A Measurement-based Message-level Timing Prediction Approach for Data-Dependent SDFGs on Tile-based Heterogeneous MPSoCs

Ralf Stemmer, Hai-Dang Vu, Sébastien Le Nours, Kim Grüttner, Sébastien Pillement, Wolfgang Nebel

applied sciences

MDPI

Article

A Measurement-Based Message-Level Timing Prediction Approach for Data-Dependent SDFGs on Tile-Based Heterogeneous MPSoCs

Ralf Stemmer [1,*], Hai-Dang Vu [2], Sébastien Le Nours [2], Kim Grüttner [1], Sébastien Pillement [2] and Wolfgang Nebel [3]

1 OFFIS e.V., 26121 Oldenburg, Germany; kim.gruettner@offis.de
2 Département Électronique et Technologies Numériques, University of Nantes, 44306 Nantes, France; hai-dang.vu@univ-nantes.fr (H.-D.V.); sebastien.le-nours@univ-nantes.fr (S.L.N.); sebastien.pillement@univ-nantes.fr (S.P.)
3 Department für Informatik, Faculty II, University of Oldenburg, 26111 Oldenburg, Germany; wolfgang.nebel@uol.de
* Correspondence: ralf.stemmer@offis.de

Abstract: Fast yet accurate performance and timing prediction of complex parallel data flow applications on multi-processor systems remains a very difficult discipline. The reason for it comes from the complexity of the data flow applications w.r.t. data dependent execution paths and the hardware platform with shared resources, like buses and memories. This combination may lead to complex timing interferences that are difficult to express in pure analytical or classical simulation-based approaches. In this work, we propose the combination of timing measurement and statistical simulation models for probabilistic timing and performance prediction of Synchronous Data Flow (SDF) applications on MPSoCs with shared memories. We exploit the separation of computation and communication in our SDF model of computation to set-up simulation-based performance prediction models following different abstraction approaches. We especially propose a message-level communication model driven by a data-dependent probabilistic execution phase timing model. We compare our work against measurement on two case-studies from the computer vision domain: a Sobel filter and a JPEG decoder. We show that the accuracy and execution time of our modeling and evaluation framework outperforms existing approaches and is suitable for a fast yet accurate design space exploration.

Keywords: system-level modeling; multi-processor; timing prediction

Citation: Stemmer, R.; Vu, H.-D.; Le Nours, S.; Grüttner, K.; Pillement, S.; Nebel, W. A Measurement-Based Message-Level Timing Prediction Approach for Data-Dependent SDFGs on Tile-Based Heterogeneous MPSoCs. *Appl. Sci.* **2021**, *11*, 6649. https://doi.org/10.3390/app11146649

Academic Editor: Luigi Pomante

Received: 31 May 2021
Accepted: 16 July 2021
Published: 20 July 2021

1. Introduction

Because of the growing demand in computational efficiency, more and more embedded systems are designed on multi-processor platforms. Validation of software running on such platforms is fundamental to guarantee functional correctness and that timing requirements (e.g., response time, latency) are fully met. Intensive analysis of hardware/software architectures under potential working scenarios is thus mandatory to prevent costly design cycles. However, in the context of software performance prediction on multi-processor platforms, creation of high abstraction level models with good prediction accuracy at affordable analysis times represents a key challenge.

System-level design (SLD) approaches [1] aim to facilitate the creation of high abstraction models of embedded systems for early performance prediction. Performance models are formed as a combination of a platform model with an application workload model which expresses the communication and computation loads. Performance models are annotated with low level timings that typically come from available documentations, static analysis of source codes, or low level simulation or execution. However, two main issues make creation of efficient performance models of multi-processor systems a time-consuming effort. The first is the possible variation of the execution behavior of the software

due to data-dependent execution paths. Each parallel software entity exhibits its own fine grained complexity where the software behavior and timing depend on the processed input data. Secondly, further execution time variability is observed when executing parallel software on a multi-processor platform due to complex interactions among shared resources (e.g., shared memories, shared communication buses).

In this article, we consolidate different high abstraction level modeling approaches for fast yet accurate timing prediction of data-dependent data flow applications on tile-based (each core has its independent data and instruction memory) multi-processor platforms with shared memory for inter tile communication. The presented approaches exploit the separation of computation and communication aspects in the data flow model.

From the computation perspective, we present two modeling approaches that differ according to the way data dependency is being considered. The first approach is a gray box model that considers knowledge of the executed algorithm to separate input data into different classes. We introduced this approach in [2]. For more complex application, a new black box approach is used. The execution delays get measured based on representative input data. On the resulting samples, Kernel Density Estimation (KDE) [3,4] is applied.

From the perspective of communication resources, we aim to reduce the simulation run-time of analyzing multiple FIFO communication channels mapped on a shared bus with a shared memory. We compare three communication models that use different levels of abstraction. The first model is a cycle accurate model we presented in [5]. This model considers each instruction and data communicated with shared resources in detail. In [6], we presented a more abstract model on message level. There we adopt an analytical model to formulate the time dependencies between the elementary communication phases taking into account potential penalty delays due to contention at shared resources. It is used in a message-level simulation model of the communication infrastructure whose combination demonstrates good scalability for performance prediction of different possible mappings compared to the cycle accurate and transaction-level model.

The contribution of this paper especially lies in the systematic evaluation of our proposed approaches through different use-cases. We systematically evaluate the benefits and limitations of our created models for timing prediction of two different data-dependent data flow applications mapped on a tile-based multi-processor platform. This setup is used for different case-studies and mapping configurations. For each studied mapping, we compare the obtained simulation results of our proposed message-level performance models against detailed transaction-level models and measurements on a real prototype. We also compare our different computation models. The evaluation metrics were accuracy of the estimations and analysis time. Moreover, scalability of the analysis approach was evaluated with respect to different use-cases and different mappings.

This leads to the following contributions:

1. A transaction level model proposed and evaluated as a compromise between the detailed cycle accurate model from from [5] and the abstract message level model from [6].
2. Systematic addressing data dependency by applying Kernel Density Estimation on measured characterization data.
3. An exhaustive evaluation combining our new advanced computation model and previous computation models from [2] with our new hybrid transaction level communication model and previous communication models from [5,6].

The paper is organized as follows. In the next section, we compare to existing modeling approaches for calibration of system-level models. In Section 3, we provide an overview of the main concepts associated with our approach. Section 4 presents our modeling workflow and the contributions to communication and computation modeling. In Section 5, we present our use-cases, the experimental setup, our comparison results and finally a discussion of these results. The paper closes with an outlook on future work and a summary.

2. Related Work

Different simulation-based approaches have been proposed to evaluate multi-processor system performance early in the design process. Examples of academic approaches are presented in [1]. Industrial frameworks such as Intel CoFluent Studio [7], Timing Architect [8], ChronSIM [9], and Space Codesign [10] have emerged also. In the proposed approaches, performance models are formed by considering an application workload model expressing computation and communication loads that an application causes when executed on platform resources. Captured performance models are then translated in a SLD language such as SystemC [11] to provide some estimations through simulation. The execution time of each load primitive is approximated as a delay, which is typically estimated from measurement on a real prototype or analysis of low level simulations. In [12], different methods are illustrated to calibrate performance models but with few consideration on data dependency and shared resources effects. We compare here our work to simulation-based timing prediction approaches that deal with these two aspects.

In [13], a performance models calibration approach is presented in the context of the Sesame simulation framework [14]. Delays of computation and communication statements are estimated using a measurement-based approach based on low-level simulation and synthesis. In our work, we extend a measurement-based approach with a message-level modeling for which communication delays are computed during system model simulation through an analytical model. Measurements are used to appropriately annotate this analytical model. From the computation perspective, we consider probabilistic methods to capture the influence of multiple execution paths.

Probabilistic models have been considered as a means of capturing system variability coming from system sensitivity to environment and low level effects of hardware platforms. In [15], a statistical inference process is proposed to capture low level platform effects on computation execution time. Based on multiple runs of the studied application on the targeted true platform, the Distribution Fitting method [16] 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 [17], 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 our work, we also combine simulation with analytical expression of shared resource effects on communication. This approach allows time-consuming simulation events to be significantly reduced while maintaining good level of accuracy. A similar combination is presented in [18] to address the timing accuracy problem in temporally decoupled transaction level models. 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. Our presented analytical model is sensitive to the simulated situation and thus our contention delay model can dynamically adapt to communication changes. A similar adaptation approach was considered in [19] but for a different bus protocol. In our case, we concentrate on FIFO channel communications mapped on a shared bus with first-come first served arbitration bus protocol.

Performance evaluation of data-dependent applications requires an accurate consideration of possible execution paths in the running software. In the data flow domain, we have also put in the focus of our work, Castrillon et al. propose a trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms [20], which has finally been integrated into the MAPS [21] framework. This framework performs a performance analysis on parallel scheduled KPN application based on recoded execution trace, which are fed back into a trace replay engine that is capable to analyze different possible trace interleaving based on known target platform shared resource characteristics and the chosen

scheduling policy. In [22], the previous work has been extended for Dynamic Data Flow (DDF) applications. In comparison to our approach, this trace based analysis relies on a fine grained instruction accurate trace replay engine, while our approach can be integrated in fast host based simulations.

Source-level simulation of software has been proposed to preserve simulation accuracy [23]. The source code of the software is annotated with timing extracted from machine code analysis. In our work, we use measured execution time and probabilistic inference methods to estimate execution time variability in data-dependent software. The Kernel density estimation (KDE) method is adopted to infer distribution laws based on a limited set of evaluated execution paths.

Our contribution lies thus in the definition of two complementary modeling methods to create fast yet accurate performance models of data-dependent SDF applications mapped on multi-processor platforms with shared bus and memory.

3. Workflow and Preliminaries

In this section, we provide a brief review of the main concepts, assumptions and constraints associated with our proposed approach.

Figure 1 illustrates our workflow, annotated with the section numbers where we describe the individual steps. All figures within this article use a consistent color scheme. All computation related parts in magenta. All hardware specific parts in purple. Everything related to observed execution is colored in green. Data dependency related parts in brown. The simulation parts are visualized in blue. The Figure 1 can be split into three columns, the system description, the characterization and modeling and the evaluation of our analysis. The system we describe consists of software described by a computation model (Magenta, Section 3.2) and hardware described by an architecture model (Purple, Section 3.1). For analysis, we characterize the hardware and software via measurements (Green, Section 3.3). This results in performance models for the computational (Blue, Section 4.1) and for the communicational (Blue, Section 4.2) aspect of our system. With these performance models, we then can simulate different system configurations with respect to its timing behavior (Blue, Section 4.3). In the evaluation Section 5, we compare the results of our analysis with the actual observed timing behavior of the simulated system setup.

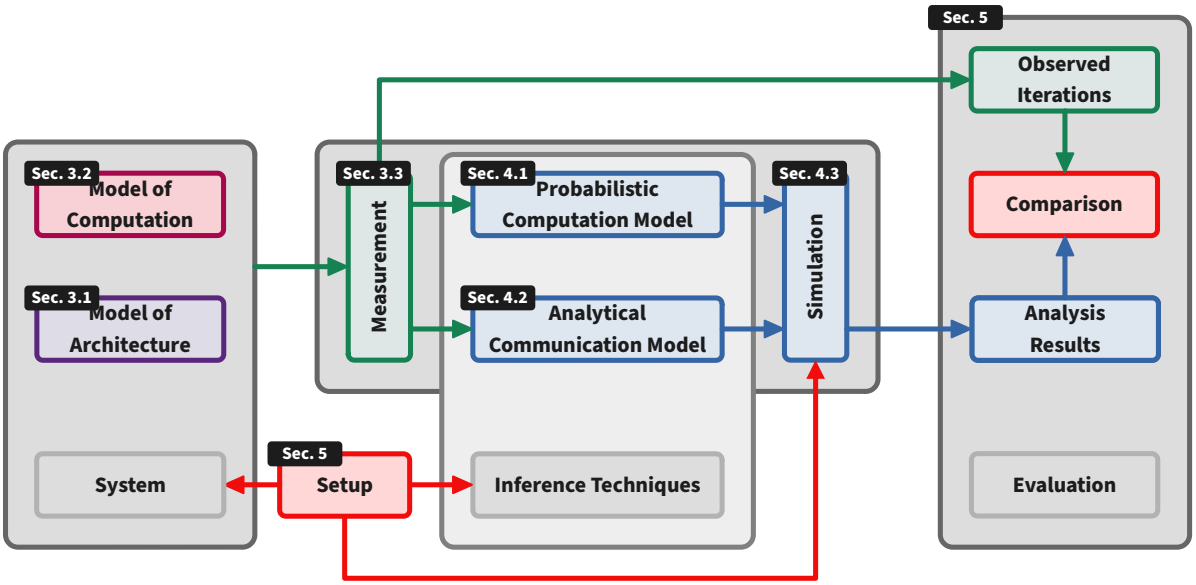


Figure 1. Workflow showing all modeling, characterization and evaluation steps.

3.1. Model of Architecture (MoA)

Our hardware model is shown in Figure 2a. The purple parts show the components of a tile-based platform described in this section. The green components of the platform show two measurement infrastructures used to characterize our system. They are described in Section 3.3.

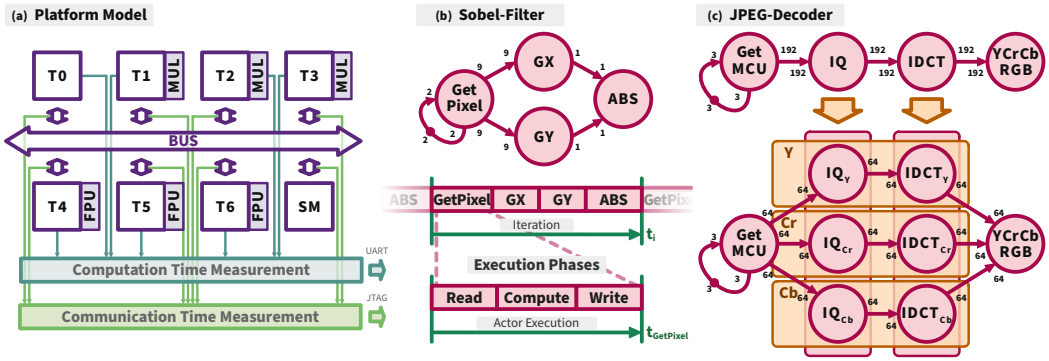


Figure 2. (**a**) Shows the architecture model of our evaluation platform (purple) and the measurement infrastructure (green) for characterization. (**b**) Shows details of the computation model on our Sobel-Filter Use-Case. (**c**) Shows our JPEG decoder Use-Case with and without considering data dependent behavior.

We adopt a tile-based platform organization where each tile consists of a processing element (PE) with private instruction and data memories, and a communication interface. A tile can thus execute software without interfering with other tiles as long as the software does not explicitly access shared resources. Processing elements of different tiles can come with different architectures. In our evaluation platform (Figure 2a), we have three different kinds of PEs. Tile *T0* comes with a minimal MicroBlaze processor without any hardware accelerators. *T1, T2, T3* are equipped with a hardware multiplier (MUL) that allows integer multiplication using a single instruction. Tiles *T4, T5, T6* provide a floating point unit (FPU) that extends the MicroBlaze architecture with floating point instructions. An execution platform consists of a finite set of tiles (Tx), a finite set of shared memories (SM) and a shared communication bus (BUS) that implements the ARM AXI4Lite standard. In the scope of our work, we consider a first-come-first-served (FCFS) bus arbitration protocol.

We assume that there are no caches and all components are connected to the same clock. Furthermore, all memories have a static timing behavior where each read and write access always takes the same amount of clock cycles.

3.2. Model of Computation (MoC)

We use synchronous data flow [24] (SDF) as model of computation. This model describes the application data flow between *Actors* via communication *Channels*. The SDF graphs of our use-cases are shown in Figure 2b,c. A complete execution of the graph is called *Iteration*. Figure 2b shows one possible valid execution of the Sobel-Filter example. The Sobel-Filter consists of four actors and five channels. The SDF model offers a strict separation of a computation phase (*Compute*) and communication phases (*Read* and *Write*).

During the *Compute* phase, only local memory of a tile is used. This guarantees that there will be no interference with other actors while an actor gets executed. During the *Read* and *Write* phase, data tokens get transferred between actors via channels.

Channels are FIFO buffers that can be mapped to any memory that is accessible by the producer and the consumer actor. For communication with actors mapped on a different tile, the channels need to be mapped on shared memory. During the *Read* phase of an actor, tokens get read from a channel buffer into a local memory buffer. During the *Write* phase, tokens get written into a channel buffer. The amount of tokens that a producer writes or a consumer reads is called *token rate*. It is denoted by integer values on the SDF graphs edges (See Figure 2b,c). As FIFO access is blocked, an actor can only switch to its

computation phase after it reads all tokens from all incoming channels. In our SDF model, we assume that all tokens have the same size of exact one data word of the interconnect. The buffer sizes needed for communication are determined before deploying the software on the platform [25]. These buffer sizes are guaranteed to never overflow [25]. Channels can be initialized with tokens before executing the application to avoid deadlocks.

In our Sobel-Filter example (Figure 2b), the first actor that needs to be executed is the *GetPixels* actor. This actor has to read two tokens from an incoming channel. The dot on the channel visualizes that there are two initial tokens available. Our JPEG-Decoder example (Figure 2c) can be modeled in two different ways. All color channels can be decoded sequentially as shown on the top. Because the three color channels (*Y*, *Cr*, *Cb*) can be decoded independently, it is possible to decode them in parallel as shown on the bottom. This also allows to characterize the decoding actors (*IQ* and *IDCT*) for each set of data (*Y*, *Cr*, *Cb*). The characteristics of the data of those three sets are systematically different due to the JPEG compression algorithm. This knowledge about the algorithm can be used to improve the accuracy of model.

In our work, all channels buffers are mapped onto shared memory, while the actors use the private memory of the tiles they get executed on. The execution order of the actors is predefined and static during run time [26]. This setup is fully composable such that the computation phases of any actor can be considered independent from communication phases.

3.3. Measurement Infrastructure

For our performance models (See Figure 1) we need to characterize the duration of the computation phase of each actor as well as the duration of the communication between actors. The measurement infrastructures used to characterize the computation and communication of our system are shown in Figure 2a (green). The *Computation Time Measurement* (dark green) part is used to characterize the computation model, the *Communication Time Measurement* (bright green) part is used to characterize the communication.

For the computation phase, we measure the amount of clock cycles a compute phase needs to be executed. We use the measurement infrastructure presented in details in [27]. It has been designed in a way that starting or stopping a measurement does not cause any interference. The measurement infrastructure is connected to the tiles using a separate peripheral bus so that there will be no interference with the communication on the system bus. From each tile, a common clock cycle counter can be started and stopped which allows characterizing the compute phase of each actor on each different tile architecture. It is also possible to measure the whole iteration of the SDF graph from the begin of the read phase of the first actor to the end of the write phase of the last actor.

For the communication phases, a measurement infrastructure is adopted to monitor and store the signals that are relevant to estimate the communication durations. It is first used to measure the durations of each elementary states (called the *elementary delays*) involved in communication. Second, it allows us to observe different contention situations at shared resources during the program execution. An analytical model is then created to compute the communication delays during these contentions. In the scope of this paper, we consider an IP, called System Integrated Logic Analyzer (SystemILA) provided by Xilinx to analyze a FCFS bus arbitration policy AXI4LITE. SystemILA connects to the AXI4LITE bus and observes the signals relating to the communication process. For the writing process, the write address valid signal (*AWVALID*) indicates that the channel is signaling valid write address and control information. The write response valid signal (*BVALID*) indicates that the channel is signaling a valid write response. For the reading process, we observed the signals *ARVALID* and *RVALID*. These signals are sent to the host computer via a JTAG cable. Further details about AXI4 protocol can be found in [28].

4. Computation and Communication Performance Modeling Approaches

This section describes the performance modes for the communication and the computation used in our simulation (See Figure 1). Section 4.1 describes the computation model. In Section 4.2, two communication models are described. The last section (Section 4.3) shows how these models are used inside our simulation.

4.1. Computation Modeling Approach

The computation model represents the temporal behavior of the compute phase of an SDF actor. We use the measurement infrastructure introduced in Section 3.3 to observe the execution time (delay) of an actor. The actor gets executed with representative stimuli data. The observed delays are the input to our modeling approach. Modeling the compute time of an actor comes with two challenges. First (Section 4.1.1), the inference process to build a delay model out of the observed execution times. Second (Section 4.1.2), the consideration of data-depended execution behavior which can lead to tremendous different delays.

4.1.1. Compute Time Representation

The most simple method we use to model the execution time of an actor is calculation the average of the observed delays.

Different compute delays can lead to different communication and bus contention behavior and so to different iteration delays of the whole application. Fitting a Gaussian distribution allows a more precise representation of the execution behavior. With this approach, the fact that an actor can have different execution times gets introduced into the model. The width of the distribution also pushes the execution time for a possible worst case higher, and for the best case lower than the worst/best observed execution time.

Generalizing the delays of software with a single Gaussian distribution comes with an huge error. To address the individual distributions of different actors, we apply Kernel Density Estimation (KDE). This is a method to estimate the probability density function based on an incomplete set of samples. Basically KDE assumes that each sample of our measured data represents the median of a well defined distribution function (in our case Gaussian distribution). The KDE processed data represents the sum of all individual distribution functions. This approach allows us to not only cover observed execution time during our simulation but also other delays that are likely to appear when using the actor with different stimuli. We apply a standard KDE with multiple Gaussian distributions as kernels. The bandwidth of the kernel is 1% of the worst observed execution time in the data set the KDE gets applied on. Using a large bandwidth makes it more likely to cover the best and worst execution time. For the KDE, we use *scikit-learn* [29].

4.1.2. Data Dependency Consideration

Considering data dependency can improve the accuracy of execution time prediction, because software often comes with different execution paths triggered by the input data processed by the software. Each path may have a different execution time.

In Figure 3 three abstraction levels of data dependency handling are shown. Beginning with a fine granular abstraction at the top of the figure, and ending with a sound model at the bottom. For each abstraction level, the data dependent execution time modeling process is shown for an example actor. In magenta, the actors executed on the target platform are shown. Brown color is used to highlight data dependent parts of the software. The execution time observed during the execution of the actor on the target platform is shown in green. The blue part shows how the data dependent execution time model is used inside the simulation.

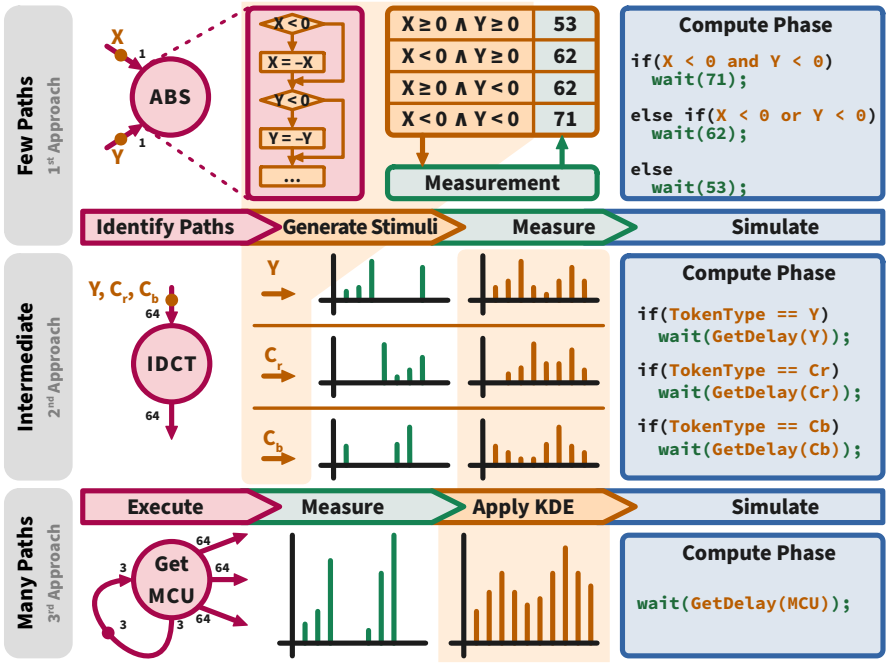


Figure 3. Modeling process with explicit execution path identification shown at Sobel-Filter example actor *GetPixel* (**top**), using Kernel Density Estimation shown at JPEG-Decoder example actor *GetMCU* (**bottom**) and a combination of both for actor *IDCT* (**middle**).

Figure 3 shows the process from left to right to create a data dependent execution time model for an actor. This is a manual, non-automated process that can be applied to any algorithm being executed on any hardware platform and improves our concept of observation based characterization of the computation time of an actor. It does not require static analysis tools that are restricted to specific platforms and does not come with restrictions to the algorithm like bounded loops. The more abstract the model becomes, the less knowledge about the algorithm and the hardware platform is required.

The *first approach* shown in Figure 3 (Few Paths) considers all possible individual execution paths of an actor. In the example, the source code of the *ABS* actor of the *Sobel-Filter* use-case gets analyzed. For each of the two input tokens, there is a branch with the sign of the input value as condition. This leads to four different execution paths, depending on the sign of the input tokens X and Y. With this knowledge, a set of stimuli-data can be generated and used to characterize the execution time of each path. Then, knowing the exact execution time for each path, these paths can be modeled in detail inside the simulation.

This first approach provides an accurate execution time model but comes with a lot of effort, especially for actors with many execution paths. It does not scale for real applications. In some cases, it is not even possible to do this on source level. For example simple multiplications can be replaced by loops during compilation, when the target platform does not provide a hardware multiplier.

The *second approach* shown in Figure 3 (Intermediate) considers different scenarios for the execution of an actor. The *IDCT* actor of the *JPEG-Decoder* use-case is used to demonstrate this approach. This actor implements an Inverse Discrete Cosine Transformation that needs to be applied to all three color channels (Y, C_r, C_b) of an JPEG image. The JPEG standard applies different compression strength to these three color channels. Therefore, the characteristics of the three color channels are very different, while the data of the color channels for each MCU (Minimum coded Unit—8×8 pixels) have similar characteristics. This knowledge about the algorithm is used to split the execution time model for the *IDCT* actor into three individual execution time models ($IDCT_Y$, $IDCT_{Cr}$, $IDCT_{Cb}$)—one for each scenario. The same strategy was applied to the *IQ* actor that performs an Inverse Quantization, a matrix multiplication that becomes data dependent on platforms without hardware

multiplier. The different amount of zeros inside the different quantization tables for each color channel has a huge impact on the execution times. This process also influences the way the JPEG decoder can be represented as SDF graph as shown in Figure 2c.

After identifying and characterizing the different scenarios, we use Kernel Density Estimation (KDE) to increase the coverage of possible execution paths. This approach allows us to not only cover observed execution time during our simulation but also other delays that are likely to appear when using the actor with different stimuli.

In the *third approach* shown in Figure 3 (Many Paths), the actor is seen as a black box. Its execution time gets measured, but not assigned to any input data and therefore execution paths. This makes characterization more easy because no insight into the source code or knowledge about the algorithm is needed. The downside of this approach is that there is also no knowledge about the coverage of execution paths inside the actor. The *GetMCU* actor used as example reads a Minimum Coded Unit of an JPEG encoded image. This included Run-Length and Huffman decoding which comes with a lot of possible execution paths. Instead of identifying and triggering each path with specific stimuli data, only a set of data that is representative for later use cases is used as input for an Actor. The Actor execution time gets then measured. The resulting observers execution times may only represent a subset of all possible execution paths. To address this issue we apply Kernel Density Estimation (KDE) to the measured delays, as we do in the second approach.

4.2. Communication Modeling Approach

The SDF model strictly separates communicational and computational parts of an application (See Section 3.2). When implementing an application following SDF semantics, these separations vanish, 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 on private memory so that instruction execution does not interfere with data communication in the context of SDF application execution.

We follow two approaches on different abstraction levels to model the SDF communication from implementation perspective. We also introduce a new intermediate model to improve the simulation speed.

The *first approach* (Section 4.2.1), first presented in [5], comes with a low abstraction Cycle Accurate (CA) model 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 Figure 4.

In [6] this model has been called Transaction Level Model. To distinguish between this old model is called Cycle Accurate Model in this article. The new model which also focus on modeling the transaction between tiles is called Transaction Level Model. Despite the old model, the new one is no longer cycle accurate.

The *second approach* (Section 4.2.2), first presented in [6], is an abstract Message Level (ML) model focussing on communicating a whole set of Tokens at once. Individual communication and instruction execution steps are abstracted to simple messages. Therefore a worst case interconnect usage is pre-calculated based on active communicating processing elements. This model is visualized at the bottom of Figure 4.

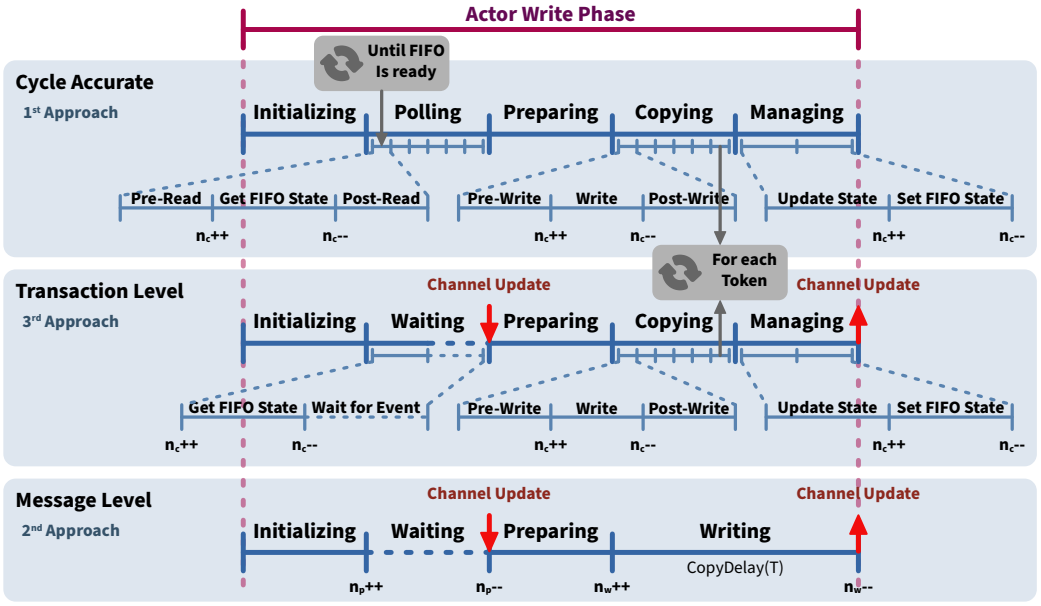


Figure 4. Simulation steps of the model of an actor write phase using two different communication models. cycle accurate [5] model at the top, message level [6] at the bottom and new transaction level model in the middle.

The *third and new approach* we show in this article is a hybrid model based on the other two models. The cycle accurate model requires a lot of simulation time compared to the message level model, as a comparison of simulation time in [6] showed. For simulation time optimization we introduce an Transaction Level (TL) model (Section 4.2.3) where polling on a channel FIFO buffer is realized using events instead of modeling the polling process cycle accurate. This model is visualized in the middle of Figure 4. The TL model is an enhancement of the CA model with the knowledge gained by the ML model.

All models consider a First Come First Server communication policy for communicating tokens between actors using a shared memory. The different variables n in Figure 4 denote the amount of tiles that are accessing a shared resource simultaneously. For the CA and TL models, all communicating tiles n_c are considered equally. The 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 models consider each token individually, the ML models' formulas for calculating certain time spans consider all consumed or produced tokens at once, denoted by T.

4.2.1. Cycle Accurate Model

This section summarizes our work published in [5]. We split the SDF communication phases *Read* and *Write* into sub-phases. These sub-phases are shown in Figure 4 (Top: Cycle Accurate).

Each sub-phase can consist of computational parts where instructions are executed and communication parts where data is transferred from or to a shared memory via a shared interconnect. Further, more sub-phases may be executed in loops. In this case, the loop-instructions are part of the computational overhead of the sub-phase.

There are five sub-phases for reading or writing tokens: Initialization, Polling or Waiting, Preparation, Copying and Management. Both reading and writing have the same structure and may only vary in the amount of clock cycles for computation, interconnect accesses and loop cycles.

The *Initialization-Phase* represents everything necessary to prepare for initiating the overall communication process with a FIFO buffer on the shared memory. This also includes programming language specific setup of the context of a function but also algorithmic features like initializing temporary variables used inside the function.

The *Polling-Phase* models the process of polling for a valid FIFO state. For reading, the FIFO must contain enough tokens to read, for writing, the FIFO must contain enough space for tokens to write. In [2,5] we modeled the polling process in detail considering the computational overhead for the polling loop and each interconnect access for checking the FIFO state.

The *Preparation-Phase* represents setting up the copy-loop to transfer the tokens between local memory of an actor and shared memory the channels are mapped to.

The *Copy-Phase* models the token transfer from a producer actor into the FIFO buffer of a channel or from the FIFO buffer of a channel into the local memory of a consuming actor. This is usually a loop that gets executed as many as tokens need to be transferred and is string related to the consume or produce rate of an actor for a specific channel. Each token is handled individually considering the computation overhead and the communication part of the transfer.

The *Management-Phase* updates the meta data of the communication channel like the fill-state of the FIFO buffer. This phase also includes the instruction to return from the communication function.

While the executed instructions on the processing element can be simply represented by waiting the corresponding amount of cycles, interconnect accesses need to be modeled in more details. The 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 Figure 4 n_c).

This model is cycle accurate from perspective of executed instructions on the connected processing element. It is not signal accurate from perspective of the interconnect. The detailed arbitration process (in our model, a First-Come-First-Serve (FCFS) arbitration) is hidden behind a look up table that defines observed penalty delays depending on the amount of contender n_c.

4.2.2. Message Level Model

This section summarizes our work published in [6]. Compared to the previously presented cycle accurate modeling approach, we aim here to reduce the simulation runtime of FIFO channel communication through a shared bus with a shared memory. To achieve still good level of accuracy, we adopt a hybrid approach that mixes simulation and analytical models. The analytical model is used to formulate the time dependencies between the elementary communication phases taking into account potential penalty delays due to contention at shared resources. It is used in a Message Level simulation model of the communication infrastructure whose combination demonstrates good scalability for performance prediction of different possible mappings. In the bottom part of Figure 4, we illustrate four sub-phases for writing tokens: Initialization, Waiting, Preparation and Writing. The same structure can be considered for reading tokens.

The *Initialization-Phase* and *Preparation-Phase* are similarly defined as in the Cycle Accurate model.

The *Waiting-Phase* is modeled by using synchronization events between reading and writing tokens on a same FIFO buffer. These events are shown in Figure 4 by the red arrows. Once the reading tokens finishes accessing the buffer, it sends an event to trigger the writing tokens to start writing data. This synchronization method reduces the number of polling states considered by the simulation kernel.

The *Writing-Phase* models the transfer of multiple tokens to the written buffer. The writing duration is computed using the analytical model built from the knowledge of communication phases and the bus arbitration policy. This analytical model is detailed in [6]. 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 [30]. The adoption of the TPN formalism allows to describe different communication situations with different numbers of tiles and simultaneous reading and writing tokens

processes. It is thus possible to systematize the obtaining of the equations that give the instants when shared resources are accessed. As it makes possible to express synchronization and mutual exclusion, it can be adopted for different bus protocols and arbitration policies. We adopted it here in the case of the FCFS bus arbitration policy.

The global variables n_p, n_w, n_r are used to update the communication situation by counting the number of on-going phases *Waiting-Phase*, *Writing-Phase* and *Reading-Phase*. At the beginning of each phase, the variables are incremented, and they are decremented at the end of each phase. Depending on the identified variables, the writing duration is computed following the expressions given in Figure 4 during simulation. The expressions take into account potential penalty delays caused by interferences between different communications denoted as $DelayOffset$ in Figure 4. Once the writing duration is computed, the processing element simply waits the corresponding amount of cycles. It finally sends an event to the reading tokens notifying that the writing process is finished.

4.2.3. Transaction Level Model

The Transaction Level model shown in the middle of Figure 4 is an improved Cycle Accurate model that uses events similar to the Message Level model to reduce the number of polling states and improve the simulation speed.

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. This causes lots of avoidable simulation steps decreasing the simulation speed tremendously. Therefore, not only in the Message Level but also in the Transaction Level model, this polling is implemented using events. When the current FIFO state is not as needed, the simulated actor waits for a channel exclusive event that the state of the FIFO buffer changed. This state update event is triggered after another actor accessed the FIFOs tokens, so the simulation kernel can easily skip the polling activity without simulating each individual polling attempt.

Similar to the Cycle Accurate model, this model also considers a fixed delay offset for the shared memory access and an additional contention penalty depending on how many processing elements try to access the shared interconnection (See Figure 4 n_c). In contrast to the Cycle Accurate model, the Transaction Level model assumes continuous bus access by the polling actors and ignores any computational overhead of executing the polling loop on the processing element the actor is executed on.

4.3. Simulation Model

In our simulation, the execution of an SDF application for a pre-configured mapping and scheduling on a pre-defined hardware platform is simulated. The amount of consecutive iterations of the SDF application can be configured as well. To represent the timing behavior of the simulated application, the computation delay is defined by the compute model (Section 4.1). The communication delay is defined by the communication model (Section 4.2). For each communication attempt and each execution of an actor, the delays get calculated. The overall concept is visualized in Figure 5. The simulation has been implemented in SystemC [11].

Figure 5 shows an abstract architecture of our simulation. On the top, the SystemC Module structure is shown. There exists a SystemC Thread (`SC_THREAD`) for each tile that shall be simulated. These tiles are communicating over the interconnect module with the shared memory module for data exchange as well as maintaining contention information required for the communication models.

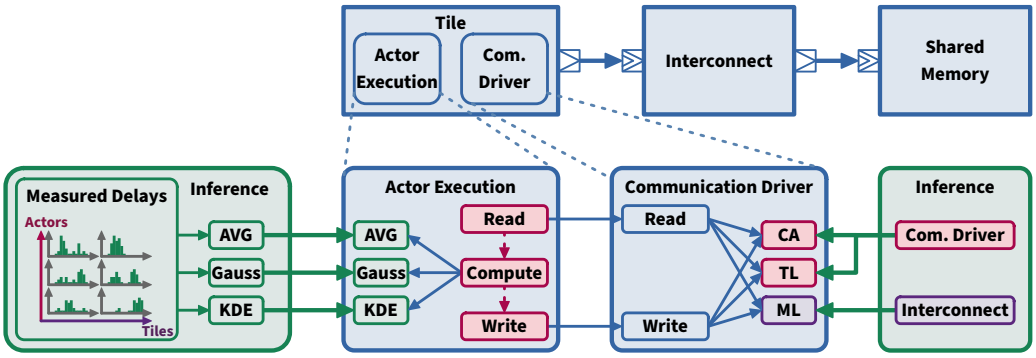


Figure 5. Architecture of our SystemC simulation showing how our models are integrated.

The delay models themselves (including hardware delay) are part of the SDF execution on a tile within the simulation. The model of executing code on a tile is split into two parts. The Actor Execution corresponds to the actual execution of an SDF actor on a tile. The Communication Driver represents the communication via an interconnect. In this communication driver representation, resource contention is included.

When simulating an actor, the three phases (read, compute, write) get simulated individually. For the read and write phase, the communication delay model gets executed. Depending on the configuration of the simulation, the Cycle Accurate model [5] (Section 4.2.1), the Message Level model [6] (Section 4.2.3) or the Transaction Level model (Section 4.2.2) is used. For the computation phase, the computation model is executed. The computation phase is represented by a single delay drawn from a set of possible delays, following a specific distribution that has been defined before the simulation. The distribution of delays can be configured and will be generated before the simulation starts. In our experiments, we compare three different distributions:

- A single average delay. (First presented in [2])
- A Gaussian distribution fitted to the observed executions times of an actor. (First presented in [5])
- A new, more sophisticated fitting approach using kernel density estimation.

5. Experiments

This section describes the setup of our experiments in Section 5.1 and the Use-Cases for the experiments in Section 5.2 The results are shown in Section 5.3 and discussed in Section 5.4. The experiments show advantages and disadvantages of the new communication and computation models compared the ones presented in [2,6].

5.1. Experiment Setup

We used the Xilinx ZC702 Evaluation Board as platform for our evaluation. On the Zynq, FPGA we instantiated a hardware platform as described in Section 3.1. In Figure 2a, we present our heterogeneous multiprocessor system that was used for all the experiments. On the upper part of the figure, the platform in purple contains 7 tiles that are connected to a shared memory via a shared interconnect. Three PEs are equipped with a hardware multiplier, and three with a floating point unit. The individual tiles use a MicroBlaze soft core that was equipped with private instruction and data memories. The measurement infrastructure is presented in green on the lower part of Figure 2a. This platform is used to characterize software following the computation model described in Section 3.2. We then simulate and analyze different mappings and compare the results with the observed behavior on the real platform.

In Section 3, we argued that communication and computation do not interfere and they can be considered as independent.

We characterized the compute phase of each actor using the measurement infrastructure described in Section 3.3. For the characterization, we measured the execution time of

the compute phase of each actor 1,000,000 times with different characteristic input data. These measurements have been done once each different kind of processing element.

The elementary delays of the communication phases are characterized using the Xilinx System Integrated Logic Analyzer (SystemILA). We simply built a simple workload model containing `WriteTokens` and `ReadTokens` functions of the communication phase. We observed specific signals of the communication process to measure the elementary delays. For each function (`WriteTokens`, `ReadTokens`), we measured the elementary delays of 20 different iterations. The measured elementary delays are constant in all iterations. Since the characterization is independent from application, we can use the elementary delays to build communication model of different mappings and applications. The communication characterization phase is not included in the simulation overhead of our proposed approach.

For the analysis results, we simulated 1,000,000 iterations. We considered parallel simulation by splitting our simulations into 16 processes. Each process simulated 62,500 iterations.

We provide a git repository (*Git repository:* https://zenodo.org/record/4876805, accessed on 31 May 2021) for all characterization data and results as well as the source code of the simulation and evaluation tools.

5.2. Use-Cases

In this section, two different use-cases and various possible mappings were considered to evaluate the efficiency of the created models in terms of accuracy and analysis time. For every considered use-cases and mappings we predicted the average iteration delay of the application, as well as the possible distribution of execution times. We executed all these experiments on a real platform and measured the actual execution times of each iteration. The obtained predictions were compared between simulation and measured results.

We compare the simulation results from the Cycle Accurate communication model first introduced in [5], the Message Level model proposed in [6] and the Transaction Level model. In Figure 2b,c, we present the applications to validate our proposed approach. In Figure 2a our heterogeneous multiprocessor system that was used for all the experiments is shown. In our experiments, we considered two SDF applications to validate our proposed approach:

The *Sobel-Filter* (see Figure 2b) which comes with simple compute phases and relatively high communication delays, is used to stress the communication models. Due to the low computational part of this application, errors in the communication model become more visible. In this use-case, the communication part takes most of the execution time. When we executed the application on tiles with an hardware multiplier, the computation part is highly predictable. In contrast to the JPEG-Decoder, the execution paths of the Sobel-Filter can be identified manually For the *GetPixel* actor, there were 9 execution paths depending on the position of tokens (3×3 pixels matrix) in the image. The *ABS* actor has 4 execution paths depending on the input tokens. There is only one execution path for *GX* and *GY* actors. We used a noise image as input of these Sobel-experiments. The analysis results of this application show the validation of our proposed communication model. We used the noise image as input of the applications. The image has a size of 48×48 Pixels

The *JPEG-Decoder* (see Figure 2c) with a wide range of possible execution times an relatively short communication delays is used to stress the computation models because errors in the communication model have a low impact on the simulation results. This use-case presents a huge computation part with an unknown and unmanageable amount of execution paths in most of the actors except for the IQ-actors that contain only one execution path when using hardware multiplier.

For each use-case different mappings were applied as shown in Table 1. The experiment name consists of the Use-Case application, the used communication model, and the amount of processing elements used for the mapping. We simulated these mappings with different communication and computation models as shown in Figure 6. We simu-

lated 3 mappings for Sobel Filter: *Sobel-1*, *Sobel-2* and *Sobel-4*. For JPEG Decoder, we also considered 3 mappings: *JPEG-1*, *JPEG-3* and *JPEG-7*. For each Use-Case and mapping, the different communication models *CA* (Cycle Accurate), *TL* (Transaction Level) and *ML* (Message Level) are used, as well as the different computation models *Avg* (Average), *Gauss* (Gaussian Distribution) and *KDE* (Kernel Density Estimation).

		Average Execution Time			Gaussian Distribution			Kernel Density Estimation		
		Cycle Accurate	Transaction Level	Message Level	Cycle Accurate	Transaction Level	Message Level	Cycle Accurate	Transaction Level	Message Level
Sobel	1 Tile	Sobel-CA1-AVG	Sobel-TL1-AVG	Sobel-ML1-AVG	Sobel-CA1-Gauss	Sobel-TL1-Gauss	Sobel-ML1-Gauss	Sobel-CA1-KDE	Sobel-TL1-KDE	Sobel-ML1-KDE
	2 Tiles	Sobel-CA2-AVG	Sobel-TL2-AVG	Sobel-ML2-AVG	Sobel-CA2-Gauss	Sobel-TL2-Gauss	Sobel-ML2-Gauss	Sobel-CA2-KDE	Sobel-TL2-KDE	Sobel-ML2-KDE
	4 Tiles	Sobel-CA4-AVG	Sobel-TL4-AVG	Sobel-ML4-AVG	Sobel-CA4-Gauss	Sobel-TL4-Gauss	Sobel-ML4-Gauss	Sobel-CA4-KDE	Sobel-TL4-KDE	Sobel-ML4-KDE
JPEG	1 Tile	JPEG-CA1-AVG	JPEG-TL1-AVG	JPEG-ML1-AVG	JPEG-CA1-Gauss	JPEG-TL1-Gauss	JPEG-ML1-Gauss	JPEG-CA1-KDE	JPEG-TL1-KDE	JPEG-ML1-KDE
	3 Tiles	JPEG-CA3-AVG	JPEG-TL3-AVG	JPEG-ML3-AVG	JPEG-CA3-Gauss	JPEG-TL3-Gauss	JPEG-ML3-Gauss	JPEG-CA3-KDE	JPEG-TL3-KDE	JPEG-ML3-KDE
	7 Tiles	JPEG-CA7-AVG	JPEG-TL7-AVG	JPEG-ML7-AVG	JPEG-CA7-Gauss	JPEG-TL7-Gauss	JPEG-ML7-Gauss	JPEG-CA7-KDE	JPEG-TL7-KDE	JPEG-ML7-KDE

Figure 6. Overview of all experiments done using different computation and communication performance models. Experiments that challenge our approach are highlighted in blue and will be discussed in detail.

For all the experiments the instruction and local data of an actor were mapped on the private memory of a tile. The applications are executed without any operating system. Interrupts are disabled so that it is guaranteed that there is no preemption. All buffers are static allocated. There is no dynamic memory management. The scheduling of the actors is static as well.

Table 1. Mapping of the Sobel and JPEG experiments on the tiles of the hardware platform. Cp. [2].

Experiment→ Actor ↓	***Jpeg1***	***Jpeg3***	***Jpeg7***	**Exp. → Actor ↓**	***Sobel1***	***Sobel2***	***Sobel4***
Get MCU	0	0	0	***GetPixel***	0	1	1
IQ_Y	0	1	1	***GX***	0	2	2
IQ_{Cr}	0	1	2	***GY***	0	1	3
IQ_{Cb}	0	1	3	***ABS***	0	2	0
$IDCT_Y$	0	4	4				
$IDCT_{Cr}$	0	4	5				
$IDCT_{Cb}$	0	4	6				
YCrCb RGB	0	0	0				

5.3. Results

In this section, we present the results of the experiments considering different criteria: accuracy, analysis time, and scalability. First we present the impact of the different computation abstraction: Kernel Density Estimation (KDE), Gaussian Distribution (Gauss) and Average execution time (Avg). Then we aim to demonstrate the validation of our proposed communication models by comparing the average iteration delay of our Message Level (ML) model with the Transaction Level (TL) model, the Cycle Accurate (CL) model and the measured data.

In Table 2, we compare the analyzed results of the CA model, TL model and ML model with the measured data. In the first column (*Experiment*), the experiments are listed. The second column (*Measured*) presents the average execution time in clock cycles of 1,000,000 iterations. This is the observed measured execution time of the application on a real hardware platform. The next three columns *Average*, *Gaussian* and *KDE* show the results of the simulation using different inference techniques: The average computation time of an actor, a fitted Gaussian distribution of possible delays, and a fitted distribution using kernel density estimation. In parentheses, the error in percent (first number) compared to the measured delay is shown, as well as the Bhattacharyya distance [31] (second number) of the analyzed distribution of possible execution times compared to the distribution of measured execution times. The error represents the difference between the average execution time

for all iterations. The Bhattacharyya distance shows the similarity between the distribution of the execution times (lower value is better).

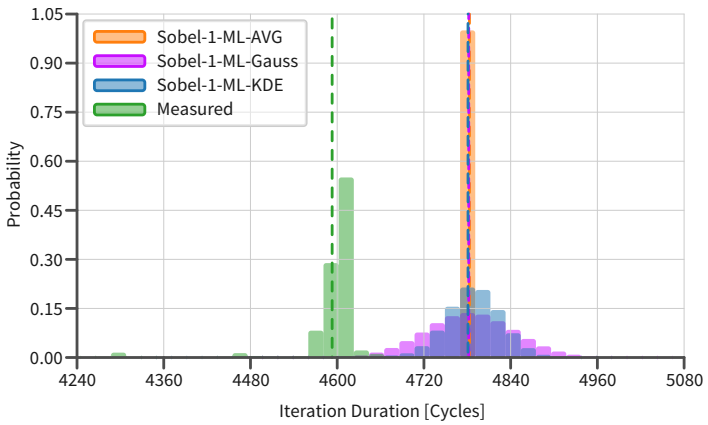
Table 2. Analysis results of the experiments listed in Figure 6. The different communication models (CA, TL, ML) are compared as well as the different computation models (Average, Gaussian Distribution and KDE). 1,000,000 iterations have been measured or simulated. The results are given in clock cycles, errors and Bhattacharyya distance to the observed timing behavior are in parentheses.

Experiment	Measured	Average		Gaussian		KDE	
Sobel-CA1	4593.26	4445.00	(−3.23%, 3.388)	4444.84	(−3.23%, 0.436)	4443.33	(−3.26%, 2.007)
Sobel-TL1	4593.26	4435.00	(−3.45%, 3.388)	4434.84	(−3.45%, 0.404)	4433.32	(−3.48%, 2.442)
Sobel-ML1	4593.26	4783.00	(4.13%, 3.388)	4782.84	(4.13%, 0.428)	4781.33	(4.09%, 2.291)
Sobel-CA2	2902.53	3100.00	(6.80%, 2.210)	3092.60	(6.55%, 0.825)	3091.98	(6.53%, 1.941)
Sobel-TL2	2902.53	2907.00	(0.14%, 3.134)	2906.79	(0.15%, 0.947)	2905.78	(0.11%, 1.607)
Sobel-ML2	2902.53	3039.00	(4.70%, 2.212)	3038.79	(4.69%, 0.802)	3037.78	(4.66%, 1.800)
Sobel-CA4	3097.43	4623.00	(49.25%, 2.665)	4636.61	(49.69%, 2.528)	4639.62	(49.79%, 4.891)
Sobel-TL4	3097.43	2939.99	(−5.08%, 2.564)	2939.79	(−5.09%, 1.487)	2938.78	(−5.12%, 1.330)
Sobel-ML4	3097.43	3105.00	(0.24%, 3.927)	3104.80	(0.24%, 0.502)	3103.78	(0.21%, 1.330)
JPEG-CA1	2,385,860.12	2,384,114.00	(−0.07%, 1.877)	2,384,156.45	(−0.07%, 0.354)	2,384,088.70	(−0.07%, 0.061)
JPEG-TL1	2,385,860.12	2,384,094.00	(−0.07%, 1.877)	2,384,136.45	(−0.07%, 0.354)	2,384,068.00	(−0.08%, 0.061)
JPEG-ML1	2,385,860.12	2,389,605.00	(0.16%, 1.877)	2,389,647.45	(0.16%, 0.445)	2,389,579.70	(0.16%, 0.059)
JPEG-CA3	940,836.44	955,621.41	(1.57%, 2.422)	955,688.20	(0.12%, 0.115)	955,623.81	(1.57%, 0.071)
JPEG-TL3	940,836.44	941,122.00	(0.03%, 2.422)	941,190.64	(0.04%, 0.185)	941,115.53	(0.03%, 0.161)
JPEG-ML3	940,836.44	941,004.00	(0.02%, 2.422)	941,068.01	(0.02%, 0.185)	940,992.80	(0.02%, 0.162)
JPEG-CA7	941,059.40	1,071,080.02	(13.82%, 2.420)	1,071,133.51	(13.82%, 0.185)	1,071,073.86	(13.82%, 0.198)
JPEG-TL7	941,059.40	927,239.00	(−1.47%, 2.422)	927,303.56	(−1.46%, 0.114)	927,235.61	(−1.47%, 0.075)
JPEG-ML7	941,059.40	941,170.91	(0.01%, 2.422)	941,234.74	(0.02%, 0.184)	941,160.56	(0.01%, 0.160)

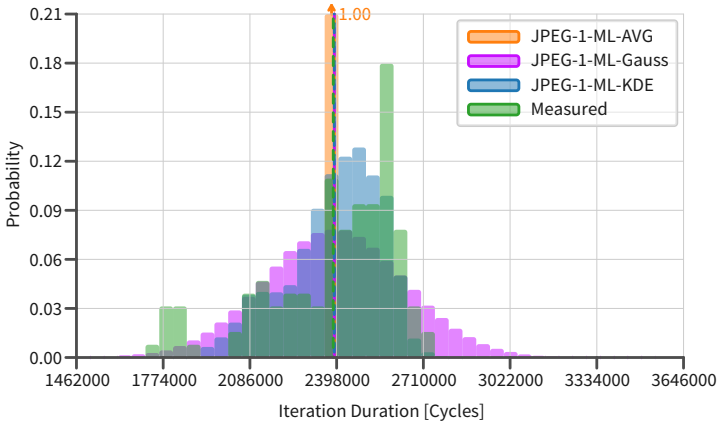
For all experiments, the Bhattacharyya distance between the simulated distribution of execution times and the observed distribution is within a range of 3.388 for *Sobel-ML/TL1-Avg* and 0.059 for *JPEG-ML1-KDE*. The JPEG use case with less communication relative to its computation times has a much lower Bhattacharyya distance compared to the Sobel experiments for all computation models. The less influences the communication has, the closer comes the analyzed distribution to the observed one. The communication heavy Sobel application analysis shows higher similarities in its distribution with the Gaussian distribution of computation delays. The simulated distribution of the computation heavy JPEG use-case is best with the kernel density estimation inference technique.

The experiments using our ML communication model show a high accuracy result that over-approximate the measured data. The error is within a range of 4.7% for *Sobel-ML2-Avg* and 0.01% for *JPEG-ML7-Avg/KDE*. The TL communication model leads to errors within the range of −5.12% for *Sobel-TL4-KDE* to 0.11% for *Sobel-TL2-KDE*. All JPEG decoder experiments using the TL communication model show very low errors with the highest value of −1.47% for *JPEG-TL7-Avg/KDE*. Compared to this, the Cycle Accurate (CA) model leads to relative high errors of up to 1.57% for *JPEG-CA3*. For the *Sobel-CA4* experiments the error is up to 49.69%.

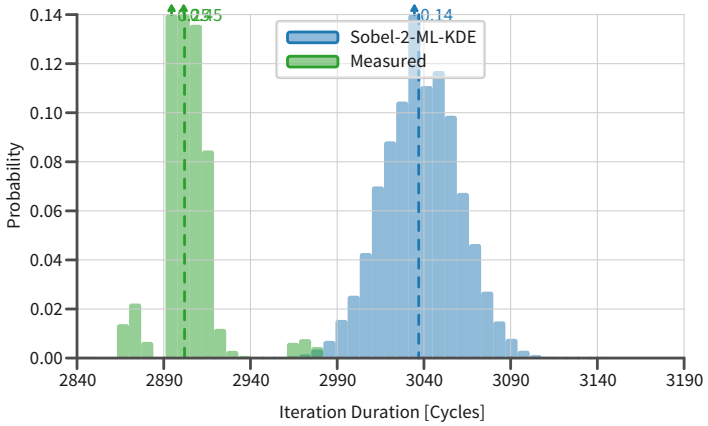
In Figure 7, we compare the iteration delay distribution of the measured data and the analyzed results of the experiments. The influence of different data on the applications execution time is shown for the experiments *Sobel-TL1* and *JPEG-TL1* in Figure 7a,b. It can be clearly seen that the consideration of different data for the analysis is valuable. The different shape of the distribution as well as the lower average iteration duration were correctly predicted by the analysis.



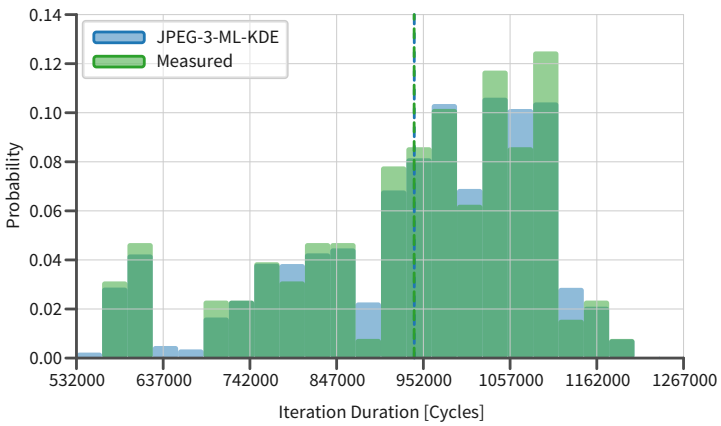
(**a**) Sobel on 1 Tile with ML communication model and different computation models compared to measured data



(**b**) JPEG Decoder on 1 Tile with ML communication model and different computation models compared to measured data



(**c**) Sobel on 2 Tiles (Realistic Mapping) using best communication and computation model.



(**d**) JPEG Decoder on 3 Tiles (Realistic Mapping) using best communication and computation model.

Figure 7. Comparison of the observed distribution of execution times with the simulated ones using different computation models.

In the lower part of Figure 7, we compare the distribution of the iteration delays of our ML model with data dependency (in blue) to the measured data (in green) of the Sobel and JPEG experiments. This comparison shows the efficiency of the computation time representation approaches (i.e., the data dependency, the KDE techniques) to the iteration delay. The distributions of simulated results show a similar shape between simulation and measured data. This means that the execution paths were well characterized and applied in our ML model.

In Table 3, we compare the duration to measure 1,000,000 iterations of an application running on the real hardware platform with simulating the same application with the same mapping using different communication models. The table only shows the simulation runs with using the KDE based computation model. Since the inference technique is done before the simulation starts it only has a small impact on the overall simulation time. The KDE approach is the most compute intensive one. The simulation time does not include the time it takes to characterize the actors and communication. The factors within the parentheses show the speed-up of the simulation compared to the measurement. The *Speed-up* column shows the speed-up of the ML communication model compared to the TL model.

Table 3. Simulation time (HH:MM:SS) with Kernel Density Estimation computation model for 1,000,000 iterations.

Experiment	Measured	CA Model		TL Model		ML Model		TL/ML Speed-up
Sobel-1-KDE	0:07:40	0:03:12	(2.40)	0:03:04	(2.50)	0:01:44	(4.42)	1.77
Sobel-2-KDE	0:07:03	0:12:32	(0.56)	0:05:05	(1.39)	0:02:11	(3.23)	2.33
Sobel-4-KDE	0:07:13	0:20:00	(0.36)	0:05:21	(1.35)	0:02:18	(3.14)	2.33
Jpeg-1-KDE	13:14:31	0:52:29	(15.14)	0:51:35	(15.40)	0:19:03	(41.71)	2.71
Jpeg-3-KDE	5:12:58	67:31:00	(0.08)	1:20:56	(3.87)	0:23:41	(13.22)	3.42
Jpeg-7-KDE	5:13:02	56:19:42	(0.10)	1:20:57	(3.87)	0:25:29	(12.28)	3.18

AMD Opteron™ Processor 6328 (3.5 GHz) at OFFIS e.V. www.offis.de, accessed on 24 May 2021. Simulation split into 16 processes, each on a dedicated processor.

For the communication heavy Sobel application, the amount of (Processing Elements) PEs does not have a huge impact to the overall execution time when using the TL or ML model. All three mappings result in an execution time of approximate 7 min for 1,000,000 iterations. The cycle accurate (CA) model is slower than measuring the mappings with multiple tiles due to the excessive simulation overhead during the polling phase of the communication.

The execution time of the JPEG application highly depends on the amount of PEs used to execute them. Measuring single PE mapping takes more than 13 h for 1,000,000 iterations. Executing some actors in parallel and making use of different architecture features like the FPU decreased the execution time to about 5 h. All MicroBlazes were operated with a clock speed of 100 MHz.

The simulation time increases with the amount of communication load. Simulating the execution of the JPEG decoder takes up to 1 h 20 min for a mapping using 7 tiles using the TL communication model. The same experiment using the more abstract ML model takes 25 min. The CA model requires multiple days for multi-processor mappings.

In our experiments, the simulation with the TL communication model is up to 15 times faster than the execution of the real application on the actual hardware platform. The simulation using the ML model is up to 41 times faster.

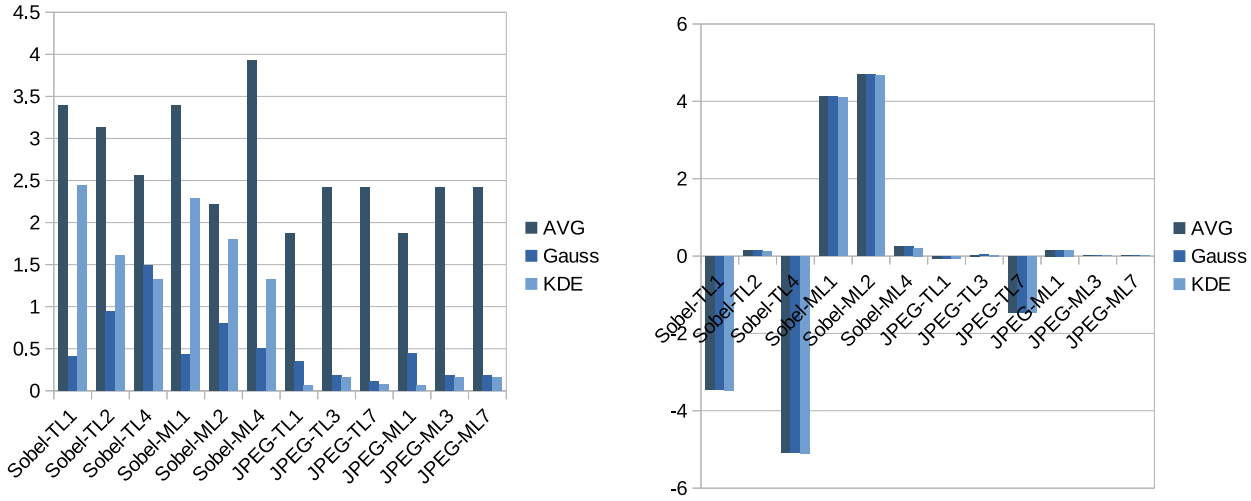
5.4. Discussion

The Kernel Density Estimation is the best computation modeling approach of the three inference techniques we applied. For the JPEG Use-Case, the resulting execution time distribution is closest to the observed one using KDE. In most cases, the overall error of the average execution time is also smallest with the exception of experiment *JPEG-CA3* where the Gaussian distribution leads to the lowest error. Table 2 shows that for the Sobel experiment, the Gaussian distribution is a bit better than the KDE approach when it comes

to similarity of the distribution function. Considering the overall error, the KDE approach is in many cases also slightly better for the Sobel use-case.

For the ML model applying the KDE technique for the Sobel filter experiments *Sobel-MLx-KDE*, the analyzed results present a similar error as in the ML model applying the Gaussian Distribution. The Bhattacharyya distance in this experiment is much higher than in the ML model applying KDE. This can be explained because in the case of the *GX* and *GY* actors, their measured computation time were constant. However, the KDE technique created unexpected computation time for these actors. This led to variation in the iteration delay of the ML model KDE for Sobel filter application which made the Bhattacharyya distance higher. Our ML model applying the KDE technique for the JPEG decoder experiments also shows a very good level of accuracy to the measured data.

Figure 8a compares all Bhattacharyya distances of the different experiments compared to their observed behavior, using different delay distributions for modeling the individual actors. For the more complex compute intensive JPEG decoder, applying a kernel density estimation leads to a much more accurate distribution of iteration delays compared to other inference techniques. For simple applications like the Sobel-Filter, a Gaussian distribution is more suitable. This is because its distribution of actor's computation delays is closer to the average delay, which is beneficial for actors with low variance in its execution time.



(**a**) Bhattacharyya distance of simulated distribution to observe distribution

(**b**) Error compared to measured delay

Figure 8. Comparison of the simulation results to the observed behavior of the Use-Cases on a real platform.

Figure 8b shows that there is no significant change in accuracy regarding analyzing the average execution time between different distribution functions. Since the different inference techniques have also no influence to the simulation time, they are only relevant when the possible distribution function of the iterations of the application shall be analyzed. For only estimating the average execution time, the selection of communication model is more relevant.

Table 2 shows that the communication model performs different depending on the amount of communicating tiles. For mappings with less communication, the Transaction Level (TL) model leads to the lowest error (Sobel, 2 Tiles and JPEG, 1 Tile). For the 1 Tile mapping of the Sobel Use-Case, the Cycle Accurate (CA) model is slightly better than the TL model. Simulated mappings with many tiles have a lower error when the Message Level (ML) model is used (Sobel, 4 Tiles and JPEG, 7 Tiles). Because the CA and TL models are very explicit, small errors can accumulate to huge errors when lots of communication takes place. The ML model models a communication phase in total, so that the error is independent from the amount of polling attempts and communicated tokens.

The impact of the communication model on the simulation results can be seen in Figure 8b. While using the transaction level model can lead to negative errors, the message level model always leads to positive errors. The ML model becomes more accurate for high communication load (many tokens to communicate, much inferences). Beside the higher accuracy, the ML model improves also the simulation speed.

The results in Table 3 show that simulation using the message level communication model is fastest. They also show that there is only an increase in simulation time between single and multi-processor systems. The increase in simulation time between a 2 Tile and a 4 Tile system for the Sobel experiment, or the 3 Tile and 7 Tile mapping for the JPEG experiment is minimal.

The fast yet accurate simulation allows using our approach for Design Space Exploration (DSE) over different architectures and mappings.

6. Future Work

For the consideration of more complex bus system with multiple layers and other scheduling policies, we have to extend our analytical bus model to a probabilistic bus model. In such a model, each master port or initiator has its individual probabilistic access delay time model, which depends on the bus configuration and the access scheduling for the remaining bus masters.

By moving from a hybrid (probabilistic and analytic) to a fully probabilistic simulation model the application of more advanced probabilistic analysis methods, such as Statistical Model Checking (SMC) [5,15] become feasible. 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 approaches reduces the required number of simulation runs by using the 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. Furthermore, such an approach could be adopted to evaluate different properties of the created models such as the probability to miss a deadline.

For more complex applications tool support is required to automatically identify all relevant actor internal execution paths, annotate them with execution times or execution time PDFs on multiple similar execution paths, and finally create a simulation model from it. This automation process must also exceed source-level analysis because different instruction sets may introduce further execution paths, e.g., by replacing multiplications by loops of additions in cases where the Arithmetic Unit does not directly support multiplications.

7. Conclusions

In this paper, we presented a composable performance and timing prediction approach for data-dependent SDF application mapped on a tile-based multi-processor platform with shared memory. We compared different techniques to model data dependency and communication via shared resources. The results are promising but have been conducted under several constraints that have to be removed or lifted in future work for demonstrating the applicability of our approach for industrial scale use-cases.

One identified constraint on our current approach is the manual selection of representative input data to capture data dependencies during the measurement phase. In future work, we aim at automatic execution path identification (though CDFG analysis) and the application of execution path coverage metrics for controlling the accuracy of our measurement and timing characterization phase.

To demonstrate the feasibility of our proposed approach and for evaluation purposes, we have started with a simple, well controllable and observable hardware platform. For this reason, we have decided to use an FPGA with 3rd party IP components. Due to space restrictions in the currently chosen Zynq platform (FPGA), we cannot instantiate more tiles (MicroBlazes with BRAM). This is the main constraint for our applications and explains why we are currently not able to execute larger and more complex software. In

future work, we are planning to activate the MicroBlaze cache and allow applications to be loaded from the Zynq's processing system main memory (DDR). Furthermore, we have already analyzed different COTS platforms like AURIX™ that we want to use in the future. Regarding on-chip communication, we aim at considering more complex bus communication schemes, including DMA transfers and split-burst transactions supported by state-of-the-art COTS MPSoC platforms to optimize overall system performance. Even though we implemented a custom timing measurement unit inside our FPGA platform, it is possible to apply our approach to COTS platforms that provide independent GPIO pins to trigger our measurement infrastructure. This could be modified and used as a stand-alone system or realized as a software implementation on a dedicated MCU.

With these planned extension towards automatic identification of execution paths, the application of a path coverage based metric for automatic creation of simulation models and the shift to COTS MPSoC platforms, the scalability of our approach for more complex applications could be demonstrated.

Author Contributions: Conceptualization, R.S. and H.-D.V.; methodology, R.S.; software, R.S. and H.-D.V.; validation, R.S. and H.-D.V.; formal analysis, R.S.; investigation, R.S. and H.-D.V.; resources, S.L.N., K.G., S.P. and W.N.; data curation, R.S.; writing—original draft preparation, R.S., H.-D.V. and S.L.N.; writing—review and editing, R.S., H.-D.V., S.L.N., S.P. and K.G.; visualization, R.S.; supervision, S.L.N., K.G., S.P. and W.N.; project administration, S.L.N. and K.G.; funding acquisition, S.L.N. and K.G. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been partially sponsored by the DAAD (PETA-MC project under grant agreement 57445418) with funds from the German Federal Ministry of Education and Research (BMBF). This work has also been partially sponsored by Campus France (PETA-MC project under grant agreement 42521PK) with funds from the French ministry of Europe and Foreign Affairs (MEAE) and by the French ministry for Higher Education, Research and Innovation (MESRI).

Data Availability Statement: All characterization data and results as well as the source code of the simulation and evaluation tools are available at https://zenodo.org/record/4876805, accessed on 31 May 2021.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gerstlauer, A.; Haubelt, C.; Pimentel, A.D.; Stefanov, T.P.; Gajski, D.D.; Teich, J. Electronic System-Level Synthesis Methodologies. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2009**, *28*, 1517–1530. [CrossRef]
2. Stemmer, R.; Vu, H.D.; Grüttner, K.; Le Nours, S.; Nebel, W.; Pillement, S. Towards Probabilistic Timing Analysis for SDFGs on Tile Based Heterogeneous MPSoCs. In Proceedings of the 10th European Congress on Embedded Real Time Software and Systems, Toulouse, France, 29–31 January 2020; p. 59.
3. Rosenblatt, M. Remarks on Some Nonparametric Estimates of a Density Function. *Ann. Math. Statist.* **1956**, *27*, 832–837. [CrossRef]
4. Parzen, E. On Estimation of a Probability Density Function and Mode. *Ann. Math. Statist.* **1962**, *33*, 1065–1076. [CrossRef]
5. Stemmer, R.; Vu, H.D.; Grüttner, K.; Le Nours, S.; Nebel, W.; Pillement, S. Experimental Evaluation of Probabilistic Execution-Time Modeling and Analysis Methods for SDF Applications on MPSoCs. In Proceedings of the 2019 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), Samos, Greece, 7–11 July 2019; pp. 241–254.
6. Vu, H.D.; Le Nours, S.; Pillement, S.; Stemmer, R.; Grüettner, K. A Fast Yet Accurate Message-level Communication Bus Model for Timing Prediction of SDFGs on MPSoC. In Proceedings of the Asia and South Pacific Design Automation Conference, Online, 18–21 January 2021; p. 1183.
7. Intel. Intel CoFluent Studio. Available online: https://www.intel.com/content/www/us/en/cofluent/cofluent-studio.html (accessed on 20 July 2021).
8. Timing-Architect. Available online: http://www.timing-architects.com (accessed on 20 July 2021).
9. ChronSIM. Available online: http://www.inchron.com/tool-suite/chronsim.html (accessed on 20 July 2021).
10. SpaceCoDesign. Available online: www.spacecodesign.com (accessed on 20 July 2021).
11. IEEE Standards Association. *IEEE Standard for Standard SystemC Language Reference Manual*; IEEE Computer Society: Washington, DC, USA, 2012.
12. Kreku, J.; Hoppari, M.; Kestilä, T.; Qu, Y.; Soininen, J.; Andersson, P.; Tiensyrjä, K. Combining UML2 Application and SystemC Platform Modelling for Performance Evaluation of Real-Time Embedded Systems. *EURASIP J. Embed. Syst.* **2008**. [CrossRef]

13. Pimentel, A.D.; Thompson, M.; Polstra, S.; Erbas, C. Calibration of Abstract Performance Models for System-Level Design Space Exploration. *J. Signal Process. Syst.* **2008**, *50*, 99–114. [CrossRef]
14. Pimentel, A.D.; Erbas, C.; Polstra, S. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput.* **2006**, *55*, 99–112. [CrossRef]
15. Nouri, A.; Bozga, M.; Moinos, A.; Legay, A.; Bensalem, S. Building faithful high-level models and performance evaluation of manycore embedded systems. In Proceedings of the ACM/IEEE International Conference on Formal Methods and Models for Codesign, Lausanne, Switzerland, 19–21 October 2014.
16. Le Boudec, J.Y. *Performance Evaluation of Computer and Communication Systems*; EPFL Press: Lausanne, Switzerland, 2010.
17. Bobrek, A.; Paul, J.M.; Thomas, D.E. Stochastic Contention Level Simulation for Single-Chip Heterogeneous Multiprocessors. *IEEE Trans. Comput.* **2010**, *59*, 1402–1418. [CrossRef]
18. Lu, K.; Müller-Gritschneder, D.; Schlichtmann, U. Analytical timing estimation for temporally decoupled TLMs considering resource conflicts. In Proceedings of theDesign, Automation Test in Europe Conference Exhibition (DATE), Grenoble, France, 18–22 March 2013; pp. 1161–1166.
19. Chen, S.; Chen, C.; Tsay, R. An activity-sensitive contention delay model for highly efficient deterministic full-system simulations. In Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE), Dresden, Germany, 24–28 March 2014; pp. 1–6.
20. Castrillon, J.; Velasquez, R.; Stulova, A.; Sheng, W.; Ceng, J.; Leupers, R.; Ascheid, G.; Meyr, H. Trace-Based KPN Composability Analysis for Mapping Simultaneous Applications to MPSoC Platforms. In Proceedings of the Design, Automation and Test in Europe, European Design and Automation Association, Dresden, Germany, 8–12 March 2010; pp. 753–758.
21. Castrillon, J.; Leupers, R.; Ascheid, G. MAPS: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs. *IEEE Trans. Ind. Inform.* **2013**, *9*, 527–545. [CrossRef]
22. Michalska, M.; Casale-Brunet, S.; Bezati, E.; Mattavelli, M. High-Precision Performance Estimation for the Design Space Exploration of Dynamic Dataflow Programs. *IEEE Trans. Multi-Scale Comput. Syst.* **2018**, *4*, 127–140. [CrossRef]
23. Bringmann, O.; Ecker, W.; Gerstlauer, A.; Goyal, A.; Mueller-Gritschneder, D.; Sasidharan, P.; Singh, S. The next generation of virtual prototyping: Ultra-fast yet accurate simulation of HW/SW systems. In Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE), Grenoble, France, 9–13 March 2015; pp. 1698–1707.
24. Lee, E.A.; Messerschmitt, D.G. Synchronous data flow. *Proc. IEEE* **1987**, *75*, 1235–1245. [CrossRef]
25. Geilen, M.; Basten, T.; Stuijk, S. Minimising buffer requirements of synchronous dataflow graphs with model checking. In Proceedings of the 42nd Design Automation Conference, Anaheim, CA, USA, 13–17 June 2005; pp. 819–824.
26. Bhattacharyya, S.S.; Lee, E.A. Scheduling synchronous dataflow graphs for efficient looping. *J. VLSI Signal Process. Syst. Signal Image Video Technol.* **1993**, *6*, 271–288. [CrossRef]
27. Schlaak, C.; Fakih, M.; Stemmer, R. Power and Execution Time Measurement Methodology for SDF Applications on FPGA-based MPSoCs. *arXiv* **2017**, arXiv:1701.03709.
28. AMBA® AXI™ and ACE™ Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite. Available online: https://developer.arm.com/documentation/ihi0022/e/ (accessed on 20 July 2021).
29. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.
30. Baccelli, F.; Cohen, G.; Olsder, G.; Quadrat, J. *Synchronization and Linearity, an Algebra for Discrete Event Systems*; Wiley & Sons Ltd: New York, NY, USA, 1992.
31. Bhattacharyya, A. On a measure of divergence between two statistical populations defined by their probability distributions. *Bull. Calcutta Math. Soc.* **1943**, *35*, 99–109.