

# A Methodology for Automated Design of Hard-Real-Time Embedded Streaming Systems

Mohamed A. Bamakhrama, Jiali Teddy Zhai, Hristo Nikolov and Todor Stefanov  
Leiden Institute of Advanced Computer Science  
Leiden University, Leiden, The Netherlands  
Email: {mohamed, tzhai, nikolov, stefanov}@liacs.nl

**Abstract**—The increasing complexity of modern embedded streaming applications imposes new challenges on system designers nowadays. For instance, the applications evolved to the point that in many cases hard-real-time execution on multiprocessor platforms is needed in order to meet the applications’ timing requirements. Moreover, in some cases, there is a need to run a set of such applications simultaneously on the same platform with support for accepting new incoming applications at run-time. Dealing with all these new challenges increases significantly the complexity of system design. However, the design time must remain acceptable. This requires the development of novel systematic and automated design methodologies driven by the aforementioned challenges. In this paper, we propose such a novel methodology for automated design of an embedded multiprocessor system, which can run multiple hard-real-time streaming applications simultaneously. Our methodology does not need the complex and time-consuming design space exploration phase, present in most of the current state-of-the-art multiprocessor design frameworks. In contrast, our methodology applies very fast yet accurate schedulability analysis to determine the minimum number of processors, needed to schedule the applications, and the mapping of applications’ tasks to processors. Furthermore, our methodology enables the use of hard-real-time multiprocessor scheduling theory to schedule the applications in a way that temporal isolation and a given throughput of each application are guaranteed. We evaluate an implementation of our methodology using a set of real-life streaming applications and demonstrate that it can greatly reduce the design time and effort while generating high quality hard-real-time systems.

## I. INTRODUCTION

The increasing complexity of embedded streaming applications in several domains (e.g., defense and medical imaging) imposes new challenges on embedded system designers. In particular, the applications evolved to the point that in order to meet their timing requirements, in many cases, the applications need hard-real-time execution on multiprocessor platforms [1]. Moreover, in some other cases (e.g., software defined radio), multiple applications have to run simultaneously on a single multiprocessor platform with the ability to accept new incoming applications at run-time [1]. The way to deal with these new challenges is to design time-predictable architectures which employ run-time resource management solutions. These solutions must provide fast *admission control*, i.e., ability to determine at run-time whether a new application can be scheduled to meet all deadlines, and *temporal isolation*, i.e., ability to start/stop applications at run-time without affecting other already running applications. Addressing all these new challenges increases significantly the complexity of system design. However, the design time must remain acceptable, which requires novel systematic, and moreover, automated design methodologies driven by the aforementioned challenges.

Current practices in designing embedded streaming systems employ *Model-of-Computation (MoC)*-based design [2], which

allows the designers to express important application properties, such as parallelism, in an explicit way. Also, the MoC-based design facilitates analysis of certain system properties such as timing behavior. In most current MoC-based design methodologies [2], an application is typically modeled as either a dataflow graph, e.g., Synchronous Dataflow (SDF) [3] and Cyclo-Static Dataflow (CSDF) [4], or a process network, e.g., Polyhedral Process Network (PPN) [5]. Although there is a large diversity of design methodologies nowadays, most of them do not consider systems running multiple applications simultaneously. The problem of automated design of such systems has been addressed in [6] and [7]. However, these methodologies do not provide timing guarantees for each application as they provide only best-effort services. Timing guarantees are provided by other design approaches, e.g., [8]–[10]. Nevertheless, most of the methodologies share the following two shortcomings. First, there is a lack of tools for automated parallelization of legacy applications. It is common knowledge that manually building a parallel specification of an application is a tedious and error-prone task. Second, the design methodologies rely on complex Design Space Exploration (DSE) techniques to determine the minimum number of processors needed to schedule the applications and the mapping of tasks to processors. Depending on the application and target platform complexity, DSE may take considerable amount of time. Therefore, these two particular shortcomings in combination may lead to a very long design cycle, especially if we consider designing a system that runs multiple applications simultaneously with guaranteed hard-real-time execution. In this paper, we address these shortcomings and propose a novel methodology for automated design of a multiprocessor embedded system that runs multiple applications simultaneously and provides temporal isolation and guaranteed throughput for each application.

A major part of the design methodology we propose is based on algorithms from the hard-real-time multiprocessor scheduling theory [11]. These algorithms can perform fast admission and scheduling decisions for incoming applications while providing timing guarantees and temporal isolation. So far, these algorithms received little attention in the embedded streaming community because they typically assume *independent* periodic or sporadic tasks. In contrast, modern embedded streaming applications are usually specified as graphs where nodes represent tasks and edges represent *data-dependencies*. Recently, it has been shown that embedded streaming applications modeled as acyclic CSDF graphs can be scheduled as implicit-deadline periodic tasks [12]. This enables applying hard-real-time multiprocessor scheduling algorithms to a broad class of streaming applications.

## A. Paper Contributions

We devise a methodology for the automated design of an embedded Multiprocessor System-on-Chip (MPSoC) that runs a set of automatically parallelized hard-real-time embedded streaming applications. One of the main contributions is that our methodology does not require DSE to determine the minimum number of processors needed to schedule the applications and the mapping of applications' tasks to processors. We realize our methodology as an extension to the Daedalus design flow [13]. The extended flow, named Daedalus<sup>RT</sup>, accepts a set of streaming applications (specified in sequential C) and derives automatically their equivalent CSDF graphs and PPN models. The automated derivation of CSDF graphs is another main contribution of this paper because it enables fast schedulability analysis in Daedalus<sup>RT</sup>. The derived CSDF graphs are used in such analysis which applies hard-real-time multiprocessor scheduling theory to schedule the applications in a way that temporal isolation and a given throughput of each application are guaranteed. The PPN models are used for the generation of the parallel code that runs on the processors of the target multiprocessor system. We evaluate the Daedalus<sup>RT</sup> flow and demonstrate that it can greatly reduce the designer effort while generating high quality systems.

## B. Scope of Work

The methodology, we propose, considers as an input a set of independent streaming applications specified as Static Affine Nested Loop Programs (SANLP) [5] without cyclic dependencies. We assume that the applications process periodic input streams of data. The target hardware platform is a tiled homogeneous MPSoC with distributed memory.

The remainder of this paper is organized as follows: Section II gives an overview of the related work. Section III introduces the preliminary material. Section IV presents the proposed methodology and Daedalus<sup>RT</sup> design flow. Section V presents the results of an empirical evaluation of Daedalus<sup>RT</sup>. Finally, Section VI ends the paper with conclusions.

## II. RELATED WORK

Several design flows for automated mapping of applications onto MPSoC platforms are surveyed in [2]. These flows use DSE procedures to determine the number of processors and the mapping of tasks to processors. In contrast, in the design flow proposed in this paper, we replace the complex DSE with very fast and accurate schedulability analysis to determine the minimum number of processors, needed to schedule the applications, and the mapping of tasks to processors. In addition, our design flow generates hard-real-time systems and derives the CSDF model used in the analysis in an automated way.

PeaCE [8] is an integrated HW/SW co-design framework for embedded multimedia systems. Similar to our approach, it also uses two MoCs. In contrast, however, it employs Synchronous Piggybacked Dataflow (SPDF) for computation tasks and Flexible Finite State Machines (fFSM) for control tasks. PeaCE uses DSE techniques and HW-SW co-simulations during the design phase in order to meet certain timing constraints. In contrast, we avoid these iterative steps by applying hard-real-time multiprocessor scheduling theory to guarantee temporal isolation and a given throughput of each application running on the target MPSoC.

CA-MPSoC [9] is an automated design flow for mapping multiple applications modeled as SDF graphs onto Commu-

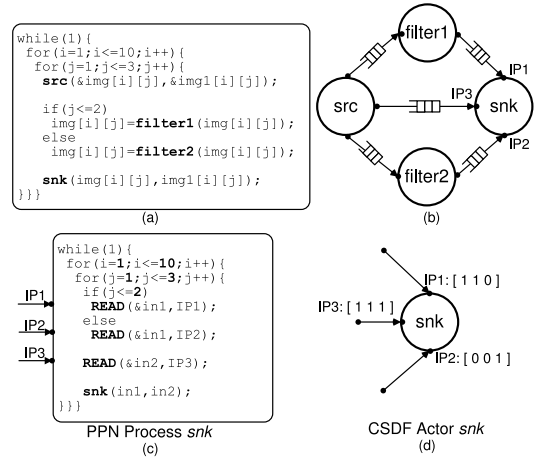


Fig. 1. A SANLP program with its PPN and CSDF representations. The flow uses non-preemptive scheduling to schedule the applications. In contrast, we consider only preemptive scheduling, because non-preemptive scheduling to meet all the deadlines is known to be NP-hard in the strong sense even for the uniprocessor case [14]. Moreover, we consider a more expressive MoC, namely the CSDF model.

An automated design flow for mapping applications modeled as SDF graphs onto MPSoCs while providing timing guarantees is presented in [10]. The flow requires the designer to specify a sequential implementation (in C) of each actor in an SDF graph. The generated implementation uses either Self-Timed or Time-Division-Multiplexing (TDM) scheduling to ensure meeting the throughput constraints. In contrast, our design flow starts from sequential applications written in C, and consequently, the code realizing the functional behavior of the PPN processes is automatically extracted from the initial C programs. Moreover, we schedule the tasks as implicit-deadline periodic tasks, which enables applying very fast schedulability analysis to determine the minimum number of required processors. Instead, [10] applies DSE techniques to determine the minimum number of processors and the mapping. Finally, our methodology supports multiple applications to run simultaneously on an MPSoC, while the work in [10] does not support multiple applications.

## III. PRELIMINARIES

In this section, we provide a brief overview of the PPN and CSDF models, hard-real-time scheduling of CSDF, and the Daedalus flow. This overview is essential for understanding the contributions presented in Section IV.

### A. PPN and CSDF

Both of the PPN [5] and CSDF [4] are parallel MoCs, in which processes/actors communicate only explicitly via FIFO channels. The execution of a PPN process is a set of iterations. This set of iterations is represented using the polytope model [15] and is called *process domain*, denoted by  $d_{process}$ . Accessing input/output ports of the PPN process is represented as a subset of the process domain, called *input/output port domain*. Compared to PPN processes, accessing input/output ports of CSDF actors is described using repetitive production/consumption rates sequences. Another key difference is that synchronization in PPN is implemented using blocking reads/writes, while in CSDF it is implemented explicitly using a schedule. It has been shown in [16] that a non-parametrized and acyclic PPN is equivalent to a CSDF graph where the

production/consumption rates sequences consist only of 0s and 1s. A '0' in the sequence indicates that a token is not produced/consumed, while a '1' indicates that a token is produced/consumed. Below, we use an example to illustrate both PPN and CSDF models.

Consider the sequential C program given in Fig. 1(a). The equivalent PPN model that can be derived using the `pn` compiler [5] is shown in Fig. 1(b). For the implementation of process `snk` shown in Fig. 1(c), its process domain is given as  $d_{snk} = \{(w, i, j) \in \mathbb{Z}^3 \mid w > 0 \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 3\}$ . Reading tokens from input port `IP1` to initialize function argument `in1` of `snk` is represented as input port domain  $d_{IP1} = \{(w, i, j) \in \mathbb{Z}^3 \mid w > 0 \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 2\}$ . The equivalent CSDF graph has the same topology as the PPN model in Fig. 1(b). The CSDF actor `snk` corresponding to process `snk` is shown in Fig. 1(d). The execution of actor `snk` is represented using consumption rates sequences. Reading tokens from input port `IP1` is described as the consumption rates sequence  $[1, 1, 0]$ .

### B. Hard-Real-Time Scheduling of Applications Modeled as Acyclic CSDF Graphs

Below, we summarize the main theoretical results proved in [12]. Let  $G$  be a consistent and live CSDF graph with periodic input streams. Let  $N$  be the set of actors in  $G$ , and  $E$  be the set of communication channels in  $G$ . The authors in [12] proved the following:

**Theorem 1.** *For any acyclic  $G$ , a periodic schedule  $\mathcal{S}$  exists such that every actor  $n_j \in N$  is fired in a strictly periodic manner with a constant period  $\lambda_j \in \bar{\lambda}^{min}$ , where  $\bar{\lambda}^{min}$  is given by Lemma 2 in [12], and every communication channel  $e_u \in E$  has a bounded buffer capacity given by Theorem 4 in [12].*

Theorem 1 states that the tasks of any application modeled as an acyclic CSDF graph can be scheduled as implicit-deadline periodic tasks. This result enables using a wide variety of hard-real-time multiprocessor scheduling algorithms to schedule the applications. The result extends trivially to multiple applications. Let  $\Upsilon = \{G_1, G_2, \dots, G_N\}$  be a set of applications. Then, a super task set  $\Gamma_\Upsilon$  can be formed by the union of all the individual task-sets representing the applications. That is:

$$\Gamma_\Upsilon = \bigcup_{G_i \in \Upsilon} \Gamma_{G_i} \quad (1)$$

$\Gamma_{G_i}$  is given by Corollary 2 in [12] as  $\Gamma_{G_i} = \{\tau_1, \tau_2, \dots, \tau_K\}$ .  $\tau_j \in \Gamma_{G_i}$  is a task represented by a 3-tuple  $\langle \mu_j, \lambda_j, \phi_j \rangle$ , where  $\mu_j$  is the Worst-Case Execution Time (WCET) of the task,  $\lambda_j$  is the task period, and  $\phi_j$  is the task start time. Each task  $\tau_j$  corresponds to an actor  $n_j \in N$ . Corollary 2 states also that it is possible to select any hard-real-time scheduling algorithm for asynchronous sets of implicit-deadline periodic tasks to schedule  $\Gamma_\Upsilon$ . Once a scheduling algorithm is selected, we derive the minimum number of processors needed to schedule the applications (See Equations 5, 6, and 7 in [12]), and the mapping of tasks to processors. Deriving the minimum number of processors is based on computing the utilization factors of the tasks. The utilization of a task  $\tau_j$  is  $U_j = \mu_j / \lambda_j$ . Similarly, the utilization of a task-set  $\Gamma_{G_i}$  is  $U_{\Gamma_{G_i}} = \sum_{\tau_j \in \Gamma_{G_i}} U_j$ . The practical applicability of the aforementioned theoretical results will be demonstrated in Section IV.

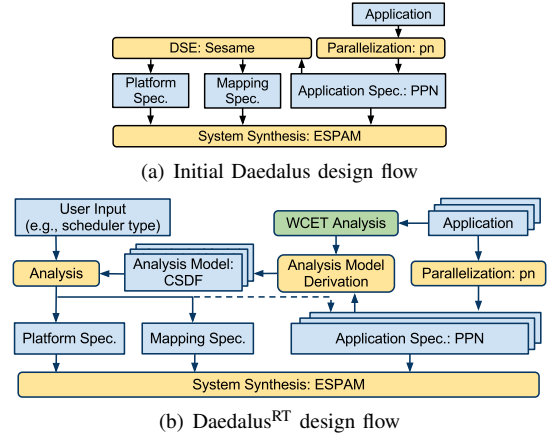


Fig. 2. Comparing the initial Daedalus and Daedalus<sup>RT</sup> flows.

### C. Daedalus Flow

Daedalus [13] is a flow for automated design of embedded streaming systems. An overview of Daedalus is shown in Fig. 2(a). The flow consists of three main phases. The Parallelization phase is realized using the `pn` compiler [5] which takes a single application (in the SANLP form) as an input and produces the PPN model of the application. The Design Space Exploration (DSE) phase is realized using Sesame [17] which takes the PPN model of the application as an input and generates a Pareto-optimal set of design points. A design point is a tuple consisting of platform and mapping specifications. The System Synthesis phase is realized using ESPAM [13] which takes as an input the PPN model of the application together with the platform and mapping specifications, and produces an implementation of the system. ESPAM supports several back-ends such as Xilinx Platform Studio (XPS) for FPGA synthesis.

### IV. PROPOSED METHODOLOGY AND ITS REALIZATION

In this section, we present our design methodology and its realization, i.e., the Daedalus<sup>RT</sup> design flow.

#### A. Overview

Our methodology is based on the initial Daedalus methodology. However, it differs in the following aspects: 1) support for multiple applications. The initial Daedalus flow supports only a single application, whereas our Daedalus<sup>RT</sup> flow supports multiple applications, 2) replacing the complex DSE with very fast yet accurate schedulability analysis to determine the minimum number of processors needed to schedule the applications, and 3) using hard-real-time multiprocessor scheduling algorithms providing temporal isolation to schedule the applications.

An overview of our Daedalus<sup>RT</sup> flow is shown in Fig. 2(b). The DSE phase of Daedalus is replaced with the analysis model derivation (described in Sec. IV-B) and analysis (described in Sec. IV-C) phases. Note that this replacement is possible only under the assumptions described in Sec. I-B. In the analysis model derivation phase, the CSDF graphs equivalent to the PPN models are derived. The derivation phase uses WCET analysis to determine the WCET of each actor. Determining the WCET can be done through either static code analysis or exact measurements on the target platform. After that, the CSDF models are analyzed to determine the platform and mapping specifications. In our methodology, we use CSDF as an analysis model, and PPN for code generation. As mentioned in Sec. III-A, [16] shows that a PPN has an equivalent CSDF graph. However, the authors in [16] do

**Algorithm 1:** Procedure to derive the CSDF model

---

**Input:** A PPN  
**Result:** The equivalent CSDF graph

- 1 Derive the topology of the CSDF graph;
- 2 **foreach** CSDF actor in the CSDF graph **do**
- 3     Derive process variants for its corresponding PPN process;
- 4     Derive a repetitive pattern of process variants;
- 5     **foreach** input/output port of the CSDF actor **do**
- 6         **foreach** process variant in the derived pattern in line 4 **do**
- 7             Generate consumption/production rate;

---

not provide any procedure for deriving the equivalent CSDF automatically. Thus, a procedure for automatic derivation of the equivalent CSDF constitutes a major contribution of this paper.

**B. Deriving the Analysis Model (CSDF)**

The procedure to derive a CSDF graph from its equivalent PPN is depicted in Algorithm 1. It consists of two main steps, namely 1) topology derivation (line 1 in Algorithm 1) and 2) consumption/production sequence derivation for input/output ports of each CSDF actor (line 3-7 in Algorithm 1). Deriving the topology of the CSDF graph is straightforward. That is, the nodes, input/output ports, and edges in the CSDF graph have one-to-one correspondence to those in the PPN. Below, we discuss the second step which derives the consumption/production rates sequences for the input/output port of a CSDF actor.

The second step consists of three sub-steps. In the first sub-step (line 3 in Algorithm 1), for the PPN process corresponding to each CSDF actor, we find the access pattern of the PPN process to its input/output ports. To realize this, we introduce the notion of *process variant*, which captures the consumption/production behavior of the process.

**Definition 1** (Process Variant). A process variant  $v$  of a PPN process is defined by a tuple  $\langle d_v, ports \rangle$ , where  $d_v$  is the variant domain,  $d_v \subseteq d_{process}$ , and  $ports$  is a set of input/output ports.

For example, consider process *snk* shown in Fig. 1(c). One of the process variants is  $\langle d_v, \{IP1, IP3\} \rangle$ , where  $d_v = \{(w, i, j) \in \mathbb{Z}^3 \mid w > 0 \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 2\}$ . According to Definition 1, for all iterations in domain  $d_v$  during the execution of process *snk*, this process always reads data from input ports *IP1* and *IP3*.

The infinite repetitive execution of a PPN process is initially represented by an unbounded polyhedron. (e.g., see  $d_{snk}$  in Sec. III-A). Therefore, we project out dimension  $w$  which denotes the *while*-loop from all the domains because it is irrelevant for the subsequent steps. As a result, the execution of a PPN process is represented by a bounded polyhedron. Algorithm 2 applies standard integer set operations to the domains to derive all process variants of a PPN process. The basic idea is that, each port domain bound to a process function argument is intersected with all other port domains. The intersected domain (line 13 in Algorithm 2) and the difference between two port domains (lines 14 and 15 in Algorithm 2) are then added to the set of process variants. In this way, all process variants are iteratively derived.

Consider process *snk* in Fig 1(c). Its process function  $snk(in1, in2)$  has two arguments represented as a set  $A = \{in1, in2\}$ , which is the input to Algorithm 2. The port domains bound to *in1* are  $d_{IP1}$  and  $d_{IP2}$ , while the port domain bound to *in2* is  $d_{IP3}$ . These port domains are illustrated in Fig. 3(a), surrounded by bold lines. Following the

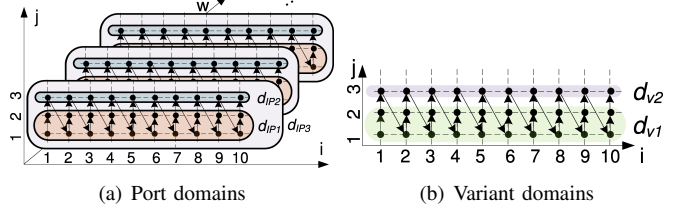
**Algorithm 2:** Procedure to derive process variants of a process

---

**Input:**  $A$ : the set of process function arguments  
**Result:**  $V$ : a set of process variants

- 1  $V \leftarrow \emptyset$ ;
- 2 **foreach**  $a \in A$  **do**
- 3     **foreach**  $d_{port}$  bound to  $a$  **do**
- 4         Project out dimension  $w$  in  $d_{port}$ ;
- 5          $y \leftarrow \langle d_{port}, \{port\} \rangle$ ;
- 6         **if**  $V = \emptyset$  **then**
- 7              $V \leftarrow \{y\}$ ;
- 8         **else**
- 9              $T \leftarrow V$ ;
- 10            **foreach**  $v \in V$  **do**
- 11                 $d_{intersect} \leftarrow v.d_{port} \cap y.d_{port}$ ;
- 12                **if**  $d_{intersect} \neq \emptyset$  **then**
- 13                     $x_{intersect} \leftarrow \langle d_{intersect}, \{v.ports\} \cup \{y.ports\} \rangle$ ;
- 14                     $x_{diff1} \leftarrow \langle v.d_{port} - y.d_{port}, \{v.ports\} \rangle$ ;
- 15                     $x_{diff2} \leftarrow \langle y.d_{port} - v.d_{port}, \{y.ports\} \rangle$ ;
- 16                     $T \leftarrow T \cup \{x_{intersect}\}$ ;
- 17                    **if**  $x_{diff1}.d_{port} \neq \emptyset$  **then**
- 18                         $T \leftarrow T \cup \{x_{diff1}\}$ ;
- 19                    **if**  $x_{diff2}.d_{port} \neq \emptyset$  **then**
- 20                         $T \leftarrow T \cup \{x_{diff2}\}$ ;
- 21                **else**
- 22                     $T \leftarrow T \cup \{y\}$ ;
- 23              $V \leftarrow T$ ;
- 24 **foreach**  $v \in V$  **do**
- 25     **if**  $|input\ ports\ in\ v.ports| \neq |input\ arguments\ in\ A|$  **then**
- 26          $V \leftarrow V \setminus v$ ;

---

Fig. 3. Domains of process *snk* in Fig. 1(c).

procedure described in Algorithm 2, we start with projecting out dimension  $w$  in the port domains, which yields:

$$\begin{aligned} in1 : d'_{IP1} &= \{(i, j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \wedge 1 \leq j \leq 2\} \\ &= \{(i, j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \wedge j = 3\} \\ in2 : d'_{IP3} &= \{(i, j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \wedge 1 \leq j \leq 3\} \end{aligned}$$

Algorithm 2 produces the set of process variants  $V = \{v_1, v_2\}$  shown below:

$$\begin{aligned} v_1 &= \langle d_{v1}, \{IP1, IP3\} \rangle \\ d_{v1} &= d'_{IP1} \cap d'_{IP3} = \{(i, j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \wedge 1 \leq j \leq 2\} \\ v_2 &= \langle d_{v2}, \{IP2, IP3\} \rangle \\ d_{v2} &= d'_{IP2} \cap d'_{IP3} = \{(i, j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \wedge j = 3\} \end{aligned}$$

Process variant domains  $d_{v1}$  and  $d_{v2}$  are also illustrated in Fig. 3(b). Process *snk* reads data from input ports *IP1* and *IP3* in domain  $d_{v1}$ , whereas it reads data from input ports *IP2* and *IP3* in  $d_{v2}$ .

In the second sub-step (line 4 in Algorithm 1), we find a one-dimensional, repetitive pattern of the process variants derived in the first sub-step. To find the repetitive pattern, we first project out dimension  $w$  in the process domain  $d_{process}$  to obtain domain  $d'_{process}$ . Then, we build a sequence of the iterations in  $d'_{process}$  according to their lexicographic order, see the arrows in Fig. 3(b). Next, we replace each iteration in the sequence with the process variant to which the iteration belongs.

For example, for process *snk*, the sequence of the iterations

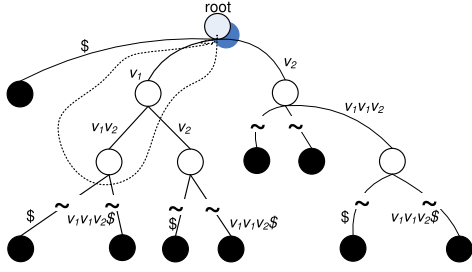


Fig. 4. Suffix tree for the sequence of process variants

in  $d'_{snk}$  is:  $(1, 1), (1, 2), (1, 3), (2, 1), \dots, (10, 3)$  (see the arrows in Fig. 3(b)), where  $d'_{snk} = \{(i, j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \wedge 1 \leq j \leq 3\}$ . The corresponding sequence of the process variants of process  $snk$  is:

$$S_{snk} = \{v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2\}$$

Since the length of the derived sequence is equal to  $|d'_{process}|$ , in general, the derived sequence might be very long. Thus, we express the sequence using the shortest repetitive pattern covering the whole sequence. This shortest repetitive pattern can be found efficiently using a data structure called *suffix tree* [18]. A suffix tree has the following properties: 1) A suffix tree for a sequence  $S$  of length  $L$  can be built in  $O(L)$  time [18], 2) Each edge in the suffix tree is labeled with a subsequence starting from character  $S[i]$  to character  $S[j]$ , where  $1 \leq i \leq j \leq L$ , and 3) No two edges out of a node in the tree can have labels beginning with the same character. Now, we construct a suffix tree for the sequence of process variants. Then, we search the tree for the shortest repetitive pattern that covers the whole sequence  $S$ . To find this pattern, for every path starting from the root to any internal node, we concatenate the labels on the edges. This concatenation results in a subsequence  $S_{sub}$  which occurs  $k$  times in the original sequence  $S$ . Finally, we select the subsequence  $S_{sub}$  with the largest occurrence  $k$  that satisfies:  $|S_{sub}| \times k = |S|$ .

Consider the sequence of the process variants of process  $snk$ . The constructed suffix-tree is illustrated in Fig. 4 (due to space limitation, we only show a part of the suffix tree). The root node is marked with shadowed circle and the leaf nodes are marked with solid circles. The edges are labeled with the subsequences consisting of process variants. The shortest repetitive pattern that covers the whole sequence of process variants is  $\{v_1, v_1, v_2\}$ , surrounded by a dashed line in Fig. 4.

In the last sub-step (lines 5-7 in Algorithm 1), a consumption/production rates sequence is generated for each port of a CSDF actor. This is done by building a table in which each row corresponds to an input/output port, and each column corresponds to a process variant in the repetitive pattern derived in the second sub-step. If the input/output port is in the set of ports of the process variant, then its entry in the table is 1. Otherwise, its entry is 0. Each row in the resulting table represents a consumption/production rates sequence for the corresponding input/output port.

Considering process  $snk$ , the consumption/production rates sequences of CSDF actor  $snk$  are generated as follows:

| Input/output ports | Repetitive pattern |       |       |
|--------------------|--------------------|-------|-------|
|                    | $v_1$              | $v_1$ | $v_2$ |
| $IP1$              | 1                  | 1     | 0     |
| $IP2$              | 0                  | 0     | 1     |
| $IP3$              | 1                  | 1     | 1     |

We see that the consumption/production rates sequences for the ports are the same as the ones shown in Fig. 1(d).

TABLE I  
APPLYING THE SCHEDULABILITY ANALYSIS TO  $\Upsilon$

| App. Task             | $G_1$  |                |                 |                | $G_2$          |                 |                 |                |
|-----------------------|--|----------------|-----------------|----------------|----------------|-----------------|-----------------|----------------|
|                       | src  | filter1        | filter2         | snk            | src            | filter1         | filter2         | snk            |
| WCET ( $\mu$ )        | 5  | 8              | 25              | 4              | 3              | 15              | 15              | 3              |
| Period ( $\lambda$ )  | 18   | 27             | 27              | 18             | 10             | 15              | 15              | 10             |
| $U_j$                 | $\frac{5}{18}$   | $\frac{8}{27}$ | $\frac{25}{27}$ | $\frac{4}{18}$ | $\frac{3}{10}$ | $\frac{15}{15}$ | $\frac{15}{15}$ | $\frac{3}{10}$ |
| $U_{\Gamma G_i}$      | $\frac{15+16+50+12}{54} = \frac{93}{54} \approx 1.72$  |                |                 |                |                |                 |                 |                |
| $U_{\Gamma \Upsilon}$ | $\frac{93+78}{54+30} = \frac{167}{270} \approx 4.32$   |                |                 |                |                |                 |                 |                |
| $M_{OPT}$             | $\lceil U_{\Gamma \Upsilon} \rceil = 5$  |                |                 |                |                |                 |                 |                |
| $M_{PEDF}$            | $\min\{x : B \text{ is } x\text{-partition of } \Gamma_{\Upsilon} \wedge U_y \leq 1 \forall y \in B\} = 5$ |                |                 |                |                |                 |                 |                |
| Mapping               | $P1$   | $P1$           | $P2$            | $P1$           | $P3$           | $P4$            | $P5$            | $P3$           |

### C. Analysis Phase - Deriving the Platform and Mapping Specifications

After deriving the CSDF graphs, it is possible to apply the analysis from [12] to schedule the CSDF actors as periodic tasks. For a set of applications  $\Upsilon = \{G_1, G_2, \dots, G_N\}$ , a super task set  $\Gamma_{\Upsilon}$  can be found according to Equation 1. Upon deriving  $\Gamma_{\Upsilon}$ , we can select any hard-real-time scheduling algorithm [11] for asynchronous sets of implicit-deadline periodic tasks to schedule the actors. Once a scheduling algorithm, denoted  $\mathcal{A}$ , is selected, we derive the minimum number of processors needed by  $\mathcal{A}$  to schedule the tasks, and the mapping of tasks to processors under  $\mathcal{A}$ . The mapping is necessary for partitioned and hybrid schedulers [11] and can be derived, for example, using bin packing allocation. For global schedulers [11], the mapping is optional and serves as an initial mapping. Finally, the PPNs generated in the parallelization phase are updated with the buffer sizes found by Theorem 4 in [12] and the schedule information (indicated with the dashed line in Fig. 2(b)).

For example, let  $\Upsilon$  be a set of applications consisting of two instances  $G_1$  and  $G_2$  of the example application shown in Fig. 1(b). Consider that  $G_1$  and  $G_2$  will run simultaneously on the same platform. For the purpose of illustrating the analysis, consider also that  $G_1$  and  $G_2$  have different WCETs. Table I illustrates how we apply the analysis presented in Section III-B to determine the minimum number of processors needed to schedule the applications and the mapping of tasks to processors.

The WCET ( $\mu$ ) row shows the WCET of each actor measured in time-units. This row together with the CSDF graphs are the inputs to the analysis phase. Then, the analysis proceeds by computing the periods ( $\lambda$ ) according to Lemma 2 in [12]. Afterwards, the utilization of the tasks, task-sets, and super task-set is computed (i.e.,  $U_j$ ,  $U_{\Gamma G_i}$ , and  $U_{\Gamma \Upsilon}$ ). Once the utilization factors have been computed, we compute the number of processors needed to schedule the applications under different algorithms. For instance, if we select the scheduling algorithm  $\mathcal{A}$  to be an optimal one (e.g., Pfair algorithms [11]), then  $\Upsilon$  can be scheduled to meet all the deadlines on 5 processors. If we select  $\mathcal{A}$  to be a partitioned algorithm (e.g., Partitioned Earliest-Deadline-First (PEDF) [11]), then  $\Upsilon$  can be scheduled to meet all the deadlines also on 5 processors. Suppose that  $\mathcal{A}$  is PEDF, then we derive the mapping of tasks to processors by applying first-fit allocation to  $\Gamma_{\Upsilon}$ . The resulting mapping is shown in the mapping row in Table I, where  $P_i$  denotes the  $i$ th processor in the platform.

## V. EVALUATION RESULTS

We evaluate the Daedalus<sup>RT</sup> design flow by performing an experiment to synthesize a system running a set of real-life streaming applications. In this experiment, we illustrate the total time/effort needed to build the system using our

TABLE II  
EXECUTION TIMES FOR THE PHASES OF DAEDALUS<sup>RT</sup> AND [10] FLOWS

|                               | Daedalus <sup>RT</sup> |            | [10]     |            |
|-------------------------------|------------------------|------------|----------|------------|
| Num. of applications          | 3                      |            | 1        |            |
| Phase                         | Time                   | Automation | Time     | Automation |
| Parallelization               | 0.48 sec.              | Yes        | < 3 days | No         |
| WCET analysis                 | 1 day                  | No         | 1 day    | No         |
| Deriving the CSDF             | 5 sec.                 | Yes        | 5 min.   | No         |
| Deriving the platform/mapping | 0.03 sec.              | Yes        | 1 min.   | Yes        |
| System synthesis              | 2.16 sec.              | Yes        | 16 sec.  | Yes        |
| Total                         | ~ 1 day                | -          | ~ 4 days | -          |
| Total (excl. WCET analysis)   | ~ 7.67 sec.            | -          | ~ 3 days | -          |

design flow and the throughput resulting from strictly periodic scheduling. We use three applications which are synthesized to run simultaneously on the same platform. The first application is an edge-detection filter (Sobel operator) from the image processing domain, the second application is the Motion JPEG (M-JPEG) video encoder from the video processing domain, and the third application is the M-JPEG video decoder. We used Xilinx Platform Studio (XPS) back-end to synthesize the platform. The synthesized platform consists of tiles interconnected via an AXI crossbar switch. Each tile in turn consists of a MicroBlaze processor that has its own program, data, and communication memories.

The second and third columns in Table II list the total time needed to execute the Daedalus<sup>RT</sup> flow phases and the automation of the different phases. The experiment was conducted on a machine with Intel Core 2 Duo T9400 processor with 4 GB of RAM running Ubuntu 11.04 (64-bit). The very short time of 0.03 seconds needed for deriving the platform/mapping specifications, see row 7 in column 2, confirms the following advantage of our methodology; the complex DSE is replaced with very fast schedulability analysis.

We also compare our flow to the flow presented in [10] as shown in Table II. The flow in [10] is selected as it is the only related flow which provides detailed information about its execution time to allow direct comparison. We compare both flows under the assumption that the input to both of them is a sequential application written in SANLP form. The authors in [10] reported the execution times of the different phases of the flow when synthesizing a platform running a single application (M-JPEG decoder). In our case, we report the execution times for synthesizing three applications (sobel, M-JPEG encoder and decoder) that run simultaneously on the platform. It can be seen from Table II that our flow outperforms [10] significantly. The reasons are: 1) Our flow provides the benefit of automatic parallelization, as well as automatic derivation of the analysis model, and 2) Our flow performs the analysis to derive the platform/mapping in around 30 milliseconds for a set of three applications. Such fast analysis is the key to handle the growing number of simultaneously running applications on a single platform.

Now, we demonstrate the quality of the generated system. Table III compares the throughput (i.e., rate) guaranteed by our flow, denoted by  $R$ , with the maximum achievable throughput resulting from self-timed scheduling, denoted by  $R_{\max}$ , for the sink actor. The most important column in the table is the last one which shows the ratio of the guaranteed throughput by our flow to the maximum achievable counterpart. It can be seen that our flow guarantees an optimal or near-optimal throughput in all the cases. Since the set of applications is known at design-time, a schedule table can be built for each application at design-time using offline scheduling algorithms [11]. The guaranteed throughput, column 2 in Table III, is achieved using

TABLE III  
GUARANTEED THROUGHPUT VS. MAXIMUM ACHIEVABLE THROUGHPUT

| Application    | $R$      | $R_{\max}$                | $R/R_{\max}$ |
|----------------|----------|---------------------------|--------------|
| Sobel          | 1/125    | $8.474584 \times 10^{-3}$ | 0.94         |
| M-JPEG encoder | 1/126386 | $7.912 \times 10^{-6}$    | 1.0          |
| M-JPEG decoder | 1/140113 | $7.137 \times 10^{-6}$    | 1.0          |

a platform consisting of 7 processors running 17 tasks with a mapping that mixes tasks from different applications on the same processor. The maximum achievable throughput, column 3 in Table III, is computed assuming unlimited resources.

If the set of applications is unknown at design-time or changes at run-time (i.e., by accepting new applications), then an online scheduling algorithm must be used. We implemented a run-time scheduler based on the PEDF algorithm and evaluated its overhead. The worst-case overhead of the scheduler for the set of applications considered in this experiment is 30  $\mu$ s when it is executed on a Xilinx MicroBlaze processor running at 100 MHz. Such low overhead affirms the viability of our methodology. It is important to note that this overhead depends on the specifics of the scheduler implementation and might vary between implementations.

## VI. CONCLUSIONS

We present a novel methodology for the automated design of an MPSoC running a set of hard-real-time streaming applications. We show that this methodology replaces the complex DSE with very fast schedulability analysis. Moreover, this methodology provides automatic derivation of the analysis model. We implement the methodology in the form of the Daedalus<sup>RT</sup> design flow and evaluate it using a set of real streaming applications. Evaluation results show that the methodology reduces significantly the design time and effort while synthesizing systems that achieve near-optimal throughput.

## ACKNOWLEDGMENTS

This work is supported by CATRENE/MEDEA+ TSAR project and the STW Netherlands STreaming (NEST) project.

## REFERENCES

- [1] L. Karam *et al.*, "Trends in multicore DSP platforms," *IEEE SPM*, 2009.
- [2] A. Gerstlauer *et al.*, "Electronic System-Level Synthesis Methodologies," *IEEE TCAD*, 2009.
- [3] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, 1987.
- [4] G. Bilsen *et al.*, "Cyclo-Static Dataflow," *IEEE TSP*, 1996.
- [5] S. Verdoolaege *et al.*, "pn: A Tool for Improved Derivation of Process Networks," *EURASIP J. Embedded Syst.*, 2007.
- [6] J. Castrillon *et al.*, "Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms," in *DATE*, 2010.
- [7] A. Kumar *et al.*, "Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA," *ACM TODAES*, 2008.
- [8] S. Ha *et al.*, "PeaCE: A hardware-software codesign environment for multimedia embedded systems," *ACM TODAES*, 2007.
- [9] A. Shabbir *et al.*, "CA-MPSoC: An automated design flow for predictable multiprocessor architectures for multiple applications," *JSA*, 2010.
- [10] R. Jordans *et al.*, "An Automated Flow to Map Throughput Constrained Applications to a MPSoC," in *PPES*, 2011.
- [11] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, 2011.
- [12] M. Bamakhrama and T. Stefanov, "Hard-real-time scheduling of data-dependent tasks in embedded streaming applications," in *EMSOFT*, 2011.
- [13] H. Nikolov *et al.*, "Systematic and Automated Multiprocessor System Design, Programming, and Implementation," *IEEE TCAD*, 2008.
- [14] K. Jeffay *et al.*, "On non-preemptive scheduling of periodic and sporadic tasks," in *RTSS*, 1991.
- [15] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model*. Springer-Verlag, 1996.
- [16] E. Deprettere *et al.*, "Affine Nested Loop Programs and their Binary Parameterized Dataflow Graph Counterparts," in *ASAP*, 2006.
- [17] A. Pimentel *et al.*, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE TC*, 2006.
- [18] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, 1995.