INTEGRATION, the VLSI journal 51 (2015) 118–138

Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

System-level synthesis of multi-ASIP platforms using an uncertainty model

Laura Micconi *, Jan Madsen, Paul Pop

DTU Compute, Technical University of Denmark, Denmark

ARTICLE INFO

Article history:
Received 1 December 2014
Received in revised form
11 July 2015
Accepted 13 July 2015
Available online 22 July 2015

Keywords:
System-level design
Multi-ASIP
Probabilistic model
Early design

ABSTRACT

In this paper we propose a system-level synthesis for MPSoCs that integrates multiple Application Specific Instruction Set Processors (ASIPs). Each ASIP is customized for a specific set of tasks. The system-level synthesis is responsible for assigning the tasks to the ASIPs, exploring different platform alternatives. We can allocate tasks to the different ASIPs and determine if the applications are schedulable only knowing the worst-case execution time (WCET) of each task. We can estimate the WCET only after establishing the micro-architecture of the ASIP. At the same time, an ASIP micro-architecture can be derived only knowing the assignment of tasks to ASIP. To address this circular dependency, we propose an Uncertainty Model for the WCETs, which captures the performance of tasks running on a range of possible ASIP implementations. We propose a novel stochastic schedulability analysis to evaluate each multi-ASIP platform. We use an Evolutionary Algorithm-based approach to explore the design space of macro-architecture possibilities and we evaluate it using real case studies.

1. Introduction

Embedded platforms are used for executing a wide variety of applications from the automotive, multimedia and networking domains. Flexibility and performance are the key design constraints for these platforms. General Purpose Processors (GPPs) are flexible platforms and run applications from various domains, but they fall behind on performance in comparison to ASICs. On the other hand, ASICs are designed to run specific applications and therefore lack flexibility. ASIPs combine the best of both worlds by incorporating application specific custom instructions, thereby giving more flexibility than ASICs and better performance than GPPs. ASIPs are designed such that they are optimized to run a specific set of functions. Recently, an increasing number of ASIPs is used in heterogeneous multi-processor SoC for the implementation of real-time systems (especially image/video processing systems) [1–3].

Designing heterogeneous multi-ASIP platforms for real-time application is a complex and time-consuming task, involving inter-dependent decisions on hardware and software architectures at macro-architecture (i.e., platform) and micro-architecture (i.e., ASIP) levels. These decisions affect the number of ASIPs, their micro-architectures and interconnections together with the assignment of tasks to the different ASIPs and they turn the optimization of the multi-ASIP platform into a NP-complete problem [4]. In this paper we propose a technique for the synthesis of a multi-ASIP platform given one or multiple applications as input. In particular, we focus on the macro-architecture (or system-level) synthesis that is in charge of assigning the tasks to the different ASIPs exploring different macro-architecture alternatives. Additionally, we show how the macro-architecture synthesis can be integrated into the flow. We use a design space exploration (DSE) to evaluate the schedulability of the applications on the candidate platform solutions to select the proper one.

To perform a schedulability analysis, we need to know the WCETs of the tasks. However, it is possible to know the WCETs only after all the ASIPs are synthesized. The synthesis of an ASIP (Fig. 1) starts with the identification of the tasks which have to be implemented by the ASIP. We call *task clustering* the partitioning of the application into multiple ASIPs and *task cluster* each group of tasks that corresponds to a single ASIP. Depending on the number of tasks and ASIPs included in the platform, a very large number of task clusters have to be evaluated during platform DSE. The micro-architecture synthesis of a *single* ASIP [5] involves a number of steps (see Fig. 1). Further, the design space of ASIP micro-architectures is very large depending on the number and data widths of registers (*RF*) and memory blocks (*MEM*) and the number of functional units (*FU*). Hence, platform synthesis with multiple ASIPs is non-trivial as it needs to take the design space of ASIP micro-architectures into consideration when exploring

* Corresponding author.
E-mail addresses: lmic@dtu.dk, laura.micconi@gmail.com (L. Micconi), jama@dtu.dk (J. Madsen), paupo@dtu.dk (P. Pop).

http://dx.doi.org/10.1016/j.vlsi.2015.07.006
0167-9260/

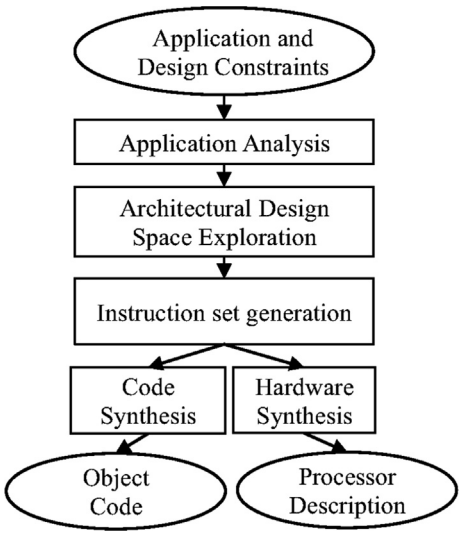


Fig. 1. Example of ASIP micro-architecture synthesis flow [5].

various platform solutions. We will use the term *micro-architectural configuration* to indicate the micro-architecture resulting from a specific ASIP synthesis.

There has been a significant effort in the development of platform synthesis methods. First, there are platform synthesis approaches that do not consider ASIPs. There is a large body of work in this category [6–12] and the assumption is that the details of each component are known. In particular [12,8] propose simulation frameworks for the mapping of applications on fixed multi-processor platforms, providing separate models for the application and the platform that are combined during the mapping. In [9], the authors implement a UML simulation framework for evaluating different mappings and platforms, but considering only a small set of processors. In [6,10], the authors explore different macro-architecture possibilities but always considering components taken from a library with well known performance and cost. The same happens in [11] where different mapping are explored to optimize throughput of applications running on a pre-defined platform where only the number of processors is varied. Point-to-point connections are used among the processors. In [7] the mapping is arbitrarily selected: each application is modeled as a synchronous data flow graph and each actor (task) is assigned to a different processor. Moreover, FIFOs are used for interconnecting the different processors, while in the approach presented in this paper, we consider different kinds of buses as interconnection type.

Second, there are platform synthesis approaches, which consider multiple ASIPs. Most of these approaches [13–15] assume that the ASIPs have been synthesized, whereas in [16,3], a small set of micro-architectural configurations is considered. In [16], the authors focus on pipelined multi-ASIP systems and explore different task graph partitionings, but the processors to be included in the platform are selected from a library of pre-configured ASIPs. They limit the interconnection network to a set of FIFO queues. Hence, these approaches severely limit the design space, with the risk of disregarding good solutions as they do not take into account the ASIP micro-architecture design space during platform synthesis.

Third, there are design approaches in which both ASIPs and their interconnections are optimized [17]. However, the ASIPs are synthesized starting from a template micro-architecture and the application is arbitrarily partitioned by the designer among a predefined number of ASIPs. In this approach, the authors do no evaluate different task clustering solutions that can also lead to a different interconnection selection.

All prior works address the *circular dependency* between the ASIP micro-architecture and WCET values by considering that the ASIPs, or a limited set of micro-architectural configurations of the ASIPs, are given. Therefore, to efficiently use these approaches, the designer needs to have some pre-existing experience and knowledge on the system that he expects as output. Hence, these works discard potentially very good solutions. To the best of our knowledge, there is no work on platform synthesis with multiple ASIPs, where the ASIPs are not synthesized beforehand and different task clustering are explored.

Contributions: In this paper we propose a method that addresses this circular dependency using an Uncertainty Model (*UM*) for the WCETs. This model captures the performances of a wide range of ASIP micro-architectural configurations. We use it to implement our *macro-architecture DSE* to guide the synthesis of a platform with heterogeneous processing elements (PEs) including multiple ASIPs, such that the applications have a high probability of meeting their timing constraints under given cost constraints. Our macro-architecture DSE is intended to be used in the very early phases of the design when no platform is available. Using an evolutionary algorithm (EA), we decide the clustering of tasks onto ASIPs and explore different types of bus-based platforms, considering a variable number of ASIPs and using the uncertainty model to predict the schedulability probability of each explored platform. Once the task clusters are established, we can synthesize each single ASIP, i.e., perform micro-architecture synthesis. This paper extends the work presented in [18]; the main differences are the use of a synchronous data flow graph (SDFG) to model the application (instead of a task graph), a new scheduling algorithm for exploiting both task-level and pipeline parallelism and the introduction of a new selection policy for EA. Additionally, in this paper we present the integration of the macro-architecture with the micro-architecture synthesis and their cooperation for the generation of a multi-ASIP platform using real case studies. For the implementation of the multi-ASIP platform and for obtaining a cycle accurate execution estimation of the application, we used the Silicon Hive (now part of Intel Corp.) tools for ASIP development [19].

The paper is organized as follows: Section 2 describes the system model for the macro-architecture DSE and the *UM* for the WCET, Section 3 defines the platform synthesis problem and highlights the main challenges using a motivational example. The scheduling algorithm and the EA implemented for performing the DSE are presented in Section 4. The evaluation of our approach, combined with the micro-architecture synthesis is presented in Section 5.2. The conclusions are in Section 6.

2. System model

The synthesis of a multi-ASIP platform requires the definition of both the macro- and micro-architectures. In this work we focus on the macro-architecture DSE and show how it can be included in the global design flow for multi-ASIP platform synthesis. This section is organized as follows: Sections 2.1 and 2.2 describe the application and platform models used for the macro-architecture DSE, respectively, while Section 2.3 introduces some concepts about ASIP micro- and macro-architecture synthesis that are necessary to motivate this paper, and finally Section 2.4 presents the *UM* for the WCET.

2.1. Application model

We assume to have an application A_i, modeled as a synchronous data flow graph (SDFG). We use a SDFG to capture both task-level and pipeline parallelism that characterize the streaming

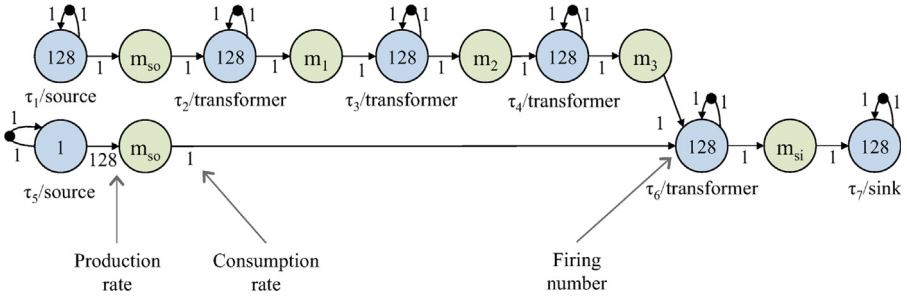


Fig. 2. SDFG model of MJPEG encoder.

applications that we are targeting. A SDFG is defined as a tuple $A_i = (\Gamma_i, \Omega_i, I_i, O_i, Z_i)$ [20] where each element in Γ_i is an *actor* (i.e., a *task*) and each element in Ω_i is an *edge* (i.e., a *message*) that models the communication between actors. A task is indicated as τ_j and a message is indicated as m_g.

Each task might have multiple input and output messages and executes by reading *tokens* (i.e., data) from its input messages, and by writing the results of the computation as tokens to the output messages. A message m_g, which models the communication between tasks τ_p and τ_c, has two values associated: the *consumption* and the *production rate*. The first corresponds to the number of tokens available on m_g and that are required by τ_c to fire, while the second is the number of tokens that the execution of τ_p produces on m_g. *Consumption* and *production rates* are contained in the sets I_i and O_i, respectively. A property of SDFGs is that every time a task executes, it consumes the same amount of tokens from its input messages and produces the same amount of tokens on its output messages. A task can start only after the required tokens are available at its input edges. We use the term *firing* to indicate a task execution. Z_i is the set of the *initial tokens*, i.e., the tokens already available on the edges before the execution of the tasks. Moreover we consider SDFGs without *auto-concurrency*: we do not allow multiple and simultaneous firings of the same task. This property can be forced adding a self-loop to each task with an initial token [21].

We also consider *consistent* SDFGs [22]: a graph is consistent if we can fire each task a fixed number of times and this will bring the SDFG to its original state, i.e., with the same distribution of tokens over all edges (messages). Each message has associated a value M_g that corresponds to the amount of data transmitted, i.e., the size of a token in bits.

An example of SDFG graph for a motion JPEG (MJPEG) encoder is shown in Fig. 2, in which the number inside each task indicates the number of firing of that task. Additionally, we identify *source*, *sink* and *transformer* tasks. Source tasks have no input edges, while sink actor has no output edges. They do not contain specific computation, but they are an interface with the environment. For the MJPEG encoder application for example, we have two source actors that set some environment variables and that simulate the arrival of frames for the encoding algorithm and a sink node that registers if the application terminated correctly. As these tasks are not part of the application, they are not assigned to any ASIPs and their WCET is set to zero. The tasks that are neither sink nor source are transformer (computation) actors.

Moreover, we use as additional information to model the applications, the value $iter_i$ that indicates the number of times that we want to execute the SDFG of the application A_i. For example the SDFG in Fig. 2 models the encoding of one frame of data. Therefore, a value $iter_{MJPEG} = 15$ expresses the encoding of 15 frames of data. Each application read A_i has a deadline d_i^{iter} that is the deadline associated to $iter_i$ execution of the SDFG.

Unlike the work done in [18], we have introduced a SDFG model as it allows exploiting both pipeline and task level parallelism. When it is possible to model the same application as a task graph and as a SDFG, we can observe that a task in the task graph corresponds to its associated task in the SDFG repeated a certain number of times [23]. Therefore, a single execution of a task in the SDFG is usually shorter; if multiple repetitions of the same task are independent, they can be scheduled separately; moreover dependent tasks can be scheduled earlier without waiting for the completion of all iterations of the task.

2.2. Platform model

Heterogeneous bus-based multi-processor platforms may contain multiple processing elements (PE) as GPPs, ASICs, digital signal processors (DSPs) or ASIPs. In this work, we focus on platforms containing multiple ASIPs, where the number of ASIPs and their configured micro-architectures are unknown and will be defined through our design method. A platform instance contains a number of processors interconnected through a bus system.[1] The kth processor is denoted by PE_k and has a frequency f_{PE_k}. A bus $b_w^{f_b}$ is characterized by a frequency f_b and a bandwidth w. We calculate the transmission time for message m_g on the bus $b_w^{f_b}$ as $t_g = \frac{M_g}{w * f_b}$. The processing elements and the bus can have different frequencies.

During the platform synthesis, our method explores different clustering solutions. A solution is composed of a set of clusters, i.e., groups of tasks executed on the same PE. A solution contains also the association of the messages to the interconnection architecture. For each clustering solution evaluated, our approach allocates an appropriate number of ASIPs. At platform level, our exploration also accounts for different types of buses. We use a *static scheduling policy* with non-preemption for the execution of the tasks on the platform. Additionally, we assume that each task reads data from local memory and writes them to remote memory. Let us consider two communicating tasks τ_1 and τ_2 assigned to two different processors PE_1 and PE_2. In our schedulability analysis, τ_1 reads the input data from the local memory of PE_1 and accesses the memory of PE_2 as a remote memory to write the data that produces (Fig. 3 shows an example).[2]

2.3. ASIP synthesis

Fig. 1 highlights the generic steps for the synthesis of a single ASIP, according to [25]. Examples of ASIP synthesis approaches are [26–28] that use LISATek Toolkit, [29] that uses Tensilica processor and the micro-architecture synthesis of the ASAM design flow [30] that we use in our case studies. To perform the ASIP synthesis, we

[1] Preliminary results using a Network-on-Chip are shown in [24].

[2] In this work, we did not consider different communication models such as shared memories; however it is possible to include them easily modifying the code of each task to allow both task producer and consumer to access an external shared memory, and considering the time to access the shared resource and the interconnection network during the schedulability analysis.

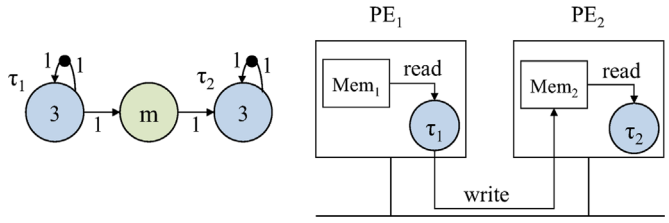


Fig. 3. Reading and writing policy: τ_1 reads from local memory and writes to remote memory.

need to know which tasks are assigned to it (task clusters). After the analysis of the code of these tasks, a micro-architectural DSE is performed in order to synthesize an ASIP compliant with the input constraints (e.g., performance, cost). The starting point for this exploration is an ASIP template (selected from a library) that most likely will satisfy the tasks' characteristics (according to the application profiling and the designer's experience). The definition of a specific ASIP micro-architecture includes the identification of the appropriate number/type of functional units, memory, issue slots, etc., in order to satisfy the functionalities required by the tasks assigned to the ASIP. After an initial micro-architecture is defined, the instruction set is generated, which can include custom instructions. Moreover depending on the tasks' code, it is possible to identify and implement custom instructions that can speed up the execution.

The next steps are the generation of the code and the HW synthesis of the ASIP. This design flow is not fully automated and it can take one or several days to complete [31,32]. In a multi-ASIP platform, every time we want to evaluate a task clustering, we would have to run a complete ASIP synthesis flow for each ASIP. As mentioned, an ASIP synthesis can be time consuming, so it cannot be done during the DSE at the platform level. Moreover to perform the DSE during platform synthesis, we need the WCET of the tasks to evaluate the schedulability, but the WCETs can only be known after the platform has been fully synthesized. This circular dependency drastically limits the number of platform alternatives that can be considered during DSE.

The approach commonly used in the industry is to leave to the designer the responsibility of manually identifying the task clustering. However, this approach relies on the experience of the designer and it is time consuming. Another possible alternative that is adopted in many design methods described in the literature (e.g., [16,3]) is to consider only a small set of predefined micro-architecture configurations for each ASIP so that the design flow of a multi-ASIP platform falls back into a classic MPSoC design. In this case the risk is to ignore potentially good solutions as the micro-architecture DSE is limited beforehand.

This paper offers an alternative approach to break this *circular dependency*, using an *Uncertainty Model* for the WCETs that enables a fast evaluation of more platform alternatives.

2.4. Modeling WCET uncertainties

In our problem, the WCET value depends on the ASIP micro-architecture, which is synthesized depending on how tasks are clustered. In this paper we propose a model to capture the design space of possible ASIP micro-architecture configurations: the WCET of each τ_j is modeled as a stochastic variable C_j and the associated probability distribution function. Such uncertainty models are used in practice in the early design stages [33].

Note that the variability of the *worst-case* execution time C_j of a task τ_j is due to the variation among the possible ASIP configurations on which task τ_j will run. It does not reflect the variation in execution time, which is due to variations in the input data and architectural features, such as branch prediction. The final implementation of the ASIP running task τ_j will only be available after the time-consuming ASIP micro-architecture synthesis. We use the probability distribution of C_j during DSE in order to avoid synthesizing every ASIP micro-architecture resulting from a change in task clustering.

We assume that the designer captures the probability distribution function of the WCET C_j of a task τ_j using two bounds: the smallest WCET value C_j^l (lower bound) and the largest value C_j^u (upper bound). The designer can arrive at these two values based on his or her knowledge of the functionality of the task and the possible range of ASIP micro-architectures. These values can also be estimated; the lower WCET bound can correspond to the expected WCET when τ_j is executed on an ideal processor according to an as soon as possible (ASAP) scheduling without architectural constraints. The upper WCET bound can correspond to a sequential execution of τ_j on the slowest possible ASIP. Within these two values, we use a normal distribution for C_j that models the WCETs of the task executing on an undefined ASIP that has not been synthesized yet. The reasons for using a normal distribution are provided in Section 5.1.

More formally, the cumulative distribution function (CDF) F_j of C_j is denoted as $F_j(x) = P(C_j \leq x)$, where the probability $F_j(x)$ is an indicator of the number of ASIP configurations that lead to a C_j smaller than a value x. The distribution is built such that $P(C_j \leq C_j^u) \approx 1$. This means that task τ_j will finish in C_j^u time units or less on *all* possible ASIP micro-architecture configurations. At the same time, we also assume that $P(C_j \leq C_j^l) \approx 0$.[3] This means that according to the designer's evaluation, *none* of the possible micro-architecture configurations will finish faster than C_j^l time units. Fig. 5(a) shows an example of CDFs for three different tasks.

Regarding messages, we assume that we know the size (in bits) of each message m_g. In this paper, we consider bus-based systems, and our DSE can explore different types of buses (for frequency and data width). Hence, we know the transmission time C_{m_g} of each message m_g. C_{m_g} is a single value and *not* a stochastic variable: for each type of bus that we want to explore, we have a different C_{m_g}.

3. Problem formulation

Given an application A_i (see Section 2.1) with deadline d, and a platform cost constraint $PC_{\max}$, the problem is to synthesize a system-level multi-ASIP platform, such that the probability of having a schedulable implementation is maximized under the specified cost constraint $PC_{\max}$.

Synthesizing a system-level platform means performing a DSE to decide the clustering of tasks and the interconnection. Our uncertainty model takes as input the SDFG of the application, its deadline and the cost constraint PC_{max} that is defined as the maximum number[4] of ASIPs that can be included in the platform, i.e., the maximum number of task clusters. We can consider a library of buses with different speeds and bandwidths, from where the DSE selects the appropriate bus. Also, there can be a set of legacy components that have to be used in the architecture and it is also possible that some tasks might be clustered on some specific PEs by the designer. Our optimization takes these constraints into account.

The designer, based on his/her knowledge or an analysis of the task code, provides the upper and lower bounds for the WCETs as presented in Section 2.4. Given the size in bits of each message and the library of buses, it is possible to determine the communication

[3] The CDF of the normal distribution does not reach the one and zero values.
[4] We will consider the ASIP area cost in our future work.

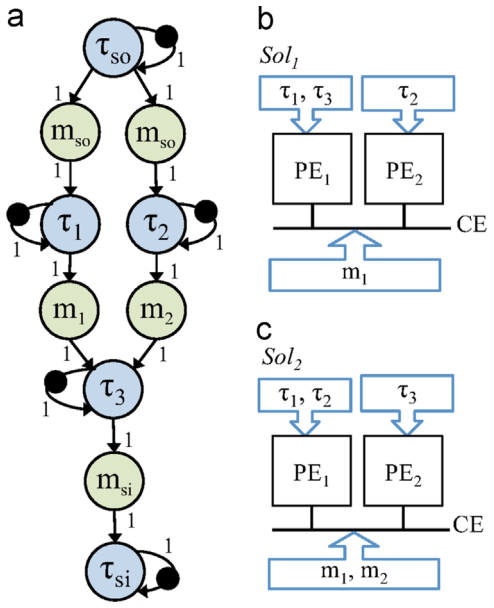


Fig. 4. Example of evaluation of clustering solutions.

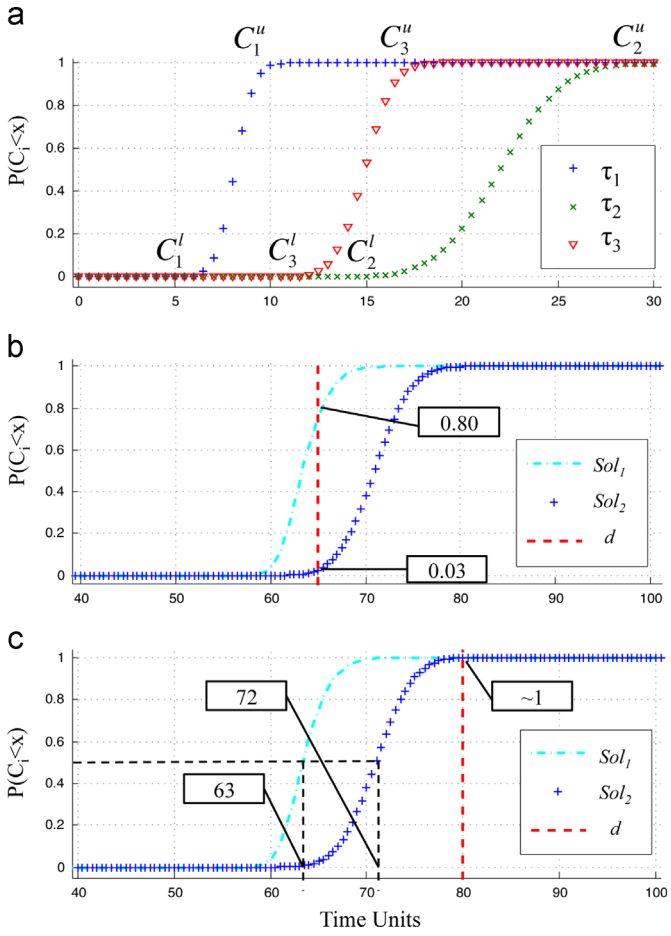


Fig. 5. (a) Input and (b,c) output CDFs for the example in Fig. 4.

time for each message. The DSE evaluates different clustering solutions as presented in Section 4.2 and selects the one which maximizes the probability of having a schedulable implementation. After DSE, we use an ASIP synthesis flow (e.g., Fig. 1) to synthesize an ASIP for each task cluster.

At the output of our multi-ASIP platform synthesis approach, we get a platform architecture, consisting of several ASIPs and possibly also legacy components, and their interconnection. For each ASIP, we have its micro-architecture, and the interconnection consists of a bus with a certain speed and bandwidth. This synthesis is performed under the platform cost constraint $PC_{\max}$. Fig. 6 shows our flow for the synthesis of a multi-ASIP platform.

The schedulability analysis of a task clustering for a given application A_i checks if the application deadline d_i^{iter} is satisfied. We consider a multi-ASIP bus-based platform, in which the ASIPs have not been designed yet. The maximum number of PEs that should be included into the final system (our platform cost, $PC_{\max}$) is provided as input. Additionally, we have the *UM* of each task, τ_j in A_i, and the transmission time, C_{m_g}, for each message (according to the bus type).

Let us consider the application SDFG in Fig. 4a and the *UM* of each task in Fig. 5a. $PC_{\max}$ is set to two. We evaluate two different task clustering solutions Sol_1 and Sol_2 (Fig. 4b and c); the results of our schedulability analysis are in Fig. 5b: we calculate the CDF of each task clustering solution. We obtain it combining the CDF of each task and the WCET of the messages. We evaluate each solution according to the application deadline (d_i^{iter}) and we obtain a probability p_{sol}.

Sol_1 has a probability $p_1 = 0.80$ while Sol_2 has $p_2 = 0.03$. This indicates that the first task clustering solution is better than the second one: a higher probability indicates that the task clustering is more likely to meet the application deadline when the platform is designed.

3.1. Motivational example

In this section, we use a small real case study (Fig. 7a) as motivational example to demonstrate the effectiveness of our approach. Given the application SDFG in Fig. 7a, we want to synthesize the platform architecture with multiple ASIPs such that the probability of having a schedulable implementation is maximized under the platform cost constraint $PC_{\max} = 2$. We assume that each of the tasks in Fig. 7a is fired only once and we assume $iter = 1$. For simplicity in Fig. 7a, as we consider a single iteration of the SDFG and each task is fired only once, we omit the self-loop to limit the auto-concurrency. Without the use of our *UM*, a designer, willing to identify the proper task clustering for its input applications, has to use a template of the ASIP micro-architecture. The designer can characterize the WCET of each task τ_j executing the task on the template processor. We denote this WCET as reference WCET, C_j^{ref} and this design approach as straightforward method (*SFM*). We use the *SFM* as a baseline to compare the results obtained with our *UM* approach.

Table 1 contains the upper and lower bounds for the *UM* and C_j^{ref} for the *SFM* for the example in Fig. 7a. These values are derived as follows for each of the tasks in Fig. 7a. For each task, we considered a simple functionality, consisting of a loop and operations such as multiplication and addition. We used the ASIP design flow of Silicon Hive [19] to design the ASIPs. We implemented a three-issue slot VLIW ASIP, and we ran the tasks to obtain their WCETs. We considered this WCET value as the reference WCET C_j^{ref}.[5] Furthermore, we varied the micro-architecture of this ASIP to obtain two extremes. The WCETs obtained using the slowest ASIP were considered as the upper bounds C_j^u, whereas the WCETs obtained with the fastest ASIP were considered as the lower bounds C_j^l. These values are presented in Table 1.

[5] Even if the value obtained executing the tasks on the ASIP is not a real and theoretical WCET determined through analysis, we believe this value is a good approximation for our experiments: the loop bounds are known at compile time and the input data are hard-coded in the tasks' code.

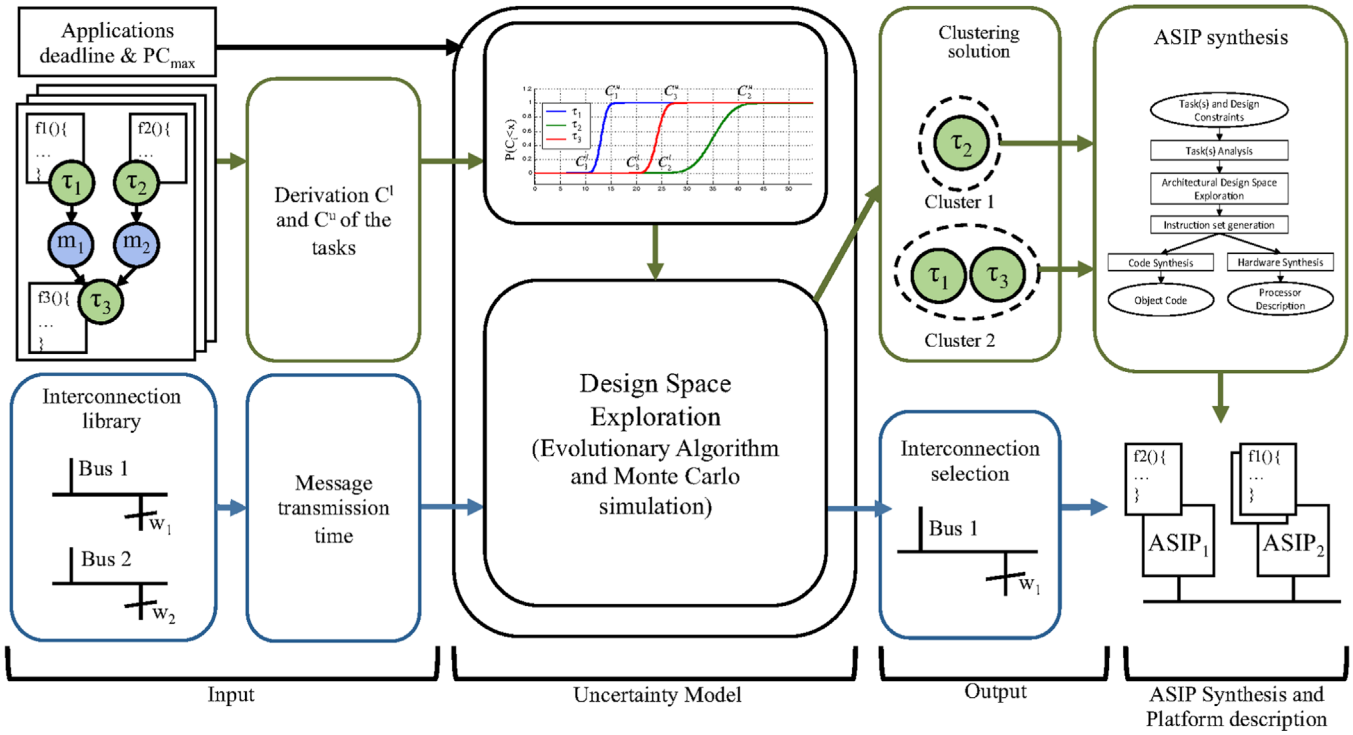


Fig. 6. Multi-ASIP platform design flow.

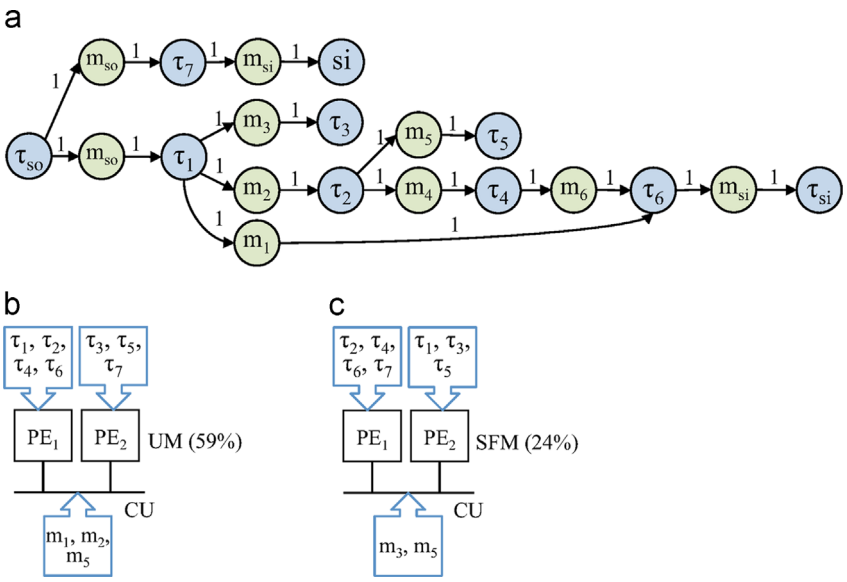


Fig. 7. Comparison between *UM* and *SFM* approaches.

Table 1
C values for the example of schedulability analysis (in μs).

C	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7
C_j^u	702	3437	8801	702	702	702	3437
C_j^l	450	180	300	450	450	450	180
C_j^{ref}	602	3237	400	602	602	602	3237

Each task clustering solution is evaluated using the schedulability analysis from Section 4, which gives the probability p of a solution to be schedulable, once it is implemented. We have performed an exhaustive DSE to explore the space of task clustering solutions. We did not assign the source and sink tasks to any ASIPs and we consider their WCET equal to zero. The best clustering solution obtained with *UM* is shown in Fig. 7b, having a $p=59\%$. Then, we perform an exhaustive DSE of all possible clustering solutions with the *SFM* approach, aiming at minimizing the schedule length, considering the given C_j^{ref}. The clustering obtained with *SFM* is shown in Fig. 7c. In order to compare the solutions obtained with the *UM* and *SFM* approaches, we calculate the probability p of the SFM solution to be schedulable using the WCET uncertainty model, as in the *UM* approach. Thus p for the solution with *SFM* approach is 24%.

To validate the results of the comparison between *UM* and *SFM*, we have synthesized the platform solutions in Fig. 7b and c, produced by *UM* and *SFM*, respectively. In the implemented multi-ASIP platform, the source and sink tasks are modeled by an host

processor (more details are available in Section 5.2). Next, we have determined the WCETs C_j of each task τ_j on the respective ASIP. Then, we have calculated the schedule lengths for the two cases. The schedule length in the case of the *UM* platform solution is 1955 μs, whereas for *SFM* is 2275 μs. For a deadline of 2000 μs, *UM* solution is schedulable, while *SFM* solution is not. This infers that if a *UM* solution has higher chances to be schedulable compared to a *SFM* solution according to our evaluation (Section 4.2), this is also true in the final implementation, as our synthesis using the Silicon Hive tools has shown. The comparison of the two approaches shows that with our *UM* approach, we are able to identify a solution that has a higher probability to be schedulable, once it is implemented.

4. Platform synthesis using an evolutionary approach

Given an application modeled as a SDFG, we evaluate different task clustering solutions to find the one that maximizes the probability of having a schedulable implementation. The DSE is responsible for evaluating different task clustering solutions; for each evaluated task clustering solution, we use a scheduling algorithm to define the static-order schedule that determines the firing order of the tasks. The scheduling algorithm is also used in conjunction with a Monte Carlo-based schedulability analysis for evaluating a given task clustering solution with our UM. In this section, we present the scheduling algorithm (Section 4.1), the schedulability analysis using the UM (Section 4.2) and the evolutionary algorithm for exploring the design space of task clustering solutions (Section 4.3).

4.1. SDFG scheduling algorithm

Given a SDFG and a task clustering solution, we use a *static* non-preemptive scheduling algorithm to define the execution order of the tasks on the different ASIPs providing a periodic admissible parallel schedule (PAPS) as defined in [34]. In this section, we present the algorithm assuming that the WCET of each task in the SDFG is available, while in Section 4.2, we describe how the algorithm can be used for schedulability analysis when considering the *UM* and, therefore, stochastic WCETs.

There are multiple approaches in the literature that propose scheduling algorithms for SDFGs that cover both the allocation of tasks to the multi-processor platform and the definition of their execution order. In our approach the DSE is responsible for evaluating the allocation of tasks (i.e., task clustering). We propose a scheduling algorithm that given a task clustering, determines the execution order of the tasks.

A list of scheduling algorithms for SDFG is available in [35]. There is a large group of approaches (e.g., [36–38]) in which the SDFG is transformed into its corresponding Homogeneous SDFG (HSDFG). An HSDFG has production and consumption rates for each actor equal to one and it has a node for each iteration of an actor in the initial SDFG [36]. The expansion of a SDFG to a HSDFG simplifies the scheduling algorithm, but can lead to very slow or memory consuming scheduler performance [39]. There are also examples of scheduling algorithm directly applied to the SDFG model as [40,41]. In [40] the authors propose an algorithm for minimizing the latency when an unlimited number of processing resources are available so that any enabled actor can be fired and all executions are feasible, while in [41], the authors define a static-order schedule assigning time slices to each actor optimizing the throughput.

In this paper, we define a scheduling algorithm for SDFGs that returns a static-order schedule. We do not focus on the definition of an optimal scheduling algorithm for SDFG, but on an algorithm that can be easily implemented on Silicon Hive's ASIPs reducing the application code size and the amount of data memory of each ASIP. We produce a scheduling order where we fire all tasks according to the firing rates they have in the original SDFG. We interleave the execution of the tasks and avoid consecutive firing of the same task and, therefore, we consume the available tokens as soon as possible (according to the SDFG semantic), limiting the amount of memory required.

With static non-preemptive scheduling, we identify the schedule length for the application A_i, modeled as a SDFG, as δ_{A_i}. To identify the execution order of the tasks, we consider a single iteration ($iter_i=1$) of the SDFG and we indicate as δ'_{A_i} the corresponding schedule length. The pseudo-code of the implemented scheduling algorithm is available in Algorithm 1. It takes as input the SDFG of A_i and a task clustering solution, Sol_c. We build a schedule assuming that the WCET C_j of each task and the transmission time C_{m_g} of each message are known. If communicating tasks are assigned to the same processing unit, the message between them is ignored as the communication time is negligible. A task can be fired

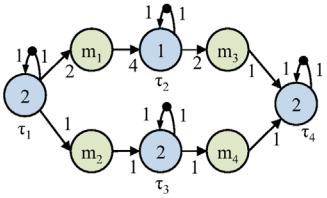
- as soon as there enough tokens on its input edges (as described in the SDFG semantic),
- as soon as the hardware resource to which the task is assigned is available.

The variable *activeTaskList* contains the list of tasks that can be scheduled because their data dependencies are satisfied (source tasks or tasks with enough tokens at their input messages). At each iteration of the algorithm (*while* loop at line 1), we update *activeTaskList*; the head element of the list is the next task to schedule. Function 2 describes the steps for scheduling a single task τ_j. We need to determine the starting time for τ_j (lines 1–4) that depends on the availability of the hardware resource. Then we can calculate the finishing time of τ_j, update the number of tokens produced (lines 6–8) and consumed (lines 9 and 10) and schedule the messages in output of τ_j (lines 11–13). Every time we schedule a task τ_j, we also schedule its output messages if they are connecting τ_j with a task in a different cluster. Once the scheduling of a task is completed, we need to update the *activeTaskList* as specified in Function 3.

Finally, Function 4 is used to determine which is the next task in *activeTaskList* that can be scheduled. For each message we calculate the rate between the number of firing of the target task (task consumer) and the number of firing of the source task (task producer) and we call this *idealRate*. At each iteration of the algorithm, we calculate, for each message, the *realRate* that is the rate between the actual number of times the task consumer and producer have been fired. When the algorithm starts, we initialize the *realRate* variable to ∞. Then we estimate the relative error (*distanceRate*) between the *realRate* and the *idealRate* of each message and we sort the messages in descending order according to their relative error. The sorted order of the messages determines the firing order for the tasks. If the *realRate* for a message m_g is bigger than the *idealRate*, then we should fire the task producer associated to m_g, otherwise the task consumer (the task must be available in the *activeTaskList*).

In Fig. 8c there is an example of eight iterations of Function 4 for the application in Fig. 8a. We reported the steps for the scheduling of one iteration of the SDFG (Fig. 8a). Fig. 8b contains the values of the *idealRate* variable for each message. At each iteration, we fire τ_{next} and we update the values of the *realRate* and *distanceRate*. Then we consider one message at a time, starting with the one with the highest *distanceRate* (when multiple messages m_g have the same *distanceRate*, we prioritize them according to their identifier g). For the selected message, we verify

a



b

idealRate			
m_1	m_2	m_3	m_4
0.5	1	2	1

c

Time	#firing				realRate				distanceRate (1-(Real rate/Ideal rate))				activeTask List	(realRate > idealRate)? source:target				Next task
	τ_1	τ_2	τ_3	τ_4	m_1	m_2	m_3	m_4	m_1	m_2	m_3	m_4		m_1	m_2	m_3	m_4	τ_{next}
1	0	0	0	0	∞	∞	∞	∞	-∞	-∞	-∞	-∞	τ_1	τ_1	τ_1	τ_2	τ_3	τ_1
2	1	0	0	0	0.0	0.0	∞	∞	1	1	-∞	-∞	τ_1, τ_3	τ_2	τ_3	τ_2	τ_3	τ_3
3	1	0	1	0	0.0	1.0	∞	0.0	1	0	-∞	1	τ_1	τ_2	τ_1	τ_2	τ_4	τ_1
4	2	0	1	0	0.0	0.5	∞	0.0	1	0.5	-∞	1	τ_2, τ_3	τ_2	τ_3	τ_2	τ_4	τ_2
5	2	1	1	0	0.5	0.5	0.0	0.0	0	0.5	1	1	τ_3, τ_4	τ_2	τ_3	τ_4	τ_4	τ_4
6	2	1	1	1	0.5	0.5	1.0	1.0	0	0.5	0.5	0	τ_3	τ_2	τ_3	τ_2	τ_3	τ_3
7	2	1	2	1	0.5	1.0	1.0	0.5	0	0	0.5	0.5	τ_4	τ_2	τ_1	τ_2	τ_4	τ_4
8	2	1	2	2	0.5	1.0	2.0	1.0	0	0	0	0	-					

Fig. 8. (a) Input SDFG, (b) idealRate for each message of the SDFG, (c) results of the first eight iteration of Function 4.

if we should fire its source or target task; for clarity, in Fig. 8c (columns 15–18), we reported the task that we should fire according to the *realRate* and *idealRate* of each message. The selected source/target task must be available in the *activeTaskList*, otherwise we select the message with the next highest *distanceRate* until it is possible to assign the variable τ_{next}. If no task can be assigned according to this policy, we assign to τ_{next}, the last task that has been added to *activeTaskList*. As an example, let us consider the row corresponding to *Time*=4 in Fig. 8c: we calculate the *realRate* and *differenceRate* for each message. Then, we sort the messages according to their *differenceRate* in descending order, obtaining the following sorted list: m_1, m_4, m_2 and m_3. We start considering the first message in this sorted list: m_1. Afterwards, we determine if we should fire the source or target task of m_1 (column 15 in Fig. 8c). As the *realRate* of m_1 is smaller than its *idealRate* (0 and 0.5, respectively), we select the target task (i.e., τ_2). As the last step, we verify if τ_2 is in the *activeTaskList* (column 14); if this is the case, we can set $\tau_{next}=\tau_2$, otherwise, we proceed with the next message in the sorted list (i.e., m_4), until we find a task that can be fired. Using Function 4, we guarantee that during the schedulability analysis we fire the tasks in a order that can be easily reproduced on the Silicon Hive's ASIPs.

Algorithm 1. Scheduling algorithm, δ_{A_i} for Sol_c.

1: *activeTaskList*: source tasks in A_i
2: **WHILE** (*activeTaskList NOT EMPTY*) **do**
3: τ_j:=*activeTaskList.head*
4: $\{t_j^f, S_{PE_k}^s, S_{PE_k}^f, S_{CE}^s, S_{CE}^f\}$:= Schedule τ_j (Function 2)
5: Update *activeTaskList* (Function 3)
6: Select from *activeTaskList* the next task τ_{next} to be scheduled (Function 4)
7: Set τ_{next} as *activeTaskList.head*
8: **end while**
9: δ'_{A_i}: Find the maximum t_j^f of the sink tasks

Function 2. Schedule τ_j.

1: *dataDependencyList*:=Find data dependencies of τ_j
2: $S_{PE_k}^f$:=Find last time the resource PE_k has been used (τ_j assigned to PE_k)
3: t_{dep}^f:=Find the maximum of t_{j1}^f for each τ_{j1} in *dataDependencyList*
4: t_j^s:=Find the maximum between t_{dep}^s and $S_{PE_k}^f$
5: t_j^f: Calculate the finishing time
6: **if** first time PE_k used **then**
7: $S_{PE_k}^s := t_j^s$
8: **end if**
9: $S_{PE_k}^f := t_j^f$
10: **fOR** EACH of the input messages m_g of τ_j **do**
11: Update the number of token available on m_g (token consumed)
12: **end for**
13: **for** EACH of the output messages m_g of τ_j **do**
14: Update the number of token available on m_g (token produced)
15: **if** m_g connect τ_j to a task in a different cluster **then**
16: Schedule m_g on the bus *CE*
17: **if** first time *CE* used **then**
18: $S_{CE}^s := t_{m_g}^s$
19: **end if**
20: $S_{CE}^f := t_{m_g}^f$
21: **end if**
22: **end for**

Function 3. Update *activeTaskList*.

1: **for** EACH of the input messages m_g of τ_j **do**
2: **if** NOT enough token in input **then**
3: Remove τ_j from *activeTaskList*
4: **end if**
5: **end for**
6: **for** EACH of the output messages m_g of τ_j **do**
7: Identify target task τ_{j1} (connected to m_g)
8: **for** ALL input messages m_{g1} of τ_{j1} **do**
9: **if** enough token in input **then**
10: Add τ_{j1} to *activeTaskList*
11: **end if**
12: **end for**
13: **end for**

Function 4. Select from *activeTaskList*, the next task τ_{next} to be scheduled.

1: **if** First invocation of Function 4 **then**
2: **for** EACH of the messages m_g of A_i **do**
3: idealRate[m_g]:= Calculate (Total number of firing of target task of m_g)/(Total number of firing of source task of m_g)
4: **end for**
5: **end if**
6: **for** EACH of the messages m_g of A_i **do**
7: realRate[m_g]:= Calculate (actual number of firing of target task of m_g)/(actual number of firing of source task of m_g)

8: distanceRate[m_g]:= Calculate relative error between realRate[m_g] and idealRate[m_g]
9: **end for**
10: *sortedMessagesList*:=Sort messages in descending order based on the distanceRate
11: **for** EACH of the messages m_g in *sortedMessagesList* **do**
12: **if** realRate[m_g] > idealRate[m_g] **then**
13: τ_{temp}:=source task of m_g
14: **else**
15: τ_{temp}:=target task of m_g
16: **end if**
17: **if** τ_{temp} is in activeTaskList **then**
18: $\tau_{next}=\tau_{temp}$
19: exit
20: **end if**
21: **end for**
22: τ_{next}:=last task that added to *activeTaskList*

4.2. Schedulability analysis

Our schedulability analysis determines the likelihood of a clustering solution to be schedulable once the corresponding multi-ASIP platform is implemented. With our UM, the WCET of each task is modeled with a stochastic variable; therefore, we cannot calculate the schedule table and perform a schedulability check. Instead, we can perform a schedulability analysis to determine the probability of having a schedule length δ_{A_i} that meets the deadline. Thus, the probability that an application A_i would meet the deadline d_i^{iter} is defined as the *schedulability probability* $p=P(\delta_{A_i}\le d_i^{iter})$.

The schedulability analysis using stochastic variables is done combining the scheduling algorithm presented in Section 4.1 and Monte Carlo simulation (*MCS*). We use *MCS* to extract n random samples of C_j for each task τ_j according to the normal distribution of the WCET. For the messages, we have arrays that contain n equal values, i.e., the transmission time C_{m_g} associated to the message. This means that for evaluating a task clustering solution, we run n times the scheduling algorithm and we collect the results of the iterations to obtain the schedulability probability.

Note that even if we use the same naming convention for the stochastic variable C_j and for the array of n elements that are modeling the probability distribution of the variable, we use the boldface formatting to indicate the n-element array (the same convention applies to the WCET of a message and its corresponding n-element array).

The operations needed by our analysis are +(sum), − (subtract), ∗ (multiply) and **max** (maximum element selection). Each operation is performed element by element on arrays of n samples.

In our analysis we consider both task level and pipeline parallelism at macro-architecture level (among different ASIPs and interconnections). This analysis is executed during DSE and we want to reduce the computation time for the evaluation of a single clustering solution (as Monte Carlo simulation is already time consuming). Therefore instead of estimating the *schedulability probability* for a SDFG A_i that is periodically repeated $iter_i$ times (as explained in Section 2.1), we perform two separated analysis. First, we perform a schedulability analysis of A_i for a single iteration ($iter_i=1$) that we indicate as δ'_{A_i}. Second, we use the output of this first analysis and the pipeline properties to estimate the schedulability probability for the required number of iterations (e.g $iter_i=5$) to obtain δ_{A_i}. We call these two analysis Task-Level Analysis (*TLA*) and Pipeline Analysis (*PA*). The *TLA* uses the scheduling algorithm defined in Section 4.1.

In the following subsections, we use the application in Fig. 9a as an illustrative example of our schedulability analysis with stochastic variables. For simplicity we assume that each task produces and consumes a single token and that it is fired a single time. Additionally we set $iter=3$. We cluster the application in Fig. 9a on a two-ASIP platform according to the task clustering in Fig. 9b where tasks are assigned to PE_1 and PE_2 and the messages are assigned to the bus (indicated as a communication element, *CE*).

4.2.1. Example of task-level analysis

Fig. 10 shows the TLA for the example in Fig. 9. We perform a schedulability analysis of a single iteration of the entire SDFG. For each firing of a task or execution of a message we calculate its starting and finishing time. For a task τ_j, we indicate the starting time as $\boldsymbol{t_j^s}$ and the finishing time as $\boldsymbol{t_j^f}$. For a message m_g, we use the symbols $\boldsymbol{t_{m_g}^s}$ and $\boldsymbol{t_{m_g}^f}$. Each starting and finishing time is a stochastic variable and is represented by an array of n samples. The maximum finishing time of all sink tasks is the output of the TLA, δ'_{A_i}. When multiple tasks and messages are assigned to the same resource and are ready for execution, their scheduling order is determined by our scheduling algorithm presented in Section 4.1 (Function 4). For this analysis we use the operators **max** and + (described in Section 4.2).

When it starts, the TLA identifies the tasks that can be fired (i.e., source tasks and *transformer* tasks that have enough tokens at their input edges). For the example in Fig. 10, we assume that the WCET of source and sink tasks is equal to zero and we also assume zero bits of data for the m_{so} and m_{si}. Note that the designer can include the source and sink tasks in the schedulability analysis specifying an input option to the DSE tool.

We schedule the source task (τ_{so}) and the output message (m_{so}) and we set their starting and finish times to zeros. Then we can schedule τ_1: its starting time $\boldsymbol{t_1^s}$ is given by the maximum of the finishing times (we apply the **max** operator) of all the tasks and messages that τ_1 depends on, in this case only m_{so}. The finishing time $\boldsymbol{t_1^f}$ of τ_1 is given by the sum of the estimated starting time and the n samples extracted by the WCET probability distribution of task τ_1, i.e., $\boldsymbol{t_1^s}+\mathbf{C}_1$ (+ operation). We can then schedule the remaining tasks following the same rules. All steps are described in Fig. 10. The finishing time of task τ_{si} is a set of n samples that

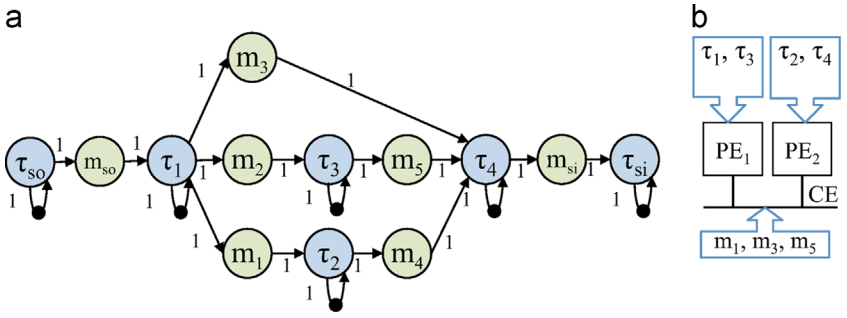


Fig. 9. Example of application and clustering for δ_{A_i} calculation.

corresponds to δ'_{A_i}. The algorithm and functions used for TLA are described in Section 4.1 (Functions 2–4).

During the TLA, we also collect additional information that is needed by the PA. To estimate the pipeline parallelism at macro-architecture level, we need to identify the size of the pipeline stages. In our case, each pipeline stage S corresponds to an hardware resource, i.e., *PEs* and *CE*. We use S_{PE_k} (or S_{CE}) to indicate the size of a pipeline stage: these are also stochastic variables and their probability distribution is modeled by a set of n samples (obtained through *MCS*). We estimate the size of the pipeline stage S_{PE_k} as described in Eq. (1), where $\boldsymbol{S}^{s}_{PE_k}$ and $\boldsymbol{S}^{f}_{PE_k}$ represent the probability distribution of the first and last time the resource PE_k has been used during the TLA. We use the $-$ operator between the *MCS* n-element arrays. More precisely, $\boldsymbol{S}^{s}_{PE_k}$ is equal to the starting time of the first task scheduled on PE_k (first firing of the task), while $\boldsymbol{S}^{f}_{PE_k}$ is equal to the finishing time of the last task scheduled on PE_k (last firing of the task). For example, for PE_2 in Fig. 11a and b we have $\boldsymbol{S}^{s}_{PE_2} = \boldsymbol{t}^{s_2}$ and $\boldsymbol{S}^{f}_{PE_2} = \boldsymbol{t}^{f}_{4}$. The same approach is used for the messages to evaluate S_{CE}

$$\boldsymbol{S}_{PE_k} = \boldsymbol{S}^{f}_{PE_k} - \boldsymbol{S}^{s}_{PE_k} \tag{1}$$

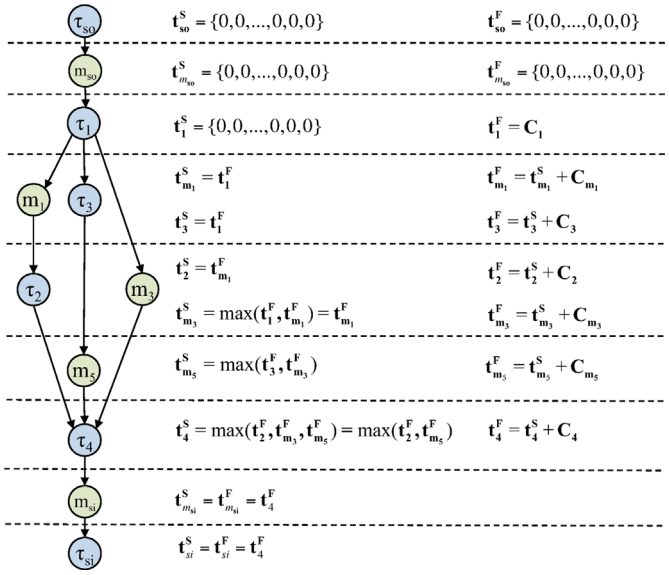


Fig. 10. Calculation of the δ_{A_i} for the example in Fig. 9.

4.2.2. Example of pipeline analysis

In general, the pipelined execution of N elements can be estimated as the sum of the time required by the first element to go through the entire pipeline plus the time required by the other $(N-1)$ elements to complete their execution when all pipeline stages are fulfilled. When the stages of the pipeline have different sizes, the time needed for the elaboration of one of the $(N-1)$ elements corresponds to the stage of the pipeline with the maximum size. We use this definition of pipelined execution to define our PA. The time required by the first element to go through the entire pipeline is the δ'_{A_i} produced by the TLA. From the TLA we also have the probability distributions modeling the WCET of each pipeline stage (S_{PE_k} or S_{CE_k}). Then we can apply Eq. (2) to obtain the set of n samples modeling the biggest pipeline stage $\boldsymbol{S}_{max}$ and Eq. (3) to get δ_{A_i}. In Fig. 11b and c, there is an example of the different pipeline stages and of the computation of the n-element array $\boldsymbol{\delta}_{A_i}$. Having the n-element array, we can calculate the distribution of the n samples and obtain the CDF associated to $\boldsymbol{\delta}_{A_i}$

$$\boldsymbol{S}_{max} = \mathbf{max}(\boldsymbol{S}_{PE_1}, \ldots, \boldsymbol{S}_{PE_{PC_{max}}}) \tag{2}$$

$$\boldsymbol{\delta}_{A_i} = \boldsymbol{\delta}'_{A_i} + \boldsymbol{S}_{max} * (\boldsymbol{iter}_i - \boldsymbol{1}) \tag{3}$$

The PA can be applied to speed up the schedulability analysis *only if there are no data dependencies between successive iterations of the SDFG*. In case of dependencies, it is possible to use TLA to compute the schedulability analysis for all the iterations of the SDFG; the main disadvantage is a higher computation time to perform the schedulability analysis.

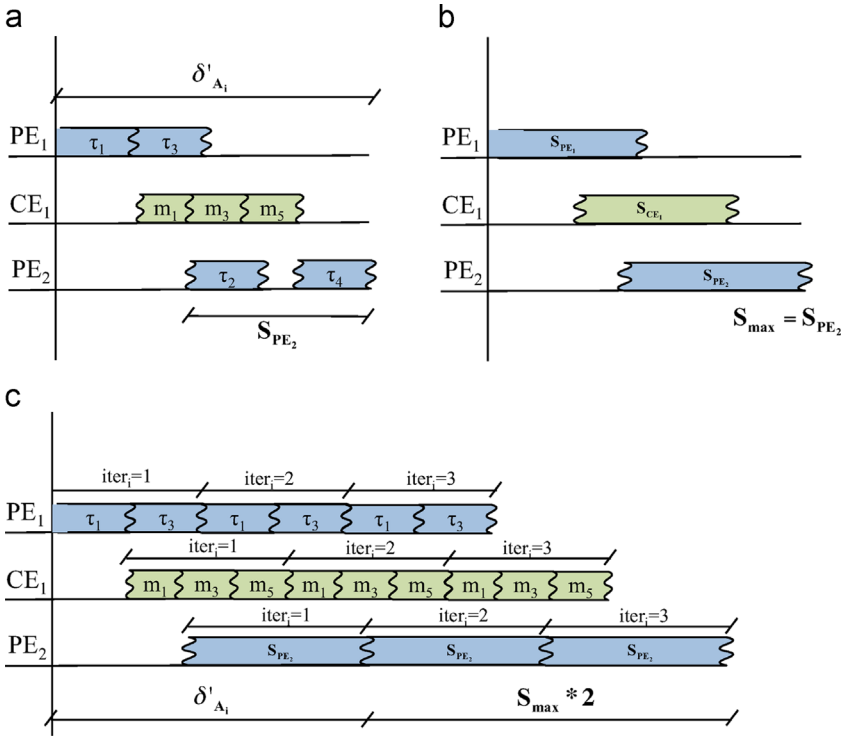


Fig. 11. Calculation of the δ_{A_i} for the example in Fig. 9.

4.2.3. Comparison of clustering solutions

Given the δ_{A_i} of multiple task clustering solutions, we compare them calculating their probability of meeting the deadline d_i^{iter}. Let us consider the task graph in Fig. 4a. The CDFs of the WCET of the tasks are in Fig. 5a. We evaluate two clustering solution: Sol_1 (Fig. 4b) and Sol_2 (Fig. 4c). The resulting $p = P(\delta_{A_i} \leq d_i^{iter})$ are shown in Fig. 5b: Sol_1 has a probability p_i of 0.82 while Sol_2 of 0.03. This indicates that the first clustering solution is much more likely to meet the application deadline $d_i^{iter} = 65$ when the platform is synthesized. Additionally, during DSE, we may need to compare clustering solutions with the same probability (e.g. Fig. 5c where both solutions have $p \approx 1$ for a deadline $d_i^{iter} = 80$). To discriminate between these solutions we use the inverse of the CDF, the quantile function $C^{0.5} = P^{-1}(p_{0.5})$: we select the solution Sol_{id} that has the *smallest* WCET $C^{0.5}$ at a probability $p_{0.5} = 0.5$. In the example in Fig. 5c, the value of quantile function at $p_{0.5}$ is 63 time units for Sol_1 and 72 time units for Sol_2. Therefore during our exploration we will select Sol_1.

4.3. Evolutionary algorithm

We use a Steady State Evolutionary Algorithm (SSEA) [42] to decide the clustering of tasks. SSEA takes as input the application (including the uncertainty model), the legacy components and task assignment constraints, and the maximum number of ASIPs allowed, $PC_{\max}$. The algorithm returns one or more task clustering solutions, which maximize the schedulability probability p, under the given cost constraint $PC_{\max}$. When multiple solutions have the same probability we select the one with the smaller quantile function $P^{-1}(p_{0.5})$ as discussed in Section 4.2.3.

SSEA is inspired from the process of natural evolution, where a set of solutions is called a *population* and each solution is encoded using a string called a *chromosome*. The population is evolved by performing recombination and mutation, and the population is replaced with the *offspring* population, which has better *fitness* according to the cost function. SSEA has been chosen because it is suitable for the case when the computation of the cost function is time-consuming due to *MCS* (a small portion of the population is replaced at each new generation). The algorithm works by adding the offspring of the individuals selected from each generation to the pre-existing one, so individuals are retained between generations.

We define the chromosome (a single clustering solution) as an array of tasks and messages; the value of each element (*gene*) represents the identifier of the *PE* or bus on which the tasks and messages are respectively clustered. We assume that a task is always assigned to the same processor for all its firings. We use a two-point crossover operator [42]. The parameters used for the execution of the SSEA are crossover probability P_c, mutation probability P_m and the population size *Pop*. SSEA finishes when a given time-limit has been reached (≤ 30 min). The tuning of these parameters has been done running multiple executions of the algorithm with different synthetic applications. The parameters used for the execution of the SSEA are $P_c = 30\%$, $P_m = 10\%$ and $Pop = 100$. Using *MCS* together with evolutionary algorithms may lead to scalability problems. In order to reduce the impact of *MCS* on the computation of the cost function (i.e. our schedulability analysis), we have moved the generation of the n random samples outside of the DSE loop. In this way, we extract them only once (during the initialization phase of the evolutionary algorithm), then we store them and we re-use them for the evaluation of every design point.

5. Experimental evaluation

5.1. Validation of the normal distribution for UM

In this section we describe how we have validated the proposed WCET uncertainty model for which we use a normal distribution. We have performed two different types of evaluations:

1. We have considered one task at a time and, using an ASIP simulator, we have verified the distribution of the WCETs obtained from running the task on a considerable number (~ 500) of ASIP micro-architecture configurations.
2. We have considered an entire application A_i and we have verified that the normal distribution enables our DSE to find the best task clustering solution. We have substituted the normal with a Gumbel and uniform distributions. We have compared the results obtained with a reference solution found by a DSE in which the WCET of each task is a deterministic and well-known value.

In the next subsections, we present the details of evaluations 1 and 2.

5.1.1. Evaluation 1

We use this evaluation to verify the probability distribution of a single task. We have applied this evaluation to two tasks, τ_{jpeg} and τ_{mp3} of different sizes and complexities. τ_{jpeg} contains the sequential code of a JPEG decoder [43], while τ_{mp3} contains the sequential code of a MP3 decoder, part of the MAD library [44].

We are interested in determining how the WCET of these tasks varies depending on the micro-architecture features, and if our WCET uncertainty model proposed in Section 2.4 is able to capture this variation. We have run these tasks on a VLIW architecture similar to the ASIP architectures considered in this paper. We have used the VLIW Example (VEX) [45], which is a VLIW compiler and simulator developed at HP Laboratories. VEX is highly configurable; we have used a set of configurations which captures the variability of a micro-architecture design, considering the features of VLIW processors available on the market (e.g., [46]) and the characteristics of the tasks considered. Table 2 presents the micro-architecture design space used for the experiments. Thus, we have varied the number of arithmetic and logic units (ALU), multipliers (MUL), registers in the register file (RF), the issue, load and store slots, the data cache size and the data cache line size. For each micro-architecture configuration, VEX performs the compilation of the C code of the task, simulates its execution and returns the number of execution cycles.

Using the parameters in Table 2 we have evaluated a large number of micro-architecture configurations. In total, we have simulated 490 micro-architecture configurations for the MP3 decoder task and 560 for the JPEG decoder task.

Table 2
Micro-architecture features explored.

Task	*Issue width*	*num. ALU*	*num. MUL*	*RF size*	*Load slot*	*Store slot*
MP3 decoder	1, 2, 3, 4, 5, 6, 7, 8	4, 5, 6, 7, 8	2, 3, 4, 5, 6, 7, 8	32, 64	4	2
JPEG decoder	2, 3, 4, 5, 6, 7, 8	4, 5, 6, 7, 8	1, 2, 3, 4, 5, 6, 7, 8	32, 64	4	2

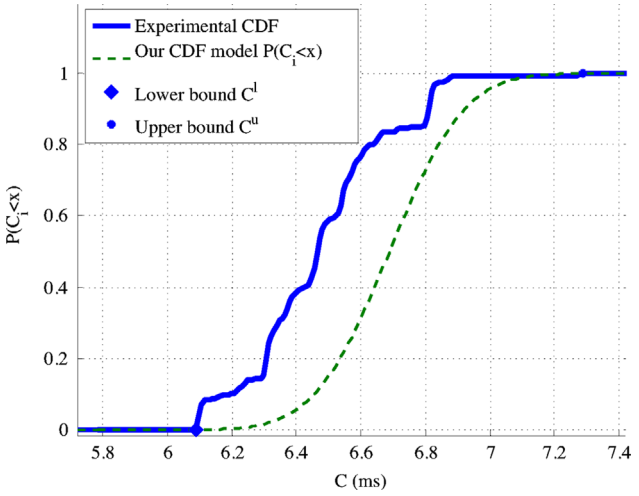


Fig. 12. Comparison of our proposed CDF ($P(C_i < x)$) with the simulation results obtained with VEX for MP3 decoder task. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

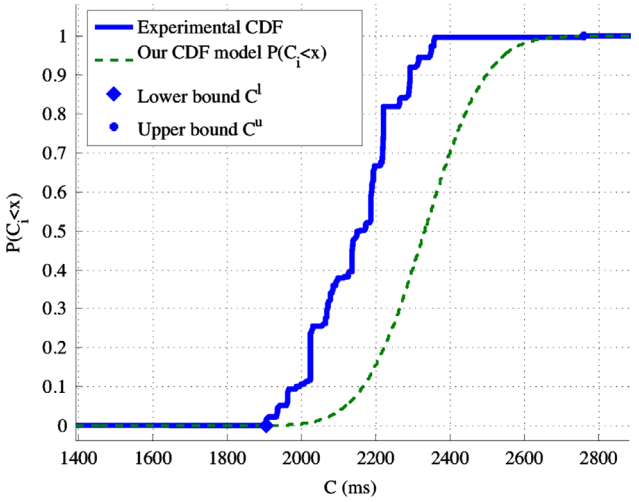


Fig. 13. Comparison of our proposed CDF ($P(C_i < x)$) with the simulation results obtained with VEX for JPEG decoder task. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

For each micro-architecture configuration, we have compiled and run the task. We have used the m3explorer tool [47] for performing DSE in an automatic way; m3explorer is a generic tool for DSE that can be interfaced to any simulation/evaluation tool using XML files. We have used the tool to perform an exhaustive DSE. Using scripting languages, we have created the interfaces between VEX and m3explorer: the scripts automatically generate the micro-architecture configuration file and collect the results (number of cycles) produced by VEX. For each micro-architecture configuration explored, we assume a frequency of 100 MHz to calculate the execution time in ms (given the characteristics of the micro-architectures explored, we can safely assume a frequency of 100 MHz by comparison with other commercial VLIW processors, e.g., [46]). For a particular micro-architecture, after simulations with multiple input files, we have considered as WCET the largest value of the execution time. We know that such a value does not represent the WCET, which is a theoretical upper bound determined through analysis, but we believe this value is a good approximation for our model validation experiments.

The results for the MP3 decoder are presented in Fig. 12 and those for JPEG in Fig. 13.

Table 3
Microarchitectures associated with the WCET upper and lower bounds.

Task	*WCET*	*Issue width*	*num. ALU*	*num. MUL*	*RF size*	*Load slot*	*Store slot*
MP3 decoder	C_l	5	8	5	64	4	2
	C_u	1	4	2	64	4	2
JPEG decoder	C_l	7	5	3	64	4	2
	C_u	2	8	6	64	4	2

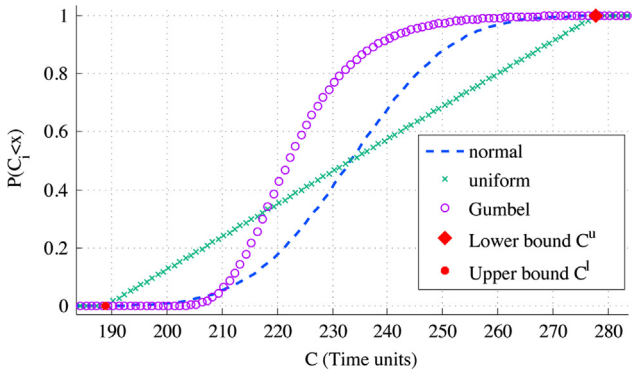


Fig. 14. Cumulative distribution function (CDF).

Each figure shows two CDF curves: the CDF resulted after experiments (depicted with a continuous blue line) and the CDF obtained by using our model (the green dotted line). Our WCET model (the green dotted CDF) is obtained as explained in Section 2.4, considering a normal distribution between a lower bound C^l and an upper bound C^u of the WCET (we took the fastest and lowest micro-architecture configurations). The micro-architectures, corresponding to the upper and lower bounds of the WCET for the two tasks, are summarized in Table 3.

This experiment shows that the WCET of multiple micro-architecture configurations can be modeled as a normal distribution and that our proposed uncertainty model is a valid and safe approximation. Note that the CDF of our model leads to more pessimistic (larger) WCETs compared to experimental measurements. This is acceptable as the WCETs produced by our experiments might be optimistic (smaller), since they are not a theoretical upper bound obtained through analysis. It is important to mention that the proposed WCET uncertainty model is used only for design space exploration, and not for providing timing guarantees in the final implementation.

5.1.2. Evaluation 2

With this evaluation, we want to verify if a WCET model based on the normal distribution can guide our DSE and find the best clustering solution when compared with other probability distributions and with a *deterministic* DSE in which we know the exact value of the WCET of each task (i.e., there is no probability distribution associated with it).

We consider three probability distributions: normal (N), Gumbel (G) and uniform (U). We select the Gumbel distribution as it is commonly used to model the WCET [48,49] and the uniform distribution for its simplicity. In Fig. 14 there is an example of the CDF for normal, Gumbel and uniform distributions.

Given an application A_i, we have modeled the WCET of its tasks using the three distribution types (we have available the $[C_j^l, C_j^u]$ of each task τ_j) and we have run our DSE. Additionally, we have run a

deterministic DSE: having well-known values for the WCET of the tasks, we can calculate the exact scheduling length.

DSE_{UM}^{type} indicates our DSE with UM where *type* is the distribution used ($type = \{N, G, U\}$); DSE_{det} indicates the DSE using deterministic WCETs. We want to compare the task clustering solutions obtained with DSE_{UM}^{type} and DSE_{det}; therefore we need to guarantee that the two DSE are comparable. We need to run the DSE_{det} multiple times using different WCETs for each tasks and we need to use the same optimization function for both DSE. In the deterministic DSE the optimization function is the minimization of the scheduling length. Instead, in our DSE with UM, the optimization function is the maximization of the probability of meeting the application deadline. We have modified our DSE_{UM}^{type}, so that the exploration is guided by the minimization of the scheduling length at different probabilities. We use the inverse of the CDF, i.e. the quantile function $C_i = F^{-1}(p)$, to obtain the C_i of a clustering solution at a specific probability p_i, where $p_i \in P_i = \{0.02, 0.50, 0.98\}$. We have selected three different probabilities to take the shape of the different CDFs into account and not to favor any distribution types.

For each pair $\{p_i, type\}$ we have run the DSE_{UM}^{type} for a total of nine times. For each execution we have obtained a different task clustering solution.

Then we have run the DSE_{det} multiple times: for each execution of the DSE_{det} and for each task τ_j in the application A_i, we assign a deterministic WCET randomly extracted from the range $[C_j^l, C_j^u]$. We assume that this WCET corresponds to a specific ASIP micro-architecture configuration. For each execution of the DSE_{det}, we have saved the *set* of WCETs used. We have run the DSE_{det} 5000 times.

Once we have collected all the results from the nine executions of the DSE_{UM}^{type} and of the 5000 executions of the DSE_{det}, we need to compare them. The schedule length obtained with DSE_{det} represents the optimal scheduling that we can obtain when knowing the exact values of the WCET for each task. We take the nine clustering solutions produced by the DSE_{UM}^{type} and we calculate the scheduling length of each of them using the 5000 *sets* of WCET generated during the DSE_{det}. This means that we obtained 5000 scheduling lengths for each of the nine task clustering solutions.

Then we have compared the scheduling length of the 5000 clustering solutions found through DSE_{det} with the scheduling length of the clustering solutions found with DSE_{UM}^{type}, which we have evaluated with the same *sets* of WCET values. This comparison is used to identify which probability distribution type allows finding a task clustering solutions with the closest scheduling length to the one found with DSE_{det}.

We have run this evaluation on six synthetic case studies, which characteristics are specified in Table 4. In Table 5, for each case study, and for each pair $\{p_i, type\}$ we have the *average relative error* in the scheduling length obtained with the DSE_{UM}^{type} when compared to the DSE_{det}. The *average relative error* is calculated as follows. Let us consider the task clustering obtained using a normal distribution (DSE_{UM}^{N}) at $p_i = 0.50$. We have evaluated this task clustering solution with one of the *sets* of WCETs generated during the DSE_{det} and we have obtained its scheduling length. We have compared it with the scheduling length obtained by the DSE_{det} when using the same *set* of WCETs and we have calculated the relative error among them. We have repeated this comparison for each of the 5000 *sets* of WCETs. The average of these relative errors returns the *average relative error*.

Table 4
Case studies for the comparison of CDF types.

Case study ID	1	2	3	4	5	6
No. of apps.	4	5	18	10	39	48
No. of tasks	14	15	24	26	44	60
No of ASIPs	11	14	14	11	10	13

Table 5
Results of the comparison of CDF types (% average relative error).

Case study ID	{**0.02, N**}	{**0.50, N**}	{**0.98, N**}	{**0.02, G**}	{**0.50, G**}	{**0.98, G**}	{**0.02, U**}	{**0.50, U**}	{**0.98, U**}
1	0.02	0.02	0.02	0.02	0.02	0.02	0.04	0.05	0.04
2	0.06	0.13	0.06	0.11	0.04	0.07	0.11	0.11	0.11
3	0.16	0.05	0.05	0.18	0.05	0.05	0.08	0.16	0.98
4	0.05	0.06	0.06	0.08	0.22	0.05	0.07	0.06	0.15
5	0.07	0.09	0.11	0.10	0.09	0.13	0.12	0.13	0.12
6	0.06	0.05	0.06	0.05	0.06	0.05	0.06	0.06	0.08

We can observe that the uniform distribution is the one with the bigger errors. Even if normal and Gumbel distributions return comparable errors for some case studies, the first one is a better fit for most of them. Figs. 16 and 17 represent the distributions of the relative error, i.e. percentage difference in the scheduling length for case studies 1 and 4 (Table 4). For each case study, the normal, Gumbel and uniform distributions are represented. For each distribution, we have grouped together the results obtained for the different p_i, for a total of 15,000 evaluated schedulings. The normal distribution is the one that returns schedulings with length closer to the DSE_{det}. In fact for all case studies is the one with the highest number of scheduling length difference equal zero. For example, let us observe Fig. 16. The histogram containing the results obtained with a normal distribution has most of the schedulings ($\sim$8000) with 0% error when compared to the schedulings obtained with DSE_{det}. This value decreases to $\sim$3000 for Gumbel and $\sim$1800 for the uniform distributions. Depending on the case studies, the distribution of the errors can vary: we observed that for the case studies with a higher number of tasks, the errors are centered around zero, but are distributed on a wider range (see Fig. 17). However, in all cases, we verified that the normal distribution produces scheduling lengths closer to the one obtained using a deterministic DSE.

5.2. Evaluation of the DSE with UM

To evaluate our approach for multi-ASIP platform synthesis, we have used three real case studies taken from the multimedia and medical domains: the motion JPEG (MJPEG) encoder [50], the Spatial Coding (SC) algorithm from MPEG4 encoder (property code of STMicroelectronics [51]) and the Electrocardiogram (ECG) applications (code provided by [52]).

We have applied the design flow[6] shown in Fig. 15 to the three applications: starting from the C code we have implemented a multi-ASIP system using our DSE and *UM* to determine the task clustering and the number of ASIP to use.

The design flow requires as inputs:

- the sequential C code of the application A_i,
- the deadline d_i^{iter} of the application,
- the desired working frequency f for the multi-ASIP system,
- initial platform description with the corresponding platform cost ($PC_{\max}$) and bus types that we want to explore.

For our case studies, we are limiting the interconnection exploration to a single bus type, b_{32}^f (i.e. a 32 bit width bus with the same frequency f of the multi-ASIP system) as that is the one that we

[6] This design flow has been proposed in the European research project ASAM [53] for the automation of the design and the construction of ASIP-based MPSoCs

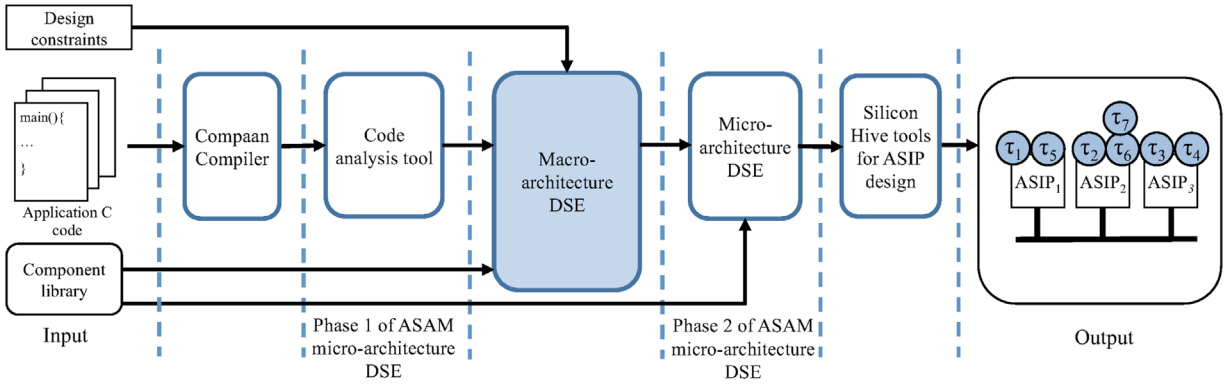


Fig. 15. Semi-automatic design flow for multi-ASIP synthesis.

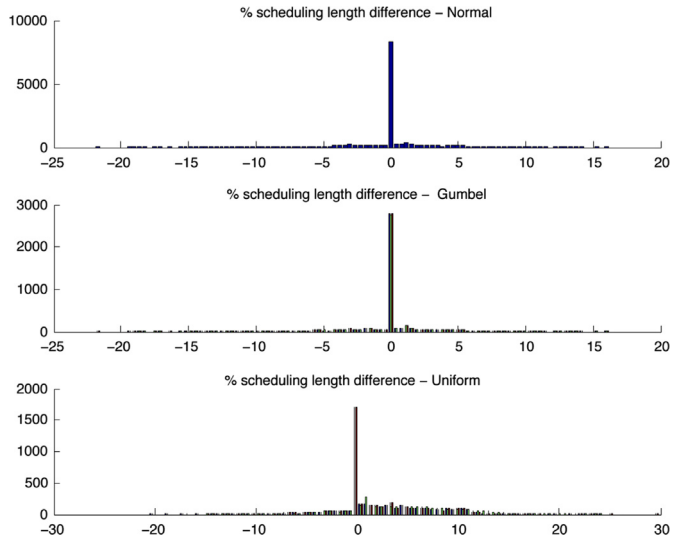


Fig. 16. Histogram of the percentage (%) differences in the scheduling length for case study 1.

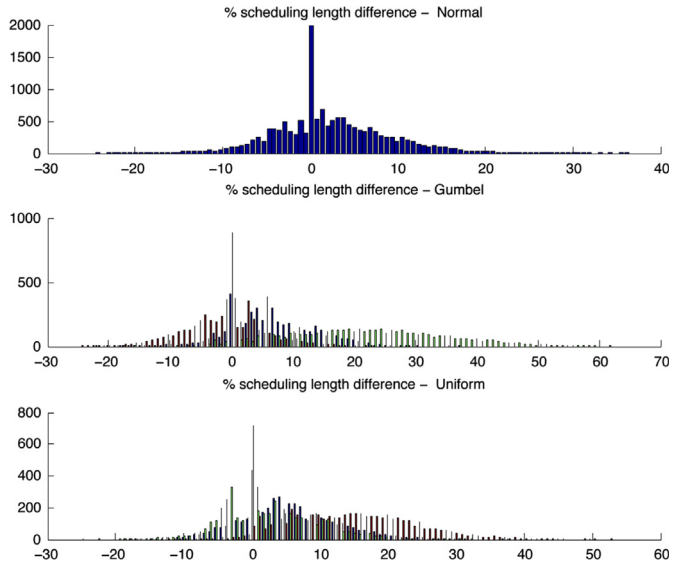


Fig. 17. Histogram of the percentage (%) differences in the scheduling length for case study 4.

have available during platform synthesis.[7] The input constraints for each of the analyzed case studies are summarized in Table 6.

In the design flow, we use the support of external tools. We use Compaan tool [54] for the partition of the application into tasks. Compaan elaborates sequential C code and builds the corresponding Kahn Process Network (KPN). From Compaan KPN model of the application, we build the corresponding SDFG.

Then we use the code analysis tool described in the ASIP DSE (Phase 1 of ASAM micro-architecture DSE) of [30] to determine the upper and lower bounds (C^l and C^u) for each of the task of the application. The code analysis tool profiles the application code (using LLVM compiler [55]) and, for each task, it estimates the number of cycles required by a sequential execution (C^u) and by a parallelized execution (C^l) of the code. As mentioned in Section 2.1, the source and sink actors of each applications are used for data initialization (i.e. for writing the input data into a local or external memory of the multi-ASIP platform that we want to design), and for providing feedback to the user about the completion and exit status of the application. For this reason, we consider their execution time equal to zero and they will not be mapped to any ASIPs.

Then we can build the CDF for each task using the estimated C^l and C^u and the input frequency f, and execute our macro-architecture DSE to identify the task clustering solution with higher chances of being schedulable after synthesis. For each cluster of tasks found, we invoke the micro-architecture DSE (Phase 2 of ASAM micro-architecture DSE [30]). It defines a single ASIP given as input a task cluster and a library of predefined *ISs* built using Silicon Hive tools. Silicon Hive ASIPs are single-threaded VLIWs that are configurable depending on the functionalities required by the applications. An ASIP is composed of one or more *ISs*. The micro-architecture DSE uses a library of predefined *ISs*: they contain a *RF*, multiple functional units and a data memory. Moreover there is a default *IS* that is always included and contains the program counter, the instruction memory, a default data memory (also used as stack memory) and a fixed number of FIFO ports. After the definition of the ASIP micro-architecture, we use the Silicon Hive tools to build a retargetable compiler for the ASIPs and a cycle-accurate simulation environment for the estimation of the execution of the application C code

Table 6
Input constraints for MJPEG encoder, ECG and SC.

Case study	*d* (*μs*)	*f* (*MHz*)	PC_{max}	*Bus type*
MJPEG encoder	500,000	166	3	b_{32}^{166}
ECG	16,000,000	1	2	b_{32}^{1}
SC	205,000	1600	3	b_{32}^{1600}

[7] This limitation does not derive from Silicon Hive's tools, but from our definition of the ASIP micro-architectures.

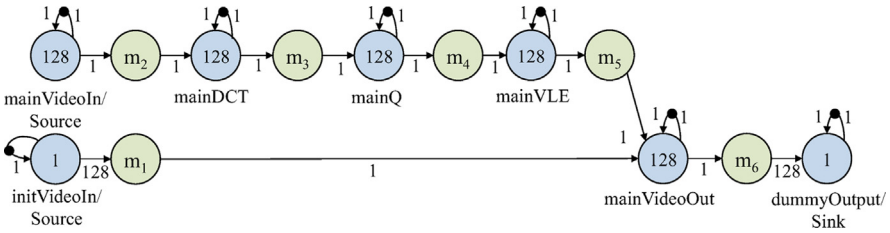


Fig. 18. SDFG model for MJPEG encoder.

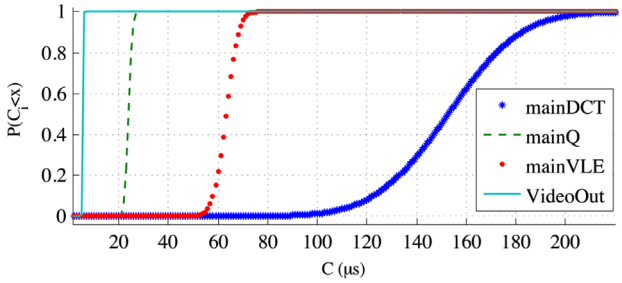


Fig. 19. Cumulative distribution functions for the tasks of the MJPEG encoder application (with $f=166$ MHz).

on the multi-ASIP platform. Silicon Hive tools allow the definition of a multi-ASIP platform in which the ASIPs are connected among them and to external memories through a hierarchy of buses.

In the following sections we analyze our three case studies.

5.2.1. MJPEG encoder

In the second row of Table 6, there are the input constraints for the MJPEG encoder application: we consider the elaboration of 15 frames and a desired throughput of 30 frames per second (fps) that give a deadline d^{15}_{MJPEG} of 0.5 s.

The SDFG of the MJPEG encoder obtained starting from the Compaan KPN model is depicted in Fig. 18.

We have used the code analysis tool described in the ASIP DSE (Phase 1) of [30] to determine the upper and lower bounds (C^l and C^u) of the tasks of the application. The code analysis tool does not return a theoretical estimation of the WCET. It returns an estimated number of cycles of a profiled execution of the MJPEG encoder. Therefore, we have run the tool with multiple input data (frames with same size, but different content) and we took the ones producing the highest estimated execution time. We have used this input to set the upper and lower bounds. We consider that these values are good enough to verify our *UM* as the variability in the estimated execution time given by the different input data is 2%.

We have used the estimated C^l and C^u and the input frequency $f=166$ MHz to build the CDFs (Fig. 19). Looking at the CDFs, it is possible to identify the most consuming tasks and how the WCET of each task varies depending on the exploited instruction level parallelism. The amount of data expressed in bits of each message is shown in Table 7. Wehave calculated the transmission time of the messages on the bus considering the bus b^f_{32}.

Then we can execute our macro-architecture DSE that performs the schedulability analysis of different clustering solutions. We have run the SSEA for 200 s and we have used $n=5000$ for the Monte Carlo simulation. We indicate with Sol_{DSE} the task clustering solution found by our DSE with UM, which is available in the second row of Table 8. Sol_{DSE} has a $p\sim 1$ (where $p=P(\delta_{A_{MJPEG}} < d^{15}_{MJPEG})$) to meet the deadline and uses two ASIPs. In the second row of Table 8 (columns 2–4) there are the task clustering solution, the probability of the application to meet the deadline and the quantile function value at a probability of 0.5 ($C^{0.5}=P^{-1}(p_{0.5})$).

Table 7
Message sizes (in bits) for MJPEG encoder

m_1	m_2	m_3	m_4	m_5	m_6
128	8192	8192	8192	4096	32

To verify our result, we have used the micro-architecture synthesis tool described in [30] and we have obtained a description of the micro-architecture of the two ASIPs, each of them with three *IS*s (including the default one).

Then with Silicon Hive tools, wehave implemented the two-ASIP platform and we have mapped the application code to the ASIPs as suggested by our *UM*.

Moreover, we have added two FIFOs between each couple of ASIPs that needs to exchange data. The FIFOs are used only for synchronization purposes while the data are transferred on the bus. We have also adjusted our algorithm for the evaluation of a clustering solution to be consistent with Silicon Hive simulator as follows. We have added offsets in the schedulability analysis for modeling the time required for starting the execution of the tasks on the ASIPs, for modeling the synchronization time (access to the FIFOs), and also for considering additional bus parameters as the hand-shake time to gain access to the bus and the setup time for transferring the data.

In columns 5 and 6 of Table 8 there are the number of execution cycles obtained from Silicon Hive simulator and the corresponding time in μs (at a frequency $f=166$ MHz). After the synthesis of the multi-ASIP platform, we have also verified that our implementation is schedulable.

It is important to note that our DSE works through comparison: we can evaluate different task clustering solutions and determine which one has the highest chances to produce a schedulable implementation once the final platform is available, but we are not guaranteeing the schedulability of the application. We use our approach in the very early phases of the design when there is no implementation available for the platform and it can help the designer to determine the platform composition and the partitioning of the application. To demonstrate the effectiveness of the result provided by our DSE (Sol_{DSE}), we compare it with other clustering solutions that we have arbitrarily selected: we evaluate them with our schedulability analysis with *UM* and we implement and simulate them with Silicon Hive's tools. The results that we have obtained are shown in Table 8; they confirm that our DSE is able to determine which solution is better than the other. The last column in Table 8 shows that only Sol_{DSE} is schedulable.

We have arbitrarily selected those task clustering solutions with two ASIPs (Sol_1 and Sol_2) that are a fair alternative to the clustering solution found by our DSE. Additionally, we have verified the performance of a solution with a single processor (Sol_4) and the performance that can be achieved using three ASIPs (Sol_3 and Sol_5). We have selected these task clustering solutions considering that *mainDCT* is the task with the highest number of cycles, and, therefore, there should be a dedicated ASIP for its execution. Additionally, it is better to cluster successive tasks on

Table 8
Comparison of clustering solutions for MJPEG encoder.

Sol_{ID} DSE	Clusters			p	$C_{0.5}$ (μs)	*sim* (cycles)	*sim* (μs)	sched
	PE_1	PE_2	PE_3					
	mainDCT	mainQ, mainVLE, mainVideoOut	–	~1	292,700	79,088,561	476,437.11	Yes
1	mainDCT, mainQ	mainVLE, mainVideoOut	–	~1	345,300	85,740,371	516,508.26	No
2	mainDCT, mainQ, mainVLE	mainVideoOut	–	0.85	460,100	124,194,971	748,162.48	No
3	mainDCT	mainQ, mainVLE	mainVideoOut	~1	294,700	83,617,556	503,720.22	No
4	mainDCT, mainQ, mainVLE, mainVideoOut	–	–	0.77	470,800	126,635,428	762,864.02	No
5	mainDCT	mainQ	mainVLE, mainVideoOut	~1	294,700	83,636,411	503,833.80	No

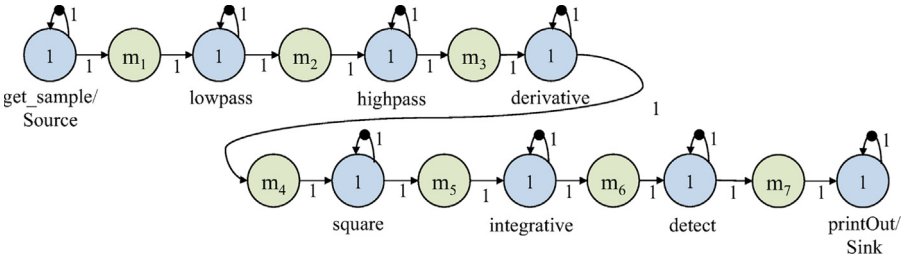


Fig. 20. SDFG model for ECG.

the same ASIP: for example, clustering *mainDCT* and *mainVLE* together (and *mainQ* and *mainVideoOut* together) is inconvenient as it forces multiple exchanges of data between the two ASIPs and produces bigger pipeline stages at platform level (this implies also that the processors will stall waiting for the data). Moreover, we have verified that using an additional processor (Sol_3 and Sol_5), there are no improvements in the performances: with having a dedicated processor for *mainDCT* and splitting *mainVLE*, *mainQ* and *mainVideoOut* into two processors we cannot not speed up the execution. In fact *mainDCT* is still the task defining the speed of the entire system and we have obtained a higher number of cycles due to the additional communication and synchronization time introduced by the extra processor. These results are confirmed by our schedulability analysis and the simulation with Silicon Hive's tools.

When multiple solutions have the same probability, we prefer the one with the smaller number of clusters and the smaller quantile value $C_{0.5}$. Sorting the task clustering solutions (from the best to the worst one) according to our schedulability analysis, we find that their order matches the results obtained with the cycle-accurate simulator from Silicon Hive. This shows that our schedulability analysis with *UM* is able to properly evaluate the different task clustering solutions and find the ones that are more promising for platform synthesis, supporting the designer and speeding up the design process.

5.2.2. ECG case study

For the ECG application we follow the same design steps of the MJPEG encoder case study (Fig. 15). The SDFG of the ECG application is shown in Fig. 20. In Table 6 (second row), there are the input constraints for the ECG case study. The amount of data expressed in bits of each message is shown in Table 9.

We have run our DSE for 200 s and with $n=5000$: the best clustering solution found has a probability $p_{ECG}=0.56$ and uses two ASIPs. The task clustering and its cost are summarized in the second row of Table 10 (Sol_{DSE}). As for the MJPEG encoder case study, we have generated the ASIPs, each of them with three-issue slots. Then we have synthesized the cores and the platform using Silicon Hive's tools and we have run the ECG code obtaining a schedulable solution. Columns 5 and 6 of Table 10 show the number of cycles and the execution time (at a frequency $f=1$ MHz) that we have obtained with Silicon Hive's simulator.

Table 9
Message sizes (in bits) for ECG

m_1	m_2	m_3	m_4	m_5	m_6	m_7
32	32	32	32	32	32	96

In Table 10, there are also the results of the schedulability analysis for other task clustering solutions (Sol_1, Sol_2, Sol_3 and Sol_4) that we have arbitrarily defined (as they are a fair alternative to Sol_{DSE}). We have compared them with Sol_{DSE} and verified that our *DSE* with *UM* is able to identify the best task clustering solution previous the actual synthesis of the multi-ASIP platform. There is a correspondence between the probability of meeting the deadline (and the value of the quantile function) that we have obtained with our schedulability analysis and the actual schedule length that we have obtained from the Silicon Hive's simulator: to a higher probability corresponds a shorter schedule length. Except for Sol_4, all solutions are schedulable.

5.2.3. SC case study

In this section, we present the Spatial Coding application (part of MPEG4) [51]. The SDFG of the SC application is shown in Fig. 21. In Table 6 (third row), there are the input constraints for the SC case study: we have considered the elaboration of 5 frames, each of them composed of 40×30 blocks.[8] As we are considering a throughput of 24 fps, for the SDFG in Fig. 21, we have a deadline $d_{SC}^{6.000} = 205,000$ μs. The size in bits of each message of the SDFG is shown in Table 11. The design space for the SC case study is bigger than the one of the MJPEG and ECG; hence, we need to run our macro-architecture DSE for 1800 s. We have used $n=5000$. Our DSE has found a task clustering solution with $p_{SC}=0.99$ and that

[8] We set the frequency to 1600 MHz to find a schedulable solution after the implementation of the platform; however, we are aware that it is not a realistic frequency and the optimization of the application code and additional processors should be used to achieved the desired performances at a lower frequency.

Table 10
Comparison of clustering solutions for ECG.

Sol_{ID} DSE	Clusters		p	$C^{0.5}$ (μs)	*sim* (cycles)	*sim* (μs)	sched
	PE_1	PE_2					
	Lowpass, highpass, derivative, square	Integral, detect	0.56	15,783,000	13,790,796	13,790,796	Yes
1	Lowpass, highpass, derivative	Square, integral, detect	0.54	15,853,000	14,000,776	14,000,776	Yes
2	Lowpass, highpass	Derivative, square, integral, detect	0.52	15,928,000	14,460,733	14,460,733	Yes
3	Lowpass, highpass, derivative, square, integral	Detect	0.23	16,942,800	15,934,010	15,934,010	Yes
4	Lowpass, highpass, derivative, square, integral, detect	–	0.22	16,991,000	16,692,594	16,692,594	No

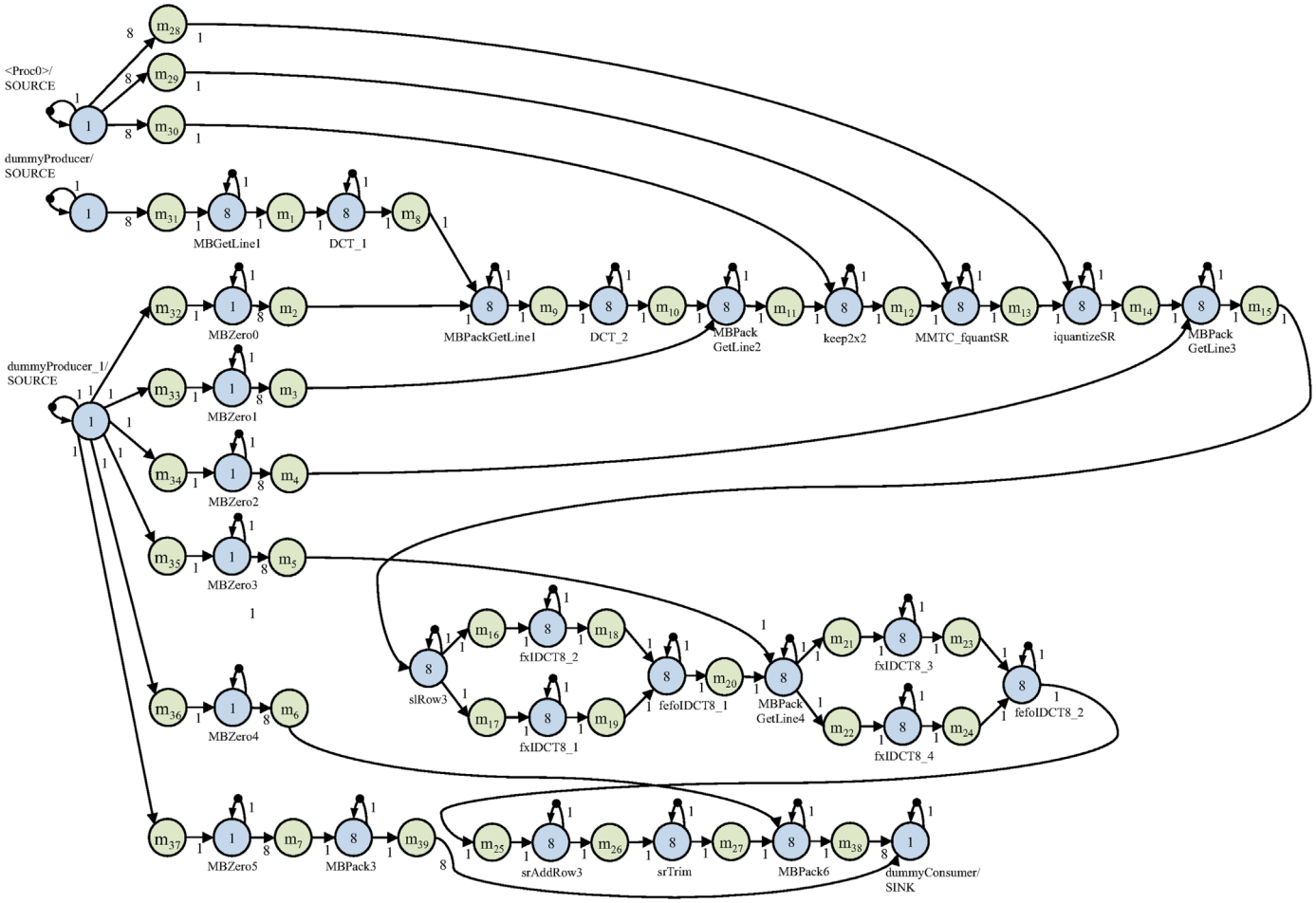


Fig. 21. SDFG model for spatial coding case study.

Table 11
Message sizes (in bits) for SC.

m_2–m_7, m_{31}–m_{39}	m_{28}–m_{30}	m_{18}–m_{19}, m_{23}–m_{24}	m_1, m_8–m_{17}, m_{20}–m_{22}, m_{25}–m_{27}
2048	224	128	256

uses three ASIPs. The details of the solution found (Sol_{DSE}) are summarized in the second row of Table 12. We have implemented Sol_{DSE} using Silicon Hive's tools and we have obtained a schedulable solution that runs for 274,847,324 cycles (171,799.58 μs at $f=1600$ MHz). As for the previous case studies, to demonstrate the validity of the solution found, we have compared it with other task clustering solutions that are summarized in Table 12: Sol_1 has a single ASIP, while Sol_2, Sol_3 and Sol_4 use three ASIPs. Note that depending on the clustering solution, the ASIPs and their interconnections may change. Except for Sol_1 that has a probability ~ 0 to meet the deadline, the other task clustering solutions, once synthesized, provide schedulable implementations. Due to the high number of tasks in the SC application, it is possible to select multiple task clustering solutions to compare with. We have selected Sol_1 in which a single ASIP is used and task level and pipeline parallelism (at system level) cannot be exploited. We have selected Sol_2 and Sol_3 because they do not differ very much from the task clustering found by our DSE: we have evaluated those solutions in which the pipeline parallelism can be conveniently exploited and in which the communication and synchronization between processors are not the bottleneck. In particular, analyzing Sol_2, we have verified that it is not convenient to cluster other tasks with MMTC_fquantSR, as it has the highest C^l and C^u; therefore, clustering it with other tasks penalizes the pipeline

Table 12
Comparison of clustering solutions for SC.

Sol_{ID} DSE	Clusters			p	$C^{0.5}$ (μs)	*sim* (cycles)	*sim* (μs)	sched
	PE_1	PE_2	PE_3					
	MBGetLine1, DCT_{1,2}, MBZero{0,1,2,3,4,5}, MBPackGetLine{1,2}, keep2x2	MMTC_fquantSR	iquantizeSR, slRow3, MBPackGetLine{3,4}, fxIDCT8_{1,2,3,4}, fefoIDCT8_{1,2}, srTrim, srAddRow3, MBPack6, MBPack3	0.99	172,000	274,847,324	171,799.58	yes
1	MBGetLine1, DCT_{1,2}, MBZero{0,1,2,3,4,5}, MBPackGetLine{1,2}, keep2x2, MMTC_fquantSR, iquantizeSR, slRow3, MBPackGetLine{3,4}, fxIDCT8_{1,2,3,4}, fefoIDCT8_{1,2}, srTrim, srAddRow3, MBPack6, MBPack3	–	–	0	242,600	573,715,348	358,572.09	no
2	MBGetLine1, DCT_{1,2}, MBZero{0,1,2,3,5}, MBPackGetLine{1,2}, keep2x2	MBZero4, MMTC_fquantSR	iquantizeSR, MBPackGetLine{3,4}, slRow3, fxIDCT8_{1,2,3,4}, fefoIDCT8_{1,2}, srTrim, srAddRow3, MBPack6, MBPack3	0.99	185,400	275,255,614	172,034.76	yes
3	MBGetLine1, DCT_{1,2}, MBZero{0,1,2,3,4,5}, MBPackGetLine{1,2}, keep2x2, iquantizeSR	MMTC_fquantSR	MBPackGetLine{3,4}, slRow3, fxIDCT8_{1,2,3,4}, fefoIDCT8_1,2, srTrim, srAddRow3, MBPack6, MBPack3	0.99	185,500	278939,568	174,337.23	yes
4	MBGetLine1, DCT_{1,2}, MBZero{0,1,2,3,4,5}, MBPackGetLine{1,2}, keep2x2	MMTC_fquantSR, iquantizeSR, MBPackGetLine3, slRow3, fxIDCT8_1	MBPackGetLine4, slRow3, fxIDCT8_{2,3,4}, fefoIDCT8_{1,2}, srTrim, srAddRow3, MBPack6, MBPack3	0.95	194,300	303,402,927	189,626.83	yes

Table 13
Comparison between the number of cycles estimated by the profiling tool [30] and the ones obtained from simulation for MJPEG.

Task Name	C_j^l	C_j^u	sim	$\%Tot_{C_j^l}$	$\%Tot_{sim}$	Err 8 $\lvert \%Tot_{C_j^l} - \%Tot_{sim} \rvert$
mainDCT	25,695,360	71,617,920	69,815,040	51.52	55.13	3.61
mainQ	6,401,280	8,868,480	12,150,620	12.83	9.59	3.24
mainVLE	16,212,735	24,024,690	40,942,298	32.51	32.33	0.18
mainVideoOut	1,565,550	1,934,250	3,168,482	3.14	2.50	0.64
Total (**cycles**)	**49,874,925**	**106,445,340**	**126,635,428**			

Table 14
Comparison between the number of cycles estimated by the profiling tool [30] and the ones obtained from simulation for ECG.

Task name	C_j^l (cycles)	C_j^u (cycles)	Sim (cycles)	$\%Tot_{C_j^l}$	$\%Tot_{sim}$	Err $\lvert \%Tot_{C_j^l} - \%Tot_{sim} \rvert$
Lowpass	400,015	450,010	750,016	3.39	4.55	1.16
Highpass	450,084	840,021	1,330,203	3.81	8.08	4.26
Derivative	10,000	10,000	350,012	0.08	2.12	2.04
Square	10,000	10,000	100,000	0.08	0.61	0.52
integrative	10,830,071	20,370,915	12,753,683	91.78	77.42	14.36
Detect	99,614	155,070	358,661	0.84	2.18	1.33
Total (**cycles**)	**11,799,784**	**21,836,016**	**16,472,592**			

parallelism (even if MBZero4 is a task with almost negligible WCET). When we have evaluated Sol_2 and Sol_3 with our *UM*, we have obtained a probability of 0.99 but with higher values in the quantile function $C^{0.5}$; these results are reflected in higher scheduling lengths. In Sol_4 we have explored a clustering solution in which the task level parallelism between tasks fxIDCT_1 and fxIDCT_2 can be exploited. In this case we have a lower probability (0.95) and a higher scheduling length than the previous solutions after synthesis. Consequently, with our *UM*, we could determine before synthesis which are the better solutions to consider for implementation.

We have demonstrated that our *UM* and the associated DSE can explore in a reasonable time (less than 1 h) multiple clustering solutions and provide a good indication of which task clustering should be selected. Even if we cannot claim to find a schedulable task clustering solution with our *UM*, we have demonstrated that we can find a clustering solution that has high probability of being schedulable after synthesis. Additionally, we have showed that the probability and the quantile function values can be used to compare different clustering solutions and that the results obtained after synthesis are consistent with our evaluations. Our approach can offer a valid starting point for a designer that has to implement a multi-ASIP platform in which the ASIPs have not being synthesized yet.

5.2.4. Accuracy of C_j^l and C_j^u

In this section, we analyze the impact of the selection of the upper and the lower bound (C_j^u and C_j^l) of each task τ_j. First, we verify how accurate the C_j^l and C_j^u found by the code analysis tool [30] are. For each case study, we have compared the upper and lower bound values estimated by the code analysis tool (C_j^l and C_j^u) with the number of cycles obtained for the execution of the entire applications on a synthesized ASIP. We have used an oversized ASIP with a large number of *IS*s to theoretically exploit all instruction level parallelism of the application (the parallelism

Table 15
Comparison between the number of cycles estimated by the profiling tool [30] and the ones obtained from simulation for SC.

Task name	C_j^l (cycles)	C_j^u (cycles)	sim (cycles)	$\%Tot_{C_j^l}$	$\%Tot_{sim}$	Err $\lvert \%Tot_{C_j^l} - \%Tot_{sim} \rvert$
MBPackGetLine3	2,556,000	3,708,000	16932000	0.83	2.95	2.12
fxIDCT8_{1,2,3,4}	7,680,000	22,416,000	23,688,000	2.50	4.13	1.62
keep2x2	792,000	936,000	8,040,000	0.26	1.40	1.14
srAddRow3	768,000	1,680,000	2,496,000	0.25	0.44	0.18
srTrim	5,856,000	7,008,000	6,240,022	1.91	1.09	0.82
iquantizeSR	10,908,003	15,090,000	37,308,000	3.56	6.51	2.95
MBPack3	1,500,000	2,268,000	12,720,000	0.49	2.22	1.73
MBPackGetLine{2,4}	1,806,000	2,574,000	17,700,000	0.59	3.09	2.50
MBPack6	750,000	1,134,000	12,720,000	0.24	2.21	1.97
slRow3	768,000	1,248,000	2,016,000	0.25	0.35	0.10
fefoIDCT8_{1,2}	768,000	1,824,000	3,216,000	0.25	0.56	0.31
MMTC_fquantSR	236,579,118	312,956,828	275,885,112	77.17	48.10	29.06
MBPackGetLine1	2,094,000	2,910,000	17,700,000	0.68	3.09	2.40
MBZero{0,1,2,3,4,5}	36,000	72,000	6,048,000	0.01	1.05	1.04
DCT_{1,2}	3,360,000	10,320,000	13,488,000	1.10	2.35	1.26
MBGetLine1	1,296,000	1,728,000	12,624,000	0.42	2.20	1.78
Total (cycles)	**306,581,121**	**469,928,828**	**573,521,134**			

during the real execution can be lower depending on the compiler optimization). For the comparison, we have used the estimated (C_j^l and C_j^u) and simulated (*sim*) values of *all iterations* of the tasks. The values obtained for MJPEG encoder, ECG and SC are summarized in columns 2–4 of, Tables 13–15, respectively. The C_j^l, C_j^u and *sim* values for each task τ_j are quite different and the *sim* values in most of the cases are not included in the range $[C_j^l, C_j^u]$ as expected (when we compare the C_j^u with the results obtained with simulation, we have relative errors up to 38% for MJPEG, 41% for ECG and up to 98% for SC).

These differences can be justified by considering some inaccuracy in the estimation tool described in [30]. The analysis of the accuracy of the code analysis tool is provided in [56]. For the case studies analyzed in [56], there is less than 10% underestimation for the evaluated number of cycles compared to the simulated one. In our case, the biggest differences between estimation and simulation results can derive from the higher complexity of the application code in which LLVM and Silicon Hive's compilers perform different types of optimization. Another factor that impacts the estimation is the number of stalls (e.g. hardware stalls) that are considered by Silicon Hive simulator and not by the code analysis tool. Moreover in Silicon Hive simulation, there are some cycles of overhead for starting the tasks execution and for synchronization with the host processor; these cycles are ignored by the code analysis tool (however, we are considering these extra cycles during our scheduling analysis to compensate the code analysis tool evaluation). It is also important to mention that the code analysis tool provides better results when we are comparing the execution of the entire applications and not the single contribution of the different tasks: in this case we have errors up to 16% for MJPEG, 32% for ECG and 18% for SC.

After these analyses we have verified which elements were influencing our design space exploration and our *UM* to understand why with such relevant errors in the upper and lower bound estimations, it is still possible to get good results for our case studies. We have noticed that the absolute value of the C_j^l and C_j^u of each task is not relevant. On the contrary it is relevant its relative value when compared to the other tasks in the application (this is true until a certain extent as there is also the influence of the messages and the interconnection network). Therefore, we have evaluated the contribution of each task to the total number of cycles of the application: we have performed this check for C_j^l and *sim* values. We have used the lower bound value because it corresponds to the most parallelized version of the application and we have run the entire application on an ASIP with a large number of ISs, also to get the most parallelized execution. The results obtained are available in columns 5 and 6 of Tables 13 (MJPEG encoder), 14 (ECG) and 15 (SC). Using the relative contribution of each task to the total number of cycles (for the estimated C^l and simulated *sim*), we have calculated the absolute error between them. The absolute error is available in column 7 of Tables 13 (MJPEG encoder), 14 (ECG) and 15 (SC). For each task, we have verified how much it contributes to the total number of cycles in the lower bound estimation compared to the task contribution to the total number of cycles in the Silicon Hive simulation (*sim*). According to this evaluation, the estimated upper bound shows which tasks are more time consuming than others and this is reflected also in the simulation results on a real ASIP. For our case studies, we have gotten errors up to 3.31% for MJPEG encoder, 14.36% for ECG and 29.06% and for SC. Additionally, we have run an experiment to check how big could be the difference between the estimated and simulated performance values before having an impact on the DSE results. We have considered the MJPEG encoder case study that has the most accurate estimated values for the C_j^l and C_j^u. We have increased the upper and lower bounds for each task of 10%, 20%, 35% and 45%. Then we have run our DSE and in all cases we have found the same task clustering obtained with the original values (with different probability and different quantile values at 50%). This suggests that our DSE is not sensitive to quite relevant variations in the upper and lower bound estimations.

6. Conclusion

In this paper we have proposed an approach for the synthesis of multi-ASIP platforms for streaming applications. We have modeled the applications as SDFGs in order to exploit both task level and pipeline parallelism. The synthesis of a multi-ASIP platform includes defining the number and type of ASIPs and their interconnection. Each ASIP is synthesized according to the cluster of tasks that it has to run. At the same time, to explore different platform alternatives, we need to perform a schedulability analysis of the application on the candidate platform. This schedulability analysis requires information about the WCET of the tasks running on a certain ASIP. This information is not available as an ASIP can be defined (and optimized) only after knowing the cluster of tasks that it has to run. Therefore, we have observed a circular dependency that we have broken using an Uncertainty Model for the WCET. The UM captures the WCET of a task running on a wide range of possible ASIP micro-architecture implementations.

We have developed a schedulability analysis that uses the UM and evaluates different task clustering solutions selecting the one

which has a high chance of meeting the application's deadline under an imposed platform cost. Through experimental evaluation we have validated the use of a Normal distribution for the UM. Additionally, we have used three real case studies to demonstrate the effectiveness of our DSE. We compared the results obtained by our DSE with the ones obtained after the multi-ASIP synthesis with Silicon Hive's tools. Our experimental results have shown that by considering the range of possible ASIP micro-architectural implementations during DSE, we can identify the task clustering solution that should be considered for platform synthesis.

Additional note

The results obtained using Silicon Hive's tools should not be used in any way as a reference to evaluate Intel technology or to compare Silicon Hive's technology with other commercial and/or research tools, as only a subset of the functionalities and optimization offered by the tools have been used and/or made available under our University license agreement.

Acknowledgments

The work on this paper has been performed in the scope of the ASAM project of the European ARTEMIS Research Program and has been partly supported by the ARTEMIS Joint Undertaking under Grant no. 100265. The authors would like to thank Rosilde Corvino, Erkan Diken and Roel Jordan for allowing the use of their research tools for micro-architecture DSE. Additionally, the authors would like to thanks Menno Lindwer from Intel Benelux and Bart Kienhuis from Compaan Compiler for the support and help in the use of the technology and tools supplied and Giuseppe Notarangelo from ST Microelectronics (Catania) for the suggestions and help for the elaboration of the case studies.

References

[1] H.C. Doan, H. Javaid, S. Parameswaran, Multi-ASIP based parallel and scalable implementation of motion estimation kernel for high definition videos, in: 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia, IEEE, Taipei, 2011, pp. 56–65.

[2] S. Saponara, L. Fanucci, S. Marsi, G. Ramponi, Algorithmic and architectural design for real-time and power-efficient Retinex image/video processing, Journal of Real-Time Image Processing. 1 (4) (2007) 267–283.

[3] H. Javaid, S. Parameswaran, Synthesis of heterogeneous pipelined multiprocessor systems using ILP, in: Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis—CODES/ISSS '08, ACM Press, New York, NY, USA, 2008, p. 1.

[4] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman & Co., New York, NY, USA, 1979.

[5] M. Jain, M. Balakrishnan, A. Kumar, ASIP design methodologies: survey and issues, in: VLSI Design 2001. Fourteenth International Conference on VLSI Design, IEEE Computer Society, Bangalore, 2001, pp. 76–81.

[6] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, J. Riihimäki, K. Kuusilinna, UML-based multiprocessor SoC design framework, ACM Trans. Embedded Comput. Syst. 5 (2) (2006) 281–320.

[7] A. Kumar, S. Fernando, Y. Ha, B. Mesman, H. Corporaal, Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA, ACM Trans. Des. Autom. Electron. Syst. 13 (3) (2008) 1–27.

[8] A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, H. Zeng, Q. Zhu, A Next-Generation Design Framework for Platform-based Design, in: DVCon 2007, 2007.

[9] J. Kreku, M. Hoppari, T. Kestilä, Y. Qu, J.-P. Soininen, P. Andersson, K. Tiensyrjä, Combining UML2 application and SystemC platform modelling for performance evaluation of real-time embedded systems, EURASIP J. Embedded Syst., 2008, Article 6.

[10] A.D. Pimentel, The Artemis workbench for system-level performance evaluation of embedded systems, IJES 3 (3) (2008) 181–196.

[11] A.K. Singh, A. Kumar, T. Srikanthan, A hybrid strategy for mapping multiple throughput-constrained applications on MPSoCs, in: Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems—CASES '11, ACM Press, New York, NY, USA, 2011, p. 175.

[12] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, B. Vanthournout, A modular simulation framework for spatial and temporal task mapping onto multi-processor soc platforms, in: Proceedings of the conference on Design, Automation and Test in Europe, vol. 2, DATE '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 876–881.

[13] O. Muller, A. Baghdadi, M. Jezequel, From parallelism levels to a multi-asip architecture for turbo decoding, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 17 (1) (2009) 92–102.

[14] C. Brehm, T. Ilnseher, N. Wehn, A scalable multi-ASIP architecture for standard compliant trellis decoding, in: 2011 International SoC Design Conference, IEEE, 2011, pp. 349–352.

[15] D.K.F. Ieromnimon, D. Kritharidis, N. S. Voros, Application of the MOSART flow on the WiMAX (802.16 e) PHY layer, in: Scalable Multi-core Architectures, Springer, 2012, pp. 197-223.

[16] Seng Lin Shee, S. Parameswaran, Design Methodology for Pipelined Heterogeneous Multiprocessor System, 2007.

[17] A. Wieferink, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, A. Nohl, A system level processor/communication co-exploration methodology for multiprocessor system-on-chip platforms, in: Design Automation Conference, 2004.

[18] L. Micconi, D. Gangadharan, P. Pop, J. Madsen, Multi-ASIP platform synthesis for real-time applications, in: 2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES), IEEE, Porto, 2013, pp. 59–67.

[19] A. Kumar, A. Hansson, J. Huisken, H. Corporaal, Interactive presentation: An fpga design flow for reconfigurable network-based multi-processor systems on chip, in: Proceedings of the Conference on Design, Automation and Test in Europe, 2007, pp. 117–122.

[20] E.A. Lee, D.G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Trans. Comput. 36 (1) (1987) 24–35.

[21] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, H. Corporaal, Schedule-extended synchronous dataflow graphs, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 32 (10) (2013) 1495–1508.

[22] A.-H. Ghamarian, M.C.W. Geilen, S. Stuijk, T. Basten, A.J.M. Moonen, M. Bekooij, B. Theelen, M. Mousavi, Throughput analysis of synchronous data flow graphs, in: Sixth International Conference on Application of Concurrency to System Design, 2006. ACSD 2006, 2006, pp. 25–36.

[23] P.-K. Huang, M. Hashemi, S. Ghiasi, System-level performance estimation for application-specific MPSoC interconnect synthesis, in: Symposium on Application Specific Processors, 2008, pp. 95–100.

[24] L. Micconi, J. Madsen, P. Pop, A probabilistic approach for the system-level design of multi-asip platforms, Kgs. Lyngby: Technical University of Denmark (DTU). 2015. (DTU Compute PHD-2014; Journal number 347).

[25] M.K. Jain, M. Balakrishnan, A. Kumar, ASIP design methodologies: Survey and issues, in: Proceedings of the IEEE/ACM International Conference on VLSI Design, 2001, pp. 76–81.

[26] K. Karuri, R. Leupers, G. Ascheid, H. Meyr, A generic design flow for application specific processor customization through instruction-set extensions (ises), in: Embedded Computer Systems: Architectures, Modeling, and Simulation, Lecture Notes in Computer Science, vol. 5657, Springer, Berlin, Heidelberg, 2009, pp. 204–214.

[27] R. Muhammad, L. Apvrille, R. Pacalet, Evaluation of ASIPs design with LISATek, in: M. Berekovic, N.J. Dimopoulos, S. Wong (Eds.), Embedded Computer Systems: Architectures, Model-ing, and Simulation, Vol. 5114 of Lecture Notes in ComputerScience, Springer Berlin Heidelberg, 2008, pp. 177-186.

[28] A. Nohl, F. Schirrmeister, D. Taussig, Application specific processor design architectures, design methods and tools, in: Proceedings of the International Conference on Computer-Aided Design, 2010, pp. 349–352.

[29] D. Goodwin, D. Petkov, Automatic generation of application specific processors, in: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES '03, ACM, New York, NY, USA, 2003, pp. 137–147.

[30] L. Jozwiak, M. Lindwer, R. Corvino, P. Meloni, L. Micconi, J. Madsen, E. Diken, D. Gangadharan, R. Jordans, S. Pomata, P. Pop, G. Tuveri, L. Raffo, G. Notarangelo, ASAM: automatic architecture synthesis and application mapping, Microprocess. Microsyst. 37 (8) (2013) 1002–1019.

[31] M. Nicola, G. Masera, M. Zamboni, H. Ishebabi, D. Kammler, G. Ascheid, H. Meyr, Fft processor: a case study in asip development, in: Proceedings of the IST Mobile & Wireless Communications Summit, Dresden, Germany, 2005.

[32] P. Karlstrom, W. Zhou, C.-h. Wang, D. Liu, Design of pioneer: A case study using nogap, in: 2010 Asia Pacific Conference on Postgraduate Research in Microelectronics and Electronics (PrimeAsia), IEEE, Shanghai, 2010, pp. 53–56.

[33] J. Axelsson, A method for evaluating uncertainties in the early development phases of embedded real-time systems, in: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, IEEE Computer Society, Washington, DC, USA, 2005, pp. 72–75.

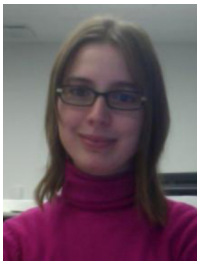
[34] E. Lee, D. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, Comput. IEEE Trans. C 36 (1) (1987) 24–35.

[35] S. Sriram, S.S. Bhattacharyya, Embedded Multiprocessors: Scheduling and Synchronization, 1st ed., Marcel Dekker, Inc., New York, NY, USA, 2000.

[36] A. Bonfietti, L. Benini, M. Lombardi, M. Milano, An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010, IEEE, Dresden, 2010, pp. 897–902.

[37] O. Moreira, J.-D. Mol, M. Bekooij, J. Van Meerbergen, Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix, in: Real Time and Embedded Technology and Applications Symposium, 2005, RTAS 2005, 11th IEEE, 2005, pp. 332–341.

[38] W. Liu, M. Yuan, X. He, Z. Gu, X. Liu, Efficient sat-based mapping and scheduling of homogeneous synchronous dataflow graphs for throughput optimization, in: Real-Time Systems Symposium, 2008, pp. 492–504.
[39] G. Zaki, W. Plishker, S. Bhattacharyya, F. Fruth, Partial expansion graphs: Exposing parallelism and dynamic scheduling opportunities for DSP applications, in: 2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2012, pp. 86–93.
[40] A. H. Ghamarian, S. Stuijk, T. Basten, M. Geilen, B.D. Theelen, Latency minimization for synchronous data flow graphs, in: 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, 2007, DSD 2007, IEEE, Lubeck, 2007, pp. 189–196.
[41] S. Stuijk, T. Basten, M.C.W. Geilen, H. Corporaal, Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs, in: Proceedings of the 44th Annual Design Automation Conference, DAC '07, ACM, New York, NY, USA, 2007, pp. 777–782.
[42] E.K. Burke, G. Kendall, Search: Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques, Springer, 2005.
[43] T.U.o.E.T. Electronic Systems, Mamps Project, Partitioned jpeg Decoder Algorithm, ⟨http://www.es.ele.tue.nl/mamps/example.php⟩(accessed July 2014).
[44] MAD, MPEG Audio Decoder, ⟨http://www.underbit.com/products/mad⟩(accessed June 2014).
[45] J.A. Fisher, P. Faraboschi, C. Young, VEX, A VLIW Example, ⟨http://www.hpl.hp.com/downloads/vex⟩(accessed July 2014).
[46] Trimedia TM-1300 Datasheet, ⟨http://www.datasheetcatalog.org/datasheet/philips/PTM1300.pdf⟩(accessed July 2014).
[47] V. Zaccaria, G. Palermo, F.C.P. di Milano, G. M. (USI), Multicube Explorer, ⟨http://home.dei.polimi.it/zaccaria/multicube_explorer_v1/Home.html⟩(accessed July 2014).
[48] S.F. Edgar, Estimation of Worst-Case Execution Time Using Statistical Analysis, University of York, Department of Computer Science—Publications-YCST.
[49] J. Hansen, S.A. Hissam, G.A. Moreno, Statistical-based WCET estimation and validation, in: Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis, 2009, pp. 1–11.
[50] Mjpeg Code, Leiden University, ⟨http://www.artist-embedded.org/artist/Benchmarks.html⟩(accessed June 2014).
[51] Mpeg4 Application, STMicroelectronics, ⟨http://www.st.com/⟩(accessed June 2014).
[52] Ecg Application, Technical University of Denmark, Not Publicly Available (June 2014).
[53] ASAM, Automatic Architecture Synthesis and Application Mapping, ⟨http://www.asam-project.org⟩(accessed June 2014).
[54] Compaan Compiler, ⟨http://www.compaandesign.com⟩(accessed June 2014).
[55] The llvm Compiler Infrastructure, ⟨http://llvm.org⟩(accessed June 2014).
[56] R. Jordans, R. Corvino, L. Jozwiak, H. Corporaal, Exploring processor parallelism: estimation methods and optimization strategies, in: 2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS), 2013, pp. 18–23.



Laura Micconi has received her Ph.D. degree from the Technical University of Denmark (Embedded System Engineering group) in 2015. Prior to this, she attended an Executive Master in Embedded System Design at the Advanced Learning and Research Institute (ALaRI) at the University of the Svizzera Italiana (USI). She got her MSc degree in Computer Science Engineering from Politecnico of Torino. Her research interests mainly lie in the field of System-level design and mapping/scheduling of application on multi-processor systems. She is currently working as ASIP design Engineer at Sigma Designs (Copenhagen, Denmark)

Jan Madsen is Full Professor in computer-based systems at DTU Compute, Technical University of Denmark (DTU). He received his Ph.D. degree in computer science from DTU in 1992. His main research interests are related to methods and tools for systems engineering of computing systems. Present research covers embedded systems-on-a-chip, wireless sensor networks (Internet-of-Things), microfluidic labs-on-a-chip and synthetic biology. Emphasis is on design, modeling, analysis and optimization of such systems, including the development of design automation tools and design methodologies. He has published more than 140 peer-reviewed conference and journal papers, 12 book chapters, 1 book and 4 edited books. He has several best paper nominations, 2 best paper award (MECO 2013, CASES 2009), 1 paper among the 30 most influential papers from 10 years of Design Automation and Test in Europe (DATE), and 3 papers among the highly cited papers in System Codesign and Synthesis. He holds 2 patents, from which he has co-founded the spin-off company Biomicore. He has served on the technical program committee of numerous conferences and has been general chair for CODES and NOCS, Program Chair for DATE, CODES+ISSS, CODES and NORCHIP. He is on the editorial board of IEEE Design & Test. At DTU Compute, he is Deputy Director of the Department and head of the Embedded Systems Engineering section.

Prof. Paul Pop is an Associate Professor at DTU Compute, Technical University of Denmark (DTU). He has received his Ph.D. degree in computer systems from Linköping University in 2003. His main research interests are in the area of system-level design of embedded systems. He has published extensively in this area, over 100 peer-reviewed international publications, 3 books and 7 book chapters. He has received the best paper award at the Design, Automation and Test in Europe Conference (DATE 2005) and at the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2010). His students have received awards for their theses, such as the EDAA Outstanding Dissertation (Ph.D. thesis) and the Embedded Skills Award (M.Sc. thesis) from the Confederation of Danish Industry (DI ITEK). He has served on the technical program committee of numerous conferences, such as DATE, ICCAD, CODES+ISSS, ASP DAC and RTSS and he is on the editorial board of two international journals. At DTU Compute, he is the coordinator of a research group focusing on safety-critical embedded systems, with a focus on mixed-criticality systems. He participates in several national and EU projects in this area.