Generalized Extraction of Real-Time Parameters for Homogeneous Synchronous Dataflow Graphs[1]

Hazem Ismail Ali
CISTER Research Centre/INESC-TEC
Polytechnic Institute of Porto, Portugal
Email: haali@isep.ipp.pt

Benny Akesson
Czech Technical University in Prague
Czech Republic
Email: kessoben@fel.cvut.cz

Luís Miguel Pinho
CISTER Research Centre/INESC-TEC
Polytechnic Institute of Porto, Portugal
Email: lmp@isep.ipp.pt

***Abstract*—Many embedded multi-core systems incorporate both dataflow applications with timing constraints and traditional real-time applications. Applying real-time scheduling techniques on such systems provides real-time guarantees that all running applications will execute safely without violating their deadlines. However, to apply traditional real-time scheduling techniques on such mixed systems, a unified model to represent both types of applications running on the system is required. Several earlier works have addressed this problem and solutions have been proposed that address acyclic graphs, implicit-deadline models or are able to extract timing parameters considering specific scheduling algorithms.**

In this paper, we present an algorithm for extracting real-time parameters (offsets, deadlines and periods) that are independent of the schedulability analysis, other applications running in the system, and the specific platform. The proposed algorithm: 1) enables applying traditional real-time schedulers and analysis techniques on cyclic or acyclic Homogeneous Synchronous Dataflow (HSDF) applications with periodic sources, 2) captures overlapping iterations, which is a main characteristic of the execution of dataflow applications, 3) provides a method to assign offsets and individual deadlines for HSDF actors, and 4) is compatible with widely used deadline assignment techniques, such as NORM and PURE. The paper proves the correctness of the proposed algorithm through formal proofs and examples.

***Keywords*-Multi- and Many-Core Systems; Real-Time Systems; Homogeneous Synchronous Dataflow (HSDF); Hard Real-Time Streaming Dataflow Applications; Algorithms; Cyclic Graphs;**

I. Introduction

Streaming applications are an increasingly important and widespread category of embedded system applications. Many streaming applications have high processing requirements and timing constraints that must be satisfied, e.g., H.264 video decoders [1]. The high processing requirements are satisfied through the adoption of parallelization models, such as the dataflow model of computation [2], that enabled streaming applications to use massive computational power [3]. Dataflow applications traditionally use static scheduling techniques, i.e. Time Division Multiple Access (TDMA) [2], [4]–[8], however, they have recently been shown to work well also with real-time scheduling techniques [9]–[12], run-time budget schedulers [13], [14] and non-starvation-free schedulers [15].

Future real-time embedded systems incorporate mixed application models with timing constraints running on the same multi-core platform. These application models are dataflow applications with latency and throughput constraints and traditional real-time applications modeled as independent tasks. These future mixed embedded systems, e.g. Automotive and Unmanned Air Vehicles [16], require real-time guarantees that all running applications will execute safely without missing their deadlines. An interesting choice is applying traditional real-time scheduling algorithms and associated analysis techniques to achieve these real-time guarantees for such mixed systems.

Mapping and scheduling mixed real-time applications (dataflow and non-dataflow) requires a unified model to represent both types of applications. A task τ_i in a traditional real-time application (non-dataflow) is represented by four parameters. These parameters are execution time C_i, start/release time s_i, period of execution T_i and Deadline D_i. These parameters enable us to apply efficient real-time analysis techniques to verify the system. In contrast, an actor in a dataflow application is represented by a different set of parameters. These parameters are Worst-Case Execution Time (WCET), Production/Consumption rate (P/C) of tokens and the throughput requirement ζ of the application. We need a method to extract the real-time properties of the actors of dataflow applications in the form of $\tau_i = (s_i, C_i, T_i, D_i)$ to be able to apply traditional real-time scheduling techniques.

In this paper, we propose an algorithm for extracting timing parameters (offsets, deadlines and periods) of real-time dataflow applications that assures real-time guarantees for the system. The real-time dataflow applications are represented as cyclic Homogeneous Synchronous Dataflow

[1]This work was partially supported by Portuguese National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project FCOMP-01-0124-FEDER-037281 (CISTER) and by FCT and EU ARTEMIS JU, within project ARTEMIS/0001/2013, JU grant nr. 621429 (EMC^2), under PhD grant SFRH/BD/79872/2011, and Czech Ministry of Education CZ.1.07/2.3.00/30.0034.

(HSDF) graphs with periodic sources. Earlier works have already considered applications represented as Directed Acyclic Graphs (DAGs) [10], [17], pipelines [18], task models without deadlines [15] or limited to implicit-deadlines [10], or application specific analysis [15], [19]. A main advantage of our proposal is that the extraction of the timing parameters is independent of the specific scheduler being used, of other applications running in the system and the details of the particular platform. The proposed algorithm: **1)** enables applying traditional real-time schedulers and analysis techniques on cyclic or acyclic HSDF applications with periodic sources, **2)** captures overlapping iterations, which is a main characteristic of the execution of dataflow applications, by modelling actors as tasks with arbitrary-deadlines, **3)** provides a method to assign offsets and individual deadlines for real-time dataflow actors, and **4)** is compatible with widely used deadline assignment techniques, such as NORM and PURE [18], [20], [21]. We also show that our algorithm give the same results as earlier work in the special case of pipeline applications.

The rest of this paper is organized as follows. Section II provides an overview of related work. Afterwards, Section III explains the main concepts necessary to understand the system model and the proposed algorithm. The proposed algorithm and its validation is detailed in Section IV and V, respectively. Section VI briefly provides experimental results. Finally, we provide some conclusions in Section VII.

II. Related Work

This section reviews techniques for extracting timing parameters (unified model) of task graphs to enable applying real-time schedulers and analysis techniques.

In [10]–[12], the authors provide an analytical framework for computing timing parameters for actors of acyclic Cyclo-Static Dataflow (CSDF) applications with single-input streaming. The actors are considered as implicit-deadline and constrained-deadline periodic tasks in [10] and [11], [12], respectively. In contrast, the proposed approach is more general and can deal with any HSDF graph (CSDF can be converted to an HSDF), single/multiple input, and actors are modelled as arbitrary-deadline tasks. Modelling the application actors as arbitrary-deadline tasks allows capturing overlapping iterations, a main characteristic of dataflow applications that increases the throughput.

Another solution is presented in [18]. The authors presented a deadline assignment approach called ORDER for dependent tasks composing real-time pipeline applications executing on a multi-core system. The proposed approach was considering the problem of scheduling a pipeline such that the end-to-end deadline is met and the amount of required resource capacity was minimal. Contrarily, in this paper we consider the general problem of deadline assignment for dependent tasks composing real-time application graphs, such as DAGs and Directed Cyclic Graphs (DCGs), which are not supported by [18], [20], [21].

In [17], the authors also address the problem of scheduling periodic DAG tasks, each consisting of subtasks. They are assigned individual deadlines and release times such that all subtasks have equal densities. They are scheduled using global Earliest Deadline First (EDF) and partitioned deadline monotonic scheduling. Another approach presented in [19] calculates offsets and deadlines for subtasks in a DAG task based on computing the interference between each subtask and the higher-priority subtasks of all DAG tasks running on the system. In contrast, we consider a more general problem where applications are represented as DCGs and the extraction of the timing parameters is independent of the scheduling algorithm being used. In addition, two different deadline assignment strategies rather than just equal task densities [17] is supported and the calculation of offsets and deadlines is not dependent on other applications running on the system as in [19].

Another technique is presented in [22]. The authors propose an exact characterization of EDF-like schedulers that can be used to correctly schedule dependent tasks, and show how preemptive algorithms, even those that deal with shared resources, can be easily extended to deal with dependencies. This was done by modifying deadlines D_i in a consistent manner so that a run-time algorithm, such as EDF, could be used without violating the dependencies. Also, [23] propose a similar approach by modifying the timing parameters of the tasks. However, this parameter modification is not only for the deadline D_i of the tasks, but also include modification of the task start time s_i. However, both works consider task parameters as already defined, which is not the case in our problem. Moreover, they are only concerned with uniprocessor platforms.

Also in [8], the authors present a method to calculate individual deadlines of HSDF actors. The method is based on an integer linear programming (ILP) optimization problem that finds the amount of slack for each actor that makes it able to extend its execution without violating the HSDF throughput and timing constraints. However, their proposed method is restricted to strongly connected HSDF graphs and the actor's offsets are calculated based on the static-order schedule of the application. In contrast, our proposed algorithm is neither restricted to strongly connected graphs nor does the offset calculation require static-order scheduling.

In [15], the authors propose a temporal analysis for dataflow application modeled as cyclic HSDF graphs under a non-starvation-free scheduler i.e. static-priority preemptive scheduler (SPP). To apply the analysis they extract timing properties like jitter (difference between best-case and worst-case offsets), periods, and execution times, but not deadlines, since SPP schedulers depend on periods not deadlines. The calculated jitter is based on the interference from the set of high-priority tasks with the task being analyzed running on the same platform. This means that the timing parameters

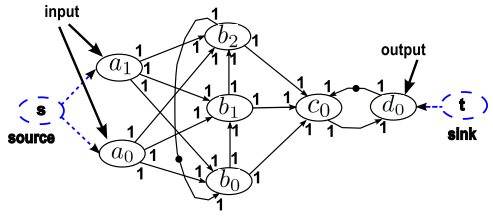


Figure 1: HSDF graph after adding source s and sink t.

calculated are dependent on the set of applications running on the platform. Contrarily, our approach is independent of the scheduler being used and other applications running on the same platform, since our proposed algorithm transforms the HSDF actors into a set of independent tasks that enables any bin-packing heuristic to be applied for mapping them on the platform.

III. Background / Preliminaries

In this section, we present background material that is essential for understanding the computational model, the system model and the proposed algorithm.

A. Homogeneous SDF (HSDF)

A synchronous dataflow application graph is a data-driven network of *actors* (nodes), where the same behavior repeats in each actor every time it is fired. An actor *fires* (executes) once all its input ports have the required *tokens* (data) for consumption. Therefore, the firing pattern of an actor depends on the arrival pattern of its input tokens. If the arrival pattern of the input tokens is periodic the firing pattern of the actor is also periodic, if it is sporadic the actor firing pattern is sporadic, and so on. Each actor has production and consumption rates associated with its ports that determine the number of input and output tokens produced and consumed in the firing process.

Homogeneous Synchronous Dataflow (HSDF) [2] is a special case of synchronous dataflow graphs in which all production and consumption rates associated with actor ports are equal to one. Therefore, when each actor is fired once, the distribution of tokens on all channels return to their initial state. This is referred to as a *complete cycle* or a *graph iteration*. Figure 1 shows an example of an HSDF graph.

Any HSDF application can be formally represented by a Directed Cyclic Graph (DCG) $G = \langle V, E, d \rangle$, where V is the set of nodes, E the edges connecting them and d the set of delays (initial tokens) on the edges of the graph. Each node in this graph is an actor and each edge is a communication channel. An HSDF application has a throughput requirement and a single or multiple latency constraints that must be satisfied for the correct execution of the application. The throughput requirement ζ_i is a performance measure that determines the minimum output data rate of the application (iterations per time unit). In contrast, the latency constraints D_{xy} are defined as actor-to-actor deadlines (maximum timing constraints) between firings of any two actors v_x and v_y in the same iteration that have a single or multiple route(s) between them, referred to as a *time-constrained path* P. Fundamentally, the D_{xy} is required to be greater than or equal to the sum of execution times C_i of all actors on the time-constrained path for the application to be schedulable. Formally, a time-constrained path P is defined as follows:

$$P = \{\langle v_x, \ldots, v_y \rangle : v \subseteq V\} \tag{1}$$

where, its latency constraint

$$D_{xy} \geq \sum_{\substack{i=x, \\ \forall v_i \in P}}^{y} C_i \tag{2}$$

If P is cyclic, it terminates in the last node before reaching an already visited node. For example, in Figure 1 (a_0, b_0, c_0, d_0) is not cyclic, because it starts at actor a_0 and ends at actor d_0, while (b_0, b_1, b_2) is cyclic because it terminates at actor b_2 before repeating itself again. Each time-constrained path P has a latency constraint D_{xy}, where x and y represent the indices of the start and end actors, respectively. For example, assume that the application in Figure 1 has two latency constraints $D_{a_0 d_0}$ and $D_{a_1 d_0}$. All time-constrained paths must start with either actors a_0 or a_1 and end up with actor d_0, any other combinations are regular paths and are not further considered in this paper.

Note that this work considers HSDF graphs with periodic sources. Also, we consider initial tokens d on back edges only. Other dataflow graphs, e.g. SDF [2] and CSDF [24] are examples of more expressive models that can be converted to an equivalent HSDF graph by using a conversion algorithm, such as the one presented in [25]. This also enables these models to be used with our approach.

B. Classical real-time model

This work considers extracting the timing parameters of real-time applications modelled as HSDF graphs, which implies changing its execution behaviour from being data-driven to being time-triggered. This means that actors are activated at their release time parameter s_i eliminating jitter effect in their execution. Formally, we consider a system $\Psi = \langle \Pi, A \rangle$ based on a homogeneous symmetrical multi-core platform, represented by the set $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$, where n is the number of cores. The platform runs a set of m periodic applications $A = \{A_1, A_2, \ldots, A_m\}$. In this model, we assume that all the applications A have periodic input sources. Therefore, each actor v_j in any application A_i can be considered a periodic task. All actors can be scheduled on Π using traditional real-time schedulers.

A periodic task $\tau_i \in V$ is represented by the 4-tuple $\tau_i = (s_i, C_i, T_i, D_i)$, where s_i is a fixed offset that specifies the start instant of an actor, C_i is the worst-case execution time, T_i is the relative period and D_i is the relative deadline

of the task. The absolute deadline $\overline{D}_i$ of task τ_i is defined as $\overline{D}_i = s_i + D_i$. The *utilization* of task τ_i is denoted by U_i and is defined as $U_i = C_i/T_i$, where $U_i \in (0, 1]$. Additionally, the *density* of task τ_i is denoted by ρ_i and is defined as $\rho_i = C_i/D_i$, where $\rho_i \in (0, 1]$. All tasks are modelled as arbitrary-deadline tasks. Other more recent task models, i.e. graph-based models [26], may also be suitable for modelling HSDF graphs, but this will be subject to future research.

C. Deadline assignment strategies

The problem of assigning individual deadlines to dependent tasks of a *pipeline application* A_p, represented by the graph $G_p = \langle V_p, E_p \rangle$, distributed on multiple processors using its end-to-end deadline has been addressed in previous research [18], [20], [21]. The pipeline application consists of a set of tasks (actors) V_p that execute in sequence. The application has a latency constraint D_{xy} that represents the end-to-end deadline of A_p, where v_x and v_y is the start and end task of A_p, respectively. Therefore, the pipeline application graph G_p contains a single time-constrained path P with a latency constraint D_{xy}. In this paper, we support two well-known deadline assignment methods for pipelines that will be used by our proposed algorithm. These methods are:

1) The NORM method **[20], [21]:** is an assignment strategy to divide the end-to-end deadline D_{xy} of a pipeline proportionally to the computation time of its tasks. Therefore, the individual deadline of a task in a pipeline D_i is computed as follows:

$$D_i = \frac{C_i}{\sum_{\forall v_j \in P} C_j} \cdot D_{xy} \tag{3}$$

From Equation (3), the NORM method assigns individual deadlines D_i to tasks with the same end-to-end deadline D_{xy}, such that all tasks have equal densities ρ_i.

$$\rho_i = \frac{C_i}{D_i} = \frac{\sum_{\forall v_j \in P} C_j}{D_{xy}} \tag{4}$$

2) The PURE method **[20], [21]:** a different deadline assignment strategy based on the distribution of the laxity ε equally among all tasks of the pipeline, such that each task have slack δ. The laxity ε on the time-constrained path P is defined as follows:

$$\varepsilon = D_{xy} - \sum_{\forall v_j \in P} C_j \tag{5}$$

Then, the slack δ of the tasks is equal to:

$$\delta = \frac{\varepsilon}{|V_p|} \tag{6}$$

where $|V_p|$ is the number of tasks in the pipeline. Therefore, the individual deadline of a task in a pipeline D_i is computed as follows:

$$D_i = C_i + \delta \tag{7}$$

Therefore,

$$D_i = C_i + \frac{D_{xy} - \sum_{\forall v_j \in P} C_j}{|V_p|} \tag{8}$$

From Equation (7), the PURE method assigns individual deadlines D_i, such that tasks have relative densities ρ_i. This means, a task with high C_i have high ρ_i relative to a task with small C_i.

$$\rho_i = \frac{C_i}{D_i} = \frac{C_i}{C_i + \delta} \tag{9}$$

IV. Timing Parameters Extraction Algorithm

The algorithm presented in this section is intended for extracting the timing parameters (s_i, C_i, T_i, D_i) of HSDF applications with periodic sources. The algorithm, presented in Section IV-C, is divided into two phases. The first phase, finds all time-constrained paths in the graph, while second phase extracts the timing parameters of individual actors. However, before going into details, we introduce a key concept that our algorithm is based on called *path sensitivity* (Section IV-A). Then, we introduce two techniques for deriving latency constraints (Section IV-B) for cyclic paths and an end-to-end latency constraint in case it is unspecified for an application. These techniques ensure that the throughput requirement ζ_i is satisfied after conversion to the unified model.

A. Path sensitivity

In this section, we define a key concept in our algorithm called *path sensitivity*, that enables supporting general HSDF graphs. Dealing with actors in general graphs implies that an actor can be present on multiple time-constrained paths of the graph. The path sensitivity parameter helps in addressing this problem by determining the order in which to consider the time-constrained paths when extracting the timing parameters.

Path sensitivity γ: is a measure of the criticality of a time-constrained path with respect to a certain parameter, e.g. utilization or density. In our case, the path sensitivity γ represents the time-constrained path *density*. The density is the measure of how tight the latency constraint D_{xy} is for a time-constrained path P compared to its execution time. γ is in the range $(0, 1]$ (because of the relation in Equation (2)), where higher values indicate higher sensitivity. It is calculated as follows:

$$\gamma = \sum_{\forall v_j \in P} \frac{C_j}{D_{xy}} \tag{10}$$

In case of NORM, substituting Equation (10) in Equation (3) gives:

$$D_i = \frac{C_i}{\gamma} \tag{11}$$

by solving for γ and substituting Equation (4) in Equation (11)

$$\rho_i = \gamma \tag{12}$$

This means that all tasks τ_i on the same time-constrained path P have densities ρ_i equal to the path sensitivity γ.

In case of PURE, substituting Equation (10) in Equation (8) gives:

$$D_i = C_i + \delta = C_i + \frac{(1-\gamma) \cdot D_{xy}}{|P|} \tag{13}$$

by dividing Equation (13) by D_i, then substitute by Equation (4) and solving for ρ_i

$$\rho_i = 1 - \frac{\delta}{D_i} = 1 - \frac{(1-\gamma) \cdot D_{xy}}{|P| \cdot D_i} \tag{14}$$

From Equations (11), (12), (13) and (14), we can draw two conclusions. First, *there is an inverse relation between the path sensitivity γ and the task relative deadline D_i for both NORM and PURE*. This conclusion is obvious from Equation (11). In case of Equation (13), since $0 < \gamma \leq 1$, an increase in the value of γ decreases the value of D_i and vice versa, confirming the inverse relation. Second, *when the sensitivity γ of a time-constrained path increases, the value of its task densities ρ_i increases too*. This is confirmed from Equations (12) and (14) and the first conclusion.

B. Deriving latency constraints

In this section, we present two techniques for deriving latency constraints for HSDF graphs. First, we derive latency constraints for cyclic paths. We then derive end-to-end latency constraints in case it is not specified in by the application.

1) Deriving cycle latency constraints: HSDF applications can have several cycles in its graph. Each cycle requires a latency constraint that satisfies the throughput requirement ζ_i of the application. A quick choice for a cycle latency constraint D_{xy}^{cycle} value is the period of the application A_i. However, such a choice of latency constraint ignores the number of tokens d involved in the cycle and limits possible pipeline parallelism in the application. Therefore, the latency constraint of a cyclic time-constrained path D_{xy}^{cycle} must take into account the number of tokens involved in this cycle d_{cycle} such that the application throughput ζ_i is not violated. The latency constraint for a cyclic time-constrained path is defined as follows [8]:

$$D_{xy}^{cycle} = C_{cycle} + (\frac{1}{\zeta_i} - \frac{C_{cycle}}{d_{cycle}}) \cdot d_{cycle} = \frac{d_{cycle}}{\zeta_i} \tag{15}$$

where C_{cycle} is the summation of execution times of the actors involved in the cycle. The latency constraint of a cycle tells us how much the execution of the actors on the cycle as a whole can be extended while still guaranteeing the desired application throughput ζ_i.

2) Deriving end-to-end latency constraint: Our proposed algorithm requires an end-to-end latency constraint for each HSDF application to satisfy the precedence constraints and the throughput requirement. In case of an HSDF application without a specified end-to-end latency constraint D_{xy}, we derive it as follows:

$$D_{xy} = \max\{T_i, \beta \cdot \sum_{\forall v_i \in CP} C_i\} \tag{16}$$

As we can notice D_{xy} is set to the maximum of two values. The first, the application period T_i (extracted from the inverse of its throughput requirement $1/\zeta_i$) which is used in case of low-throughput applications where $T_i \geq \beta \cdot \sum_{\forall v \in CP} C_i$. The second, is the sum of the C_i of actors in the critical path (CP) of the application multiplied by a constant β, where the critical path (CP) of an application is defined as its longest execution path from input to output. In contrast, the second value is used in case of high-throughput applications, where $T_i < \beta \cdot \sum_{\forall v \in CP} C_i$. The β constant has a value that ranges $[1, \infty)$. Selecting $\beta = 1$ results in unnecessarily tight actor deadline values and increases the total density of the application that makes it more critical and hard to schedule with other applications, since the actors in the application CP have $\rho_i = 1$. On the other hand, selecting higher values of β relaxes the criticality of the application and eases its schedulability with other applications. A good value for β that we use in this paper is when the sensitivity of the CP of the application γ_{CP} is equal to the maximum sensitivity of all the cycles γ_{cycle} in the application,

$$\max_{\forall cycle \in G}\{\gamma_{cycle}\} = \gamma_{CP} = \sum_{\forall v_j \in CP} \frac{C_j}{D_{xy}} = \frac{\sum_{\forall v_j \in CP} C_j}{\beta \cdot \sum_{\forall v_j \in CP} C_j} \tag{17}$$

At this value of β, the execution time of all cycles in the application graph can be extended to the maximum possible limit (latency constraint computed in Equation (15)) while still satisfying its throughput requirement ζ. Therefore, solving for β in Equation (17) defines it as:

$$\beta = \frac{1}{\max_{\forall cycle \in G}\{\gamma_{cycle}\}} \tag{18}$$

C. Proposed algorithm

In this section, we present our proposed algorithm for extracting timing parameters of HSDF applications with periodic sources. The algorithm consists of two phases: **1)** finding all time-constrained paths and **2)** extracting timing parameters. The following sections explain these two phases in detail.

1) First phase: Finding all time-constrained paths: In this phase, we calculate all time-constrained paths for a given HSDF in non-increasing order of sensitivities. A time-constrained path in an HSDF can be between any two actors that have a latency constraint.

The first phase of the algorithm is divided into two stages: ***1) Creation of source and sink actors:*** First, we search the graph G to find all input (output) actors. Actors associated with the input (output) data stream are specified as the starting-actors (ending-actors), respectively. A dummy source s (sink t) actor that has a zero execution time is

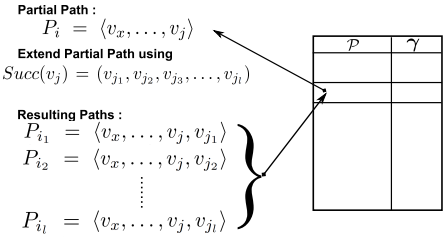


Figure 2: Enumeration of time-constrained paths.

inserted at the beginning (end) of the graph G, as shown in Figure 1. These two actors (s, t) are connected with dummy links to starting and ending actors, respectively. Adding these dummy actors with their edges converts the graph into a canonical form, since all the paths that traverse the graph from the input to the output of the graph have a uniform form that starts with s and ends with t. This is helpful when traversing multi-input/multi-output graphs, as shown in Figure 1.

2) Path enumeration: This is an iterative process where all time-constrained paths between source s and sink t actors in the HSDF are generated. In case of having latency constraints between two specific actors, the path enumeration phase generates all time-constrained paths between these two actors in addition to the ones generated from s to t. The set of all time-constrained paths between actors with latency constraints is called $\mathcal{P}$, which is arranged in non-increasing order of path sensitivities γ. It is defined as follows:

$$\mathcal{P} = \{\langle P_i, \gamma_i \rangle : V, \gamma_{i-1} \geq \gamma_i, \gamma \in (0, 1]\} \tag{19}$$

The process starts by initializing $\mathcal{P}$ with a few *partial paths*. In this case, these initial partial paths are all single hop paths generated by combining the start actors with the elements in their list of successor actors. The list of successor actors is a set of child actors that are one hop away from their parent. For example, in Figure 1, the list of successor actors for source actor s is $Succ(v_s) = (a_0, a_1)$. The starting actors can be the source actor s or any actor that starts a set of time-constrained paths v_x with a specific latency constraint D_{xy}. The list of successor actors $Succ(v_x)$ is defined as follows:

$$Succ(v_x) = (v_{x_1}, v_{x_2}, v_{x_3}, \ldots, v_{x_l}) \tag{20}$$

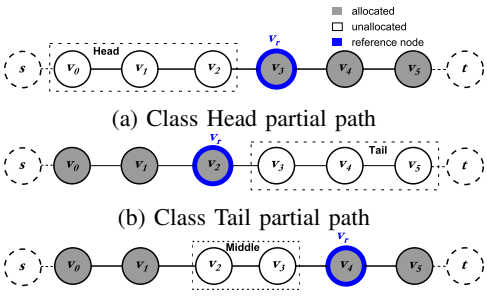
where l is the number of actors in $Succ(v_x)$. Then, the process picks up a partial path $P_i = \langle v_x, \ldots, v_j \rangle$ from $\mathcal{P}$, where v_j is not equal to the end actor v_y, and extends it to a full path (Equation (1)) as shown in Figure 2. The extension process starts by getting the $Succ(v_j) = (v_{j_1}, v_{j_2}, v_{j_3}, \ldots, v_{j_l})$. Then, it extends the partial path P_i to its l possible extended paths, $P_{i_1} = \langle v_x, \ldots, v_j, v_{j_1} \rangle, P_{i_2} = \langle v_x, \ldots, v_j, v_{j_2} \rangle, \ldots, P_{i_l} = \langle v_x, \ldots, v_j, v_{j_l} \rangle$. It then removes P_i and inserts its l possible continuations in the $\mathcal{P}$ set in non-increasing order of sensitivity. The path enumeration process continues until all partial paths in $\mathcal{P}$ are extended to full time-constrained paths.

Algorithm 1: Extracting timing parameters of HSDF

$\boldsymbol{P_i}$: A full time-constrained path in $\mathcal{P}$ set.
$\boldsymbol{D^i_{xy}}$: deadline constraint between actor v_x and actor v_y on a full time-constrained path P_i.
$\boldsymbol{\mathcal{P}}$: totally ordered set of all time-constrained paths of an application ordered according to γ, $\mathcal{P} = \{P_i : \gamma_{i-1} \geq \gamma_i\}$.
$\hat{\boldsymbol{\mathcal{P}}}$: totally ordered set of time-constrained paths from s to t of an application ordered according to D_{xy}, $\hat{\mathcal{P}} \subseteq \mathcal{P}$, $\hat{\mathcal{P}} = \{P_i : v_x = s, v_y = t, [D^{i-1}_{xy} > D^i_{xy}$ *or* $\{D^{i-1}_{xy} = D^i_{xy}, \gamma_{i-1} \geq \gamma_i\}]\}$.
$\boldsymbol{P^H_i}$: set of higher sensitivity time-constrained paths than P_i, $P^H_i = \{\langle P_1, \ldots, P_{i-1} \rangle : \gamma_{i-1} \geq \gamma_i\}$
$\boldsymbol{X_i}$: set of shared actors between P_i with higher sensitivity time-constrained paths set P^H_i, $X_i = \{v_k : v_k \in P_i, v_k \in P_j \in P^H_i\}$
$\boldsymbol{p_i}$: partial path in time-constrained path P_i.

begin
  `// Actor deadline assignment`
  foreach P_i **in** $\mathcal{P}$ **do**
    if *(*$\forall v_j \in P_i, D_j = \varnothing$*)* **then**
      foreach v_j **in** P_i **do**
        $D_j = dead\_assign(D^i_{xy})$; `// NORM/PURE`
    else `//` $X_i \subseteq P_i$
      foreach v_j **in** $P_i - X_i$ **do**
        $D_j = dead\_assign(D^i_{xy} - \sum_{\forall v_k \in X_i} D_k)$;
        `// NORM/PURE`
  `// Actor offset assignment`
  foreach P_i **in** $\hat{\mathcal{P}}$ **do**
    if *(*$\forall v_j \in P_i, s_j = \varnothing$*)* **then**
      $s_0 = 0$;
      foreach v_j **in** P_i, $j = 1..sizeof(P_i)$ **do**
        $s_j = s_{j-1} + D_{j-1}$;
        $T_j = 1/\zeta_{A_i}$
    else
      Determine all $p_i \in P_i$ with $s_j = \varnothing$.
      Determine reference actor v_r.
      foreach p_i **in** P_i **do**
        if *(*p_i **is** *Head or Middle)* **then**
          foreach v_j **in** p_i **do**
            $v_r = v_{j+1}$;
            $s_j = s_r - D_j$;
            $T_j = 1/\zeta_A$;
        else
          foreach v_j **in** p_i **do**
            $v_r = v_{j-1}$;
            $s_j = s_r + D_r$;
            $T_j = 1/\zeta_A$;
  `// Validation check`
  foreach P_i **in** $\mathcal{P}$ **do**
    if *((* $\sum_{\forall v_j \in P_i} D_j \leq D^i_{xy}$*) &* $(s_y + D_y - s_x \leq D^i_{xy})$ *)* **then**
      $Algorithm\ Succeeds$;
    else
      $Algorithm\ Fails$;

2) Second phase: Extracting timing parameters: The second phase, shown in Algorithm 1, repeats for each application in the application set A. It picks a time-constrained path P_i in order of sensitivity from $\mathcal{P}$. The selected path P_i is checked whether or not it has actors v_j with assigned deadlines D_j. If P_i has no actors with assigned deadlines ($\forall v_j \in P_i$), the algorithm assigns individual deadlines D_j for the actors v_j using *dead_assign()* function that

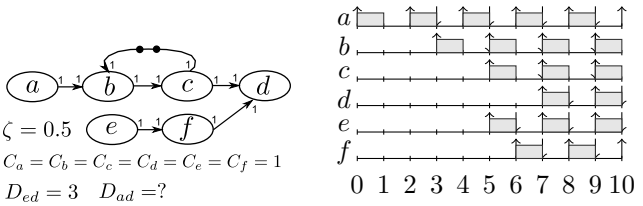


(a) Class Head partial path

(b) Class Tail partial path

(c) Class Middle partial path

Figure 3: Partial path classes for offsets setting



(a) HSDF application.

(b) HSDF timing diagram.

Figure 4: HSDF example.

implements either NORM or PURE (Equations (3) or (8), respectively), using the corresponding latency constraint D^i_{xy}.

On the other hand, if P_i has a set of actors with assigned deadlines X_i (shared actors v_k with any previously processed time-constrained paths), the algorithm assigns individual deadlines D_j to the unassigned actors v_j using either NORM or PURE based on the corresponding latency constraint, which is the difference between D^i_{xy} and the sum of individual deadlines D_k already assigned to actors, $(D^i_{xy} - \sum_{\forall v_k \in X_i} D_k)$. In all cases, the period of the actor T_j is derived from the throughput constraint ζ_A of the application. It is defined as follows:

$$T_j = 1/\zeta_A \tag{21}$$

This follows naturally for an HSDF graph, since each actor executes only once per iteration by definition.

Once the application A_i actors relative deadline are determined, the offset of the actors s_j are calculated in a similar fashion. Algorithm 1 generates a new set $\hat{\mathcal{P}} \subseteq \mathcal{P}$ containing time-constrained paths that include s and t actors only. $\hat{\mathcal{P}}$ is arranged in a non-increasing order of D_{xy}. If two paths have the same D_{xy}, they are ordered in a non-increasing order of γ. The algorithm picks a time-constrained path P_i from $\hat{\mathcal{P}}$. If the path has no actors with assigned offsets, it assigns offsets s_j for the actors v_j on the path in the direction from s to t as follows:

$$s_j = s_{j-1} + D_{j-1} \tag{22}$$

If time-constrained path P_i has a set of actors with assigned offsets (actors assigned in previously processed paths), the algorithm traverses P_i in a search for partial path segments p_i of actors with unassigned offsets. Once they are listed, the algorithm determines the reference actors v_r and classify them into one of three types: *Head*, *Middle* or *Tail*, as shown in Figure 3. This information is used to calculate the offsets s_j, as shown in Algorithm 1. If the partial path p_i is of type *Head* or *Middle*, the reference actor v_r is always on the right hand side of p_i, as shown in Figures 3a and 3c, and the offsets of p_i actors are assigned using the following equation:

$$s_j = s_r - D_j \tag{23}$$

Offset assignment of *Head* and *Middle* in this way instead of traversing the path from s to t assigning offsets using Equation (22), enables larger offset values to be assigned to actors delaying their execution allow satisfying wider range of latency constraints, as we show in Section IV-D.

If the partial path p_i is type *Tail*, the reference actor v_r is always on the left hand side of p_i, as shown in Figure 3b, and the offsets of p_i actors are assigned using the following equation:

$$s_j = s_r + D_r \tag{24}$$

After assigning deadline and offsets for the application actors, the algorithm checks the application for the validity of the assigned values and that they do not violate the latency constraints specified.

Finally, we can conclude that Algorithm 1 preserves relative deadline values D_j computed from high-sensitivity time-constrained paths. This is clear from determining the actors with unassigned deadlines in P_i, and their corresponding latency constraint $(D^i_{xy} - \sum_{\forall v_k \in X_i} D_k)$, leaving the preassigned set of actors X_i untouched. In case of using deadline-based schedulers, this property makes actors in high-sensitivity time-constrained paths have a higher priority compared to actors in low-sensitivity time-constrained paths, since they have tighter deadlines (as concluded from Equations (11) and (13)).

D. Example

In this section, we present an example, illustrated in Figure 4, to show how to apply our proposed algorithm step-by-step. The following paragraphs explains this in detail.

Figure 4a shows an HSDF graph application comprising six actors (a, b, c, d, e, f) with execution times of all actors equal to 1, throughput requirement $\zeta = 0.5$, and two end-to-end latency constraints, one is specified $D_{ed} = 3$, while the other D_{ad} is not. The example HSDF graph is not trivial, as it features multiple input actors a and e, a cycle, and multiple initial tokens. Applying the first phase of our proposed algorithm results in three time-constrained paths. The first time-constrained path is $P_1 = \langle e, f, d \rangle$

with an end-to-end latency constraint $D^1_{ed} = 3$ and sensitivity $\gamma_1 = 1$. The second time-constrained path is $P_2 = \langle b, c\rangle$ which represents a cycle in the graph with a latency constraint $D^2_{bc} = 4$ calculated by substituting with $C_{cycle} = C_b + C_c = 2$, $\zeta = 0.5$ and number of tokens in the cycle $d = 2$ in Equation (15). The sensitivity of P_2 is hence $\gamma_2 = 0.5$ (Equation (10)). The third time-constrained path is $P_3 = \langle a, b, c, d\rangle$ with a latency constraint D^3_{ad} equal to the second end-to-end deadline, which is not specified by the application. Therefore, we calculate D^3_{ad} using Equation (16) ($\beta = 1/\gamma_2 = 2$, Equation (18)) that results in $D^3_{ad} = 8$ and its sensitivity is $\gamma_3 = 0.5$. Therefore, the set of all possible time-constrained paths is $\mathcal{P} = \{\langle P_1, \gamma_1\rangle, \langle P_2, \gamma_2\rangle \langle P_3, \gamma_3\rangle\} = \{\langle (e, f, d), 1\rangle, \langle (b, c), 0.5\rangle, \langle (a, b, c, d), 0.5\rangle\}$.

The second phase of the proposed algorithm picks up P_1 and assigns individual deadlines to actors (e, f, d) equal to $(D_e = 1, D_f = 1, D_d = 1,)$, respectively, for both NORM and PURE. Picking up the next time-constrained path P_2 for deadline assignment results in $(D_b = 2, D_c = 2)$. Finally, picking up the last time-constrained path P_3 for deadline assignment results in $(D_a = 3)$.

For offset assignment, the algorithm creates the set of time-constrained paths that goes from source s to sink t, ordered according to the constraint $[D^{i-1}_{xy} > D^i_{xy}$ *or* $\{D^{i-1}_{xy} = D^i_{xy}, \gamma_{i-1} \geq \gamma_i\}]$, $\hat{\mathcal{P}} = \{\langle P_3, D^3_{ad}\rangle, \langle P_1, D^1_{ed}\rangle\}$. First, it picks the time-constrained path with the longest delay P_3 for offset assignment. Since none of its actors have assigned offsets, the actor offsets are $(s_a = 0, s_b = 3, s_c = 5, s_d = 7)$. Then, it picks P_1 where one of its actors d has already assigned offset s_d equal to 7. It discovers a single partial path of type *Head* in P_1 which is $p_1 = (e, f)$. The reference actor for p_1 is actor d. Therefore, the offsets of actors e and f are $(s_e = 5, s_f = 6)$, respectively. As noted, actor e is triggered at time $(s_e = 5)$ even though its input data is available from time instance zero to satisfy the latency constraint $(D_{ed} = 3)$ of the application. For the periods, $(T_a = T_b = T_c = T_d = T_e = T_f = 1/\zeta = 2)$. Therefore, the extracted timing parameters (s_i, C_i, T_i, D_i) for the graph actors $\{a, b, c, d, e, f\}$ are $\{(0, 1, 2, 3), (3, 1, 2, 2), (5, 1, 2, 2), (7, 1, 2, 1), (5, 1, 2, 1),$ $(6, 1, 2, 1)\}$, respectively. These extracted parameters preserve the precedence, throughput and latency constraints of the HSDF application, indicated in the timing diagram in Figure 4b. The timing diagram also shows that multiple iterations of the graph execute in parallel assuming at least three processors are available.

V. Validation of the proposed approach

This section validates the proposed algorithm by proving that it assigns individual deadlines for actors of any application graph such that it respects all its latency constraints. First, we start by the following property driven from the inverse relationship between γ and D_v (concluded from Equations (11) and (13)):

Property 1. *If there are two time-constrained paths P_i and P_j, where $\gamma_i > \gamma_j$ and there is a shared actor v between them. The deadline value D^i_v computed for actor v on P_i is less than the value D^j_v computed for the same actor on P_j, $D^i_v < D^j_v$.*

Another important property of the deadline assignment strategies NORM and PURE, derived from Equations (3) and (8) is:

Property 2. *A time-constrained path P with a latency constraint D_{xy}, whose actors v_j are assigned individual deadlines D_j, using NORM or PURE, has the following property:*

$$D_{xy} = \sum_{\forall v_j \in P} D_j \tag{25}$$

From Property 2, it follows that applying Algorithm 1 on any time-constrained path P, whose actors has no assigned deadlines, results in a time-constrained path that satisfies its latency constraints. This is for the simple case where the actors in P has no assigned deadlines. However, when P shares some actors with higher sensitivity time-constrained paths the situation gets more complex. Lemma 1 proves the correctness of this case.

Lemma 1. *If a time-constrained path P_i with a latency constraint D^i_{xy}, has a set of actors X_i shared with higher sensitivity time-constrained paths $P^H_i = \langle P_1, \ldots, P_{i-1}\rangle$ in an application graph G, Algorithm 1 assures that the sum of individual deadlines D_j of actors in P_i is equal to $D^i_{xy} = \sum_{\forall v_j \in P_i} D_j$.*

Proof: Let us assume a time-constrained path $P'_i = P_i$, except that all its actors v'_j have $D'_j = \varnothing$ (empty element). Assigning individual deadlines D'_j to the actors of time-constrained path P'_i using either NORM or PURE (Equations (3) and (8)) and its latency constraint D^i_{xy} under the system model constraint specified in Equation (2) then

$$\forall v'_j \in P'_i, \quad D'_j \geq C_j, \quad D^i_{xy} = \sum_{\forall v'_j \in P'_i} D'_j \tag{26}$$

The set of shared actors X_i in P_i has a sum of individual deadlines equal to d.

$$d = \sum_{\forall v_j \in X_i} D_j, \quad \forall v_j \in X_i, D_j \geq C_j \tag{27}$$

Here, d represents the value calculated from the higher sensitivity time-constrained paths P^H_i. Let us assume d' represents the value calculated for the same set of actors X_i on time-constrained path P'_i. Then, from Property 1:

$$d < d' \tag{28}$$

And,

$$D^i_{xy} - d > D^i_{xy} - d' \tag{29}$$

Again, let us assume that the sum of computation time of actors in X_i is c.

$$c = \sum_{\forall v_j \in X_i} C_j \tag{30}$$

Then, from Equation (27)

$$d \geq c \tag{31}$$

And, since the summation of individual deadlines of actors in P'_i such that $v'_j \in P'_i - X_i$ is

$$\sum_{v'_j \in P'_i - X_i} D'_j = D^i_{xy} - d' \tag{32}$$

Therefore, from Equations (26) and (29)

$$D^i_{xy} - d > D^i_{xy} - d' \geq \sum_{\forall v_j} C_j - c \tag{33}$$

Also, it can be written as

$$D^i_{xy} - \sum_{\forall v_j \in X_i} D_j > D^i_{xy} - \sum_{\forall v'_j \in X_i} D'_j \geq \sum_{\forall v_j} C_j - c \tag{34}$$

According to Equations (31) and (33), $D^i_{xy} - d$ and d follows the system model constraint specified in Equation (2). Then, applying NORM or PURE (Equations (3) and (8)) using the corresponding latency constraint $D^i_{xy} - d$, the sum of individual deadlines of all the actors in P_i is

$$\sum_{\forall v_j \in P_i - X_i} D_j + \sum_{\forall v_j \in X_i} D_j = D^i_{xy} - d + d = D^i_{xy} \tag{35}$$

Therefore, Algorithm 1 assures that $D^i_{xy} = \sum_{\forall v_j \in P_i} D_j$ even when actors are shared across time-constrained paths. ■

After proving that in case of a time-constrained path P sharing some actors with higher sensitivity time-constrained paths, the proposed algorithm assures that P satisfies its latency constraints. Here comes the main proof through Theorem 1 that states the validity of the proposed approach and assures that any type of application graph (DAG or DCG) satisfies its latency constraints.

Theorem 1. *Consider an HSDF DCG $G = \langle V, E, d \rangle$ with multiple latency constraints D^i_{xy}. Assuming that G is represented by a set of all possible time-constrained paths $\mathcal{P}$ ordered by non-increasing order of sensitivity γ, Algorithm 1 assures that the actors of G are assigned individual deadlines that makes any $P \in \mathcal{P}$ not exceed its specified latency constraint.*

Proof: For any time-constrained path P_i there are two cases:

Case 1: P_i has no actors with assigned deadlines,

$$\forall v_j \in P_i, D_j = \varnothing \tag{36}$$

Therefore, Algorithm 1 applies either NORM or PURE stated by Equations (11) or (13) under the system model constraint $D_{xy} \geq \sum_{\forall v_j \in P} C_j$. Therefore, from Property 2:

$$\sum_{\forall v_j \in P_i} D_j = D^i_{xy} \tag{37}$$

and, P_i does not exceed its specified latency constraint D^i_{xy}.

Case 2: P_i has a set of shared actors X_i with a set of high-sensitivity time-constrained paths P^H_i,

$$\forall v_k \in X_i, D_k \neq \varnothing \tag{38}$$

Therefore, Algorithm 1 determines the set of unassigned actors and their corresponding latency constraint ($D^i_{xy} - \sum_{\forall v_k \in X_i} D_{v_k}$). Since P_i has a set of shared actors X_i with a set of high-sensitivity time-constrained paths P^H_i, Lemma 1 assures that the sum of individual deadlines D_j of actors in P_i is equal to $D^i_{xy} = \sum_{\forall v_j \in P_i} D_j$.

Therefore, Algorithm 1 assures that the assigned deadlines of all actors in G are such that all latency constraints are satisfied. ■

Finally, we would like to show that in the special case of pipeline application graphs, the proposed algorithm behaves identically to [18], [20], [21] and gives the same results. This is proved in Corollary 1.

Corollary 1. *In case of pipeline application graph $G = \langle V, E, d \rangle$, where G is a multiple actor graph with each actor has a single input/output connected in sequence, applying the proposed algorithm will lead to exactly the same results as previous deadline assignment work for pipelines.*

Proof: Let us assume that we have a pipeline application graph $G = \langle V, E, d \rangle$, where G is a multiple actor graph, where each actor has a single input/output connected in sequence. Applying the first phase of the algorithm (finding all possible time-constrained paths) on G results in a list $\mathcal{P}$ with a single time-constrained path $P = \langle s, v_1, v_2, \ldots, v_z, t \rangle$, where z is number of actors in G. Since it is a single time-constrained path graph and its actors have no assigned deadlines, it will be covered by the first case (1) in Theorem 1. Therefore, applying the proposed algorithm will lead to exactly the same results as previous deadline assignment work for pipelines, Equations (3) and (8) will be applied in this case. ■

Corollary 1 is an important finding, since it shows that our proposed algorithm is more general and deals with any types of application graphs without any particular drawbacks.

VI. Evaluation and Results

The proposed algorithm does not need experiments to prove its correctness since it is formally verified, as shown in Section V. However, we evaluated the algorithm by testing it on cyclic and acyclic real life streaming applications from the SDF^3 Benchmark [27]. The evaluation details are left out

due to space constraints, but they are available in the technical report [28]. In brief, the evaluations show that both of the deadline assignment strategies (NORM/PURE) are applied successfully on acyclic and cyclic graphs, which is a main contribution of our approach. Also, both (NORM/PURE) show almost similar success rates, but NORM shows slightly better schedulability success rate over PURE at higher workloads.

VII. Conclusions

In this paper, we propose an algorithm for extracting the real-time properties of dataflow applications with timing constraints. The algorithm can be applied on dataflow applications modelled as HSDF graphs with periodic sources. The main novelty is that the HSDF graphs can be cyclic or acyclic and the graph actors are modelled as arbitrary-deadline tasks. In addition, it enables applying traditional real-time schedulers and analysis techniques on HSDF dataflow graphs, a method to assign individual deadlines for real-time dataflow actors and support for two deadline assignment techniques (NORM/PURE) that are widely used in the literature.

Acknowledgement

The authors would like to thank Dr. Orlando Moreira and his students for their valuable comments and insights that helped to improve this paper.

References

[1] M. Kim et. al, "H.264 decoder on embedded dual core with dynamically load-balanced functional paritioning," in *Proc. ICIP*, 2010.

[2] E. Lee and D. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, 1987.

[3] V. Pankratius et. al, "Parallelizing bzip2: A case study in multicore software engineering," *Software, IEEE*, 2009.

[4] M. Damavandpeyma et. al, "Parametric throughput analysis of scenario-aware dataflow graphs," in *Proc. ICCD*, 2012.

[5] A. Ghamarian et. al, "Parametric throughput analysis of synchronous data flow graphs," in *Proc. DATE*, 2008.

[6] S. Stuijk, "Predictable mapping of streaming applications on multiprocessors," in *Phd thesis*, 2007.

[7] S. Stuijk et. al, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *Proc. DAC*, 2007.

[8] O. Moreira et. al, "Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor," in *Proc. EMSOFT*, 2007.

[9] H. I. Ali et. al, "Critical-path-first based allocation of real-time streaming applications on 2d mesh-type multi-cores," in *Proc. RTCSA*, 2013.

[10] M. Bamakhrama and T. Stefanov, "Hard-real-time scheduling of data-dependent tasks in embedded streaming applications," in *Proc. EMSOFT*, 2011.

[11] ——, "Managing latency in embedded streaming applications under hard-real-time scheduling," in *Proc. CODES+ISSS*, 2012.

[12] D. Liu et. al, "Resource optimization for csdf-modeled streaming applications with latency constraints," in *Proc. DATE*, 2014.

[13] M. H. Wiggers et. al, "Monotonicity and run-time scheduling," in *Proc. EMSOFT*, 2009.

[14] ——, "Modelling run-time arbitration by latency-rate servers in dataflow graphs," in *Proc. SCOPES*. ACM, 2007.

[15] J. P. Hausmans et. al, "Dataflow analysis for multiprocessor systems with non-starvation-free schedulers," in *Proc. SCOPES*. ACM, 2013.

[16] G. Zhou and J. Wu, "Unmanned Aerial Vehicle (UAV) data flow processing for natural disaster response," *ASPRS*, 2006.

[17] A. Saifullah et. al, "Multi-core real-time scheduling for generalized parallel task models," *Real-Time Systems*, 2013.

[18] G. Lipari and E. Bini, "On the problem of allocating multicore resources to real-time task pipelines," 2011.

[19] M. Qamhieh et. al, "Global EDF scheduling of directed acyclic graphs on multiprocessor systems," in *Proc. RTNS*, 2013.

[20] M. Di Natale and J. Stankovic, "Dynamic end-to-end guarantees in distributed real time systems," in *Proc. RTSS*, 1994.

[21] B. Kao and H. Garcia-Molina, "Deadline assignment in a distributed soft real-time system," *IEEE TPDS*, vol. 8, no. 12, 1997.

[22] M. Spuri and J. Stankovic, "How to integrate precedence constraints and shared resources in real-time scheduling," *IEEE TC*, vol. 43, no. 12, 1994.

[23] H. Chetto et. al, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Systems*, vol. 2, 1990.

[24] G. Bilsen et. al, "Cyclo-static data flow," in *Proc. ICASSP*, 1995.

[25] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

[26] M. Stigge and W. Yi, "Combinatorial abstraction refinement for feasibility analysis," in *Proc. RTSS*, 2013.

[27] S. Stuijk et. al, "SDF^3: SDF for free," in *Proc. ACSD*, 2006.

[28] H. I. Ali et. al, "Generalized extraction of real-time parameters for homogeneous synchronous dataflow graphs," in *Technical Report [CISTER-TR-141104]*, 2014.