PathTracer: Understanding Response Time of Signal Processing Applications on Heterogeneous MPSoCs

Claudio Rubattu, Francesca Palumbo, Shuvra S. Bhattacharyya, Maxime Pelcat

PathTracer: Understanding Response Time of Signal Processing Applications on Heterogeneous MPSoCs

CLAUDIO RUBATTU, University of Sassari, Italy and INSA Rennes, IETR UMR CNRS 6164, France
FRANCESCA PALUMBO, University of Sassari, Italy
SHUVRA S. BHATTACHARYYA, University of Maryland, USA and INSA Rennes, IETR UMR CNRS 6164, France
MAXIME PELCAT, INSA Rennes, IETR UMR CNRS 6164, France and Institut Pascal, UMR CNRS 6602, France

In embedded and cyber-physical systems, the design of a desired functionality under constraints increasingly requires parallel execution of a set of tasks on a heterogeneous architecture. The nature of such parallel systems complicates the process of understanding and predicting performance in terms of response time. Indeed, response time depends on many factors related to both the functionality and the target architecture. State-of-the-art strategies derive response time by examining the operations required by each task for both processing and accessing shared resources. This procedure is often followed by the addition or elimination of potential interference due to task concurrency. However, such approaches require an advanced knowledge of the software and hardware details, rarely available in practice.

This work presents an alternative "top-down" strategy, called PathTracer, aimed at understanding software response time and extending the cases in which it can be analyzed and estimated. PathTracer leverages on dataflow-based application representation and response time estimation of signal processing applications mapped on heterogeneous Multiprocessor Systems-on-a-Chip (MPSoCs). Experimental results demonstrate that PathTracer provides i) information on the nature of the application (work-dominated, span-dominated, or balanced parallel), and ii) response time modeling which can reach high accuracy when performed post-execution, leading to prediction errors with average and standard deviation under 5% and 3% respectively.

Keywords: model-based design, dataflow, MPSoC, design space exploration, processing latency, signal processing applications

1 INTRODUCTION

Embedded systems are now the most widespread devices in both mass and industrial electronics. Their processing are managed by Multiprocessor Systems-on-a-Chip (MPSoCs) that embed a growing number of heterogeneous Processing Elements (PEs) in order to efficiently perform demanding signal and information processing. For this reason, the design of systems based on MPSoCs has become increasingly complex and several approaches aiming at limiting this complexity have appeared [13]. Among the concepts of embedded systems, Cyber-Physical Systems (CPSs) have been studied with interest by the scientific community in the last few years. These systems are capable of monitoring and controlling physical elements and consider heterogeneous components that interact with each other in different modalities depending on the context in which they operate [54]. Design and maintenance of such systems are extremely complex because of their multidisciplinary nature, their elaborate requirements, the heterogeneity of their components and the continuous communication between their physical and cyber layers [14]. Both embedded systems and CPSs have demanding requirements in terms of flexibility and efficiency. The former property requires solutions capable of performing different functionalities evaluated in various

Authors' addresses: Claudio Rubattu, University of Sassari, Sassari, Italy, 07100, crubattu@uniss.it, INSA Rennes, IETR UMR CNRS 6164, Rennes, France, 35708, claudio.rubattu@insa-rennes.fr; Francesca Palumbo, University of Sassari, Sassari, Italy, 07100, fpalumbo@uniss.it; Shuvra S. Bhattacharyya, University of Maryland, College Park, USA, ssb@umd.edu, INSA Rennes, IETR UMR CNRS 6164, Rennes, France, 35708, sbhattac@insa-rennes.fr; Maxime Pelcat, INSA Rennes, IETR UMR CNRS 6164, Rennes, France, 35708, maxime.pelcat@insa-rennes.fr, Institut Pascal, UMR CNRS 6602, Clermont-Ferrand, France, 63000, maxime.pelcat@uca.fr.

operating modes. The latter is needed for the exploitation of the system with respect to the performance and costs deriving from the design choices (e.g. regarding requirements associated with energy budget and response time). In order to facilitate these application requests, the use of parameters, associated with specific operating modes, offers high-level management of functionality.

Among the most common Key Performance Indicators (KPIs) of embedded systems and CPSs, *latency*, also called *system execution latency* or *response time*, plays an important role for the strategies considered for adapting signal processing solutions [20, 23]. Indeed, this metric represents the execution time of a complete elaboration of a functionality, starting from the acquisition of a given input data, and ending with the generation of an associated output data. With current MPSoC technology, the accurate evaluation of the response time implies complex timing analyses or strong limitations in the implementation of the hardware and software components. Indeed, although different approaches in literature aim at providing accuracy and reliability in the verification of the response time, their applicability depends on various aspects, such as: the availability of information regarding hardware and software features (cache management, bus arbitration, operating system scheduling decisions), and the amount of processing and requests in the use of system resources associated with the functionality [63]. In this context, the present paper aims to simplify MPSoC response time estimation, especially when parametric functionalities are performed on heterogeneous systems. In particular, the analysis focuses on parametric functionalities described via a dataflow computation model (which allows the use of parameters) and examined in the single-rate DAG version suitable for their scheduling. However, reducing latency evaluation complexity does not come for free, and typically leads to a large loss of response time evaluation accuracy.

This paper introduces the PathTracer methodology for modeling the system execution latency of signal processing applications described through a dataflow Model of Computation (MoC). The objective of the developed method is to extract, from an application model, a Longest-Latency Path (LLP), that is a subset of the application workload sufficient to approximate its response time. Generalizing this concept, we refer as *activity* to the share of an application workload that determines a given KPI [48]. When evaluating latency, the LLP is the application activity, and having such information on the application workload opens up to a large set of studies, ranging from the design of abstract Models of Architecture (MoAs) to scheduling optimizations and Design Space Exploration (DSE). PathTracer aims at building a model of the MPSoC workload execution that offers more insights on response time activity than the Deterministic Actor Execution Time (DAET)-based solution provided by schedulers. This paper is an extension of our previous work [55]. The main contributions of this paper with respect to this previous work are i) the explanation of the PathTracer method to extract a Longest-Latency Path (LLP) from a dataflow application, and ii) the detailed analysis on functional use cases of the causes of MPSoC response time.

The rest of the paper is organized as follows. Section 2 discusses the dataflow-based description of signal processing applications that serve as information source for PathTracer and details the input MoC used in PathTracer. Section 3 explains the concept of Longest-Latency Path and how it differs from application Critical Path. Sections 4 and 5 respectively describe the PathTracer design flow and its assessment. Section 6 discusses current limitations of PathTracer and potential improvements. Section 7 concludes the paper.

2 DATAFLOW REPRESENTATION OF SIGNAL PROCESSING APPLICATIONS

This paper focuses on signal processing applications that process streams of data through transformational operations. The MoC used to describe such a functionality impacts the DSE strategy exploitable for specific KPIs [2]. The large available set of MoCs led to the development of many tools capable to evaluate a functionality in terms of timing behavior [1, 40, 43]. Depending on the latency evaluation objectives and their requirements in terms of accuracy, DSE strategies can lead to intractable evaluation problems and long times of development and analysis. This is mainly due to the amount of information (even not always available) required in this type of investigation, which implies scalability issues. Another explanation is the presence of timing anomalies, i.e. the fact that decisions with small local influence can have a very large system-level influence [22].

Different design tools can be used depending on the constrained and optimized KPIs that have to be considered during development of the functionality. In particular, in order to achieve timing estimation and optimization for embedded systems, simulators working at different abstraction levels are exploited [44, 50]. Among these tools, DSE tools are focused on high abstraction levels and designed for obtaining fast evaluations. Conversely, low-level analyses such as the ones performed by instruction set simulators favor accuracy at the expense of an increased exploration time. For these reasons, having the possibility to include in the analysis a certain amount of information available depending on the degree of accuracy required (i.e. elasticity in the DSE) is desirable to efficiently exploit this trade-off and improve design productivity [2]. This is particularly true with large design spaces, where abstraction is needed to reduce exploration complexity [29].

Since dataflow (DF) models are optimized for modularity and abstraction, these MoCs are widely used in predicting streaming application response time. DF MoCs have in particular proven useful for modeling signal processing applications [5, 12]. When using a DF MoC, one of several alternative abstract models can be chosen to represent the computation of a functionality as a set of non-preemptive tasks which exchange messages with each other through First In, First Out data queues (FIFOs) [45]. Each DF MoC offers a different trade-off between application behavior predictability and runtime reconfiguration of the workload. Synchronous DF (SDF) [34] in particular makes it possible to precisely define an execution iteration that is indefinitely repeated after an initial transitory phase. In that case, response time can be defined as *the time between the beginning of the execution of the firstly executed data source actor in the graph iteration and the end of the execution of the last data sink actor, where a data source actor (data sink actor) corresponds to a task that acquires (provides) determined input data (output data) of the DF network* (see Figure 1). This notion of iteration is important for the present studies because it provides a formal ground for the definition of latency. Latency evaluations can be computed from a DF MoC instance and a model of the hardware architecture [38]. In that case, the accuracy of the timing analysis relies on the quality of these timing models, based on KPI estimates that define runtime behavior thresholds (such as worst-case execution time or average execution time).

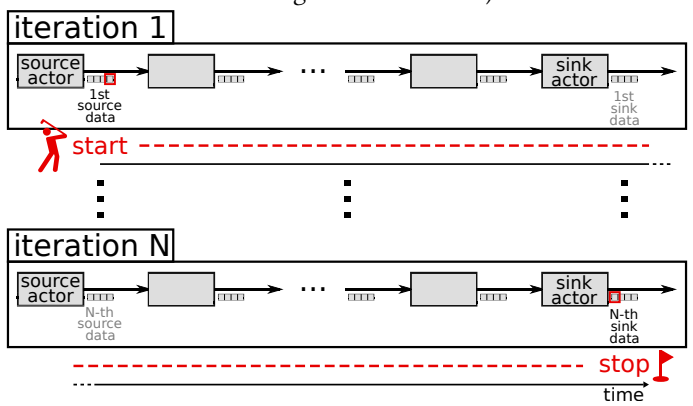


Fig. 1. Representation of response time along the graph iterations.

The proposed PathTracer method aims at estimating the latency of a DF modeled application. This modeling choice is motivated by the fact that DF MoCs give advanced information on both computation and data transfers in a stream processing application, represented as a network of tasks that communicate with each other. This approach brings modularity and abstraction that are assets for timing analysis, especially for heterogeneous architectures. Section 2.1 and Section 2.2 propose a state-of-the-art of the DF MoCs and the timing-focused tools based on these models respectively.

2.1 Dataflow Models of Computation

When modeling applications for timing analysis, various DF MoCs have been used, their semantics representing different modeling forms of execution, and interaction among tasks that describe the desired functionality. Each MoC brings specific features as reported in Table 1. This table focuses on main monodimensional DF MoCs while multidimensional MoCs [19, 41] constitute promising future directions for timing analysis of compact application representations.

MoC	Reconfiguration	Rate Variability	Determinism	Expressivity	Turing Completeness
KPN	N	N	N	high	Y
DPN	Y	Y	N	high	Y
SDF	N	N	Y	low	N
BDF	N	Y	N	high	Y
CSDF	N	Y	Y	low	N
HDF	N	Y	Y	low	N
SADF	Y	Y	N	medium	Y
PSDF	Y	Y	Y	medium	N
PiSDF	Y	Y	N	medium	N
SPDF	Y	Y	N	high	Y
BPDF	Y	Y	N	high	Y
EIDF	Y	Y	N	high	Y
CFDF	Y	Y	Y	high	Y

Table 1. Comparison of the presented DF MoCs.

Kahn Process Networks (KPNs) [30] model an application with parallel tasks communicating with each other via unbounded FIFOs. Each task represents a specific amount of computation of the whole functionality, which, at each iteration, receives input data (called *tokens*) through FIFO channels interfaces. In particular, FIFO writes and FIFO reads are non-blocking and blocking respectively. Other MoCs presented below derive from KPN [46]. As a specialization of KPN, Dataflow Process Networks (DPNs) [35] associate tasks (called *actors*) with *firing rules* that specify data arrival patterns that trigger actor-related execution modes (called *actions*), making FIFO reads potentially non-blocking. Such firings are completed without preemption and *firing rules* are based on the presence of tokens into bounded FIFOs. SDF [34] specializes DPNs by providing static firing rules, losing the capacity to reconfigure the datapath. Indeed, consumption and production rates associated with the number of tokens on FIFOs are fixed and constant for all executions. This choice improves design-time predictability and optimization. As an extension of SDF, Boolean DF (BDF) [10] introduces a new class of actors that control dynamic token consumption and production rates. This control flow makes the BDF model Turing complete. Other MoCs limited to applications that provide for the sequential execution of scenarios are the Heterochronous DF (HDF) [21] and the Scenario-Aware DF (SADF) [62]. HDF supports changing the token consumption and production rates through Finite State Machine (FSM), whose state remains fixed during a firing, and during which the system can be considered as an SDF model. SADF extends SDF with the concept of scenarios, which imply the dynamic control of the rates. An alternative way to perform reconfiguration of the data rates in a DF MoC is the Parameterized SDF (PSDF) metamodel [4]. Parameterized dataflow extends a DF model with dynamically reconfigurable parameters. As an extension of PSDF, Parameterized and Interfaced SDF (PiSDF) [15] refines the control over parametrization compared to the PSDF and adds the notion of interface [49] that insulates the different levels of hierarchy in the application representation. This insulation makes schedulability analyzable per hierarchy level in the graph. Modularity is obtained by improving graph composition and adding dependencies among parameters, whose relations are defined in a hierarchical tree. Other models that make use of parameters to set consumption and production rates are the Schedulable Parametric DF (SPDF) [17] and the Boolean Parametric DF (BPDF) [3]. These models represent

non-hierarchically reconfigurable generalizations of the SDF model. SPDF considers special actors in charge of modifying parameters after every specified number of firings, with the purpose of guaranteeing schedulability. BPDF leverages a combination of integer and boolean parameters in order to dynamically control rates and communication channels respectively. An alternative to the parametric approaches consists in using dynamic MoCs. With respect to the parametric-based MoCs, these models, such as Enable-Invoke DF (EIDF) [52] and Core Functional DF (CFDF) [51], offer more efficient implementations with different degrees of analyzability. These models are by nature less predictable than PiSDF. In the EIDF, consumption and production rates depend on the dynamic firing modes of the actors. CFDF corresponds to a subset of the EIDF, that provides for a restriction of the achievable next modes, from a series to a single and deterministic case.

In this work, the PiSDF MoC has been chosen for the availability of advanced tooling, the possibility to change parameters to adapt functionalities, and the predictability of the application representations supporting reconfiguration. However, LLP activity computation could be conducted from all forementioned dataflow models, provided that application model is representative of application behavior for the considered time window.

2.2 Timing-focused Tools for Design Automation of Dataflow Applications

In literature, various design automation tools relying on specific DF MoCs have been developed over the years. This section focuses on those tools that provide system-level analysis to optimize timing of functionality expressed with the DF models (see Table 2).

DF Tool	MoC	Architectural Model	Input Specifications	Availability
MAPS	KPN	cost funct. of target	C,CPN	academic
MPPA AccessCore	CSDF	abstract	Sigma-C	commercial
PREESM	PiSDF	S-LAM	C	open source
Ptolemy II	KPN,DPN,SDF, BDF,CSDF,HDF, PSDF	abstract	various languages	open source
SDF3	SDF,CSDF,SADF	abstract	commands, C/C++	open source
SPIDER	PiSDF	S-LAM	C++	open source

Table 2. Comparison of the main timing-focused tools based on DF MoCs.

MAPS [36] is a compilation framework that supports applications expressed with the KPN model. The main feature of MAPS consists of providing DSE in order to favor fast functional validation in systems with heterogeneous PEs, which are defined by their cost functions. Applications can be implemented in sequential C code or in its extension called C for Process Networks (CPN). Developed by Kalray, MPPA AccessCore [8, 31] is a commercial framework dedicated to MPPA multi-core systems. Applications are expressed as Cyclo-Static DFs (CSDFs) through a C-based language called Sigma-C. PREESM [47] is a rapid prototyping tool that provides design simulation and verification for applications described as a PiSDF graph. The tasks of the desired functionality are specified as C functions and mapped on a target modeled through the System-Level Architecture Model (S-LAM). In order to execute the application on the system, PREESM provides code generation for MPSoCs. Ptolemy II [53] provides modeling and simulation of heterogeneous systems expressed as combinations of various DF MoCs. A functionality is specified through a hierarchy of application graphs, whose levels conform to a determined MoC, which is represented in some modeling language (such as Java or C). SDF3 [61] is an open-source framework that offers analysis and simulation for SDF, CSDF and SADF MoCs. Functionalities are described by using command-line tools and C/C++ Application Programming Interface (API). Nevertheless, like with Ptolemy II, code generation of the application prototype for MPSoCs is not provided. SPIDER [26] offers runtime

management of applications specified as PiSDFs. The development of the functionality can take place by exploiting the PREESM environment, and the actors internal code is provided as C++ code.

In this work, PREESM is chosen for application representation since it provides a design environment for scheduling and mapping parametric functionalities at design time and generates MPSoC functional code for static and parameterized dataflow. In addition, this tool is open source and uses the S-LAM as an architectural model, which is suitable for describing heterogeneous MPSoCs with a high level of abstraction. Finally, PREESM provides automated Directed Acyclic Graph (DAG) generation when fixing application parameters and scales up to thousands of individual actor firings. As explained in the next section, this DAG serves as an input to PathTracer.

The next section leverage dataflow application representation to explain the concepts of Critical Path and Longest-Latency Path, as well as the different possible levels of latency evaluations depending on available system-level knowledge.

3 DAG-BASED TIMING ANALYSIS: KNOWLEDGE LEVELS, CRITICAL PATH AND LONGEST-LATENCY PATH

The application representation input to the PathTracer method complies to a subset of the SDF [34] MoC corresponding to a transformation into a single-rate DAG. In this form, production and consumption rates on each FIFO in the graph are made identical. Moreover, to this general graph, the following simplifications are applied: i) no cycle is considered (FIFOs from one iteration of the graph to the next are ignored) and ii) it is assumed that the number of initial data packets (*delay*) in each FIFO is zero. A formal definition follows.

The DAG model is represented as a finite directed, weighted graph $G =< V, E, m >$ where:

- V is the set of nodes called actors; each node represents a non-preemptive task that performs computation on one or more input data streams and produces one or more output data streams. Task execution is data driven: the task is fired as soon as input messages are available on all input edges.
- $E \subseteq V \times V$ is the edge set, representing messages between tasks.
- $m : E \rightarrow \mathbb{N}^*$ is the *message size* function with $m(e)$ representing the number of data indivisible units (called *tokens*) sent from the e source actor to the e sink actor in a message.

This DAG is used as an intermediate representation and is generated after consistency and liveness have been guaranteed on the PiSDF graph. Ignoring cycles and delays is consistent with the objective of analyzing latency of a single application iteration. Indeed, when the PiSDF parameters of the application are fixed, the model respects the SDF graph semantics and this semantics ensure that, if the graph is consistent and schedulable, a fixed sequence of actor firings, called graph iteration, can be repeated indefinitely to execute the graph [11]. There is then a well defined concept of a minimal sequence for achieving an indefinite execution with bounded memory [59]. The generated DAG is representing one iteration of the application and ignoring any edge holding initial data (called delays) makes it possible to analyze one iteration in isolation, the iteration number determining which source and sink task executions shall be associated to compute algorithm latency. The *single-rate* nature of the DAG makes all tasks executed only once per iteration of the graph. One may note that for the graph to be alive, all source actors, i.e. actors with no input edge, shall be fired at the same rate. This graph is a coordination language and as such, it only specifies the topology of the network, but does not give any information on the internal behavior of tasks. However, each actor is associated to a code implementing its internal functionality (e.g., a C code as in Section 5). Moreover, information is tagged on tasks and messages to model the relative cost of their management and, as discussed hereunder, latency precision accuracy depends on the level of knowledge of these tags.

Such a model is called single-rate DAG (srDAG) in the literature and has the advantage of simplicity at the cost of a lack of scalability for heavily multirate applications (e.g., see [28]). This srDAG entry point is chosen because it is common to many studies in the literature (e.g., [9, 18, 37, 39, 42]) and can be generated from many applicative representations, ranging from

advanced DF algorithms (such as SDF [34], CSDF [6] or PiSDF in our case [15]) to real-time application task sets. In the following discussions, we will refer to *DAG* or *srDAG* interchangeably, considering their meanings corresponding to the definition given in this section.

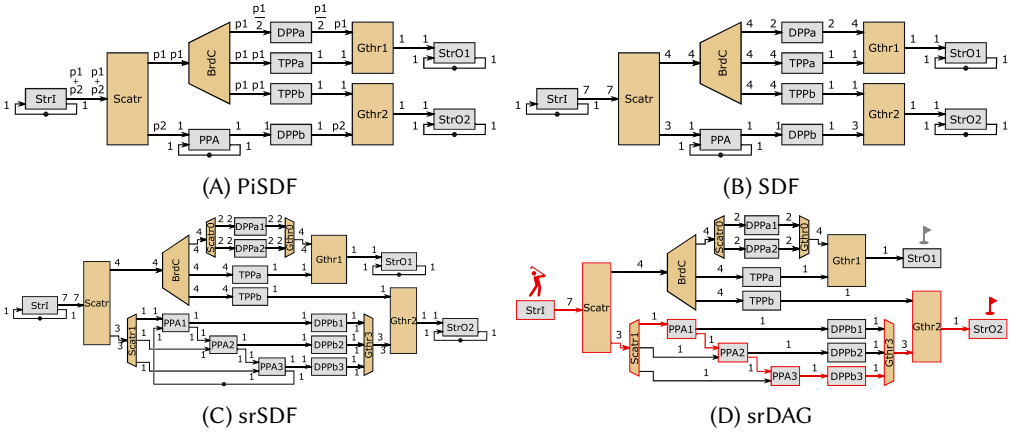
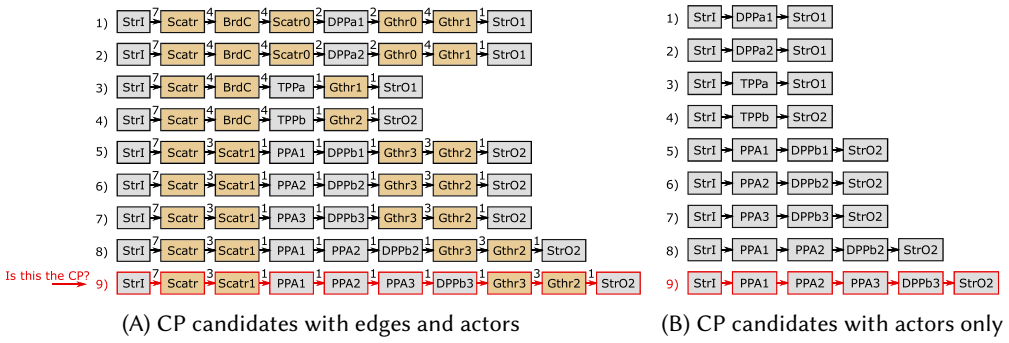


Fig. 2. Transformation steps of the application description from PiSDF to srDAG. Actors whose names are written vertically are simple data routers whose behavior is automatically generated. This is a pseudo-application with data and task parallelism, as well as pipelining. Gthr = Gather, Scatr = Scatter, BrdC = Broadcast, StrI = Stream Input, StrO = Stream Output, PPA = Pipelined Actor, DPP = Data Parallel Actor, TPP = Task Parallel Actor.

Figure 2 depicts the different steps required for obtaining a srDAG starting from a PiSDF description designed in PREESM. Figure 2A shows an example signal processing application described with PiSDF with only static parameters. In this illustrative example, N data packets (tokens) of a stream (StrI) are given as input to a scatter task (Scatr) every execution (firing), where $N = p1 + p2$ and $p1$ and $p2$ are integer parameters. Then, a broadcast actor (BrdC) conveys tokens to a data parallel actor (DPPa) and to task parallel actors (TPPa and TTPb), that fire $2 * p1$ and 1 times for each graph execution respectively, given their consumption/production rates of the communication links (edges). In the branch below, a pipelined actor (PPA) and another data parallel task (DPPb) are shown in sequence and executed both $p2$ times per graph firing. Finally, gather (reduce) actors (Gthr1 and Gthr2) collect and send the processing results to the stream output tasks (StrO1 and StrO2). Setting $p1 = 4$ and $p2 = 3$, a more specific SDF description can be derived (Figure 2B) and given as an input to the scheduler, that is capable to automatically convert it into its single-rate version (Figure 2C). In this description, the instances of the actors are made explicit, new routing blocks are added (Scatr0, Scatr1, Gthr0 and Gthr3), and the hypotheses on the number of their executions per graph firing mentioned previously can be verified. Defining latency of the application such as the execution time of a single graph firing, it can be evaluated by the difference between the start time of the first firing actor (StrI) and the end time of the last firing actor (StrO1 or StrO2). Following this mono-iteration strategy, feedback edges (with delay, represented by a black round in the link in Figure 2C) can be removed from the analysis because they represent inter-iteration dependencies, obtaining the srDAG (as in Figure 2D). We can take this hypothesis because two graph iterations are here not pipelined, i.e. a new iteration does not start as long as the previous one is running. If pipeline at a graph level was activated, additional interference would be to consider, that are not taken into account in this paper. Source and sink actors are highlighted in Figure 2D by respectively a golfer and golf holes.

In order to identify the latency contribute (or activity) of an application for latency estimation, the starting point of this work is to divide the whole application DAG into paths of tasks and edges linking sources to sinks. At this point, the Critical Path (CP) is extracted. CP is characterized by the following assumptions: i) it consists of a path that maximizes the sum of contributions, and ii) the CP determines the latency of the whole application. Each of source-to-sink paths



(A) CP candidates with edges and actors (B) CP candidates with actors only

Fig. 3. CP candidates list associated with the srDAG depicted in Figure 2D.

is a CP candidate (see Figure 3A). Then, the longest path, that will constitute the CP, can be approximated. As a further definition, the CP can be defined as the longest chain of dependent application elements causing latency. Indeed, every actor and edge (an edge in the DAG represents a message) may contribute to the CP with its specific weight in terms of time associated to its activity. Depending on the application domain, this general analysis can be simplified by focusing on the most relevant execution characteristics, removing those with negligible impact. An implementation of a signal processing application in an MPSoC is likely to require, to build an efficient system, token manipulation times to be small compared to those for their processing. In this case, the dominant activity is the processing associated with the actors, while communication weights deriving from signals/data exchange and routing shall be orders of magnitude lower. In the case of shared memory MPSoCs such as the one considered in Experiments Section 5.2, times for data reads and writes, included within task timings, depend on cache activities and directly model data transfer timings within task activity. Modeling a distributed memory MPSoCs would require separate modeling of the edges latency, not compulsory in this study. However, as will be shown in our experimental results, mapping and scheduling the activity of several CP candidates and ignoring the interference due to their combination is in general not realistic. To model these latencies, two different approaches have been considered in this study. In the first one, the CP is estimated by exploiting the information on the mapping and the pre-characterized/monitored timing of its elements. The second one extends the previous method by adding interferences to the CP, obtaining another set of latency contributions with higher accuracy: the Longest-Latency Path (LLP).

PathTracer aims at analyzing the determining factors that lead to the prediction gap between different levels of architectural information when a DAG-based application is mapped on a heterogeneous system, as motivated in [55]. The discussion starts from the levels (A, B and C) based on the information available before running the DAG on the MPSoC, such as DAET tags and architectural properties (called *a-priori information/knowledge*). On the other hand, the subsequent levels (D and E) consider tags of the start/end time, obtained only after the DAG execution and named *a-posteriori information/knowledge*. At all the defined (A-E) levels, the scheduling policy is hypothesised to be known to the simulation process so that levels represent knowledge of architecture rather than knowledge of the scheduling decisions. This notion of knowledge levels could be extended to knowledge of the platform scheduling policy but this study is kept out of the scope of the current paper. In experimental results of Section 5, the chosen scheduling is an list scheduling algorithm with As Soon As Possible (ASAP) task starting date [33] and the total order of the task list is known to the simulation process. We can list the mentioned levels of knowledge and related latency estimations:

- level A – **DAG with Single DAET Tags**: several works have analyzed the theoretical latency of the DAG depending on its CP [32, 46, 58]. As shown in Figure 3B, the DAG is divided into task chains starting from each source node (with no predecessor) and ending to every sink node (with no successor), that we will call *paths*. The latency evaluation from CP analysis is usually performed by using heuristic approaches based on path exploration and task

characterizations. For each CP candidate, DAG nodes are tagged with timing weights. In this context, since no information about the architecture is used, CP is considered to freely run on a system with unbounded (or large enough) memory and infinite (or as many as necessary) number of homogeneous PEs, as depicted in Figure 4A.

- level B – **DAG with Single DAET Tags and Architecture with a Specified Number of PEs**: with a known number of homogeneous PEs and a bounded memory, a heuristic-based scheduling/mapping strategy can be applied in order to achieve an optimized solution in terms of timing performance [56] (see Figure 4B). This procedure leads to a more realistic model of execution in which paths do not need to be analyzed in the latency representation, usually depicted with a Gantt chart. As tasks share PEs and messages share memory, contentions appear on architectural resources among tasks or communication operations. These contentions lead to interferences in DAG execution, which in turn increase latency and, if not predicted, reduce latency predictability.
- level C – **Architecture with a Specified Number of PEs of Different Types, and DAG tagged with one DAET per PEs Type**: a more accurate solution in terms of the DAG execution time can be achieved by distinguishing the types of the PEs in the target system (see Figure 4C). Nevertheless, with respect to the homogeneous case, this implies to provide latency prediction with the characterization of: i) the tasks for each type of PE, and ii) the elements of the hardware infrastructure. To do so, a DSE consisting of a statistical evaluation of the target-dependent costs associated to computation and communication may be required.
- level D (Trace) – **DAG with Start and End Time Tags and Real Architecture**: differently from the level C, the bus system linking multiple PEs is due to the real architecture, as well as the hierarchical structure of the memory. This case corresponds to an execution trace, extracting from code executions the start and end times of all tasks (see Figure 4D). It can be used as the reference Gantt chart to be looked for by models. At this post-execution stage, system latency is perfectly known. However, such a trace does not provide a full knowledge on the execution, as only start and end times of tasks are known but the causal chain of latency-causing phenomena is still unknown.
- Proposed level E: PathTracer – **DAG with Start and End Time Tags, Full Scheduling Information and Architecture with a Specified Number of PEs**: we propose PathTracer as a new level of latency explaining system model. This level extracts the LLP as the subset of tasks causing latency (see Figure 4E). At a PathTracer level, the objective of the analysis is to be able to tag the Gantt chart, knowing whether a particular task execution or message effectively belongs to the LLP, and to remove all tasks that do not participate to latency. PathTracer can be evaluated by a heuristic, evaluating start and end times, and tracing the most probable causal chain. Apart from CP tasks, the LLP may contain different types of interferences. *Scheduling interferences* are tasks whose execution delay the execution of a critical path task. *Dependency interferences* are tasks, or parts of tasks representing waiting time by a CP task for data produced by non-CP tasks. Other types of interferences can be defined and traced, to refine the level of understanding on the parallel execution.

The proposed strategy relies on an application-centric information: the application CP. By decomposing the list of elements determining latency into CP and interferences, we propose the novel concept of Longest-Latency Path as an equivalent to CP in levels B through E. Indeed, interferences are introduced by different phenomena, mainly: i) sharing of a finite number of hardware resources (processor, memory, interconnects, peripherals, etc.); ii) the nature of the application itself (task heterogeneity, scenario-based diversity in the actor characterization, task dependencies, etc.); iii) the often unknown memory access time by tasks, that prevents timing determinism [25].

With respect to the example proposed in Figures 2, 3 and 4, in level A no interference arises, since CP can run freely from the rest of the application (see Figure 4A). From B to D (see Figures 4B,4C, and 4D), the two main types of interference that can be observed are associated to:

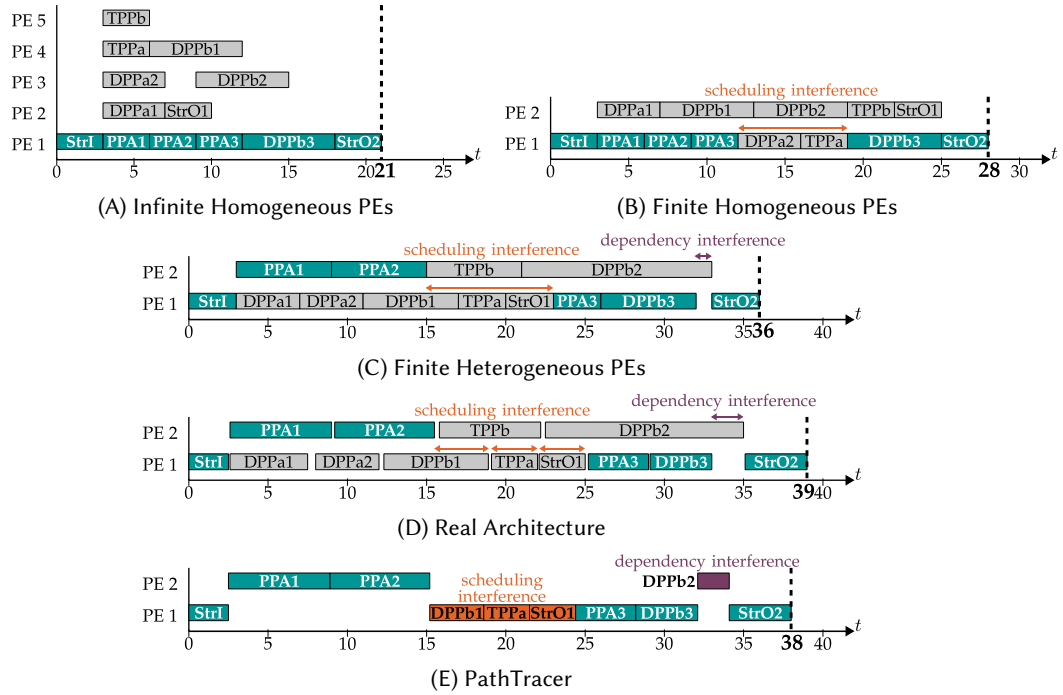


Fig. 4. Gantt chart with tasks execution times for each level of architectural information. From level C, PE 2 is of a different type than PE 1, and considered twice as slow as PE 1 (for each task).

- *scheduling*: when a CP actor execution is delayed due to other tasks that precede it in scheduling are mapped in the same PE (in Figure 4B, DPPb3 cannot be executed since DPPa2 and TPPa, not in CP, are still running);
- *dependency*: when the execution of a CP instance has to wait for tokens coming from other tasks that are not present in the CP (as shown in Figure 4C, StrO2 is waiting for a token from DPPb2).

Based on these notions of CP and LLP, the next section explains the PathTracer design flow that computes a LLP from an srDAG and architectural information.

4 FINDING THE LLP: THE PATHTRACER DESIGN FLOW

This paper aims at providing a practical method to extract level E information of Section 3. In this section, the design flow for estimating LLP for a parametric application described as a task graph is proposed. As depicted in Figure 5, the flow starts from the design of the dataflow application and the modeling of the target architecture (1, detailed in Section 4.1). Next phases can be performed in a loop for each scenario (values of parameters), starting from the scheduling step (2, described in Section 4.2). The application execution is monitored (3, see Section 4.3). A user that wants to apply the proposed PathTracer flow needs to carry out the 3 hardware (HW)- and software (SW)-specific steps by utilizing tools in order to schedule, map and monitor the DF-based application onto a heterogeneous system. After that, an automated chain of scripts analyzing system execution latency can be launched. At first, this process provides a list of CP candidates based on DAG analysis, estimates the longest one, and then evaluates interferences to the estimated CP in order to build the LLP that approximates the response time of the functionality. Indeed, while CP is the list of tasks causing response time in an idealized architecture with an unlimited number of PE, LLP is the list of tasks and communications determining response time in a real MPSoC, including many interferences of different kinds (as discussed in Section 3). The input list provided by the user for these phases consists of:

- the architectural information (number of PEs, and types, corresponding to the names, of PEs);
- the application srDAG with parameter values indicating how to explore it (note that when using the PREESM tool, this srDAG is generated from higher-level application description);
- the monitoring information including number of tasks executions and time measurements for individual tasks (this information is also provided by PREESM);
- the selected statistical metrics (e.g. average) on which to base the timing characterization and the LLP analysis.

Thus, depending on the chosen statistical metric, the timing cost of the actor and edge instances are extracted from measurements (4, see Section 4.4). In parallel, the CP candidates analysis explained in Section 4.5 is performed (5). Following to this method, the CP is obtained starting from the monitored actor times for a desired execution (6, see Section 4.6). Finally, the LLP analysis is performed (7, see Section 4.7). In the rest of this section, these PathTracer flow steps are detailed.

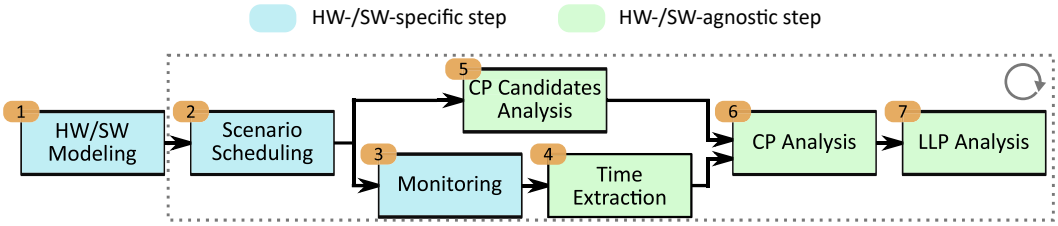


Fig. 5. Design flow implementing the PathTracer method.

4.1 HW/SW Modeling for LLP Analysis

In order to feed the LLP analysis, the model of the HW system and the application have to be provided to the scheduler. The architectural information consists of the specified number of PEs of different types (already known at level C, Section 3). On the other hand, the application is modeled as a srDAG (as in Section 3).

4.2 Scheduling Scenarios

In contexts in which instantaneous system requirements imply the reconfiguration of the application, functional changes can lead to application graphs with different parameter values, numbers of actor executions, numbers of messages and sizes of messages. For this reason, it can be useful for the user to identify specific operating modes (scenarios) of the application. The LLP analysis can then be focused separately on the srDAG structures associated with each individual scenario. This procedure gives the system designer data on the impact of parameter values on the execution in terms of latency speedup, and interferences. After a task characterization on multiple PE types from single-core executions, the proposed flow relies on the scheduling and mapping algorithm of the compilation tool (in our case PREESM) in order to generate and evaluate solutions, dependent on the number of PEs, on the scheduling strategy, and on time characterization of the actors and the communication cost. Moreover, in order to provide the most flexibility in executing actors, the HW configuration corresponds to the exploitation of all the PEs presenting in the HW target. In terms of interferences, this setup corresponds to the worst-case configuration, since the execution of the actors and their data exchanges are most likely to interfere with each other.

4.3 Monitoring: Measuring Local Tasks Timings

Due to the timing variability of the srDAG iterations, multiple measurements are required for their post-characterization. Indeed, starting and ending times of the tasks are obtained by a system monitoring, and used to obtain a CP linked to the particular running process. This CP takes part in the LLP evaluation, and computed as proposed in Section 4.6. Moreover, the measures feed the

latency evaluation of the srDAG. Please note that measurements with both cold and warm caches may be necessary to provide later analysis with realistic timings.

4.4 Task Post-Execution Latency Characterization and Relevant Executions

Depending on the objective of the exploration, the design flow makes it possible to choose the latency metric of interest. This metric can be i) the post-characterization of the time contributions for DAG actors and edges, or ii) a specific task-level monitored execution that more than the others represents the response time evaluated with respect to the chosen metric. In this second case, average latency, median latency or Worst-Case Execution Time (WCET) of all executions can be chosen. In the evaluation of the PathTracer method, the choice is made to concentrate on average latency. This choice is motivated by the fact that the considered applicative algorithms are often ported in soft real-time systems. Moreover, the static nature of their dataflow structure makes the time variations of the considered systems limited, with under 5% of latency standard deviation (Section 5.2). Average latency is thus a good tool for system analysis in this soft real-time context.

4.5 Extracting the CP Candidates

In	srDAG
Out	all CP Candidates
Pros	no application knowledge is needed
Cons	poor scalability

Algo

Algorithm 1: CP Candidates extraction among all DAG paths.

```
1  Procedure f_nxt(starting_node,Links,Paths)
2      for i ← 1 to size(Links) do
3          next_node = get_target(starting_node,Links[i]);
4          if next_node then
5              update_paths(Paths);
6              Paths = f_nxt(next_node,Links,Paths);
7          end
8      end
9      return Paths;
10 end
11 All_Sources = find_all_sources(srDAG);
12 All_Edges = find_all_edges(srDAG);
13 for i ← 1 to size(All_Sources) do
14     CP_Candidates = f_nxt(All_Sources[i],All_Edges,CP_Candidates);
15 end
16 return CP_Candidates;
```

Table 3. Overview of the Complete CP Candidates Analysis.

Starting from the application srDAG, several ways can be considered to obtain CP candidates. In this work, complete and selective analyses have been evaluated (see Tables 3 and 4). The complete analyses does not require a-priori knowledge of the application and provides the user with all the paths present in the graph. However, this could not always be convenient since scalability issues may occur. To avoid these, in the selective analysis, the evaluation is limited by setting the maximum number of paths per source to be considered. Moreover, being familiar with the application favors the reduction of computational complexity by selecting sources and sinks of the paths that have to be analyzed in priority.

4.5.1 Description of the Algorithms. Starting from each graph source actor (i.e. without predecessor), the search algorithm performs a depth-first walk (lines 14 and 16 in Algorithms 1 and 2, present in Tables 3 and 4 respectively) until all paths for each sink actor has been found (see procedure f_nxt, lines from 1 to 10 in Algorithm 1). The heuristic version of such procedure evaluates only N paths

In	srDAG, selected sources/sinks
Out	selected CP Candidates
Pros	tractable complexity
Cons	application knowledge is needed

Algo

Algorithm 2: CP Candidates extraction among selected paths.

```
1  Procedure f_nxt_N(starting_node,Links,Paths,N)
2      if size(Paths) < N then
3          for i ← 1 to size(Links) do
4              next_node = get_target(starting_node,Links[i]);
5              if next_node then
6                  update_paths(Paths);
7                  Paths = f_nxt_N(next_node,Links,Paths,N);
8              end
9          end
10     end
11     return Paths;
12 end
13 Sel_srDAG = sel_graph(srDAG,Sel_Sources,Sel_Sinks);
14 Sel_Edges = sel_edges(Sel_srDAG);
15 for i ← 1 to size(Sel_Sources) do
16     CP_Candidates_i = f_nxt_N(Sel_Sources[i],Sel_Edges,CP_Candidates,N);
17     CP_Candidates = append_paths(CP_Candidates,CP_Candidates_i);
18 end
19 return CP_Candidates;
```

Table 4. Overview of the Selective CP Candidates Analysis.

for each selected input (see f_nxt_N, lines from 1 to 12 in Algorithm 2). Procedures sel_graph and sel_edges (lines 13 and 14) are used to select only the part of the srDAG affected by the paths with the chosen sources (Sel_Sources) and sinks (Sel_Sinks) with their associated edges (Sel_Edges), instead of considering all the sources (All_Sources), sinks (All_Sinks) and edges (All_Edges).

4.5.2 Complexity of the Algorithms. A full exploration of the srDAG can easily lead to an explosion of the investigation timing depending on the number of nodes and links among them. Indeed, Algorithm 1 presents a complexity of $O(s \cdot e \cdot n)$, where s = All_Sources, e = All_Edges, and n is the maximum length of the paths in the srDAG. Hence, to make the analysis tractable on large graphs, the search space should be strategically pruned. The analysis can be limited to $O(s' \cdot e' \cdot n')$, with $s' \leq s$, $e' \leq e$, $n' \leq n$, s' = Sel_Sources, e' = Sel_Edges, and n' the maximum length of the paths in the limited srDAG. The space complexity can be reduced to $O(s' \cdot e'' \cdot n'')$ (with $e'' \leq e'$, $n'' \leq n'$) by limiting the walk to only N CP candidates per source. In the current version of PathTracer, the parameters s' and n'' that limit the complexity of the analysis are given as inputs by the designer, while e' and e'' are obtained in dependence on these. This choice of the number of considered CP candidates gives a control to the designer of a trade-off between modeling accuracy and simulation time. Indeed, if the discovered CP is substancially smaller than the real CP, interferences will be over-estimated and the model will loose its capacity to analyse latency causes. The current version of PathTracer prunes randomly the candidate paths. More advanced heuristics, based e.g. on forking and joining points in the DAG, could be imagined to make the pruning more efficient.

4.6 Choosing the Most Probable CP

In	CP Candidates, measured execution time tags
Out	CP, CP latency
Pros	analysis of runtime latency variability
Cons	poor accuracy, running information needed

Algo

Algorithm 3: Detection of the CP among the CP Candidates.

```
1  max_diff = 0;
2  for i ← 1 to size(CP_Candidates) do
3      path_lat = 0;
4      for j ← 1 to length(CP_Candidates[i]) do
5          actor_lat = Mon_Actor_Lats[CP_Candidates[i, j]];
6          path_lat = path_lat + actor_lat;
7      end
8      end_start_diff = get_diff(Gantt,CP_Candidates[i]);
9      if i == 1 then
10         path_proc_ratio = path_lat/end_start_diff;
11     else
12         path_proc_ratio = path_lat/max_diff;
13     end
14     if end_start_diff > max_diff then
15         max_diff = end_start_diff;
16     end
17     if path_proc_ratio > CP_proc_ratio then
18         CP_lat = path_lat;
19         CP_proc_ratio = path_proc_ratio;
20         CP = CP_Candidates[i];
21     end
22 end
23 return CP and CP_lat;
```

Table 5. Overview of the A-Posteriori CP Analysis.

Among the CP candidates collected in the previous phase, the critical (longest) one can be detected with strategies tailored to the available system-level information. In the proposed method (see Table 5), details of the monitored execution are exploited: scheduling, mapping, starting and ending times of each actor (available at level E). CP provides an estimation of the response time with a poor accuracy, that needs to be integrated in an LLP by adding interferences (as will be seen in Section 4.7). However, this method makes it possible to examine latency variability of the CP tasks during the execution. Among the paths provided as input (CP_Candidates), Algorithm 3 of Table 5 detects the CP as that path with the maximum difference (max_diff) between the end time of its last actor and the start time of its first one (by the procedure get_diff). As a next step, the Gantt chart describing scheduling/mapping of the real execution (Gantt) is explored. In addition, the processing ratio (path_proc_ratio) between the latency due to the all of the actors in the path (path_lat) and the one due to the whole path execution (end_start_diff) determines the choice of the CP. If more paths with the same end-to-start difference are present, the choice falls on that one with the largest processing ratio. The logic behind this decision aims to reduce the impact of the communication links (not considered in this version of the PathTracer) that can occur in the time windows present among executions of connected CP actors. Communication times have been ignored because all experiments have been conducted on an MPSoC with shared memory for which inter-PE communications use loads/stores which latencies are measured within actor execution time.

4.7 Finding the LLP

In	CP, CP latency, execution times of actors, Gantt Chart
Out	LLP, LLP latency
Pros	high accuracy
Cons	medium complexity

Algo

Algorithm 4: LLP extraction based on the detected CP.

```
1  k = 1;
2  CP_lat = 0;
3  CP_int = 0;
4  for i ← 1 to length(CP) do
5      for j ← 1 to size(NonCP_Actors) do
6          if !is_present(NonCP_Actors[j],LLP,k) then
7              if get_pe(NonCP_Actors[j]) == get_pe(CP[i]) then
8                  actor_sched_int =
                     get_sched(CP[i],NonCP_Actors[j],Gantt,LLP,k);
9                  CP_int = CP_int + actor_sched_int;
10             end
11         end
12     end
13     actor_dep_int = get_dep(CP[i],Gantt,Sel_Edges,LLP,k);
14     CP_int = CP_int + actor_dep_int;
15     CP_lat = CP_lat + get_lat(CP[i]);
16     LLP[k] = CP[i];
17     k = k + 1;
18 end
19 LLP_lat = CP_lat + CP_int;
20 return LLP and LLP_lat;
```

Table 6. Overview of the Selective LLP Analysis. See Algorithm 5 (6) for details about get_sched (get_dep).

Algorithm 5: Procedure associated with the scheduling interference.

```
1  Procedure get_sched(CP_Actor,NonCP_Actor,Gantt,LLP,k)
2      actor_sched_int = 0;
3      if is_in_btw(NonCP_Actor,CP_Actor,Gantt,LLP,k) then
4          actor_sched_int = lat_in_btw(NonCP_Actor,CP_Actor,Gantt,LLP,k);
5          if actor_sched_int then
6              LLP[k] = NonCP_Actor;
7              k = k + 1;
8          end
9      end
10     return actor_sched_int;
11 end
```

Adding the interference to the CP analysis leads to a better awareness about the factors determining response time, exploiting the information of the level E presented in Section 3. However, this further investigation increases the evaluation time and the complexity of the analysis. Indeed, for each element of the CP, an identification of potential interferences is required (see Table 6). The different types of interferences have specific weights, and one of them often dominates over the others (e.g., in the case of overlapping, only that one due to the scheduling is counted in PathTracer analysis). Thus, a selective evaluation (see Algorithm 4 in Table 6) is desirable to reduce the time and complexity of the analysis. In this work, interferences due to scheduling and dependency (defined in the Section 3 and described more in details in Algorithms 5 and 6) have been added to the selective LLP analysis. In the case of overlapping in time (i.e. when interference phenomena

Algorithm 6: Procedure associated with the dependency interference.

```
1  Procedure get_dep(LLP_Actor,Gantt,Sel_Edges,LLP,k)
2      actor_dep_int = 0;
3      actor_chain_dep_int = 0;
4      DAG_pred_Actors = get_DAG_preds(LLP_Actor,Sel_Edges);
5      for i ← 1 to size(DAG_pred_Actors) do
6          if is_in_btw(DAG_pred_Actors[i],LLP_Actor,Gantt,LLP,k) then
7              actor_dep_int = lat_in_btw(DAG_pred_Actors[i],LLP_Actor,Gantt,LLP,k);
8              if actor_dep_int and !is_present(DAG_pred_Actors[i],LLP,k) then
9                  LLP[k] = DAG_pred_Actors[i];
10                 k = k + 1;
11                 actor_chain_dep_int = actor_chain_dep_int + actor_dep_int +
                     get_dep(DAG_pred_Actors[i],Gantt,Sel_Edges,LLP,k);
12             end
13         end
14     end
15     return actor_chain_dep_int;
16 end
```

occur simultaneously), only one contribute of interference is counted in the LLP, prioritizing the contribution of scheduling interference over dependency interference. Moreover, the dependency interference is computed in a recursive manner, considering the chain of interdependent tasks (performed in order of execution given by the srDAG) that interfere with the CP. Also in this case, simultaneous contribution are handled to occur only one at a time (i.e. excluding the sum of several parallel interference contributions in the same time window). Figure 6 shows how LLP contributes to three Gantt charts representing the real execution of one DAG iteration of three different applications mapped on a heterogeneous 8-PEs MPSoC, in which PEs with index from 1 to 4 are less performing in terms of Instruction Set Architecture (ISA) than PEs with index from 5 to 8. LLP is represented as a succession of the execution time contributions of tasks associated with the CP (in teal color), scheduling interference (in orange) and dependency interference (in purple). The considered application mappings show different characteristics in terms of LLP. Indeed, Figure 6A shows a work-dominated application, that is, where the CP contribution in the LLP is lower than the interferences. In Figure 6B, the LLP is the result of a balanced mix of both contributions due to CP and interferences. Finally, Figure 6C depicts an LLP dominated by the CP (or span).

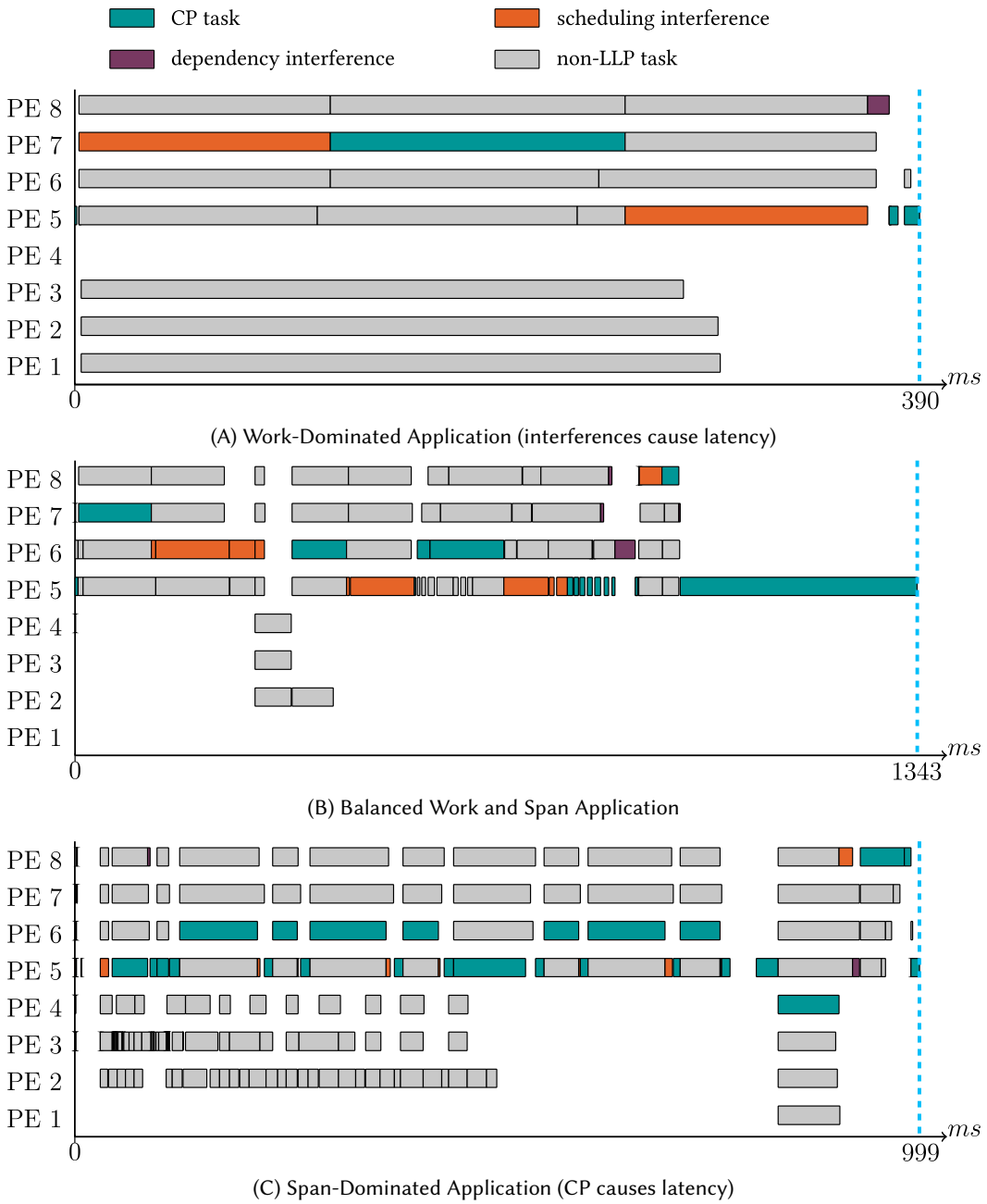


Fig. 6. Example of one iteration of DAG execution, represented as Gantt charts with real measured timings, of work-dominated, balanced work and span, span-dominated applications mapped on a heterogeneous MPSoC. These proposed execution views are associated with three DAGs respectively composed of 22, 74 and 172 actors (as will be explained in Section 5). LLP tasks are highlighted in teal, orange and purple, depending on the type of contribution, as tasks associated with: CP, scheduling interferences, and dependency interferences respectively.

5 EXPERIMENTAL RESULTS AND PATHTRACER ASSESSMENT

In this section, a comparison among the approaches described in Section 3 to analyze LLP is proposed. The flow described in Section 4 is assessed with 3 large scale PiSDF applications evaluated in several

scenarios and mapped onto an ODROID-XU3 board [24]. As explained before, PiSDF [15] is a DF MoC that models communication between tasks and algorithmic semantics handled by using high-level parameters. By using this MoC, the user can vary the consumption/production rates to obtain different structures of the srDAG. Indeed, each configuration of the PiSDF parameters represents a specific scenario of the application, having a certain number of actor instances. The PiSDF graphs of the chosen applications have been designed in PREESM [47], which embeds a non-preemptive ASAP multi-core list scheduler targeting latency minimization [33]. The monitoring process consists of 101 executions with the highest execution priority available for a process in the Linux environment for each evaluated scenario. Among these, the execution that best fits the mean value of the 100 measures (excluding the 1st execution to avoid typical initial events, such as cache warming effect) is selected to represent the response time of the scenario and the characterization of the srDAG instances. About CP candidates analysis, only 50 random paths per graph source are explored (i.e. the selective analysis as in Algorithm 2); we demonstrate that this amount of paths represents a good trade-off between complexity and DAG exploration for the use-case applications.

Regarding the target platform, the experimental setup processor is a Samsung Exynos 5422 composed of 8 ARM cores in a big.LITTLE configuration: 4 cores are of type Cortex-A7 with a cluster frequency up to 1.4GHz, and other 4 cores of type Cortex-A15 with a cluster frequency up to 2GHz. In the experiments, both clocks have been set at their maximum. All the 8 PEs present in the HW target have been used for mapping the tasks. This choice potentially leads to a high amount of interferences. The motivation for forcing the architecture to its maximum performance is to reduce the variations between different iterations of the same application. The dataflow nature of the applications, in which flowing data is written and read once, combined with these architectural choice leads to limited timing variations: latency standard deviation on the considered applications is ranging from 0.3% to 6.5% with an average of 2.8%. This latency stability makes the measured systems usable in a wide range of soft real-time systems.

5.1 Use-case Applications

This study exploits realistic use-case applications with functional code. In the context of image/video processing, the selected applications show distinct features in terms of actors, firing, and parallelism. These have been evaluated each in 6 different scenarios depending on their parameters in order to change the dataflow graph structure: CP candidates, pipeline actor stages, data parallel tasks.

5.1.1 Example of a Work-Dominated Application — Video Stabilization. This application is *work-dominated* for the considered architecture parallelism, thanks to data concurrency. This property means that the response time is influenced more by the number of PEs and architecture properties rather than by application limitations. Video stabilization reduces the effects of undesired fast camera movements due to a shaking camera during video recording. Post-processing techniques can analyze image motion, leading to the generation of a new video in which shaky movements are removed. The author thanks Karol Desnos for providing the PiSDF version of the Video Stabilization application. The proposed solution[1] has the DAG as reported in Table 7.

5.1.2 Example of a Balanced Work and Span Application — Stereo Matching. For the considered platform, the Stereo Matching application has complex behaviors in terms of response time: latency is influenced by both application CP and architecture number of PEs. From the comparison of two images from two different poses of the same scene, the Stereo Matching application obtains the scene depth information in the form of a disparity map. Indeed, disparity map pixels represent the distance between the locations of the same pixel in the two regularized views. The considered implementation[2] is detailed in terms of actors and edges in Table 7.

5.1.3 Example of a Span-Dominated Application — Scale Invariant Feature Transform (SIFT) Point Computation. This application is *span-dominated* as its response time in the architecture considered

[1] github.com/preesm/preesm-apps/tree/master/org.ietr.preesm.stabilization
[2] github.com/preesm/preesm-apps/tree/master/stereo/org.ietr.preesm.stereo

is strongly characterized by its CP. In the context of computer vision, the SIFT is an algorithm used to detect features of an image. Keypoints are extracted by the comparison between corresponding points of the input image and of the same image evaluated in its blurred versions and at different resolutions. In details, the evaluated implementation[3] [27] present actor and edge instances as in Table 7.

S	STABILIZATION		STEREO		SIFT	
	actors	edges	actors	edges	actors	edges
1	9	41	24	80	101	550
2	13	57	28	94	118	645
3	15	65	32	102	135	740
4	22	93	38	136	152	835
5	73	297	74	331	172	951
6	127	513	218	1099	222	1229

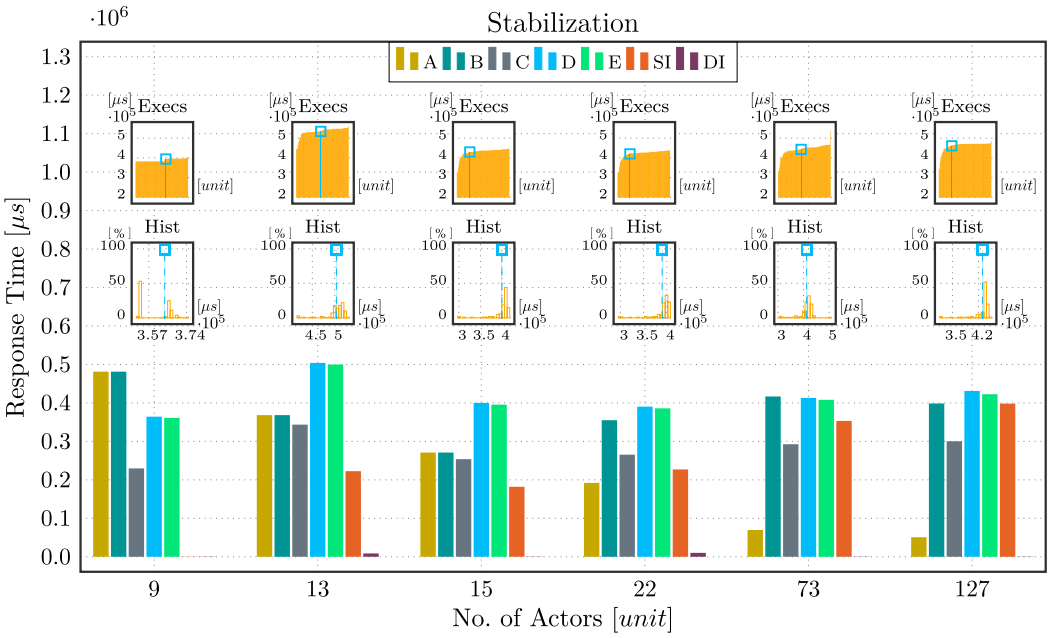
Table 7. Instances of the edges and actors in the 6 scenarios of the use-case applications.

5.2 Experimental Results

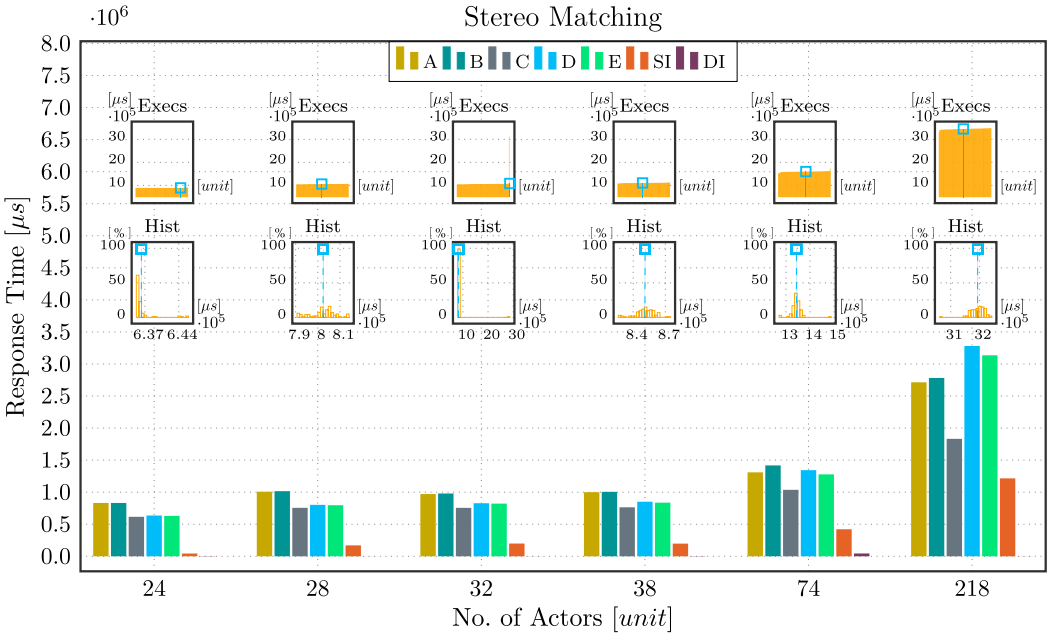
In this section, we demonstrate the insights a system designer can gain on MPSoC workloads using PathTracer. The latency evaluations possible at the knowledge levels A-E presented in Section 3 for the use-case applications are discussed. In particular, the task characterization for the a-priori knowledge levels A and B has been obtained by the average of the execution times associated to the different types of PEs present on the target board. In Figure 7, a comparison of the knowledge levels presented in Section 3 in terms of response time estimation is proposed. Figure 7 also includes two small plots for each scenario examined. These plots provide additional detail regarding the response time (Execs) and histogram over time (Hist) of the 100 measurements taken for the assessment. In the plots, the execution chosen to be analyzed by PathTracer (the average one) and its place on the histogram is highlighted, respectively. Figure 8 shows the percentage error with respect to level D, expressed as average (Avg) and standard deviation (StD).

5.2.1 Video Stabilization. Starting with the image stabilization application, Figure 7A shows an inversely proportional trend between CP and real latency, which indicates work domination. Indeed, when increasing the number of tasks, the latency estimates at level A decrease, while the interference grows. Compared to the level D (trace of the real latency) along all the scenarios, this leads to a poor accuracy when having only the knowledge of the level A. As reported in Figure 8, its error in terms of latency average and standard deviation is -36.1% and 44.4% respectively. Regarding the other a-priori knowledge levels (B and C), the estimates are more realistic since the exploitable parallelism in the application is higher than the number of the used PEs. Indeed, work domination is effective except in the first scenario (with 9 actors in total, but less than 8 with equivalent activities in parallel). Nevertheless, in these two levels, the error presents a mean value and a standard deviation equal to -2.0% and 22.2% (level B), and -33.1% and 2.9% (level C).

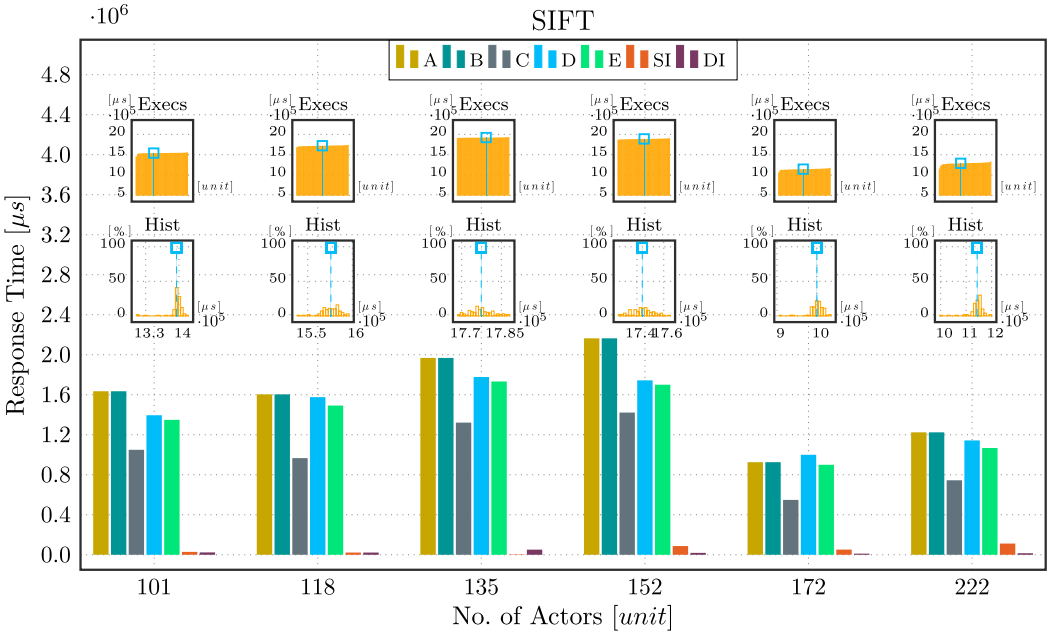
[3] github.com/preesm/preesm-apps/tree/master/SIFT



(A) Video Stabilization latency measurements and model-based evaluations from knowledge levels A to E, as well as Scheduling Interferences (SI) and Dependency Interferences (DI). Level D is the real measured latency while level E is the one evaluated by PathTracer.



(B) Stereo Matching latency measurements and model-based evaluations from knowledge levels A to E, as well as Scheduling Interferences (SI) and Dependency Interferences (DI). Level D is the real measured latency while level E is the one evaluated by PathTracer.



(C) SIFT Point Computation latency measurements and model-based evaluations from knowledge levels A to E, as well as Scheduling Interferences (SI) and Dependency Interferences (DI). Level D is the real measured latency while level E is the one evaluated by PathTracer.

Fig. 7. Latency estimates at different knowledge levels presented in Section 3. For 6 scenarios of each application, the execution is profiled for 100 measurements (displayed over time in Execs and as a histogram in Hist). Units are $10^5\,[\mu s]$ in small plots.

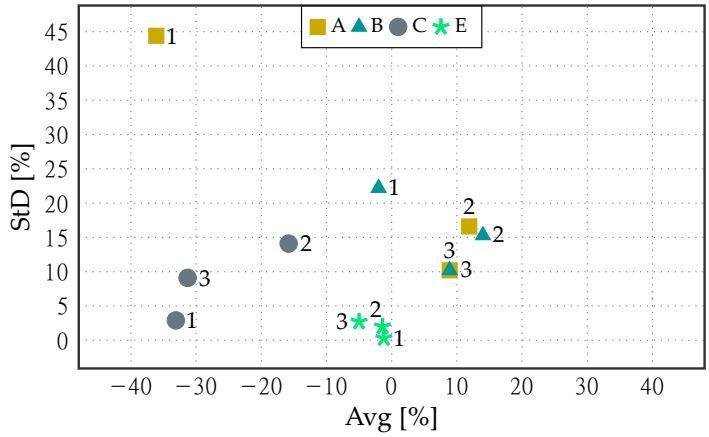


Fig. 8. Error of latency evaluation at the different levels with respect to the reference level D. For each level, percentage error in terms of standard deviation (StD) and average (Avg) along the scenarios is reported for the 3 applications: Video Stabilization (1), Stereo Matching (2) and SIFT Point Computation (3).

These results are explained by i) the wrong hypothesis in level B of a homogeneous MPSoC which leads to a large variations of the estimate around the average value, and ii) from the ignorance in level C of scheduling delays and interferences which cause underestimated response times. On the other hand, PathTracer level E achieves a highly accurate estimation, having an error with an average and a standard deviation respectively of -1.2% and 0.3%, demonstrating the interest of the PathTracer method to extract fine MPSoC execution behavior. The standard deviation of the PathTracer modeling error for a given iteration of the stabilization application is much lower than

the standard deviation of the execution latency between iterations that is ranging between 2% and 6.5% depending on the values of the stabilization parameters.

5.2.2 Stereo Matching. In the stereo matching application, the behavior is more subtile as both work and span act on latency. Activity associated to the CP and interferences from the rest of application work both influence execution time, as can be seen in all the levels in Figure 7B. Regarding the error on the latency estimates depicted in Figure 8, the levels A and B suffer from a lack of architecture knowledge. Indeed, they show an error of 11.9% (16.6%) and 14.0% (15.3%) in terms of average (standard deviation) respectively. On the other hand, estimations based on level C shows a decreasingly accurate estimate as the amount of work grows, with an mean error of -15.8% and a standard deviation of 14.1%. On the contrary, estimates obtained from the PathTracer (level E) follow the measured response times (Avg: -2.4%, StD: 2%). The good performances of PathTracer validate the heuristics composing the method. While not exploring all paths, PathTracer can still locate the causes of latency in the different applications. The standard deviation of the modeling error of PathTracer for the stereo matching application is reasonably higher than the standard deviation of the execution latency itself, ranging between 0.3% and 1.7% depending on the values of the stereo matching parameters.

5.2.3 SIFT Point Computation. As shown in Figure 7C, even if interferences do not dominate in this third application where CP dictates latency, PathTracer can still reduce the latency estimate error with respect to the other levels (A, B and C). Indeed, PathTracer presents the following characteristics: Avg = −5.0% and StD = 2.7% (see Figure 8). On the contrary, results based on the a-priori levels (A, B and C) show an error with mean values of 8.9%, 8.9% and -31.3%, and deviations of 10.2%, 10.2% and 9.1% respectively. The standard deviation of the modeling error of PathTracer for the SIFT point computation application is also a fraction higher than the standard deviation of the execution latency itself, ranging between 0.3% and 2.3% depending on the values of the SIFT parameters.

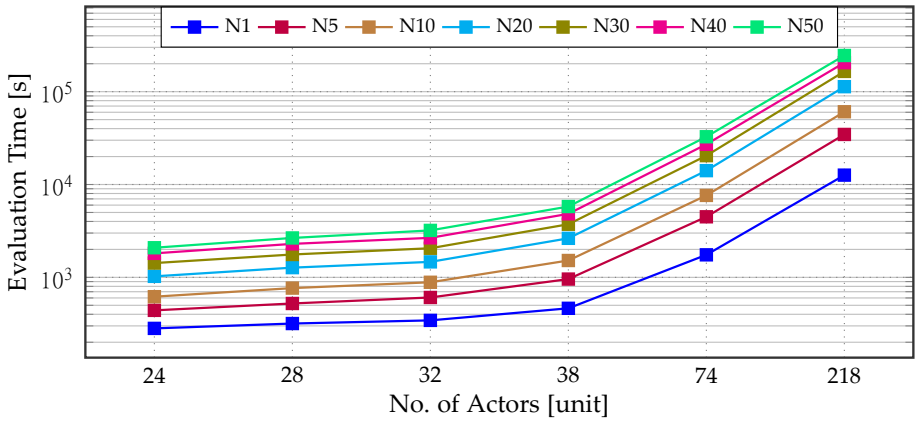
5.3 Analysis of the interferences

One can see in Figure 8 that levels A, B and C all have rather poor accuracy when attempting to match application latency. This is interesting because it means that adding knowledge on the heterogeneity of the platform without incorporating knowledge on interferences and effects due to concurrent actors on the platform is not sufficient to efficiently match the MPSoC behavior. This effect demonstrates the complexity of the problem and the need for advanced system-level information to understand the application data flow. Another conclusion from Figure 8 is that PathTracer is capable to match system latency. This element shows that the chosen two types of modeled interferences, discussed hereunder, are well adapted to the problem. It also shows that the LLP computed for the applications is good, and that all the rest of the application can be discarded while still computing accurate latency estimation.

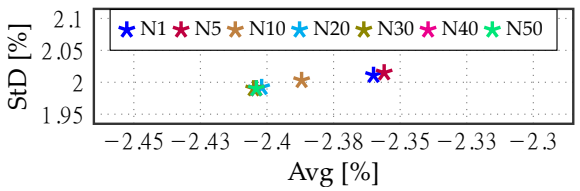
In this assessment of PathTracer, two types of interference have been evaluated. Indeed, the current LLP analysis of PathTracer evaluates only the effects due to first-order actor scheduling and actor dependencies, as detailed in Sections 3 and 4.7. Other interferences appear in the Gantt charts such as synchronization delays and data transfer delays. The choice fell on these two types since, in the context of signal processing, actor latency is expected to dominate communication latency. This hypothesis is important to explain the high accuracy of PathTracer when modeling the causes of latency. The hypothesis makes sense because the considered systems aim at processing signals with optimized processing latency and, if communication latency were dominating, the system would be inefficient, spending its time communicating data and not processing. Figure 7 also shows the subdivision of the interference for the applications and scenarios of Section 5.2 (SI and DI indicate scheduling and dependency interferences respectively). The interference of Video Stabilization grows with the amount of work of the actors and is mainly due to scheduling. Such an analysis is actually a method to determine that the application is work-dominated. This derives from the strategy applied by PREESM, that favors the mapping on the faster PEs until the parallelism does

not give any advantage. On the contrary in SIFT Point Computation, the dependency interference affects the CP until the last scenarios. However, the CP contribution mainly determines the response time, since the span dominates the execution. In Stereo Matching, both scheduling and dependency interferences grow with work. In the 5-th scenario the dependency interference starts to have a significant weight. As expected by an application in which coexist work and span effects, the combination of both types leads to an increase with the number of the actors.

5.4 Managing complexity in DAG exploration



(A) Evaluation time



(B) Error of latency evaluation

Fig. 9. Evaluation time and error of latency evaluation at the different N values (1, 5, 10, 20, 30, 40 and 50) with respect to the reference level D. Results are referred to the the Stereo Matching application, which has 2 source actors and 2 initialization actors evaluated as source nodes. Figure 9A reports high evaluation times due to the low-perfomance machine used (CPU: AMD A6-5200, RAM: 8-GB DDR3@1333MHz, storage: 1-TB HDD). In Figure 9B, percentage error in terms of standard deviation (StD) and average (Avg) along the scenarios is reported.

In this section, an analysis of the impact of the parameter N, i.e., the number of DAG paths evaluated for each actor source (as described in Section 4.5), is proposed. The analysis is presented in terms of evaluation time and estimation error, and focuses on the steps of the PathTracer flow influenced by the value of N (i.e., the last three steps in Figure 5 of Section 4). The examined use-case is the Stereo Matching application presented in the previous sections, since this application shows response times dependent on a combination of CP and interferences. This application feature indicates greater variability in DAG (e.g., CP contribution in LLP) along scenarios. Figure 9A shows how, as the number N of evaluated paths varies, the evaluation time becomes critical, especially for as the number of instances in the DAG increases. Nevertheless, low values of N still lead to the identification of an LLP that maintains accuracy in terms of response time for the scenarios examined (as depicted in Figure 9B). On the other hand, the weight of interference on the LLP decreases as N increases. This effect is related to the CP search strategy, which identifies CP as the path with the highest latency contribution associated with its tasks (see Section 4.5). As N increases, the LLP is characterized by an equal or greater weight of CP at the expense of interference.

However, in the cases examined, this variation in contribution is very small (as illustrated in Figure 10), thus leading to suboptimal but very close solutions in identifying the causes of response time. In addition to knowledge of the PiSDF, the choice of N can be made by successive iterations as a tradeoff between evaluation time and the magnitude of LLP error.

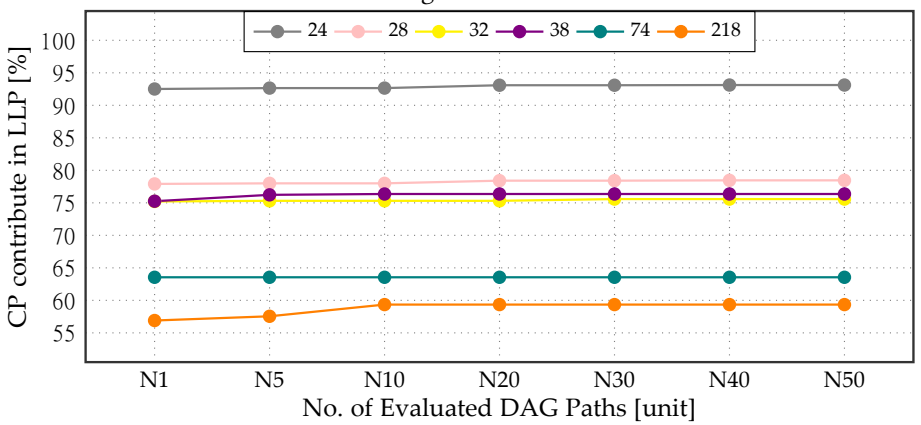


Fig. 10. CP contribute in percentage with respect to different N values (1, 5, 10, 20, 30, 40 and 50) along the scenarios of the Stereo Matching application.

6 CURRENT LIMITATIONS OF PATHTRACER

PathTracer is intended as a proof of concept of the benefits brought by extracting an LLP and explaining its components for the characterization of the average latency of MPSoC workloads. This section discusses promising future works for each step of the PathTracer design flow as proposed in Section 4. A first limitation of current version is associated with the scaling of the method. PathTracer takes an srDAG as an input (see Section 4.1) and the size of this graph may rise very fast depending on application parameters and exposed concurrency. PathTracer has been successfully tested on srDAG with more than a thousand actors but method modifications are needed to scale to larger application graphs. In case of reconfigurable applications, one srDAG analysis is required for each scenario to be analyzed (i.e. for each parameter values combination, see Section 4.2). In this regard, the use of design environments for describing applications through parametrized DF MoCs, as used in this paper, is strongly recommended. In addition to the scheduling and mapping PathTracer phases, the monitoring phase generates start and end times related to single elements of the srDAG (Section 4.3) and limits the analysis to non-preemptive executions. The analysis of monitored times is performed according to the same metric (e.g. average) for all instances of the srDAG (as mentioned in Section 4.4), without taking into account the statistical behavior of the specific tasks. The approach will gain at being combined with statistical actor time and data transfer analysis [60]. In parallel to the analysis of the monitored times, PathTracer explores the topology of the srDAG (see Section 4.5). PathTracer uses a depth-first search [16] and does not include yet an algorithm aimed at dynamically identifying the routes into the srDAG (as in route-finding problems [7]), since elements of the srDAG not present in the identified path are still counted as possible interferences. Nevertheless, this step of PathTracer suffers time complexity issues and will benefit from such optimizations. In the current version of PathTracer, LLP detection considers communication costs as embedded inside tasks costs. This hypothesis is valid in the case of shared memory MPSoCs because cache accesses and cache management operations (write-back, invalidate) are entangled with processing operations. When considering architectures with distributed memories, such as multiple networked systems, communications will need to be considered separately.

Finally, although LLP analysis is based on scheduling and dependency interferences (see Sections 4.7), some higher-order combinations of both can rise during execution, such as the following ones:

- *scheduling affected by dependency*: such interferences appear when an actor that interferes with CP through scheduling interference has to wait for a token from other tasks, which affects its execution with a dependency interference;
- *dependency affected by scheduling*: such interference appears when an actor that interferes with dependency interference has to wait for the execution of other tasks, which affect its execution with a scheduling interference.

These second-order interferences are not evaluated yet by PathTracer while, in some real cases, they do have a non-negligible impact, especially when the CP estimate is underperforming.

7 CONCLUSION

This paper has introduced the concept of Longest-Latency Path (LLP), as well as the PathTracer method to automatically compute LLP from an application dataflow representation. PathTracer aims at explaining factors that determine MPSoC execution latency. The design flow of PathTracer provides execution insights on applications described as a dataflow graph, as well as on monitored execution times. As shown by experimental results, PathTracer generates high-accuracy latency estimates based on average execution behaviors and helps determine bottlenecks as well as causes of speedup limitations. PathTracer and LLP computation have been evaluated on realistic, functional video processing applications and results show that, using PathTracer, a designer can study whether the heterogeneous MPSoC latency speedup is limited by the application (span-dominated implementation) or by the architecture parallelism (work-dominated implementation) and pinpoint the actors and messages interfering with the CP.

Experimental results show that models not considering interference when evaluating execution latency suffer from prediction errors of tens of percents. PathTracer on the contrary reaches an accuracy of a few percents. These results encourage further studies on methods to extend the applicability of LLP analysis to larger scale applications and to earlier design steps.

Finally, timing analyses are especially useful in the context of runtime management since, for the dynamic handling of task scheduling and mapping, runtime managers rely on profiling of the system execution in order to apply adaptation strategies [57]. PathTracer provides response time analysis based on measured metrics and promising future works include estimating LLP at runtime, from compile-time a-priori of application behavior.

ACKNOWLEDGMENTS

This research received funding from the European Union's under grant agreements No 783162 (FitOptiVis ECSEL project) and No 732105 (CERBERO H2020 project). Prof. F. Palumbo is grateful to the University of Sassari supporting her research activity through the "fondo di Ateneo per la ricerca 2019".

REFERENCES

[1] Ayaz Akram and Lina Sawalha. 2019. A Survey of Computer Architecture Simulation Techniques and Tools. *IEEE Access* 7 (2019).

[2] Manel Ammar and Mohamed Abid. 2018. Heterogeneity of abstractions in EDA tools: Reviewing models of computation for many-core systems targeting intensive signal processing applications. *Microprocessors and Microsystems* 59 (2018), 1 – 14.

[3] Vagelis Bebelis, Pascal Fradet, Alain Girault, and Bruno Lavigueur. 2013. BPDF: A statically analyzable dataflow model with integer and boolean parameters. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*.

[4] Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya. 2001. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing* 49, 10 (2001), 2408–2421.

[5] Shuvra S Bhattacharyya, Ed F Deprettere, Rainer Leupers, and Jarmo Takala. 2013. *Handbook of signal processing systems.* Springer.

[6] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. 1996. Cycle-static dataflow. *IEEE Transactions on Signal Processing* 44, 2 (Feb 1996), 397–408.

[7] Bing Liu. 1997. Route finding by using knowledge about the road network. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 27, 4 (1997), 436–448.

[8] Bruno Bodin, Alix Munier-Kordon, and Benoît Dupont de Dinechin. 2012. K-Periodic schedules for evaluating the maximum throughput of a Synchronous Dataflow graph. In *2012 International Conference on Embedded Computer Systems (SAMOS)*.

[9] Vincenzo Bonifaci, Andreas Wiese, Sanjoy K. Baruah, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Leen Stougie. 2019. A Generalized Parallel Task Model for Recurrent Real-Time Processes. *ACM Transactions on Parallel Computing* 6, 1 (2019).

[10] Joseph T. Buck. 1993. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model.* Ph.D. Dissertation. Department of Electrical Engineering and Computer Sciences, University of California at Berkeley.

[11] Joseph Tobin Buck and Edward A Lee. 1993. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *1993 IEEE international conference on acoustics, speech, and signal processing*, Vol. 1. IEEE, 429–432.

[12] Nicola Carta, Carlo Sau, Francesca Palumbo, Danilo Pani, and Luigi Raffo. 2013. A coarse-grained reconfigurable wavelet denoiser exploiting the Multi-Dataflow Composer tool. In *2013 Conference on Design and Architectures for Signal and Image Processing*.

[13] Alessandro Cilardo and Edoardo Fusella. 2016. Design automation for application-specific on-chip interconnects: A survey. *Integration* 52 (2016), 102 – 121.

[14] Patricia Derler, Edward A. Lee, and Alberto Sangiovanni Vincentelli. 2012. Modeling Cyber–Physical Systems. *Proc. IEEE* 100, 1 (2012), 13–28.

[15] Karol Desnos, Maxime Pelcat, Jean-Francois Nezan, Shuvra S. Bhattacharyya, and Slaheddine Aridhi. 2013. Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*.

[16] Shimon Even. 2011. *Graph Algorithms* (2 ed.). Cambridge University Press.

[17] Pascal Fradet, Alain Girault, and Peter Poplavko. 2012. SPDF: A schedulable parametric data-flow MoC. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[18] Adi Fuchs and David Wentzlaff. 2019. The accelerator wall: Limits of chip specialization. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[19] Abdoulaye Gamatié, Éric Rutten, Huafeng Yu, Pierre Boulet, and Jean-Luc Dekeyser. 2008. Synchronous Modeling and Analysis of Data Intensive Applications. *EURASIP Journal on Embedded Systems* 2008, 1 (2008).

[20] Marisol García-Valls, Diego Perez-Palacin, and Raffaela Mirandola. 2014. Time-Sensitive Adaptation in CPS through Run-Time Configuration Generation and Verification. In *2014 IEEE 38th Annual Computer Software and Applications Conference*.

[21] Alain Girault, Bilung Lee, and Edward A. Lee. 1999. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18, 6 (1999), 742–760.

[22] Ronald L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics* 17, 2 (1969), 416–429.

[23] Shah Ahsanul Haque, Syed Mahfuzul Aziz, and Mustafizur Rahman. 2014. Review of Cyber-Physical System in Healthcare. *International Journal of Distributed Sensor Networks* 10, 4 (2014), 217415.

[24] Hardkernel co. Ltd. [n.d.]. *ODROID-XU3.* https://www.hardkernel.com/shop/odroid-xu3/

[25] Mohamed Hassan and Rodolfo Pellizzoni. 2018. Bounding DRAM Interference in COTS Heterogeneous MPSoCs for Mixed Criticality Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2323–2336.

[26] Julien Heulot, Maxime Pelcat, Karol Desnos, Jean-Francois Nezan, and Slaheddine Aridhi. 2014. SPIDER: A Synchronous Parameterized and Interfaced Dataflow-based RTOS for multicore DSPS. In *2014 6th European Embedded Design in Education and Research Conference (EDERC)*.

[27] Alexandre Honorat, Karol Desnos, Maxime Pelcat, and Jean-François Nezan. 2019. Modeling Nested for Loops with Explicit Parallelism in Synchronous DataFlow Graphs. In *International Conference on Embedded Computer Systems*. Springer, 269–280.

[28] Chia-Jui Hsu, Ming-Yung Ko, Shuvra S. Bhattacharyya, Suren Ramasubbu, and José Luis Pino. 2007. Efficient Simulation of Critical Synchronous Dataflow Graphs. *ACM Transactions on Design Automation of Electronic Systems* 12, 3 (2007).

[29] Rik Jongerius, Andreea Anghel, Gero Dittmann, Giovanni Mariani, Erik Vermij, and Henk Corporaal. 2018. Analytic Multi-Core Processor Model for Fast Design-Space Exploration. *IEEE Trans. Comput.* 67, 6 (June 2018), 755–770.

[30] Gilles Kahn. 1974. The semantics of a simple language for parallel programming. *In Information Processing* 74 (1974), 471–475.

[31] Kalray. [n.d.]. *MPPA AccessCore.* https://www.kalrayinc.com/IMG/pdf/FLYER_MPPA_ACCESSCORE-2.pdf

[32] Torsten Kempf, Gerd Ascheid, and Rainer Leupers. 2011. Multiprocessor Systems on Chip: Design Space Exploration.

[33] Yu-Kwong Kwok and Ishfaq Ahmad. 1999. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *Comput. Surveys* 31, 4 (1999), 406–471.

[34] Edward A. Lee and David G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[35] Edward A. Lee and Thomas M. Parks. 1995. Dataflow process networks. *Proc. IEEE* 83, 5 (1995), 773–801.

[36] Rainer Leupers, Miguel Angel Aguilar, Juan Fernando Eusse, Jeronimo Castrillon, and Weihua Sheng. 2017. *MAPS: A Software Development Environment for Embedded Multicore Applications.* Springer Netherlands, 917–949.

[37] Yibin Li, Min Chen, Wenyun Dai, and Meikang Qiu. 2017. Energy Optimization With Dynamic Task Scheduling Mobile Cloud Computing. *IEEE Systems Journal* 11, 1 (2017), 96–105.

[38] Joel Matejka, Björn Forsberg, Michal Sojka, Premysl Sucha, Luca Benini, Andrea Marongiu, and Zdeněk Hanzálek. 2019. Combining PREM compilation and static scheduling for high-performance and predictable MPSoC execution. *Parallel Comput.* 85 (2019), 27 – 44.

[39] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. 2015. Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems. In *2015 27th Euromicro Conference on Real-Time Systems*.

[40] Christian Menard, Jerónimo Castrillón, Matthias Jung, and Norbert Wehn. 2017. System simulation with gem5 and SystemC: The keystone for full interoperability. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*.

[41] Praveen K. Murthy and Edward A. Lee. 2002. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing* 50, 8 (2002), 2064–2079.

[42] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2013. A General Constraint-Centric Scheduling Framework for Spatial Architectures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[43] Object Management Group (OMG). [n.d.]. *UML Profile for MARTE*. https://www.omg.org/spec/MARTE/1.1/

[44] Kenneth O'Neal and Philip Brisk. 2018. Predictive Modeling for CPU, GPU, and FPGA Performance and Power Consumption: A Survey. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*.

[45] Thomas M. Parks and Edward A. Lee. 1995. Non-preemptive real-time scheduling of dataflow systems. In *1995 International Conference on Acoustics, Speech, and Signal Processing*.

[46] Maxime Pelcat, Slaheddine Aridhi, Jonathan Piat, and Jean-François Nezan. 2013. *Dataflow Model of Computation*. 53–75.

[47] Maxime Pelcat, Karol Desnos, Julien Heulot, Clément Guy, Jean-Francois Nezan, and Slaheddine Aridhi. 2014. Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming. In *2014 6th European Embedded Design in Education and Research Conference (EDERC)*.

[48] Maxime Pelcat, Alexandre Mercat, Karol Desnos, Luca Maggiani, Yanzhou Liu, Julien Heulot, Jean-Francois Nezan, Wassim Hamidouche, and Shuvra S. Bhattacharyya. 2018. Reproducible Evaluation of System Efficiency With a Model of Architecture: From Theory to Practice. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 10 (2018), 2050–2063.

[49] Jonathan Piat, Shuvra S. Bhattacharyya, and Mickaël Raulet. 2009. Interface-based hierarchy for synchronous data-flow graphs. In *2009 IEEE Workshop on Signal Processing Systems*.

[50] Andy D. Pimentel. 2017. Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration. *IEEE Design Test* 34, 1 (2017), 77–90.

[51] William Plishker, Nimish Sane, and Shuvra S. Bhattacharyya. 2009. A generalized scheduling approach for dynamic dataflow applications. In *2009 Design, Automation & Test in Europe Conference & Exhibition*.

[52] William Plishker, Nimish Sane, Mary Kiemb, Kapil Anand, and Shuvra S. Bhattacharyya. 2008. Functional DIF for Rapid Prototyping. In *2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*.

[53] Claudius Ptolemaeus. 2014. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org.

[54] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. 2010. Cyber-physical systems: The next computing revolution. In *Design Automation Conference*.

[55] Claudio Rubattu, Francesca Palumbo, Shuvra S. Bhattacharyya, and Maxime Pelcat. 2021. PathTracing: Raising the Level of Understanding of Processing Latency in Heterogeneous MPSoCs. In *Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings (DroneSE and RAPIDO)*.

[56] Amit Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. 2013. Mapping on multi/many-core systems: Survey of current and emerging trends. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*.

[57] Amit Kumar Singh, Piotr Dziurzanski, Hashan Roshantha Mendis, and Leandro Soares Indrusiak. 2017. A Survey and Comparative Study of Hard and Soft Real-Time Dynamic Resource Allocation Strategies for Multi-/Many-Core Systems. *Comput. Surveys* 50, 2 (2017), 24:1–24:40.

[58] Oliver Sinnen. 2007. Task Scheduling for Parallel Systems. In *Wiley series on parallel and distributed computing*.

[59] Sundararajan Sriram and Shuvra S Bhattacharyya. 2018. *Embedded multiprocessors: Scheduling and synchronization*. CRC press.

[60] Ralf Stemmer, Hai-Dang Vu, Kim Grüttner, Sébastien Le Nours, Wolfgang Nebel, and Sébastien Pillement. 2020. Towards Probabilistic Timing Analysis for SDFGs on Tile Based Heterogeneous MPSoCs. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*. paper–59.

[61] Sander Stuijk, Marc C.W. Geilen, and Twan Basten. 2006. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*.

[62] Bart D. Theelen, Marc C. W. Geilen, Twan Basten, Jeroen P. M. Voeten, Stefan V. Gheorghita, and Sander Stuijk. 2006. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings*.

[63] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems* 7, 3, Article 36 (2008), 53 pages.