiles the actor computation time needs to be measured. Based on the observed data, different distribution functions can be derived. Furthermore, the communication interconnect needs to be characterized following at least one of the proposed models form Sec 3.4. In context of this thesis, 1 000 000 samples have been measured. From these samples the average value, a uniform distribution and a Gauss distribution can be derived. Furthermore Kernel Density Estimation using Gauss kernels has been applied. In the

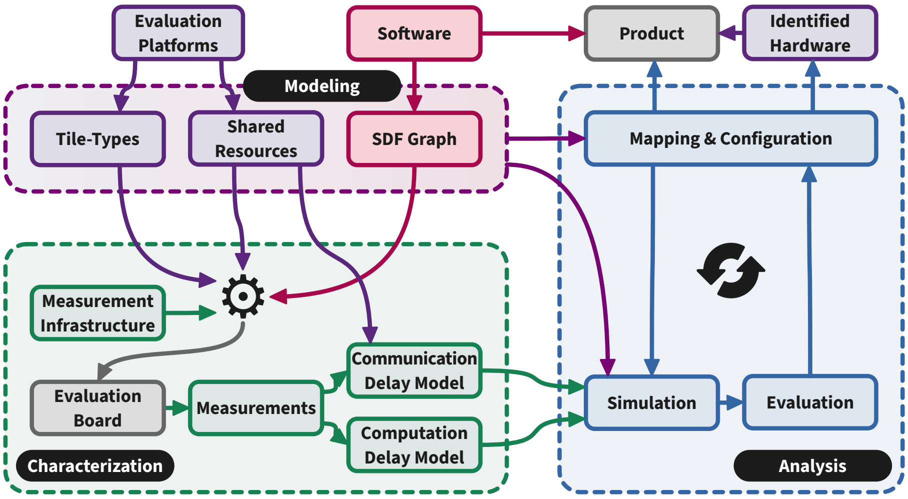


Fig. 4.11 A refined version of Fig. 1.1 showing the components of the Modeling, Characterization and Analysis processes.

example from Fig. 4.10, KDE has been selected as computation delay model and the Message Level communication model for the communication phases of an actor.

When simulating the compute phase of an actor (Fig. 4.10, second box), one delay following the distribution function that has been configured to be used for the simulation is drawn. This delay represents the computation time for the actor for one simulated actor execution.

For simulating the read and write phase, one of the three communication models gets executed, base on which one has been selected when starting the simulation (Fig. 4.10, third box). The models will proceed the simulated time based on the amount of activity on the shared interconnect.

4.4 Work Flow

In Fig. 4.11 a detailed version of the proposed work flow of Fig. 1.1 from Sec 1.5 is shown. The *Modeling*, *Characterization* and *Analysis* sections (dashed boxes) have been expanded by the actual components involved in those processes. Goal of the proposed work flow is to support the design of a product that executes a given software on hardware. Again, hardware related parts are colored in purple, software related parts in magenta. Delay related components in green, and simulation related components in blue.

During the *Modeling* process, described in Chap 3 a SDF representation of the software under analysis gets created as well as a platform model describing tiles that execute the software and shared resources for communication between tiles. These models are then used for the simulation and for configuring the simulation. To support execution time analysis, these models need to be annotated with delay which is done by the characterization process.

During the *Characterization* process (Sec 4.2), that gets the created models as input, these models get annotated with delays. For getting the delays required for the computation and communication simulation the measurement infrastructure form Sec 4.2 is required. An evaluation board that is capable to implement the modeled tiles and shared interconnects and that can execute the software is set up, configured and extended by the measurement infrastructure. The characterization requires measurements for all actors on all tile types. A tile type represent tiles with certain instruction set architectures and hardware accelerators. In the context of this thesis, three tile types exist. All have the same instruction set but different hardware accelerators. More details in Sec 5.1.3. The result of the computation delay characterization is a delay set (Sec 3.3 Def. 3.16) that represents the computation time of each actor of a software on each type of tile.

The models from the modeling process and the delay data from the characterization process are input for the *Analysis* process that has been described in Sec 4.3. The simulation process starts with the product designer choosing a certain mapping and configuration to execute the given software on. Then this mapping and configuration gets simulated. After the simulation the designer has to evaluate if the simulation results fulfill the requirements of the product. If not, a different mapping and/or configuration can be tried. If the simulation results look promising for a good execution time and hardware resource trade-off the hardware that has been identified by the analysis as best suited needs to be build or bought. Then the software can be deployed with the analyses mapping on the selected hardware configuration.

Chapter 5

Models and Simulation Evaluation

5.1 Evaluation Setup

This section describes the setup of hardware and software for the evaluation of the hardware and software models as well as the simulation itself. Two SDF applications with different complexity are used. A *Sobel-Filter* (Sec 5.1.1) as a small and easy to handle application, and a *JPEG Decoder* (Sec 5.1.2) as a more realistic but challenging example. Both SDF graphs of those applications are shown in Fig. 5.1. The evaluation setup including hardware, software, measurement infrastructure, simulation and all characterization and evaluation data are available for download[1]. The source code of both applications have been published[2] first in context of my journal article in 2021 [Ste+21]. The evaluation execution platform (Sec 5.1.3) together with the measurement infrastructure form Sec 4.1.2 gets instantiated on an FPGA. For the evaluation different mapping and configurations are applied to evaluate the computation models (Sec 5.4), communication models (Sec 5.5) and the simulation (Sec 5.6).

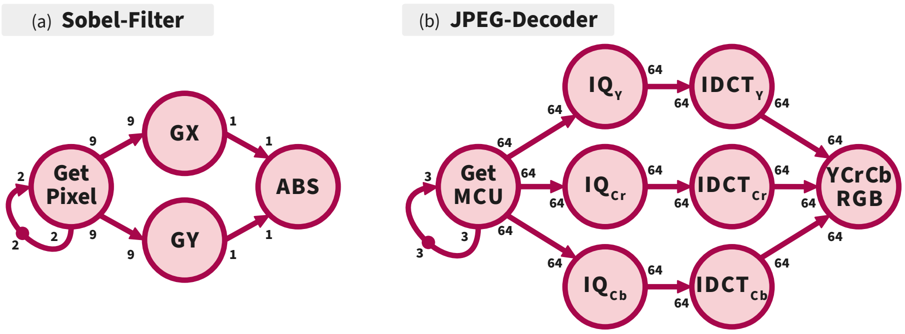


Fig. 5.1 SDF model of a Sobel-Filter (left) and a JPEG-Decoder (right). This figure has been published by me first in [Ste+20].

---

[1] Evaluation code and data: `https://zenodo.org/record/7976829`, visited 28.05.2023.

[2] Sobel-Filter and JPEG-Decoder source code: `https://zenodo.org/record/4876805`, visited 05.05.2023. Published in context of [Ste+21].

5.1.1 Sobel Filter

One of the SDF application used for the evaluation is a *Sobel-Filter* [Sob14]. Its SDF graph representation is shown in Fig. 5.1a. The Sobel-Filter is an high pass filter for two dimensional data and is usually being used for edge detection in images.

This application comes with only four actors, listed in Eqn. 5.1 Each of these Actors have relative simple computation phases with few execution paths which make them highly predictable. The execution paths of the Sobel-Filter can be identified manually. For the *GetPixel* actor, there are 9 execution paths depending on the position of tokens (3×3 pixels matrix) that get read form the input data. The *ABS* actor has 4 execution paths depending on its two input tokens. In case a hardware multiplier is available the actors *GX* and *GY* have only one execution path. If not, the delay highly depends on the input data and is no longer easy to analyze manually.

$$\mathcal{A}_{\text{Sobel}} = \{A_{\text{GetPixel}}, A_{\text{GX}}, A_{\text{GY}}, A_{\text{ABS}}\} \tag{5.1}$$

The amount of data communicated between the actors is relative high compared to the small computational part. Even though the JPEG decoder transfers more token over its channels the computation load during compute phase is much higher. The combination of simple computation and relative high communication makes modeling errors in the communication model have higher impact on the Sobel model than errors in the computation model.

Two additional features make this application an interesting case study for heterogeneous MPSoCs. The core of the Sobel-Filter algorithm is a convolution with a 3×3 matrix. The involved integer multiplication benefits from hardware multiplication. The fact that the edge detection in horizontal and vertical direction implemented by the two actors *GX* and *GY* can be parallelized allows concurrent execution of the two actors.

Input data processed by the *GetPixel* actor is a 48×48 Pixels gray scale noise image. Other patterns like plain black, a horizontal gradient from white to black and a white cross on black background have been considered as well. The noise image lead to most different execution times compared to the regular patterns. In contrast the overall average execution time was only less then nine clock cycles (0.25 %) higher. The input data is hard coded in the GetPixel actor for simplicity reasons. Output data generated by the ABS actor gets discarded. The handling of input and output data is discussed in detail in Chap 6. Only the sampling position in X and Y coordinates is used as actor input. In Fig. 5.1a these position data is shown as feedback loop of the *GetPixel* actor. The position gets initialized by $(0, 0)$.

5.1.2 JPEG Decoder

As second use-case a *JPEG-Decoder*, modeled as shown in Fig. 5.1b is used as an realistic example with high data dependent actor delays. It reads a JPEG [VJ94] encoded image from memory and decodes it into an array of RGB-Pixel data.

A JPEG encoded image consists of segments called minimum coded unit block (MCU). Each iteration of the JPEG decoder decodes one MCU with a size of 8×8 Pixels. In context of JPEG a pixel is split in three color channels: Y, $\mathrm{C_r}$, $\mathrm{C_b}$ - (Luminance, Red-Difference Chroma and Blue-Difference Chroma). The last actor $A_{\mathrm{YCrCb\_RGB}}$ of the JPEG-Decoder consolidates these three color channels and transfers them into RGB (Red, Green, Blue) color channels.

The JPEG-Decoder comes with actors that have different requirements to perform well on an embedded platform. For example the inverse discrete cosine transformation (IDCT) benefits a lot from a floating point unit. The inverse quantization (IQ) benefits from a hardware integer multiplier.

A challenge that comes with the JPEG-Decoder is the high variety of data dependent execution paths of some actors. This makes it barely possible to statically analyze the actors like the GetMCU actor. Getting the MCU inside the *GetMCU* actor involves variable length decoding, run length decoding and Huffman decoding. This leads to a lot data dependent execution paths. Modeling such an actor benefits from a characterization based on measurements.

The JPEG-Decoder use-case is selected to challenge the computation modeling approach regarding accuracy and the communication models regarding simulation time. Due to its high computational load compared to the communicated tokens, errors in the communication model have a relative low impact.

The separation of an MCU into different color channels allows performing the decoding process for each color channel in parallel as shown in Fig. 5.1b. This separation can also be used to reduce the variety of execution times by characterizing the individual actors based on the color channel they decode. In Eqn. 5.2 three individual instances of the *IQ* and *IDCT* actors are considered in the sets of actors $\mathcal{A}_{\mathrm{JPEG}}$. The functional code of the *IQ* actors or *IDCT* actors are the same. But the execution time model of each kind of the actors are different.

$$\mathcal{A}_{\mathrm{JPEG}} = \{A_{\mathrm{GetMCU}}, A_{\mathrm{IQ}_Y}, A_{\mathrm{IQ}_{Cr}}, A_{\mathrm{IQ}_{Cb}}, A_{\mathrm{IDCT}_Y}, A_{\mathrm{IDCT}_{Cr}}, A_{\mathrm{IDCT}_{Cb}}, A_{\mathrm{YCrCb\_RGB}}\} \tag{5.2}$$

The JPEG decoder shown in Fig. 5.1b as well as its implementation is purely academic and does not use any optimization except for a FastIDCT implementation of the $A_{\mathrm{IDCT}_{Y,Cr,Cb}}$ actors. The implementation of the decoder follows the recommendation from [TT93].

The input data is hard coded in the first actor (*GetMCU*) for simplicity reasons. The last actor of the SDF application does a color transformation from the $\mathrm{YC_rC_b}$ color

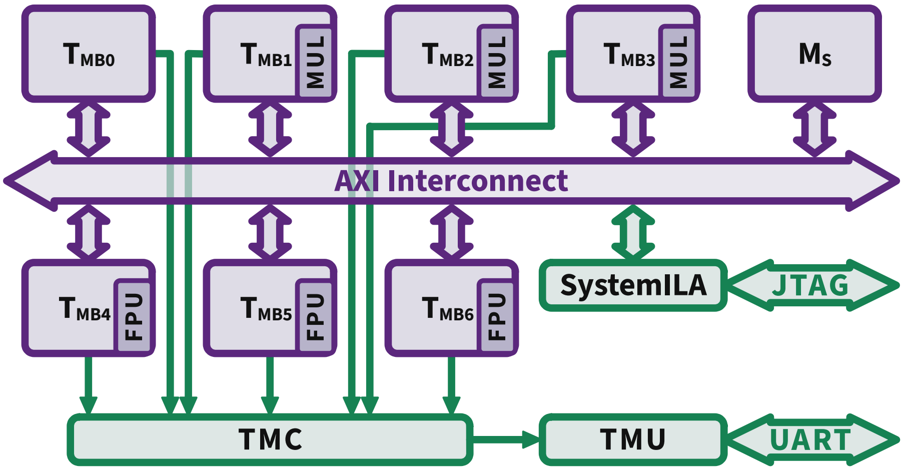


Fig. 5.2 Evaluation platform (purple) with measurement infrastructure (green). The platform has 7 tiles (T) and a shared memory (M_S). Some tiles have a Floating Point Unit (FPU) or a hardware multiplier (MUL). The measurement infrastructure consists of the Time Measurement Controller (TMC) that merges all control signals from the tiles to trigger the Time Measurement Unit (TMU). The SystemILA is used for the Message Level Communication model configuration and is removed afterwards.

model into the RGB color model. Output data generated by this actor gets discarded. The handling of input and output data is discussed in detail in Chap 6.

The execution time of an actor highly changes based on the color channel it decodes. This is because each color channel is encoded with different compression strength. For example red-difference chroma channel (C_r) usually gets encoded with higher loss than the luminance (Y) channel. Higher compression rate, or higher loss, reduces the amount of information the decode can use to restore the original image. Therefore less calculations are need to be performed.

5.1.3 Evaluation Platform

For the experiments I use a heterogeneous multi-processor system with 7 tiles as shown in Fig. 5.2. The purple part of the platform represents the actual execution platform to run the Sobel-Filter (Sec 5.1.1) and the JPEG-Decoder (Sec 5.1.2) application. The green components represent the measurement infrastructure used for characterization of the models. Furthermore the measurement infrastructure is used to observe the execution of the use-case application to evaluate the accuracy of the execution times predicted by the simulation presented in Sec 4.3.

The individual tiles of the execution platform use a Xilinx MicroBlaze [Xil21] soft core as processing element that is equipped with private memory for storing local

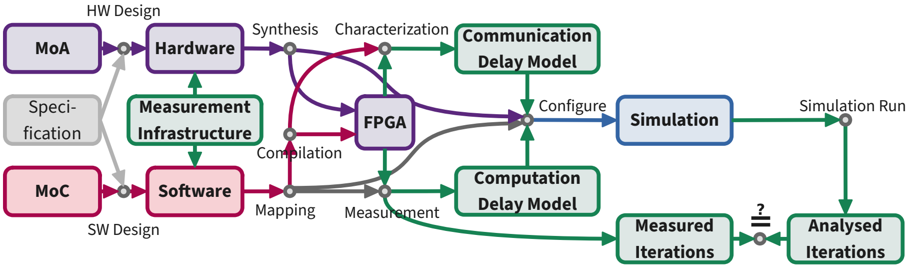


Fig. 5.3 Evaluation concept showing characterization and evaluation steps. Artifacts are shown as boxes, processes as circles. Based on [Ste+19a]

data and instructions of the software executed on that processing element. The tiles T_{MB1}, T_{MB2}, T_{MB3} are extended by a hardware multiplication unit (MUL). All other processing element use soft multiplication. Tiles T_{MB4}, T_{MB5}, T_{MB6} are extended with a floating point unit (FPU). All other processing elements do floating point operation in software. Tile T_{MB0} has no FPU and no hardware multiplication unit. Furthermore there exists a shared memory (M_S) that is used for communication between the tiles. All tiles and the shared memory are connected to the same shared bus, an AXI Interconnect.

The time measurement infrastructure consists of three components. The Time Measurement Controller (TMC) and the Time Measurement Unit (TMU) that are used to measure execution time of software or parts of software executed on one or more processing elements (See Sec 4.1.2 for details). The third component is the Xilinx System Integrated Logic Analyzer (SystemILA[3]) used for configuring the Message Level communication model as described in Sec 4.2.3

The hardware platform as shown in Fig. 5.2 including the timing measurement infrastructure has been synthesized and instantiated on the FPGA part of the Xilinx Zynq-7000[4] SoC.

5.2 Evaluation Concept

For evaluating the concept shown in Fig. 5.3 is followed. This concept is a concrete implementation of the workflow presented in Sec 4.4. In contrast to the proposed workflow, the hardware and software is designed to strictly follow the constraints of the models instead of using the models to approximate existing hardware or software. This guarantees that the hardware and software do not violate any assumptions of the models. Furthermore the measurement infrastructure is not only used for characterization but

---

[3] Xilinx System Integrated Logic Analyzer, `https://www.xilinx.com/products/intellectual-property/system-ila.html`, Visited 06.01.2023

[4] Xilinx Zynq-7000 SoC on an ZC702 Evaluation Kit, `https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html`, Visited 20.05.2023

also for measuring the overall system execution time. This execution time is then used to compare the actual observed execution time distributions with the execution time distribution predicted by the simulation for accuracy evaluation.

The rectangles represent artifacts like models, data and components. These artifacts are processed in multiple steps. The processes are visualized by small circles. The arrows represent data flow from an artifact as input of a process, and the output of a process into new artifacts.

Base on the architecture model (MoA) a tile based execution platform gets designed. The designed hardware gets equipped with measurement infrastructure. This execution platform is described in Sec 5.1.3.

The evaluation software described in Sec 5.1.1 and Sec 5.1.2 gets designed following the constraints of the applied computation model (MoC). The software gets annotated with instructions to trigger the measurement infrastructure of the evaluation hardware.

The hardware gets synthesized and instantiated on a field programmable gate array (FPGA). Then the software actors gets mapped to the individual tiles of the hardware on that they then get executed. The hardware platform instantiation is only done once because it does not change for different experiments. Still, not all tiles may be used in all experiments. Due to the tile based design, inactive tiles have no influence to the overall system behavior and therefore can be assumed to not exist.

The compilation and mapping of the software needs to be done for each experiment. It must be ensured that each time, the compiled instructions are the same to not change the execution time of the software. For characterization and all experiments the same compiler is used with the same configuration. Compile-Time optimization techniques have been deactivated to ensure that each time the code gets compiled, the resulting instructions and instruction order is the same. A less drastic measure would be to check which optimization techniques are deterministic and only apply those.

After deploying the hardware and software to the FPGA the characterization process starts. For the communication model, the interconnect gets characterized. Since some of the communication models presented in Sec 3.4 require knowledge of the instructions executed during communication, the compiled binary code needs to be known. In particular how long the execution time of the instructions take that are related to the communication between actors on different tiles. The computation delay model of the individual actors get characterized for each tile type. Once the characterization is done, the delay models can be used for different experiments.

The simulation uses these characterized models depending on the configuration of the system that shall be simulated. The simulation configuration is described in detail in Sec 4.3.7. Each configuration gets simulated for 1 000 000 iterations of the simulated SDF application. Each simulation run provides one possible execution time of one iteration. This leads to 1 000 000 iteration delays approximately representing the distribution of possible execution times of the simulated system.

For evaluation, the results of the simulation gets compared to the actual measured behavior of the simulated system. The results are compared regarding accuracy of the

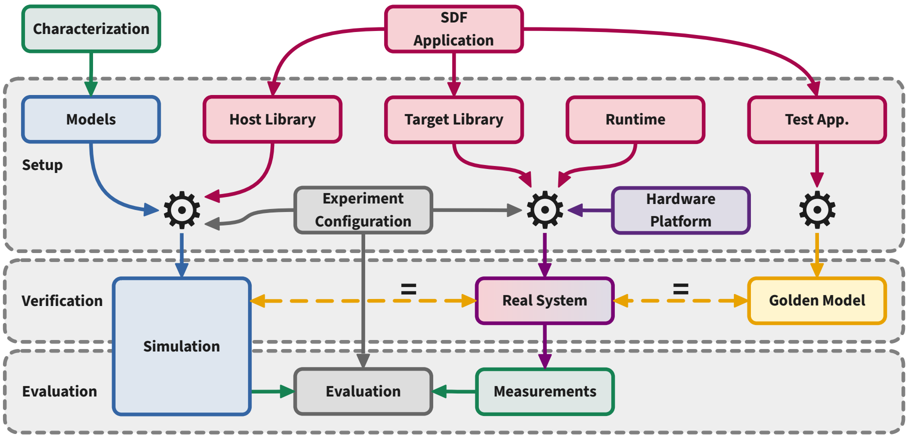


Fig. 5.4 Process of setting up an experiment and verifying that it works functional correct by comparing the results against a golden model.

average execution time as well as the similarity of the distribution functions of the execution time.

5.3 Experiment Execution and Functional Verification

Before an experiment setup as described in Sec 5.1 can be used to evaluate the models and analysis process, it must be ensured that the setup works as expected. Therefore the results of the functional execution of the software executed in simulation and on the real evaluation platform are compared against a golden model.

In Fig. 5.4 this process of running experiments is described. Each experiment consists of three phases, shown in the figure by the gray boxes with dashed lines. The *Setup* phase prepares the experiment, the *Verification* phase ensures that the hardware, software and simulation used for the experiment is working functionally correct. The *Evaluation* phase then uses the simulation and the real system to evaluate the models and processes proposed in this thesis.

The setup phase prepares the experiment as shown in Fig. 5.4. This phase uses the SDF application code that will be used for the experiments and its delay characterization needed for the simulation. Each experiment consists of three execution environments: The *Simulation* that shall be evaluated, the *Real System* that will be used to compare the results of the simulation with and the *Golden Model* that is used to verify that the simulation and the real system are working functionally correct.

The SDF application code (Sobel-Filter form Sec 5.1.1 or JPEG-Decoder form Sec 5.1.2) gets compiled three times, once for each execution environment. The *Host Library* is used to do a functional simulation of the application on the host system that runs simulation. A *Target Library* to execute the application on the actual evaluation platform. And a *Test Application* that executes the code as regular Linux application that is independent from the simulation and the simulated system.

The *Simulation* gets compiled and linked together with the host library of the SDF application. This allows to not only simulate the time behavior of the system under analysis but also the functional behavior of a certain experiment configuration.

The *Real System*, the system under analysis, gets instantiated for example on an FPGA. Therefore the synthesized hardware platform described in Sec 5.1.3 is used. For the hardware platform a prepared generic SDF runtime applications gets compiled and linked against the target library regarding the desired experiment configuration and mapping. Then the applications are written into the platform memory to be executed.

The *Golden Model* is a set of output that got generated by an application that implements the same algorithm as the SDF applications form the Host Library and Target Library implement. Running the three implementations in simulation, the system under analysis must result in the same output generated by the golden model.

In the verification phase, the functional correctness of the simulation and the real system gets checked. Both, the simulation and the system under analysis must output the exact identical results. Furthermore, to ensure that not both system are generating wrong data because of a mapping or scheduling issue, these results must also match the results of the golden model.

When it is ensured that the evaluation setup works as expected, the actual evaluation phase can begin. Therefore the simulation gets executed (optional without functional simulation to improve performance). Furthermore the execution time of running the SDF application executed on the real system gets measured. The simulated and measured iteration delays are then used to evaluate the simulation and the models used inside the simulation.

5.4 Computation Model Evaluation

To model the delay of the computational part of an SDF application, different delay representations can be used. The most simple solution can be the average observed delay. Usually there exist a best and worst observed execution time. These can be interpolated using a uniform distribution of possible execution times. Alternatively, a simple Gauss distribution function can be fitted to the observed delays. A more sophisticated inference technique is Kernel Density Estimation. This section evaluates the differences of these delay representation approaches. The same configuration gets simulated with different computation delay representations. The results are compared against the actual observed iteration execution time. As metric the accuracy of the average execution time and the similarity of the distribution of execution times is used.

Table 5.1 Analysis results for a single tile system using different actor computation delay representations. The table shows the average iteration delay d in clock cycles, the error compared to the observed delay and the Bhattacharyya distance (B.Dist.) between observed and simulated delay distribution.

	Sobel			**JPEG**		
	d	Error	B.Dist.	d	Error	B.Dist.
Measured	4124			2 376 894		
Average	4275	3.661 %	3.442	2 381 099	0.177 %	1.877
Uniform	4114	−0.242 %	0.680	2 241 058	−5.715 %	0.129
Gauss	4275	3.661 %	0.445	2 381 141	0.179 %	0.445
KDE	4273	3.613 %	2.335	2 381 074	0.176 %	0.058

Simulation split into 5 processes, each on a dedicated processor core of an Intel® Core$^{\text{TM}}$ i7-5930K (3.5 GHz).

5.4.1 Experiment Setup

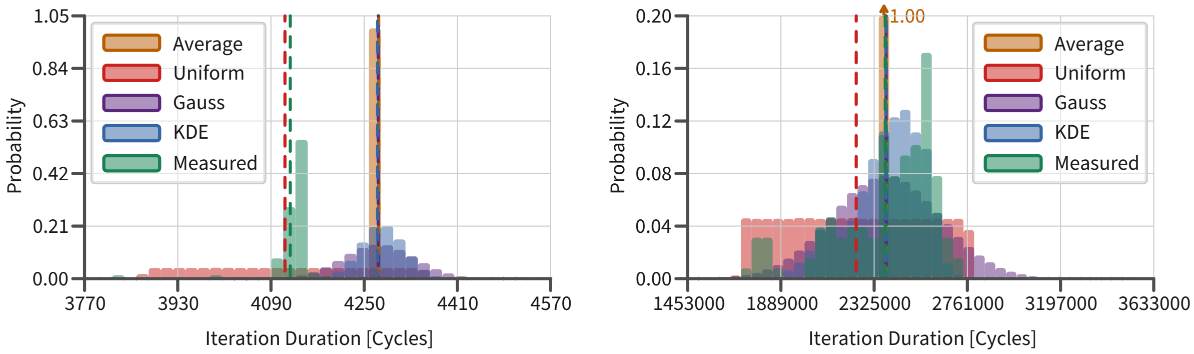
To eliminate influence of the communication model on the error, only a single tile mapping is considered. All actors are mapped to tile T_0. This tile does not provide hardware multiplication or an FPU. So all multiplications and floating point operations are done in software, leading to many possible execution paths. All channels are mapped to the private memory of tile T_0 to avoid any communication overhead.

This mapping gets executed on the real evaluation platform and 1 000 000 samples of the iteration execution times are measured. The same mapping gets simulated for one million samples for each of the distribution functions used to represent the actor computation delays.

The accuracy gets evaluated by comparing the measured average iteration execution time with the simulated one of each simulation. Furthermore the similarity of the distribution of the simulated iteration execution times gets compared against the distribution of the measured execution times. Therefore the Bhattacharyya distance [Bha43] is used as metric. The Bhattacharyya distance describes the similarity of two probability distribution functions. The lower the distance is, the more similar the distribution functions are.

5.4.2 Experiment Results

Tab. 5.1 shows the results of the computation model evaluation experiments. The overall execution time of 1 000 000 iterations have been measured for each, the Sobel and the JPEG setup. Each simulation has simulated 1 000 000 iterations for each investigated distribution function. The results show the average iteration execution time d in clock cycles, and the error of the simulated average iteration delay compared to the observed



(a) Sobel results comparison

(b) JPEG simuation results comparison

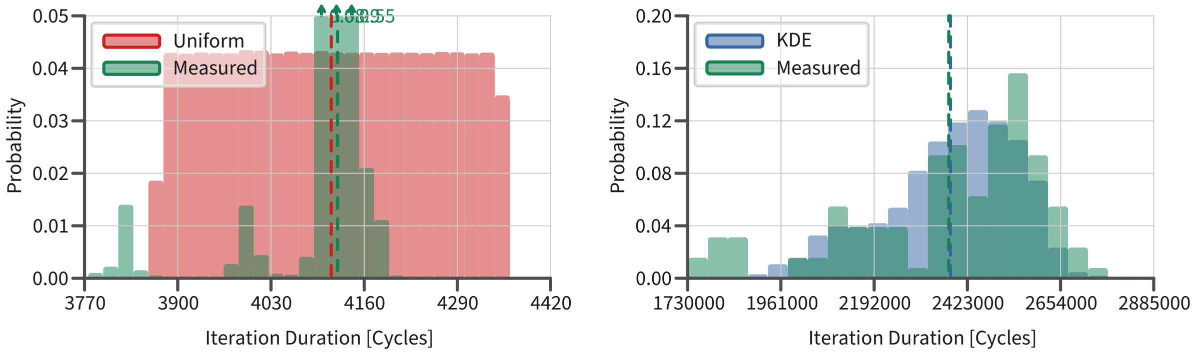
Fig. 5.5 Comparison of the observed iteration delay distribution (green) to the simulated ones using different distribution functions as representation for actor delays. The dashed lines show the average of the distribution.

ones are presented. Beside the error, the Bhattacharyya distance (B.Dist.) is given to compare the similarity of the distribution of iteration delays of the simulation compared to the observed distribution. The smaller the distance is, the better match the distributions.

The data show that using a uniform distribution for presenting actor delays lead to an under approximation of the average iteration delay. This can be seen for both use-cases, Sobel with and error of −0.2 % and JPEG with −5.7 %. For the Sobel experiment, the Gauss distribution leads to the lowest Bhattacharyya distance with 0.445. The iteration delay distribution for the JPEG is closest to the observed one using Kernel Density Estimation. There the Bhattacharyya distance is 0.058.

The different inference techniques to get the individual actor delays barely impact the simulation speed. There may be minor differences in preprocessing data which is negligible compared to other influences on the simulation speed caused by other applications running on the simulation computer. For the *Sobel* simulation all simulations took about 5 min each. The *JPEG* simulations took about 1 h and 45 min for each simulation.

The Bhattacharyya distance represents the similarity of the analyzed distribution function compared to the observed one. In Fig. 5.5 the iteration execution time distributions from the simulation as well as the observed implementation of the Sobel and JPEG application are shown. The distribution of the actually observed iteration delays distribution is shown in green. The distribution of the simulated iteration delays are shown in brown for the results when using average delays for actor execution. For uniform delay distributions the resulting distribution is shown in red, for a Gauss distribution in purple and for the Kernel Density Estimation approach in blue. The vertical dashed lines show the corresponding average execution time.



(a) Sobel simulation best result

(b) JPEG simulation best result

Fig. 5.6 Comparison of the observed iteration delay distribution (green) to the simulated ones using different distribution functions as representation for actor delays. The dashed lines show the average of the distribution.

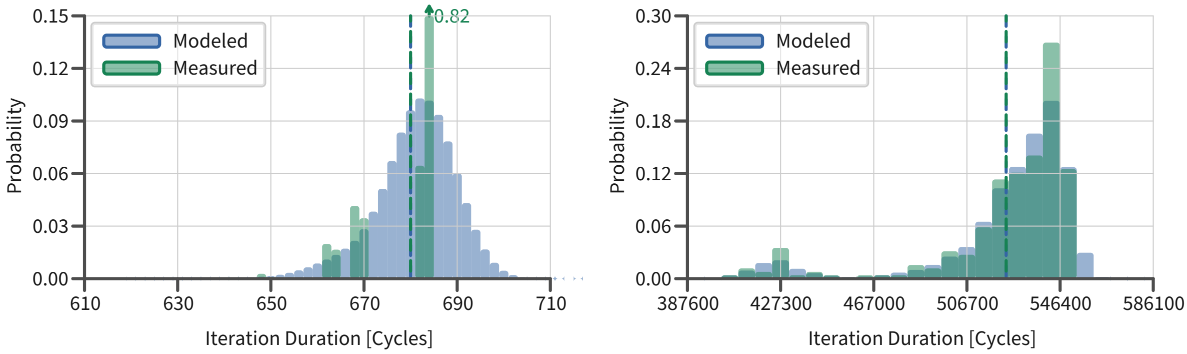
The Sobel experiment Fig. 5.5a show that the Gauss and KDE based simulations result in a similar distribution function with nearly the same average value as also Tab. 5.1 shows. In contrast the uniform representation for actors lead to an average value closer to the observed average.

The results of the JPEG experiment in Fig. 5.5b clearly show that the JPEG decoder comes with some best case peaks (green/Measured). These rare best case events are fully covered by the results of the simulation using a uniform distribution as actor delays (red/Uniform). Anyway, other distribution functions lead to a more accurate average iteration delay prediction. It is also visible that the KDE approach leads to a distribution with its highest probability right in between the two peaks of the most often observed execution times. (better visible in Fig. 5.6b).

5.4.3 Discussion

The experiments show that using a uniform distribution to describe the distribution of delays of an actor leads to under approximation. The reason for this is, that rare best observed delays get a too significant weight. The average delay is closer to the worst observed delays which can be seen on the example of the A_{GX} Actor shown in Fig. 5.7a. Still there may be some rare execution paths that lead to much lower delays. This can be explained by algorithmic optimizations that allows skipping functionality depending on input data. For example the software multiplication used when no hardware multiplier is available, can be reduced to a few instructions when one of the factors is zero.

A closer look at the measured data (Fig. 5.6a green) of the Sobel application show two peaks that are close to the best observed iteration delays. These peaks are better



(a) Actor A_{GX} of the Sobel Application (b) Actor $A_{\mathrm{IDCT\_Y}}$ of the JPEG Application

Fig. 5.7 Example of an actor compute phase delay distribution mapped on a tile without any hardware accelerators. 1 000 000 samples have been measured (green). For the modeled distribution, Kernel Density Estimation has been applied.

covered by the uniform distribution than other distributions because rare best cases have the same weight as common average cases. These characteristics are propagated from the input distribution functions to the simulation result.

In Fig. 5.6b the measured iteration execution time of the JPEG decoder compared to the simulated execution time using KDE as actor delay model shows the advantage of KDE. Similar to the Sobel application, also the JPEG application has some much faster execution paths than the average paths as shown in Fig. 5.7b on the example of the $A_{\mathrm{IDCT\_Y}}$ Actor. These faster paths are covered by the KDE approach while still maintaining the focus, the most likely execution times, around the observed average. Here the faster execution times of an actor compute phase occurs less often than values around the average case. For the simulation this leads to an overall iteration execution that is not false balanced as it happens with an uniform distribution. Still these short execution paths occur more often than using just a Gauss distribution that more likely provide delays around the average delay of an actor.

The computation model evaluation also shows that for simple applications like the Sobel filter application, assuming a Gauss distribution for actor delays may be sufficient enough. For more complex application like the JPEG decode, Kernel Density Estimation provides a better input for analysis. Regarding simulation speed, the computation delay model has no significant impact on the simulation execution time. Still, KDE requires a slightly higher (still negligible) preprocessing time for the delay data used by the simulation.

5.5 Communication Model Evaluation

For the communication model, three different abstraction levels have been proposed in Sec 3.4. The Cycle Accurate model comes with a detailed representation of individual instruction execution delays and token communication steps for each token. The abstract Message Level model comes with highest abstraction, only modeling high level macro blocks of the communication process without looking explicitly at any computation. In between the hybrid Transaction Level model is based on the detailed cycle accurate model but reduces simulation steps by abstracting polling phases to one single simulation step. This section evaluates these three different approaches regarding accuracy and simulation time.

5.5.1 Experiment Setup

The configuration of the evaluation platform focus on high communication load on the shared interconnect. The mappings of the Sobel and JPEG applications are shown in Tab. 5.2. All actors are spread over all existing tiles of the evaluation platform. This mapping causes all actors that are not in their compute phase to poll the channel status and cause lots of bus contention. The computation complexity gets reduced as good as possible by mapping actors to tiles that provide accelerators the actors benefit from with respect to minimize execution paths. All channels are mapped to shared memory even those that could be mapped to private memory to further increase communication load.

These mappings get executed on the real evaluation platform and 1 000 000 samples of the iteration execution times are measured. The same mapping gets simulated for one million samples for each of the communication models. For the simulation, only the average computation delay of an actor is used to represent the actors compute phase.

The accuracy gets evaluated by comparing the measured average iteration execution time with the simulated one of each simulation. Furthermore the simulation time gets measured by simply execute the simulation inside the Linux shell `bash` using the command `time` in front of the actual simulation executable. The real execution time is used to evaluate the simulation time.

I did these simulations and published the data in [Ste+21]. The data from those experiments have been used for Tab. 5.3.

5.5.2 Experiment Results

Tab. 5.3 shows the results of the communication model evaluation experiments. 1 000 000 iterations have been measured or simulated. The results show the average iteration execution time d in clock cycles. In parentheses the error compared to the measured

Table 5.2 Mapping for the Sobel and JPEG communication experiments. All channels have been mapped to shared memory.

	GetPixel	**GX**	**GY**	**ABS**				
Sobel	T_1	T_2	T_3	T_0				
	GetMCU	$\mathbf{IQ_Y}$	$\mathbf{IQ_{Cr}}$	$\mathbf{IQ_{Cb}}$	$\mathbf{IDCT_Y}$	$\mathbf{IDCT_{Cr}}$	$\mathbf{IDCT_{Cb}}$	**YCrCb_RGB**
JPEG	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_0

Table 5.3 Results of the communication evaluation using different communication models. The table shows the average iteration delay d in clock cycles. The error compared to the observed delay the measurement and simulation time t is given as HH:MM:SS and the speed-up. The results have been presented first in [Ste+21].

	Sobel				**JPEG**			
	d	Error	t	t_M/t_S	d	Error	t	t_M/t_S
Measured	3097		00:07:15		941 060		05:13:02	
Cycle Accurate	4623	(49.273 %)	00:19:40	0.4	1 071 080	(13.816 %)	56:29:35	0.1
Transaction Level	2940	(−5.069 %)	00:05:02	1.4	927 239	(−1.469 %)	01:24:36	3.7
Message Level	3105	(0.258 %)	00:01:49	4.0	941 171	(0.012 %)	00:23:50	13.1

Simulation split into 16 processes, each on a dedicated processor core of an AMD Opteron$^{\text{TM}}$ Processor 6328 (3.5 GHz) at OFFIS e.V. `www.offis.de`, accessed on 24. May 2021

iteration delay is given. The required simulation time t_S to simulate all one million iterations are given in hours, minutes and seconds. Furthermore the speed-up is shown to comparing the time t_M it took to measure compared to the time t_S it took to simulate.

Both use-cases show a similar picture of the communication models. The Cycle Accurate model has the highest error with 13.8 % for the JPEG application simulation. Furthermore it took more than 56 h to simulate all one million iterations. The Message Level model has the lowest error with 0.012 % for the JPEG use-case. It also provides the fastest simulation time with only about 24 min for the JPEG application. In this experiment the simulation was more than 13 times faster than measuring. The Transaction Level model provides a moderate simulation speed but results in underestimated iteration delay.

5.5.3 Discussion

For both applications, the Message Level shows best performance and accuracy. Compared to the Transaction Level model it is 3.55 times faster simulating the JPEG application and 13.13 times faster than the actual measurement.

The Cycle Accurate communication model leads to an enormous slow down of the simulation. Even though one million iterations have been simulated on 16 CPU cores in parallel, the simulation took more than ten times longer (10.828) for simulating the JPEG use-case than the actual execution on the evaluation platform. Combined with the high error of the result makes the Cycle Accurate model least suitable for analysis of possible platform configuration, and especially not for design space exploration.

The high error of the Cycle Accurate model can be explained by the fact, that this model does not differentiate between read or write access to the shared memory. Only the amount of communicating tiles is used to obtain communication delay. As Fig. 4.8 from Sec 4.2.2 shows that there can be huge difference in communication delay depending of contenders performing read or write accesses to the shared memory. The cycle accurate model only considers the worst case delay for n contender. This can lead to an error of up to 21 clock cycles for 6 contender, which is nearly as high as the best case of 26 clock cycles per transferred token when all six contender perform a write access. This error sums up with each simulated token communicated.

The Transaction Level modes has a lower error that the Cycle Accurate error and leads to faster simulation time as well. With the TL model, simulation is up to 3.7 times faster than measuring a platform configuration. The error of $-5.1\,\%$ or better makes it a reasonable communication model. Still, it is not as accurate and not as fast as the Message Level model. The Message Level model differentiates between read and write access, which leads to higher accuracy.

5.6 Simulation Time Evaluation

In this section the accuracy and performance of the simulation gets evaluated. Therefore simulations of different complexity regarding number of simulated tiles are compared. The JPEG application is used as realistic use-case and reasonable mappings are applied.

5.6.1 Experiment Setup

The JPEG use-cases will be simulated for different mappings, each mapping using a different amount of tiles. The individual mappings are defined in Tab. 5.4. The experiments are labeled *JPEGx* with x representing the amount of tiles used to execute the application.

Reasonable Heterogeneous Multi-Processor System-on-a-Chip (HMPSoC) configurations have been simulated. When possible, channels are mapped to private memory. The actors are mapped in a way that they benefit from available hardware accelerators like hardware multiplication or floating point units. Furthermore the mapping should allow pipelined execution of the applications. So a next iteration of execution can already start while the previous iteration is still running.

Table 5.4 Mapping for different JPEG experiments. Tiles 1,2,3 have hardware multipliers, Tiles 4,5,6 FPUs. When possible, channels are mapped to private memory, otherwise on shared memory.

JPEG:	**GetMCU**	**IQ$_{\mathbf{Y}}$**	**IQ$_{\mathbf{Cr}}$**	**IQ$_{\mathbf{Cb}}$**	**IDCT$_{\mathbf{Y}}$**	**IDCT$_{\mathbf{Cr}}$**	**IDCT$_{\mathbf{Cb}}$**	**YCrCb_RGB**
JPEG2	T_0	T_4	T_4	T_4	T_4	T_4	T_4	T_4
JPEG3	T_0	T_1	T_1	T_1	T_4	T_4	T_4	T_4
JPEG4	T_0	T_1	T_1	T_1	T_4	T_4	T_4	T_5
JPEG5	T_0	T_1	T_2	T_3	T_1	T_2	T_3	T_4
JPEG6	T_0	T_1	T_2	T_2	T_4	T_5	T_6	T_6
JPEG7	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_6

For the simulation of the computation delay model, the Kernel Density Estimation approach is applied. This approach showed best accuracy in the computation evaluation in Sec 5.4. As communication model, the Message Level approach will be used. In Sec 5.5 this approach showed better accuracy for high communication load and best simulation performance.

The individual mappings get executed on the real evaluation platform to measure 1 000 000 samples of the iteration execution times. The same mappings get simulated for one million samples as well.

The accuracy gets evaluated by comparing the measured average iteration execution time with the simulated one of each simulation. Additional, the best and worst measured execution time gets compared to the best and worst simulated execution timed. Furthermore the similarity of the distribution of the simulated iteration execution times gets compared against the distribution of the measured execution times. Therefore the Bhattacharyya distance [Bha43] is used as metric.

To evaluate the simulation performance the simulation gets executed inside the Linux shell `bash` using the command `time` in front of the actual simulation executable. The real execution time is used to evaluate the simulation time by comparing it with the time it took to measure the actual execution. Therefore the time between the first sample got received until the one millionth sample got received get measured.

5.6.2 Experiment Results

Tab. 5.5 shows the results of the simulation evaluation experiments. Each pair of line shows the measured iteration of a mapping from Tab. 5.4 compared to the simulation of that mapping. 1 000 000 iterations have been measured or simulated for each experiment. The minimum, average and maximum simulated or measured iteration execution times are given in clock cycles. The error given in the *Err.Avg.* column shows the error of the simulated average iteration execution time compared to the measured one. The Bhattacharyya distance in the *B.Dist.* column expresses the similarity of the simulated

Table 5.5 Results of the simulation evaluation using different mappings. The *Err.Avg.* column represents the error of the simulated average iterate execution time compared to the measured one. Measurement and simulation time is given as HH:MM:SS. The Bhattacharyya distance (B.Dist.) describes the similarity of the distribution functions, and t_M/t_S the speed-up of the simulation compared to the measurement.

Experiment	**Min.**	**Avg.**	**Max.**	**Err.Avg.**	**B.Dist.**	**Time** *t*	t_M/t_S
JPEG2 Measured	124 304	493 832	725 228			02:27:25	
JPEG2 Simulated	121 092	495 685	766 441	0.38 %	0.112	00:40:57	3.6
JPEG3 Measured	108 586	466 642	702 117			02:26:22	
JPEG3 Simulated	88 293	479 333	749 063	2.72 %	0.123	00:38:51	3.8
JPEG4 Measured	111 914	469 974	705 433			02:26:48	
JPEG4 Simulated	87 855	479 974	750 118	2.13 %	0.106	00:39:29	3.7
JPEG5 Measured	1 090 546	1 535 249	1 671 340			05:32:48	
JPEG5 Simulated	917 308	1 573 158	1 727 461	2.47 %	1.700	00:39:34	8.4
JPEG6 Measured	106 287	463 531	702 363			02:26:45	
JPEG6 Simulated	86 709	477 363	746 914	2.98 %	0.120	00:40:41	3.6
JPEG7 Measured	107 016	464 251	703 092			02:27:18	
JPEG7 Simulated	88 554	479 134	748 652	3.21 %	0.122	00:39:34	3.7

Simulation split into 5 processes, each on a dedicated processor: Intel® Core^{TM} i7-5930K (3.5 GHz).

iteration execution times distribution with the actual observed one. The smaller the distance is, the better match the distributions of iteration execution times. The *Time* column lists how long the experiments took. The last columns shows how much faster the simulation is, compared to the measurement of the one million samples.

The error of the average simulated iteration execution time compared to the observed time is increasing from 0.38 % for the mapping of the application on only two tiles to 3.21 % for the mapping of the same application on seven tiles. The error over number of tiles is plotted in Fig. 5.8. In all cases, the simulated best execution was below the observed best execution time. Also the simulated worst execution time was in all experiments above the observed worst execution time. The average measured execution time was always below the simulated one.

The Bhattacharyya distance of the measured and simulated iteration execution time distribution is within the range of 0.112 for the 2-tile mapping and 1.7 for the 5-tile mapping.

The simulation time was constant around 40 min. In any cases the simulation of one million iterations was faster than the measuring one million samples of the actual execution. In most cases simulation was about 3.6 to 3.8 times faster. For the 5-tile mapping, the simulation was 8.4 times faster.

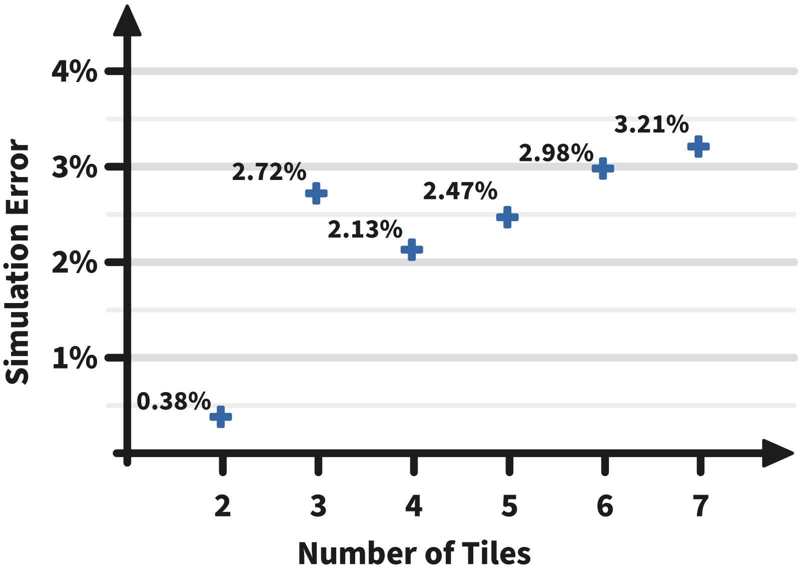


Fig. 5.8 Error of the simulated average iteration execution time for mappings using 2 to 7 tiles.

5.6.3 Discussion

In all cases shown in Tab. 5.5, the measured iteration execution times are within the boundaries of the best and worst simulated execution times. The applied Kernel Density Estimation inference technique smoothes the distribution of the measured computation delays of the characterized actors. This leads to possible computation delays of an actor higher and lower than the actual observed ones. These higher and lower delays representing individual actor compute phases widens the boundaries of best and worst simulated overall iteration execution time. This is on purpose, because rare cases of better or worse computation delays that have not been observed during the characterization of an actor might be covered as well.

Fig. 5.8 shows that with more tiles the error between measured and simulated average iteration execution times raises. This can be explained by the increase of communication and interconnect contention. The more communication takes place, the more influence of the communication model error comes on top of the computation error.

Looking towards using the simulation for design space exploration, the case of *JPEG3* versus *JPEG5* mapping is interesting. The simulated as well as the actual observed iteration execution time show that the mapping using three tiles is much faster than the mapping using 5 tiles. The 3-tiles mapping one iteration takes in average 479 333 clock cycles. With the 5-tiles mapping it takes more than 3 times longer with 1 573 158 clock cycles. The reason behind the increased execution time lays in the designers strategy for assigning the actors of the JPEG decoder to the tiles on that they will be executed. For both mappings, the goal was to allow pipelined execution. Furthermore, the actor A_{GetMCU} had to be mapped to tile T_0 because this was the only tile with enough private memory to store the data and instructions of that actor. These two were common constraints. Different constraints have been applied for using hardware accelerators. With *JPEG3*, the inverse quantization (IQ) actors have been mapped to a tile with hardware multiplier and the inverse discrete cosine transformation (IDCT) actors have been mapped to a tile with FPU. For the *JPEG5* mapping, the strategy was different. There, the A_{GetMCU} and A_{YCrCb_RGB} were mapped to separate tiles, then for the remaining six actors three tiles were still free. Since the JPEG decoder can decode each of the three color channels in parallel, for each color channel the corresponding IQ and IDCT actor have been mapped to the same tile - a tile with hardware multiplier, but without floating point unit. This design decision lead to the situation that the floating point operations of the IDCT had to be done in software which increased the execution time.

The execution time of the simulation is constant about 40 min for any mapping. In contrast the time it takes to measure the same amount of samples that have been simulated depends on the actual execution time it takes to execute the software under analysis. This can clearly be seen by the experiments *JPEG5*. The 5-tile mapping lead to slow execution of the JPEG decoder because actors with floating point operations have been mapped to tiles that do not provide a floating point unit, leading to perform the floating point operations in software. In this case the simulation was up to 8.4 times faster. For more suitable mappings the measurement of one million samples took about 2 h and 27 min. In those cases the simulation was around 3.7 times faster.

The simulations of this experiment have been split into 5 individual processes that have been executed in parallel on a multi-core CPU with 6 physical CPU cores. Each simulation process only had to simulate 200 000 of the 1 000 000 iterations. The parallel execution has significant influence on the simulation time shown in Tab. 5.5. With more available CPU cores the simulation time can be further decreased. Measuring one million samples on real hardware can also be increased by operating multiple evaluation platforms.

Chapter 6
Discussion

This section discusses the limitations of the approach and models presented in this thesis. A discussion of the research questions form Sec 1.2 follows in the conclusion Chap 7.

The proposed methods and models are developed to be used in context of design space exploration, not for safety analysis. This allows focus on an abstract and fast modeling and analysis approach to allow the system designers to narrow down the design space during early development. For a more mature product more detailed and time consuming approaches may be needed to guarantee a safe operation.

Since the characterization of software function is done following a measurement based approach, a simulation based execution time analysis makes only sense when the simulation is faster than using the characterization measurement infrastructure to simply measure an instance of the final product configuration. Still, the approach allows simulating much larger systems than used for characterization. This makes the design space exploration cost efficient by doing characterization on cheap system configurations and simulating expensive system configurations.

For characterization, representative input data is required. Not only to cover all possible execution paths of the code that gets characterized but also to get a representative distribution function of the observed execution times. For the Sobel use-case which implements an edge detection algorithm, a white noise image has been used. Compared to an image of a cross, a gradient and a pure black image, the best and worst observed execution times of these images have been covered by the white noise image. Still, the distribution function is different for each input image. The JPEG-Decoder has been characterized using a landscape photo made on a sunny day in summer in the early afternoon. The characteristics of this image is different to a photo of an object at night. Luminance and texture are different for both image classes. For example texture has an impact on the execution paths of the inverse discrete cosine transformation of the JPED-Decoder. Luminance has an impact on the importance of different color channels, and therefore change the distribution functions for each of them. The huge influence of the input data on the characterization makes it important to be careful with selecting stimuli data for this modeling approach.

Beside the importance of stimuli data, constraints on the execution environment introduced by the models need to be considered. The presented approach has strict limitations on memory hierarchy. Constraints are, that there are no memory caches and that processing elements have private local memory for code and local data. Many modern MPSoCs are equipped with large shared memory and caches for quick access. Even though there might be private local memory available for a processing element it might be only a cache that regularly gets synchronized with a larger shared memory. To make the proposed approach available to more COTS platform, addressing caches in the models is mandatory.

The presented use-cases are algorithms that process defined input data to defined output data. The implementations used in the evaluation and for characterization have their input data hard coded in the starting actor, and discard their output data produced by the ending actor. In real-world situations, the input data would come for example from a sensor and the output would be used by external actuator or other software applications. The assumption that input and output data “appear” and “disappear” in actors may be valid. The compute phase of an actor could implement sensor or actuator drivers. An sensor driver actor can get data from sensors that are then processes within the SDF-Application until an actuator driver actor gives over the data to a different hardware/software system. Still, there are some constraints for those drives, even though they can be abstracted as computational component. The driver may need to access external hardware components. This must not be done using the main system interconnect since access to this interconnect is only allowed during read and write phase, and only to communicate via channels. In most cases sensors and actuators may be connected to a peripheral bus that can be accessed without interfering the main system bus. This is also the case in the proposed execution platform of this thesis. Another constrain is, that the access to external hardware within the driver actors does not interfere with the measurement infrastructure that is also using a peripheral bus. This constraint is also fulfilled by the proposed architecture since MicroBlaze processors come with several independent periphery buses. This is not necessarily the case for other architectures.

Chapter 7
Conclusion

In this thesis I presented a modeling and analysis approach for execution time analysis of synchronous dataflow (SDF) applications on Heterogeneous Multi-Processor Systems-on-Chips (HMPSoCs). For the analysis I follow a simulation based approach using measured delay distributions.

Several research questions have been identified and presented in the introduction in Sec 1.2. To conclude this thesis, I will present the answers to these questions in the following paragraphs.

Are measured delays a suitable abstraction for timing of application functions? The model of computation used in this thesis (SDF, See Sec 3.1) allows dividing application into several interference free components, called actors. The compute phase of an actor can be represented by its computation delay for time analysis. In Sec 3.3 I showed how to deal with low level hardware effects and data dependency influencing the execution delay. Instead of a single delay like the average of measured delays, the distribution of the observed delays is used. The representation of computation delays by a delay distribution functions (Described in Sec 3.3) has been evaluated in Sec 5.4. A Gauss distribution was well suited for simple applications like a Sobel-Filter. A more complex use-case, a JPEG decoder, required a more sophisticated inference technique to obtain a delay distribution that represents the computation phase of an actor. Kernel Density Estimation showed good results and allowed to not just get a reasonable average execution time with error less than 5 % but also a representative distribution function of possible execution times as simulation result. To get a well suited distribution function for an actor, it is important to characterize this actor on all different instruction set architectures with all different combination of available hardware accelerators like floating point units or hardware multiplier. Furthermore, for the later usage of the characterized application representative stimuli data needs to be applied.

What is necessary to predict execution time distributions of an application on an MPSoC in short time? In context of this thesis a simulation should be executed faster than gathering the same amount of data by simply measuring of a real platform. Using SDF as computation model and representing the computation as delays following a certain distribution function has been shown as fast and accurate method in Sec 5.4 to model and simulation the computational part of an application. The communication model still was a challenge. Three different approaches have been presented in Sec 3.4. These models have been evaluated and compared in Sec 5.5. The Message Level model first introduced by [Vu+21] allowed the fastest simulation by still maintaining good accuracy. In Sec 5.6 the performance of the simulation presented in this thesis has been evaluated. While measuring a certain mapping and configuration usually took more than 2 h, the simulation finished within less than 45 min in most cases. In all cases the simulation was faster. The simulation was executed on a general purpose desktop computer. The simulation allows parallel execution on a compute cluster with many CPU cores that can further reduce simulation time.

How can I achieve good accuracy for analyzing realistic systems? In Sec 5.6 I showed the analysis of a JPEG-Decoder application executed on a Heterogeneous Multi-Processor System-on-a-Chip with 7 tiles and different hardware accelerators. The accuracy depends on the quality of the communication model and the computation delay model. Since the computation delay model is based on the distribution of measured delays representative stimuli data is mandatory for good accuracy. The communication models presented in this thesis rely on a deeper understanding of the software, or hardware implementation of the communication drivers or interconnects. The modeling and analysis approach presented in this thesis allowed a prediction of the average execution time of a certain mapping and configuration with an error less than 5 % (Tab. 5.5). For modeling the computation time, Kernel Density Estimation has been applied on measured execution times. For the characterization, data dependencies and hardware introduced variance in execution time has been considered. In comparison with several communication models, the Message Level model first introduced by [Vu+21] provided best accuracy. The software application fulfilled all assumptions introduced by the used computation model (Sec 3.1). The up to seven tiles MPSoC fulfilled all assumptions introduced by the architecture model (Sec 3.2). Applying the presented approach to software or hardware that violates these assumptions can have a negative impact on the accuracy.

How can I achieve good scalability for realistic systems? The evaluation showed that an important aspect for this approach, when it comes to scalability, is the right choice of abstraction of communication. In Sec 5.5 I compared three different communication models. The Message Level model by [Vu21] showed best performance. Even with high communication activity, the simulation was executed within 24 min (Tab. 5.3). Beside the simulation itself, the characterization can become a time consuming process. The evaluation platform used in this thesis consists of three different tile types. Each of the 8 actors of the JPEG-Decoder application had to be characterized for each tile type. Therefore 24 times one million samples of the actor execution have been measured. This is only required once, but still can become time consuming for a higher variety of tiles or applications with more actors. For the evaluation of the simulation in Sec 5.6 I simulated a JPEG-Decoder that consists of 8 actors on an MPSoC with up to 7 tiles within an analysis time of less than 45 min (Tab. 5.5). The simulation time was barely increasing with increasing number of tiles. In any cases the simulation was multiple times faster than measuring the execution time on a real instance of the simulated system. For the evaluation, 1 000 000 iterations have been simulated in total. These number of iterations can be distributed to several instances of the simulation. In most cases five instances of the simulation have been executed in parallel, each simulating 200 000 iterations. These processes were executed on a desktop computer. High performance computer with many CPU cores would allow more parallel instances and therefore faster simulation. Simulating only one iteration on the i7-5930K CPU used for most of the evaluation is computed in 8.8 s including the setup phase of the simulation and preprocessing the actors computation delay data (Applying Kernel Density Estimation).

What are the limitations of my approach when applying to consumer of-the-shelf platforms? The proposed analysis and characterization rely on software that can be represented as SDF graph (Sec 3.1) and a tile based hardware architecture (Sec 3.2). For the characterization, the hardware additionally need to provide a dedicated peripheral interface or GPIO port to allow connecting measurement infrastructure (Sec 4.1.2) to the hardware platform without interfere with the execution of the software under analysis. For evaluation, a custom hardware platform has been designed and instantiated on a Field Programmable Gate Array (FPGA). Beside a highly customized hardware platform of-the-shelf MPSoCs exists, that also fulfills the requirements for the proposed approach. One example is the Parallax Propeller[1]. The Propeller is a MPSoC that comes with 8 independent tiles called “Cog” connected to a shared interconnect called “Hub”. The Hub implements a round-robin arbitration protocol. Each tile comes with private memory for 512 words. For data exchange between Cogs (Tiles), a 32 kB shared memory exist. Furthermore it is possible to connect external measurement infrastructure to GPIO pins that can be accessed by the Cogs without interfering with each other.

---

[1] Parallax Inc. Propeller 1 `https://www.parallax.com/propeller/`, visited 05.01.2023

Chapter 8
Future Work

To allow more complex bus systems a probabilistic communication time model instead of the analytical ones presented in this thesis, can be beneficial. Furthermore a fully probabilistic model allows Statistical Model Checking (SMC) techniques. By moving from a hybrid simulation model using a probabilistic approach for computation and analytic approach for communication to a fully probabilistic simulation model, the application of more advanced probabilistic analysis methods such as SMC [Ste+19a; Nou+14a] become feasible. SMC approaches reduces the required number of simulation runs by using statistical algorithms such as Monte-Carlo or Sequential Probability Ratio Test (SPRT). By controlling the number of simulation runs, a trade-off between high confidence and fast analysis time is possible. The instrumentation and monitoring of SystemC models to carry out statistical analysis were presented in [NLQ16]. A first evaluation has been done in a technical report [Ste+19b] by me and Hai-Dang Vu and further investigated by Hai-Dang Vu as contribution to our publication in [Ste+19a].

In the context of WCET analysis, authors in [Caz+13a; Caz+16] proposed a measurement based approach in combination with hardware and/or software randomization techniques to conduct a probabilistic worst-case execution time (pWCET) through the application of Extreme Value Theory (EVT) [And70]. In future work, the usage of EVT for the proposed models can be assessed. EVT can potentially be applied on our execution time distributions to provide an analytic pWCET.

For more complex applications, tool support can improve handling data dependencies. Automation can be applied to identifying all relevant actor internal execution paths, and annotate them with execution times or execution time distributions. Within this automation process, automatic characterization can be integrated so that stimuli data activating certain execution paths can be generated, the actor executed and its delay distribution for the stimulated execution paths observed.

My colleague Quentin Dariol follows the analysis approach presented in this thesis for his PhD thesis . He uses some modeling and characterization techniques as well as the simulation for modeling and analyzing artificial neural networks (ANN) on MPSoCs [Dar+22]. He extends the models and characterization technique to also allow analyzing the power consumption of ANNs on MPSoCs [Dar+23].

An important topic for future work is loosening the constraints made to the hardware architecture so that the techniques presented in this thesis can be applied to more of-the-shelf hardware platforms. My colleague Hai-Dang Vu addressed the extension of the presented models to allow Dynamic RAM memory and caches in his PhD thesis [Vu21].

Chapter 9
References

My Publication

This section lists all publications I did as main author or as co-author in context of this thesis. This includes workshop and conference papers as well as journal articles. All listed publications are peer reviewed and follow scientific standards. Beside the listed peer reviewed publications, a technical report [Ste+19b] has been published. Details about the context and contributions are presented in Sec 1.4. The list is ordered by the publication date. Each entry starts with a reference to the actual reference entry in the generated reference list is the next section. Publications where I am the main contributor are marked with a star (⋆).

- [Ste+17] ⋆ Ralf Stemmer, Maher Fakih, Kim Grüttner and Wolfgang Nebel; "Towards State-Based RT Analysis of FSM-SADFGs on MPSoCs with Shared Memory Communication". In *Proceedings of the 9th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. ACM. 2017
- [SFS17] Christof Schlaak, Maher Fakih and Ralf Stemmer "Power and Execution Time Measurement Methodology for SDF Applications on FPGA-based MPSoCs". In *arXiv preprint arXiv:1701.03709*. 2017
- [Kle+19] Oliver Klemp, Maher Fakih, Kim Grüttner, Ralf Stemmer and Wolfgang Nebel "Experimental Evaluation of Scenario Aware Synchronous Data Flow based Power Management". In *Proceedings of the International Conference on Omni-Layer Intelligent Systems*. 2019
- [Ste+19c] ⋆ Ralf Stemmer, Henning Schlender, Maher Fakih, Kim Grüttner and Wolfgang Nebel "Probabilistic State-Based RT-Analysis of SDFGs on MPSoCs with Shared Memory Communication". In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. March 2019
- [Ste+19a] ⋆ Ralf Stemmer, Hai-Dang Vu, Kim Grüttner, Sébastien Le Nours, Wolfgang Nebel and Sébastien Pillement "Experimental evaluation of probabilistic execution-time modeling and analysis methods for SDF applications on MPSoCs". In *2019 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Springer. July 2019
- [Ste+20] ⋆ Ralf Stemmer, Hai-Dang Vu, Kim Grüttner, Sébastien Le Nours, Wolfgang Nebel and Sébastien Pillement "Towards Probabilistic Timing Analysis for SDFGs on Tile Based Heterogeneous MPSoCs". In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*. 2020

- [Vu+21] Hai-Dang Vu, Sébastien Le Nours, Sébastien Pillement, Ralf Stemmer and Kim Grüttner “A Fast Yet Accurate Message-level Communication Bus Model for Timing Prediction of SDFGs on MPSoC”. In *Asia and South Pacific Design Automation Conference_ASP-DAC 2021 (Virtual Conference)*. 2021
- [Ste+21] ⋆ Ralf Stemmer, Hai-Dang Vu, Sébastien Le Nours, Kim Grüttner, Sébastien Pillement and Wolfgang Nebel “A Measurement-based Message-level Timing Prediction Approach for Data-Dependent SDFGs on Tile-based Heterogeneous MPSoCs”. In *Applied Sciences* 11.14. Multidisciplinary Digital Publishing Institute. 2021
- [Dar+22] Quentin Dariol, Sébastien Le Nours, Sébastien Pillement, Ralf Stemmer, Domenik Helms and Kim Grüttner “A Hybrid Performance Prediction Approach for Fully-Connected Artificial Neural Networks on Multi-core Platforms”. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Springer. 2022
- [Dar+23] Quentin Dariol, Sébastien Le Nours, Domenik Helms, Ralf Stemmer, Sébastien Pillement and Kim Grüttner “Fast Yet Accurate Timing and Power Prediction of Artificial Neural Networks Deployed on Clock-Gated Multi-Core Platforms”. In *Proceedings of the DroneSE and RAPIDO: System Engineering for constrained embedded systems*. 2023

References

[Ahm+14] Waheed Ahmad et al. “Resource-Constrained Optimal Scheduling of Synchronous Dataflow Graphs via Timed Automata”. In: *Proceedings of 14th IEEE International Conference on Application of Concurrency to System Design (ACSD)*. IEEE, 2014.

[And70] C. W. Anderson. “Extreme value theory for a class of discrete distributions with applications to some stochastic processes”. In: *Journal of Applied Probability* 7.1 (1970), pp. 99–113. DOI: 10.2307/3212152.

[Arc] Timing Architect. http://www.timing-architects.com.

[Arp+09] T. Arpinen et al. “Performance Evaluation of UML-2 Modeled Embedded Streaming Applications with System-Level Simulation”. In: *EURASIP Journal on Embedded Systems* 2009, 826296 (2009).

[AS12] Accellera and the SystemC community. “IEEE Standard for Standard SystemC Language Reference Manual”. In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (2012), pp. 1–638. DOI: 10.1109/IEEESTD.2012.6134619.

[Bac+92] F. Baccelli et al. *Synchronization and linearity, an algebra for discrete event systems*. New York: Wiley & Sons Ltd, 1992.

[Bas+11] A. Basu et al. “Rigorous Component-Based System Design Using the BIP Framework”. In: *IEEE Software* 28.3 (May 2011), pp. 41–48. ISSN: 0740-7459. DOI: `10.1109/MS.2011.27`.

[Bha43] Anil Bhattacharyya. “On a measure of divergence between two statistical populations defined by their probability distributions”. In: *Bull. Calcutta Math. Soc.* 35 (1943), pp. 99–109.

[BHM08] Aske Brekling, Michael R. Hansen, and Jan Madsen. “Models and formal verification of multiprocessor system-on-chips”. In: *The Journal of Logic and Algebraic Programming*. The 16th Nordic Workshop on the Prgramming Theory (NWPT 2006) 77.1-2 (Sept. 2008), pp. 1–19. ISSN: 1567-8326.

[BKR07] Steffen Becker, Heiko Koziolek, and Ralf Reussner. “Model-based performance prediction with the palladio component model”. In: *Proceedings of the 6th international workshop on Software and performance*. 2007, pp. 54–65.

[Boy+13] Benoıt Boyer et al. “PLASMA-lab: A flexible, distributable statistical model checking library”. In: *Quantitative Evaluation of Systems: 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings 10*. Springer. 2013, pp. 160–164.

[BPT10] A. Bobrek, J. M. Paul, and D. E. Thomas. “Stochastic Contention Level Simulation for Single-Chip Heterogeneous Multiprocessors”. In: *IEEE Transactions on Computers* 59.10 (2010), pp. 1402–1418.

[Bri+15] O. Bringmann et al. “The next generation of virtual prototyping: Ultra-fast yet accurate simulation of HW/SW systems”. In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2015, pp. 1698–1707.

[Bük13] Matthias Büker. “An Automated Semantic-Based Approach for Creating Task Structures”. Dissertation. University of Oldenburg, 2013.

[Bul+12] P. Bulychev et al. “Statistical model checking for priced timed automata”. In: *In Proc. 10th workshop on quantitative aspects of programming languages and systems (QAPL’12)*. 2012.

[Caz+13a] Francisco J Cazorla et al. “Proartis: Probabilistically analyzable real-time systems”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 12.2s (2013), p. 94.

[Caz+13b] Francisco J. Cazorla et al. “PROARTIS: Probabilistically Analyzable Real-Time Systems”. In: *ACM Trans. Embed. Comput. Syst.* 12.2s (May 2013), 94:1–94:26. ISSN: 1539-9087. DOI: `10.1145/2465787.2465796`. URL: `http://doi.acm.org/10.1145/2465787.2465796`.

[Caz+16] Francisco J. Cazorla et al. “PROXIMA: Improving measurement-based timing analysis through randomisation and probabilistic analysis”. In: *Digital System Design (DSD), 2016 Euromicro Conference on*. IEEE. 2016, pp. 276–285.

[Che+15] M. Chen et al. “Variation-aware evaluation of MPSoC task allocation and scheduling strategies using statistical model checking”. In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2015, pp. 199–204.

[Chr] ChronSIM. http://www.inchron.com/tool-suite/chronsim.html.

[Cul+10] C. Cullmann et al. “Predictability considerations in the design of multi-core embedded systems”. In: *Proceedings of the Embedded Real Time Software and Systems Congress (ERTS^2) 2010*. 2010.

[Dam+12] Morteza Damavandpeyma et al. “Modeling static-order schedules in synchronous dataflow graphs”. In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2012, pp. 775–780.

[Dar+22] Quentin Dariol et al. “A Hybrid Performance Prediction Approach for Fully-Connected Artificial Neural Networks on Multi-core Platforms”. In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Springer. 2022, pp. 250–263.

[Dar+23] Quentin Dariol et al. “Fast Yet Accurate Timing and Power Prediction of Artificial Neural Networks Deployed on Clock-Gated Multi-Core Platforms”. In: *Proceedings of the DroneSE and RAPIDO: System Engineering for constrained embedded systems*. 2023, pp. 79–86.

[Dav+11] Alexandre David et al. “Statistical Model Checking for Networks of Priced Timed Automata”. English. In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Uli Fahrenberg and Stavros Tripakis. Vol. 6919. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 80–96. ISBN: 978-3-642-24309-7.

[Döm+08] R. Dömer et al. “System-on-Chip Environment: A SpecC-Based Framework for Heterogeneous MPSoC Design”. In: *EURASIP Journal on Embedded Systems* 2008 (2008).

[Erb+07] C. Erbas et al. “A Framework for System-Level Modeling and Simulation of Embedded Systems Architectures”. In: *EURASIP Journal on Embedded Systems* (2007).

[Fak+15] Maher Fakih et al. “State-based real-time analysis of SDF applications on MPSoCs with shared communication resources”. In: *Journal of Systems Architecture - Embedded Systems Design* 61.9 (2015), pp. 486–509. ISSN: 1383-7621. DOI: http://dx.doi.org/10.1016/j.sysarc.2015.04.005. URL: http://www.sciencedirect.com/science/article/pii/S1383762115000326.

[Fak16] Maher Fakih. “State-Based Real-Time Analysis of Synchronous Data-flow (SDF) Applications on MPSoCs with Shared Communication Resources”. PhD thesis. Universität Oldenburg, 2016.

[Gaj+09] Daniel D Gajski et al. *Embedded system design: modeling, synthesis and verification*. Springer Science & Business Media, 2009.

[Gal+07] Mark Galassi et al. “The gnu scientific library reference manual, 2007”. In: *URL http://www.gnu.org/software/gsl* (2007).

[Ger+09] A. Gerstlauer et al. “Electronic System-Level Synthesis Methodologies”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.10 (2009), pp. 1517–1530.

[Gia+12] Georgia Giannopoulou et al. “Timed Model Checking with Abstractions: Towards Worst-Case Response Time Analysis in Resource-Sharing Manycore Systems”. In: *Proc. International Conference on Embedded Software (EMSOFT)*. Tampere, Finland: ACM, Oct. 2012, pp. 63–72.

[Gus+10] Andreas Gustavsson et al. “Towards WCET Analysis of Multicore Architectures Using UPPAAL”. In: *WCET*. 2010, pp. 101–112.

[Hen+05] Rafik Henia et al. “System Level Performance Analysis - the SymTA/S Approach”. In: *IEE Proceedings Computers and Digital Techniques*. 2005.

[Hua+12] Kai Huang et al. “Embedding Formal Performance Analysis into the Design Cycle of MPSoCs for Real-time Streaming Applications”. In: *ACM Trans. Embed. Comput. Syst.* 11.1 (Apr. 2012), 8:1–8:23. ISSN: 1539-9087. DOI: `10.1145/2146417.2146425`. URL: `http://doi.acm.org/10.1145/2146417.2146425`.

[HV06] Martijn Hendriks and Marcel Verhoef. “Timed automata based analysis of embedded system architectures”. In: *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, 8–pp.

[Int] Intel. *Intel CoFluent Studio*. http://www.intel.com/content/www/us/en/cofluent/intel-cofluent-studio.html.

[JLS12] C. Jegourel, A. Legay, and S. Sedwards. “A Platform for High Performance Statistical Model Checking - plasma”. In: *In Proc. International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS’12)*. 2012, pp.498–503.

[Kan+06] T. Kangas et al. “UML-Based Multiprocessor SoC Design Framework”. In: *ACM Transactions on Embedded Computing Systems* 5.2 (May 2006), pp. 281–320.

[Kei+09] J. Keinert et al. “SystemCoDesigner-An automatic ESL synthesis approach by design space exploration and behavior synthesis for streaming applications”. In: *ACM Trans. Des. Autom. Electro. Syst.* 14.1 (Jan. 2009), pp.1–23.

[Kir+05] Raimund Kirner et al. “Using measurements as a complement to static worst-case execution time analysis”. In: *Intelligent Systems at the Service of Mankind* 2.8 (2005), p. 20.

[Kle+19] Oliver Klemp et al. “Experimental Evaluation of Scenario Aware Synchronous Data Flow based Power Management”. In: *Proceedings of the International Conference on Omni-Layer Intelligent Systems*. 2019, pp. 80–85.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. “PRISM 4.0: Verification of Probabilistic Real-time Systems”. In: *In Proc. International Conference on Computer Aided Verification (CAV’11)*. July 2011, pp. 585–591.

[Kre+08] J. Kreku et al. “Combining UML2 Application and SystemC Platform Modelling for Performance Evaluation of Real-Time Embedded Systems”. In: *EURASIP Journal on Embedded Systems* 2008 (Jan. 2008), 6:1–6:18.

[Kum09] Akash Kumar. “Analysis, design and management of multimedia multiprocessor systems”. PhD thesis. Eindhoven University of Technology, 2009.

[KW16] Joost-Pieter Katoen and Hao Wu. “Probabilistic Model Checking for Uncertain Scenario-Aware Data Flow”. In: *ACM Trans. Des. Autom. Electron. Syst.* 22.1 (Sept. 2016), 15:1–15:27. ISSN: 1084-4309. DOI: `10.1145/2914788`. URL: `http://doi.acm.org/10.1145/2914788`.

[Le 10] Jean-Yves Le Boudec. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland, 2010.

[LM87] Edward A Lee and David G Messerschmitt. “Synchronous data flow”. In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245.

[LMS13] K. Lu, D. Müller-Gritschneder, and U. Schlichtmann. “Analytical timing estimation for temporally decoupled TLMs considering resource conflicts”. In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2013, pp. 1161–1166.

[Lv+10] M. Lv et al. “Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software”. In: *2010 31st IEEE Real-Time Systems Symposium*. 2010, pp. 339–349.

[Mar63] George Marsaglia. “Generating discrete random variables in a computer”. In: *Communications of the ACM* 6.1 (1963), pp. 37–38.

[MG13] Avinash Malik and David Gregg. “Orchestrating Stream Graphs Using Model Checking”. In: *ACM Trans. Archit. Code Optim.* 10.3 (Sept. 2013), 19:1–19:25. ISSN: 1544-3566. DOI: `10.1145/2512435`. URL: `http://doi.acm.org/10.1145/2512435`.

[Mir] MirabilisDesign. www.mirabilisdesign.com.

[NLQ16] Van Chan Ngo, A. Legay, and J. Quilbeuf. “Statistical Model Checking for SystemC Models”. In: *2016 IEEE 17th International Symposium on High Assurance Systems Engineering* (2016), pp. 197–204.

[Nou+14a] A. Nouri et al. “Building faithful high-level models and performance evaluation of manycore embedded systems”. In: *ACM/IEEE International conference on Formal methods and models for codesign*. 2014.

[Nou+14b] Ayoub Nouri et al. “Statistical model checking QoS properties of systems with SBIP”. In: *International Journal on Software Tools for Technology Transfer* 17.2 (2014), pp. 171–185.

[NWY99] Christer Norstrom, Anders Wall, and Wang Yi. “Timed automata as task models for event-driven systems”. In: *Real-Time Computing Systems and*

Applications, 1999. RTCSA'99. Sixth International Conference. IEEE, 1999, pp. 182–189.

[OMG17] Object Management Group (OMG). *OMG Unified Modeling Language (OMG UML)*. Dec. 2017.

[Par62] Emanuel Parzen. "On Estimation of a Probability Density Function and Mode". In: *Ann. Math. Statist.* 33.3 (Sept. 1962), pp. 1065–1076. DOI: `10.1214/aoms/1177704472`. URL: `https://doi.org/10.1214/aoms/1177704472`.

[Per+09] Simon Perathoner et al. "Influence of Different Abstractions on the Performance Analysis of Distributed Hard Real-time Systems". In: *Des. Autom. Embedded Syst.* 13.1-2 (June 2009), pp. 27–49. ISSN: 0929-5585. DOI: `10.1007/s10617-008-9015-1`. URL: `http://dx.doi.org/10.1007/s10617-008-9015-1`.

[Ros56] Murray Rosenblatt. "Remarks on Some Nonparametric Estimates of a Density Function". In: *Ann. Math. Statist.* 27.3 (Sept. 1956), pp. 832–837. DOI: `10.1214/aoms/1177728190`. URL: `https://doi.org/10.1214/aoms/1177728190`.

[SFS17] Christof Schlaak, Maher Fakih, and Ralf Stemmer. "Power and Execution Time Measurement Methodology for SDF Applications on FPGA-based MPSoCs". In: *arXiv preprint arXiv:1701.03709* (2017).

[Ske+15] Mladen Skelin et al. "Model checking of finite-state machine-based scenario-aware dataflow using timed automata". In: *Industrial Embedded Systems (SIES), 2015 10th IEEE International Symposium on*. IEEE, 2015, pp. 1–10.

[Sob14] Irwin Sobel. "An Isotropic 3x3 Image Gradient Operator". In: *Presentation at Stanford A.I. Project 1968* (Feb. 2014).

[Spa] SpaceCoDesign. www.spacecodesign.com.

[Ste+17] Ralf Stemmer et al. "Towards State-Based RT Analysis of FSM-SADFGs on MPSoCs with Shared Memory Communication". In: *Proceedings of the 9th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. ACM. Jan. 2017, p. 6.

[Ste+19a] Ralf Stemmer et al. "Experimental Evaluation of Probabilistic Execution-Time Modeling and Analysis Methods for SDF Applications on MPSoCs". In: *2019 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Springer. July 2019, pp. 241–254.

[Ste+19b] Ralf Stemmer et al. *Feasibility Study of Probabilistic Timing Analysis Methods for SDF Applications on Multi-Core Processors*. Research Report. IETR ; OFFIS, Mar. 2019. URL: `https://hal.archives-ouvertes.fr/hal-02071362`.

[Ste+19c] Ralf Stemmer et al. "Probabilistic State-Based RT-Analysis of SDFGs on MPSoCs with Shared Memory Communication". In: *2019 Design,* *Automation & Test in Europe Conference & Exhibition (DATE)*. Mar. 2019.

[Ste+20] Ralf Stemmer et al. "Towards probabilistic timing analysis for SDFGs on tile based heterogeneous MPSoCs". In: *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*. 2020, paper–59.

[Ste+21] Ralf Stemmer et al. "A Measurement-Based Message-Level Timing Prediction Approach for Data-Dependent SDFGs on Tile-Based Heterogeneous MPSoCs". In: *Applied Sciences* 11.14 (2021), p. 6649.

[Stu+08] Sander Stuijk et al. "FSM-based SADF". In: *Technical report, Eindhoven University of Technology, Department of Electrical Engineering* (2008).

[Stu07] Sander Stuijk. *Predictable mapping of streaming applications on multiprocessors*. 2007.

[Tra] TraceAnalyzer. http://www.symtavision.com/products/symtastraceanalyzer/.

[TS15] Rajesh Kumar Thakur and Y. N. Srikant. *Efficient Compilation of Stream Programs for Heterogeneous Architectures: A Model-Checking based approach*. Tech. rep. IISc-CSA-TR-2015-2. Indian Institute of Science, India, 2015. URL: http://www.csa.iisc.ernet.in/TR/2015/2/TechReport2015.pdf.

[TT93] International Telegraph and The Telephone Consultative Committee. "ITU-T Recommendation T. 81". In: (1993).

[Vir+20] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[VJ94] CCITT Study Group VIII and the Joint Photographic Experts Group. *Information technology – Digital compression and coding of continuous-tone still images: Requirements and guidelines*. Standard. International Organization for Standardization, Feb. 1994.

[Vu+21] Hai-Dang Vu et al. "A Fast Yet Accurate Message-level Communication Bus Model for Timing Prediction of SDFGs on MPSoC". In: *Asia and South Pacific Design Automation Conference_ASP-DAC 2021 (Virtual Conference)*. 2021, p. 1183.

[Vu21] Hai Dang Vu. "Fast and Accurate Performance Models for Probabilistic Timing Analysis of SDFGs on MPSoCs". PhD thesis. Universite de Nantes, 2021.

[Wil+10] Reinhard Wilhelm et al. "Static Timing Analysis for Hard Real-Time Systems". In: *Verification, Model Checking, and Abstract Interpretation: 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*. Ed. by Gilles Barthe and Manuel Hermenegildo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 3–22. ISBN: 978-3-642-11319-2. DOI: 10.1007/978-3-642-11319-2_3. URL: https://doi.org/10.1007/978-3-642-11319-2_3.

[Xil21] Xilinx. *MicroBlaze Processor Reference Guide*. UG984 (v2021.2). Infineon Technologies AG. Oct. 2021.

[Yan+10] Yang Yang et al. "Automated bottleneck-driven design-space exploration of media processing systems". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '10. Dresden, Germany: European Design and Automation Association, 2010, pp. 1041–1046.

[Zha11] W. Zhang. "Bounding Worst-Case Performance for Multi-Core Processors with Shared L2 Instruction Caches". In: *Journal of Computing Science and Engineering* 5.1 (2011), pp. 1–18.

[Zhu+15] Xue-Yang Zhu et al. "Static Optimal Scheduling for Synchronous Data Flow Graphs with Model Checking". In: *FM 2015: Formal Methods*. Springer, 2015, pp. 551–569. URL: `http://link.springer.com/chapter/10.1007/978-3-319-19249-9_34`.