Scheduling of Synchronous Dataflow Graphs with Partially Periodic Real-Time Constraints

Alexandre Honorat
Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164
F-35000 Rennes, France
ahonorat@insa-rennes.fr

Karol Desnos
Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164
F-35000 Rennes, France
kdesnos@insa-rennes.fr

Shuvra S. Bhattacharyya
Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164
F-35000 Rennes, France
University of Maryland
College Park, MD 20742, USA
sbhattac@insa-rennes.fr

Jean-François Nezan
Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164
F-35000 Rennes, France
jnezan@insa-rennes.fr

ABSTRACT

Modern Cyber-Physical Systems (CPSs) are composed of numerous components, some of which require real-time management: for example, management of sensors and actuators requires periodic deadlines while processing parts do not. We refer to these systems as partially periodic. In a partially periodic system, precedence constraints may exist between periodic and aperiodic components. It is notably the case in CPSs where sensors measuring physical variables at a fixed sampling rate are typically feeding data to one or more processing part.

A critical challenge for any real-time CPS software is its scheduling on an embedded computing platform. The increasing number of cores in such platforms (as Kalray MPPA Bostan having 288 cores) makes offline non-preemptive scheduling techniques efficient to respect real-time constraints, but requires new analysis and synthesis algorithms. In this paper, we study the schedulability of partially periodic systems modeled as Synchronous Data Flow (SDF) graphs. Our contributions are a few necessary conditions on any live SDF graph, and a linearithmic offline non-preemptive scheduling algorithm on vertices of any directed acyclic task graph. The presented algorithm has been evaluated on a set of randomly generated SDF graphs and on one real use-case. Experiments show that our proposed non-preemptive scheduling algorithm allocates thousands of tasks in less than a second. In the last experiment, the computed schedules achieve a throughput close to that one obtained with global Earliest Deadline First (EDF) scheduling.

KEYWORDS

CPS, SDF, real-time, periodic, scheduling

ACM Reference Format:
Alexandre Honorat, Karol Desnos, Shuvra S. Bhattacharyya, and Jean-François Nezan. 2020. Scheduling of Synchronous Dataflow Graphs with Partially Periodic Real-Time Constraints. In *28th International Conference on Real-Time Networks and Systems (RTNS 2020), June 9–10, 2020, Paris, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3394810.3394820

RTNS 2020, June 9–10, 2020, Paris, France

ACM ISBN 978-1-4503-7593-1/20/06…$15.00
https://doi.org/10.1145/3394810.3394820

1 INTRODUCTION

Cyber-Physical Systems (CPSs) are real-time systems: they are constrained by deadlines on tasks. The tasks are scheduled so that the deadlines are met, inside threads of an operating system or directly on bare-metal. In order to perform their analysis and execution, such real-time systems are modeled with tasks having extra periodicity constraints for the deadlines, and precedence constraints for the data. For systems with only periodic tasks, synchronous languages and related tools as Esterel [3] and SynDEx [19] are a good choice to check the schedulability and to compute a schedule. On the contrary, a few online schedulers [16, 30] focus on the execution of aperiodic, sporadic and periodic tasks together, but these schedulers do not consider precedences. Yet modern CPSs have periodic components interacting with aperiodic components, and with precedence constraints here expressed in the Synchronous Data Flow (SDF) model. This paper aims to analyze the schedulability of such CPSs, called *partially* periodic, and to schedule them systematically and efficiently. A few necessary conditions and an offline non-preemptive scheduling algorithm are introduced for this purpose. Both have been implemented in the PREESM tool [35].

Image signal processing systems and visual servoing are typical examples of partially periodic CPSs where certain components are periodic. For example, a camera films at a periodic framerate and the images arrive to the aperiodic processing components as a stream. Other components may also be periodic, as the input of servo-motors which must be regularly updated. Thus the processing part often depends on periodic inputs and must provide periodically one or more outputs, but does not have to be periodic itself. The flexibility to deviate significantly from periodic operation arises, for example, if data is buffered between components. One possible use-case is the Simultaneous Localization And Mapping (SLAM) application: it constantly retrieves information from a camera or a LIDAR and then processes data to reconstruct a map of the environment and to move according to this map [43]. Sensor fusion [45] or other techniques [17] take benefit of camera *and* LIDAR at the same time.

This paper focuses on CPSs with periodic and aperiodic components, which are modeled as SDF graphs [27]. SDF is commonly used to model image processing applications, as for SLAM with one camera [37]. SDF graphs abstract data transfer between components, called *actors*, being the graph vertices. The SDF graph edges correspond to buffers, in which transmitted data are stored. SDF graphs of CPSs often have imposed periodic inputs and outputs. However our approach is more flexible as any component of the system can be periodic. This flexibility is helpful in the case where multiple processing parts rely on different sensors.

Modeling systems is the first step of the design process. The systems then have to be verified and scheduled. Unfortunately the offline non-preemptive scheduling time complexity is exponential in the number of tasks to get the optimal solution because it is in general NP-complete [25]. This complexity limits the design of CPSs since optimal schedulers do not scale. In contrast, our approach gives results that are not optimal but that can be used to quickly build and assess prototypes of large applications. In other words, our approach is useful for the design space exploration of scheduling solutions. Optimal schedulers and timing property checkers may still have to be used. However, if they are used, it would only be after the prototyping step, on a small set of prototypes.

We focus on offline non-preemptive scheduling because of two main reasons. First, modern systems embed multicore processors where preemption, useful to perform multi-tasking on a unicore processor, is not required anymore, especially when executing a single application. The absence of preemption prevents the overhead caused by context switching [28] and simplifies the timing analysis. Second, as SDF graphs model only systems where all tasks and their precedences are known in advance, there is no necessity to have a reactive online scheduler. In our case a static schedule on each core is used for a global self-timed execution of the system.

In this paper, we consider applications modeled with an SDF graph, where some actors have periodic release times with implicit deadline. Our model is a restriction of the Polygraph model [14] to SDF graphs, but extended with deadlines. We say that such graph has *partially periodic constraints*. Given a number of identical cores to execute the application and the Worst Case Execution Time (WCET) of each actor, the addressed problems are:

(1) first, to quickly check the schedulability, without computing a schedule;
(2) second, to compute an offline non-preemptive schedule satisfying the periodicity and precedence constraints.

In the context of this paper, a schedule lists the start times of all tasks and the cores on which they are allocated.

The notations used in this paper and details about SDF graphs are introduced in Section 2. Then necessary conditions for the non-preemptive scheduling of SDF graphs with some periodic actors are expressed in Section 3. Section 4 discusses the algorithm checking if SDF graphs respect the necessary conditions. A greedy algorithm to schedule graphs with some periodic actors is presented in Section 5. Finally, a discussion on this work, including an evaluation of the scheduling algorithm, is located in Section 6. The related work is located in Section 7 and is followed by a conclusion.

2 BACKGROUND

This work is related to real-time systems and dataflow graphs, which are discussed in the next two subsections.

2.1 Real-time systems

Real-time systems are composed of multiple computational tasks to execute before their deadlines. In this paper, each task τ has either no real-time constraint or a periodic hard deadline. T_τ denotes the period of a periodic task τ. Tasks without periodic deadlines are called *aperiodic*. For periodically released tasks, their deadline d_τ (relative to their release time) is implicit, which means equal to their period. The WCET of each task τ is denoted C_τ.

During the execution of a real-time system, the tasks must be ordered and mapped to the cores in such a way that all tasks meet their deadlines (if any), which is not always possible. In this paper, a *schedule* refers to the order and to the static mapping of the tasks. When there is no schedule respecting the deadlines, the system is said to be not schedulable. In this work only offline data-driven non-preemptive schedulers are considered; the system repeats indefinitely a precomputed schedule. We consider that the system has m identical cores.

2.2 Sychronous Dataflow graphs

SDF graphs [27] model systems where the computational parts, called actors, correspond to the vertices, and where the means of communication, called buffers, correspond to the edges. G denotes the SDF graph being analyzed. $G = (V, E)$ is a directed multi-graph with V the set of actors, and E the set of buffers. A buffer $e \in E$ links its source actor $\text{src}(e) \in V$ to its destination $\text{dst}(e) \in V$. Only weakly connected SDF graphs are considered in this paper [1].

Actors exchange data through their incoming and outgoing buffers. A *token* is the unit of data used in the graph, e.g. one byte. A token production rate $\text{prod}(e) \in \mathbb{N}^*$ is defined on the source side of a buffer e, and a consumption rate $\text{cons}(e) \in \mathbb{N}^*$ is defined on the destination side. The tokens produced by one execution of $\text{src}(e)$ are available to be consumed only after the end of the execution of $\text{src}(e)$. The number of tokens initially present on a buffer e is denoted $d_0(e)$; these tokens are called *delays*.

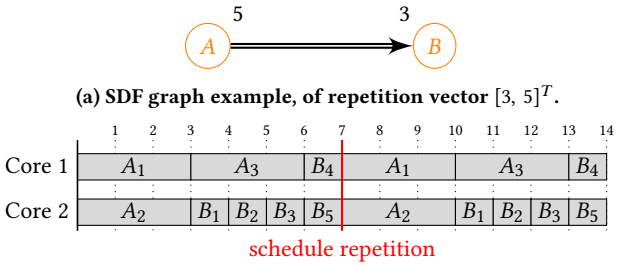
As the rates may not be equal on both sides of a buffer, there can be multiple executions of an actor $\alpha \in V$ in order to avoid underflow or overflow on the buffer. The graph is *consistent* if it ensures that buffers have bounded sizes. Then, it is possible to compute a unique repetition vector $\vec{r}$ giving the minimal number of executions of each actor needed to put the graph back to its initial state with the same number of tokens in each buffer. This defines a graph *iteration* in which the actor executions are called *firings*, or equivalently *jobs* in the literature. Consistency implies Equation (1):

$$\forall e \in E, \quad \vec{r}[\text{dst}(e)] \times \text{cons}(e) = \vec{r}[\text{src}(e)] \times \text{prod}(e) \tag{1}$$

Two examples of a schedule for an SDF graph are given in Figure 1, where the repetition vector is $[3, 5]^T$ (indexed by actor names in the lexicographic order).

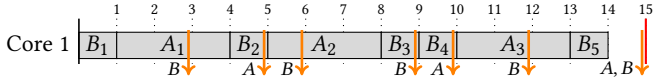
In this paper α_j is the j-th firing of α in one graph iteration. The WCET of an actor α, denoted C_α, is the same for each firing

[1] A directed graph is weakly connected when its undirected induced graph is connected (i.e. a path exists between each pair of distinct vertices).



(a) SDF graph example, of repetition vector $[3, 5]^T$.

(b) Schedule example of 1a on two cores. Two graph iterations are represented, separated by a red vertical line.



(c) Unicore schedule example of 1a, not respecting the following precedences: $A_1 \rightarrow B_1$, $A_2 \rightarrow B_2$ and $A_3 \rightarrow B_4$. The orange arrows are periodic implicit deadlines, if considering both actors A and B periodic.

Figure 1: SDF graph scheduling examples.

of α. Actors have uppercase names for examples, and lowercase names for formula variables. $\mathcal{P}$ denotes the set of periodic actors in G, and $\mathcal{N}$ denotes the set of aperiodic actors. The periods of the actors in $\mathcal{P}$ are defined by the user; but the graph consistency restricts their possible values. Indeed all periods are linearly related, as expressed in Section 3.1. According to the Polygraph model [14], $\mathcal{P}$ corresponds to actors having a *frequency*.

Delays on buffers are allowed, with some restrictions for cycles in G. Indeed SDF graphs deadlock if there is no delays in cycles, and G is *live* if no deadlock occurs. We assume in the analysis that G is live, thanks to delays set by the user on a specific edge of each cycle. Then, such specific edges will not be considered during the analysis and thus, only SDF graphs being Directed Acyclic Graph (DAG) are considered in this work. The case of cycles that are self-loops on an actor is also considered. Self-loops disable auto-concurrency and, as cycles, require delays in order to be consistent. Auto-concurrency implies that multiple firings of an aperiodic actor can be executed at the same time on different cores. For a self-loop $l \in \mathcal{L} \subseteq E$, we assume $\text{cons}(l) = \text{prod}(l) = d_0(l)$.

3 PARTIALLY PERIODIC CONSTRAINTS

Non-preemptive scheduling is often not the best strategy when considering only periodic actors since it may lead to use the cores below their full capacity. For example, consider that two periodic actors A and B are scheduled as in Figure 1c. Actors A and B are periodic, but the only functional requirement on B is that there are 3 executions of A for 5 of B, according to the repetition vector of the SDF graph in Figure 1a. In Figure 1c, the periods are $T_A = 5$ and $T_B = 3$. When ignoring the precedences, the system is schedulable on one core with the WCET respectively $C_A = 3$ and $C_B = 1$, and the core even idles during 1 time unit. If the execution time of A is now $C_A = 3.1$ (instead of 3), the core is still not used to its full capacity but the system is not schedulable anymore since B_3 would miss its deadline in any case. However, if B is not required to be periodic, the system is schedulable on 1 core with $C_A = 3.1$. Thus in this case partially periodic constraints help to fully use the capacities of the cores in the context of non-preemptive scheduling. In this section, we focus on necessary conditions for schedulability of SDF with partially periodic constraints. The generic processor utilization necessary condition is recalled in Section 3.1 while a more precise one is established in Section 3.2.

3.1 Plain schedulability condition

A widely used necessary condition for schedulability of periodic tasks derives from the processor utilization factor [31] metric $U = \sum \frac{C_\tau}{T_\tau}$, without unit. $U \leq m$ is a necessary but not sufficient condition, for all preemptive and non-preemptive schedulers of tasks with and without precedence constraints: if $U > m$ the system is not schedulable [22].

This paper focuses on weakly connected SDF graphs, thus all actors are connected and specifying the period of one actor π is equivalent to specifying a period for the whole graph. Indeed the graph period T_G will be $\vec{r}[\pi] \times T_\pi$ time unit. In Figure 1c, the graph period is 15, according to the periods $T_A = 5$ and $T_B = 3$, and to the repetition vector $[3, 5]^T$. Formally, in one graph iteration, the start time of the k-th firing of a periodic actor $\pi \in \mathcal{P}$ with an implicit deadline must occur in the following time interval:

$$[\![kT_\pi; (k+1)T_\pi - C_\pi]\!], \text{with } k \in [\![0; \vec{r}[\pi][\![\tag{2}$$

Consequently, in average $\vec{r}[\alpha]$ firings of an aperiodic actor α are executed during each graph period T_G, since the repetition vector imposes $\vec{r}[\alpha]$ firings of α for $\vec{r}[\pi]$ firings of π.

As each periodic actor π defines a graph period $T_G = \vec{r}[\pi] \times T_\pi$ deriving from the unique repetition vector, this implies that all the obtained graph periods must be equal: $\exists T_G, \forall \pi \in \mathcal{P}, T_G = \vec{r}[\pi] \times T_\pi$. Then, the processor utilization factor metric may be reformulated in the context of partially periodic constraints, considering the average amount of aperiodic firings per graph period T_G. Equation (3) is a necessary but not sufficient schedulability condition for partially periodic SDF graphs.

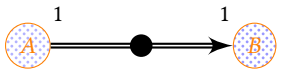
$$m \geq U = \frac{\sum_{\alpha \in \mathcal{N}} \vec{r}[\alpha] \times C_\alpha}{T_G} + \sum_{\pi \in \mathcal{P}} \frac{C_\pi}{T_\pi} \tag{3}$$

3.2 With no scheduler iteration overlapping

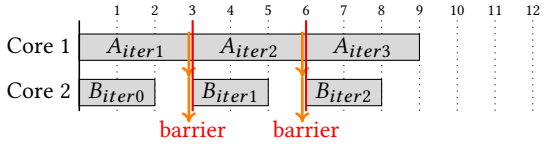
Results of this subsection assume the following assumption on the scheduler. Under Assumption 1, indeed another necessary condition can be derived from the path lengths in the SDF graph with partially periodic constraints.

Assumption 1 (A1). *For every actor α, as many firings as specified in the repetition vector $\vec{r}[\alpha]$ must have been completely executed before the next scheduler iteration begins.*

A *scheduler* iteration is the static scheduling of the application that is indefinitely repeated until the application is stopped. This assumption is made to ease the scheduling, the code generation, and the memory allocation. Under Assumption 1, if there is a graph period, all firings of one scheduler iteration must be done during a time interval equal to this graph period. Assumption 1 is present in the PREESM SDF graph scheduler [35] where scheduler iterations cannot overlap in time and are separated by a synchronization barrier between all cores. One graph period separates two successive

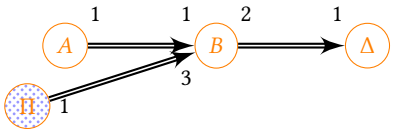


(a) SDF graph having two periodic actors with a dependency and one delay. The repetition vector is $[1, 1]^T$.

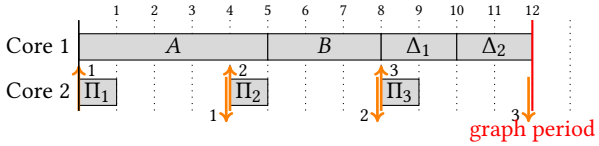


(b) Schedule example of 2a on two cores. The barriers happen at the same time as the graph period, they define a scheduler iteration. Note that three scheduler iterations are represented while only two graph iterations are completed (iterations 1 and 2).

Figure 2: Scheduler iteration example, under Assumption 1.



(a) Dependencies on a periodic actor. $nblf^{\uparrow}_{\Pi}(B) = 1$, $nblf^{\uparrow}_{\Pi}(\Delta) = 2$



(b) Schedule example of graph 3a, generating underflow.

Figure 3: Periodic actor Π generating an underflow.

barriers, so it ensures that for each actor α, there are $\vec{r}[\alpha]$ executions of α between two barriers. Note that Assumption 1 is similar to *K-periodic scheduling* [8] with K being the repetition vector $\vec{r}$ in our case. While K-periodic scheduling imposes a periodic schedule of $K[\alpha]$ firings independently for each actor α, Assumption 1 enforces these periodic schedules to be synchronized with barriers.

Assumption 1 still accepts schedules with delays; but then, all firings of one *graph* iteration do not occur in the same *scheduler* iteration: this is *pipelining*. Figure 2 gives an example of such a pipelined schedule with delays: the firing of B consuming the data produced by the last firing of A happens one scheduler iteration after. Graph iterations are denoted *iter* in the Gantt diagram of Figure 2b. Equation (2) is also respected inside one *scheduler* iteration.

Scheduling a partially periodic SDF graph without taking care of data dependencies may lead to buffer underflows and overflows as illustrated in Figure 3 where the SDF graph is however consistent. In this example the period of the actor Π is 4 time units (and $C_\Pi = 1$), and the graph period is 12. The Gantt diagram in Figure 3b respects the periodic constraint but not the data dependencies. An underflow occurs since B and Δ are executed before having received the data produced by the last firing of Π. Note that the delays are predefined by the user, and are not computed nor checked by Algorithm 1, presented at the end of this section. However, such data dependency errors are checked easily on static schedules.

Figure 3 illustrates an intuitive necessary condition to check the schedulability: all actors depending on the tokens produced by the last execution of Π must be executed in the slack time of Π. This necessary condition derives from Assumption 1. The slack time of Π is defined by $T_\Pi - C_\Pi$. A symmetrical necessary condition can be computed for the first execution of any periodic actor, this time with all its incoming data dependencies, which are all actors on a directed path leading to the periodic actor. In order to formalize these necessary conditions, some notations and functions are introduced in the next paragraphs.

$\mathcal{D}^{\uparrow}_{\pi}$ denotes the set of actors in G that are transitively data dependent on an actor π: if $\alpha \in \mathcal{D}^{\uparrow}_{\pi}$, the last firing of π enables at least one firing of α. It can be computed by a graph traversal from π. The subgraph restriction of G containing only the actors in $\mathcal{D}^{\uparrow}_{\pi}$, is denoted $G^{\uparrow}_{\pi}$. All presented equations can be applied on the transpose of G, considering the first firing of π instead of the last. The transpose G^T of a graph G is its mirror, where all edges are directed in the opposite direction, also exchanging token production and consumption rates. For brevity, equations on G^T are not shown.

The main metric to compute is the numbers of actor firings, enabled by a single firing of a periodic actor π. These numbers of firings allow us to compute lower bounds of the processor utilization factor. The analysis is simplified by restricting it to the last firing of π and all induced firings of its successors in $\mathcal{D}^{\uparrow}_{\pi}$. Note that the number of remaining dependent firings can be computed for two adjacent actors connected by a single buffer e: k firings of $\text{src}(e)$ enable $\max\{0, \left\lceil \frac{k\times\text{prod}(e)-d_0(e)}{\text{cons}(e)} \right\rceil\}$ firings of $\text{dst}(e)$. The term $k \times \text{prod}(e)$ corresponds to the new tokens incoming on the buffer e. The ceiling operator is needed since the previous execution of the producer $\text{src}(e)$ may have left unused tokens on e. At the end of the graph iteration, e contains exactly $d_0(e)$ delays.

The function computing the number of firings enabled by the last firing of a periodic actor π is denoted $nblf^{\uparrow}_{\pi}$, defined in Equation (4). $nblf^{\uparrow}_{\pi}$ is a recursive function, depending on the predecessor actors in the graphs $G^{\uparrow}_{\pi}$. The set of incoming edges to an actor α in $G^{\uparrow}_{\pi}$ is denoted $IE^{\uparrow}_{\pi}(\alpha)$; this set excludes self-loops $l \in \mathcal{L}$.

$$nblf^{\uparrow}_{\pi}(\alpha) = \max_{e \in IE^{\uparrow}_{\pi}(\alpha)} \left\{ 0, \left\lceil \frac{nblf^{\uparrow}_{\pi}(\text{src}(e)) \times \text{prod}(e) - d_0(e)}{\text{cons}(e)} \right\rceil \right\} \quad (4)$$

The recursion stops at the root actor π, having no incoming edges (there is only one root, by construction of $G^{\uparrow}_{\pi}$), where $nblf^{\uparrow}_{\pi}$ holds the value 1 if π is periodic and 0 otherwise. Hence if the root actor is not periodic, $nblf^{\uparrow}_{\pi}$ takes the value 0 on all vertices and it does not help to find any necessary condition. For brevity, the proof of Equation (4) is given for one direct predecessor only of $\text{dst}(e)$.

Proof. To prove Equation (4), let us consider the firings of the actor $\text{dst}(e)$ before those that are induced by the last firing of π. By definition this number of firings is $\vec{r}[\text{dst}(e)] - nblf^{\uparrow}_{\pi}(\text{dst}(e))$. Under Assumption 1, there are exactly $\vec{r}[\alpha]$ firings of each actor α during

one scheduler iteration. So this number can also be computed considering all tokens produced on e by src(e) during one scheduler iteration, before the last firing of π: this is why $\vec{r}[\text{src}(e)] - nblf^{\uparrow}_{\pi}(\text{src}(e))$ multiplies the production rate in the following equality.

$$\vec{r}[\text{dst}(e)] - nblf^{\uparrow}_{\pi}(\text{dst}(e)) = \left\lfloor \frac{(\vec{r}[\text{src}(e)] - nblf^{\uparrow}_{\pi}(\text{src}(e))) \times \text{prod}(e) + d_0(e)}{\text{cons}(e)} \right\rfloor$$

Thus $nblf^{\uparrow}_{\pi}(\text{dst}(e))$ is equal to:

$$\vec{r}[\text{dst}(e)] - \left\lfloor \frac{\vec{r}[\text{src}(e)] \times \text{prod}(e) - nblf^{\uparrow}_{\pi}(\text{src}(e)) \times \text{prod}(e) + d_0(e)}{\text{cons}(e)} \right\rfloor$$

Knowing that $\forall x \in \mathbb{R}, -\lfloor x \rfloor = \lceil -x \rceil$, it becomes:

$$nblf^{\uparrow}_{\pi}(\text{dst}(e)) = \left\lceil \frac{\vec{r}[\text{dst}(e)] \times \text{cons}(e)}{\text{cons}(e)} + \frac{-\vec{r}[\text{src}(e)] \times \text{prod}(e) + nblf^{\uparrow}_{\pi}(\text{src}(e)) \times \text{prod}(e) - d_0(e)}{\text{cons}(e)} \right\rceil$$

From the consistency of G, see Equation (1), $\vec{r}[\text{dst}(e)] \times \text{cons}(e) - \vec{r}[\text{src}(e)] \times \text{prod}(e) = 0$, so the last formula can be simplified to Equation (4) (without the maximum, needed when they are multiple direct predecessors).

□

The following formula is then a necessary condition for schedulability under Assumption 1. Equation (5) corresponds to the processor utilization factor of all firings depending on the last firing of a periodic actor. This processor utilization factor is computed over the slack time of the periodic actor, hence the division by $T_\pi - C_\pi$.

$$\forall \pi \in \mathcal{P}, \frac{\sum_{\alpha \in \mathcal{D}^{\uparrow}_{\pi}} nblf^{\uparrow}_{\pi}(\alpha) \times C_\alpha}{T_\pi - C_\pi} \leq m \tag{5}$$

Note that the maximal length of any graph path starting at a periodic actor π also provides a simple necessary condition of the schedulability: this length must be less than the slack time of π. Also, actors with self-loops have a strong impact on the maximal length because they do not have auto-concurrency. This necessary condition for actors with self-loops is formalized in Equation (6).

$$\forall \alpha \in \mathcal{D}^{\uparrow}_{\pi} \cap \mathcal{L},\ nblf^{\uparrow}_{\pi}(\alpha) \times C_\alpha \leq T_\pi - C_\pi \tag{6}$$

Equation (6) can be extended to all paths between a periodic root π and leaves of the DAG $G^{\uparrow}_{\pi}$. On each of these paths, each actor will be executed at least once except if there are enough delays before the actor; if no delays, the path length is the actor WCET sum. Again, all these path lengths must be lower than the slack time $T_\pi - C_\pi$.

Algorithm 1 checks the schedulability of a periodic actor π in $G^{\uparrow}_{\pi}$ thanks to the aforementioned necessary conditions: the one derived from the processor utilization factor, and the one derived from the critical path minimal execution time. The efficiency of Algorithm 1 is discussed in the next section.

Algorithm 1: Modified Breadth-First Search (BFS) to compute $nblf^{\uparrow}_{\pi}$ and related necessary conditions

```
1  function nblfExt(π)
2      forall α ∈ G do
3          timeTo(α) ← 0;    nblf↑π(α) ← 0;
4          nbVisits(α) ← 0;
5      C_tot ← 0;    queue ← ∅;    addLast(queue, π);
6      D↑π ← ∅;    nblf↑π(π) ← 1;                ▷ π is periodic.
7      while queue ≠ ∅ do
8          α ← pop(queue);    D↑π ← D↑π ∪ {α};
9          forall e ∈ OE↑π(α) do
10             dest ← dst(e);
11             nbVisits(dest) ← nbVisits(dest) + 1;
12             timeTo(dest) ← max{timeTo(dest), timeTo(α)};
13             nblf↑π(dest) ←
                   max{ nblf↑π(dest), ⌈ (nblf↑π(α)×prod(e)−d0(e)) / cons(e) ⌉ };
               ▷ See Equation (4).
14             if nbVisits(dest) = #IE↑π(dest) and
                   nblf↑π(dest) > 0 then
15                 addLast(queue, dest);    fctr ← nblf↑π(dest));
16                 if e ∉ L then                  ▷ See Equation (6).
17                     fctr ← max{1, ⌊ fctr/m ⌋};
18                 timeTo(dest) ← timeTo(dest) + C_dest × fctr;
                   ▷ Update the path length to dest with an
                   underestimation of its execution time.
19                 if timeTo(dest) > T_π − C_π then
20                     return System not schedulable.
21     forall α ∈ D↑π − {π} do
22         C_tot ← C_tot + nblf↑π(α) × C_α;        ▷ See Equation (5).
23     if C_tot / (T_π − C_π) > m then
24         return System not schedulable.
```

4 DISCUSSION ON ALGORITHM 1

Algorithm 1 has a linear complexity in the number of edges in $G^{\uparrow}_{\pi}$. Thus if all actors in G are periodic, it may not be efficient to execute Algorithm 1 on each one: in specific cases the overall complexity can be more than quadratic in the number of vertices in G, as for the star graphs with directed paths going to/from a central vertex. In order to perform the algorithm on a subset of the periodic actors, a heuristic is presented in Section 4.1. Algorithm 1 faces another problem: as it relies only on necessary conditions, there are cases where the algorithm fails to detect a non-schedulable system. This point is discussed in Section 4.2.

4.1 Heuristic to run Algorithm 1 efficiently

In this subsection a heuristic is given to execute Algorithm 1 on a small set of periodic actors: the one having small slack time

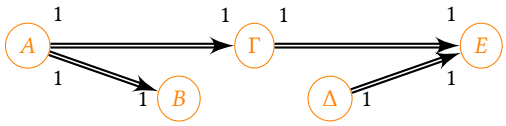


Figure 4: Sample SDF graph

and a low topological rank in G. Indeed a small period T_π will reduce the denominator in Equation (5), while a low topological rank may increase the numerator because it means that more actors are located after π. Thus this heuristic selects the actors being more discriminative regarding to the schedulability tests of Algorithm 1.

As we consider synchronous dataflow DAG, there is always a topological sort existing. One topological sort is used to select actors: it corresponds to an As Soon As Possible (ASAP) schedule of G^T, not constrained by the number of cores. The ASAP topological sort on G^T is denoted o^T and is used to select the periodic actors on which Algorithm 1 is called. Notice that the actor WCETs are not taken into account in this topological sort, only the structure of the graph is used. Such topological sort can be computed with a BFS, having a linear complexity in the number of edges in G.

Considering the SDF graph in Figure 4, there are three topological ranks: one per actor in the longest graph path which is $A \longrightarrow \Gamma \longrightarrow E$. Thus $o^T(A) = 2$, $o^T(\Gamma) = 1$, $o^T(E) = 0$. The other actors have the lowest topological ranks at which they can be executed, that is $o^T(B) = 0$, $o^T(\Delta) = 1$. Indeed B has no incoming edges in G^T so its rank is 0. The ASAP sort on the transpose graph of Figure 4 can be equivalently represented as follows: $o^T \equiv \{E, B\} \prec \{\Gamma, \Delta\} \prec \{A\}$.

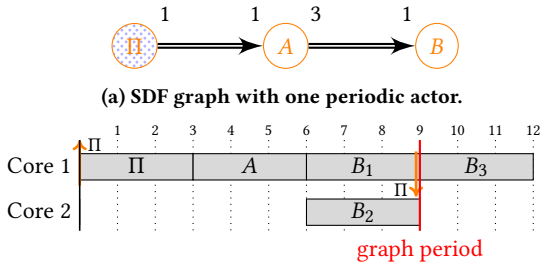
Formally, the heuristic selects the periodic actors having the lowest $\frac{T_\pi - C_\pi}{o(\pi)}$. The number of selected actors with this heuristic is arbitrarily chosen by the user. Note that vertices with ASAP topological rank equal to 0 are not of interest since it means that they have no successors; Algorithm 1 is not run on such vertices.

4.2 A false positive to Algorithm 1

Algorithm 1 performs two schedulability tests. One is using the processor utilization factor U, Equation (5) lines 21-24, and thus does not consider the precedences between actors. Considering multiple cores, it may lead to keep invalid schedules where $U < m$, but where a path from a periodic actor is longer than the slack time of this actor. This situation is precisely checked by the other schedulability test using path lengths, lines 18-20 in Algorithm 1[2]. Between these two situations, the algorithm may miss that U is too large on a small portion of the slack time: for example, if $nblf^\uparrow$ of the last actor is greater than m. Thus, Algorithm 1 fails to find that the graph represented in Figure 5 is not schedulable with 2 cores. Yet the critical path starting from Π in Figure 5a is equal to the slack time of Π and $U = \frac{15}{9}$ is less than the number of cores $m = 2$.

Algorithm 1 may compute other false positive answers: only $G_\pi^\uparrow$ is considered and thus, periodic actors having a period smaller than T_π in G but not being in $G_\pi^\uparrow$ are not taken into account. To avoid that, Algorithm 1 can be refactored with a subfunction performing

[2]The equation on line 17 is actually a simplification when there is auto-concurrency. Inside cycles (including self-loops), the auto-concurrency is limited by the repetition vector of the cycle, which divides the original repetition vector.



(a) SDF graph with one periodic actor.

(b) Shortest schedule of graph in Figure 5a.

Figure 5: A false positive to Algorithm 1: B_3 cannot be scheduled before the graph period.

computations of lines 2-22. The subfunction is called on π and on all the periodic actors having a period smaller than T_π, and not being connected by any path in G. For brevity, the full algorithm is not presented here.

However, even if false positives appear, the system designer will logically continue its work by calling a scheduler. The scheduler will give a final answer: schedulable or not. In the next section, an offline scheduler is presented under Assumption 1. This scheduler is a heuristic and thus does not attempt to find an optimal scheduling, it rather focuses on giving quickly an answer to the designer.

5 SCHEDULING OF SDF GRAPHS WITH PARTIALLY PERIODIC CONSTRAINTS

The problem studied in this section is the quick scheduling of G^* with partially periodic constraints. G^* is the Single-Rate Data Flow (SRDF) graph corresponding to the SDF graph G. In other words, G^* is the unrolled version of G, where data dependencies are expressed between firings instead of actors. Since all dependencies are explicitly expressed in G^*, it enables computing a static schedule. Each SDF actor α in G has $\vec{r}[\alpha]$ corresponding vertices in the SRDF graph G^*, and on each edge e of G^* the rates of both sides are equal, i.e. prod(e) = cons(e). In this section, *tasks* refer to vertices in G^*. As in G, delays break cycles so G^* is a DAG.

Scheduling of DAG of tasks has been widely studied [25] however the periodic case is specific since the start times of periodic actors are bounded in an interval of the form of Equation (2).

For example, the FAST algorithm [26], based on a list scheduling heuristic and a neighborhood search, is not appropriate for periodic schedules. The main difference between the problem studied here and the one solved by standard list scheduling algorithms is that, because of partially periodic constraints, we have a bound on the schedule length. This bound is the graph period.

The main difficulty to design a greedy list-scheduling scheduling algorithm is to order the vertices before trying to schedule them. The FAST algorithm cannot be used as is but some of its techniques are reused in the presented Algorithm 2. As in FAST, Algorithm 2 relies on ASAP and As Late As Possible (ALAP) orderings.

The minimum start time ns of each task in G^* as well as the maximum start time xs are computed first. This is done in two successive rounds: 1) for all periodic actors, with Equation (2), 2) for all actors, with ASAP and ALAP. During round 2), ALAP schedule

has a global deadline equal to the graph period. If any task τ has $ns(\tau) > xs(\tau)$, the algorithm stops: the system is not schedulable.

Algorithm 2 is executed once all ns and xs have been computed, starting with the `schedule` procedure. The tasks are sorted according to their average start time $\frac{xs+ns}{2}$. This sorting criterion is a heuristic to balance the task executions over time. The list of tasks to allocate, l line 22, contains only vertices having all their dependencies satisfied, initially the one having no incoming edges. If several tasks in l have the same average start time, ns is used to break the tie, by increasing order. The algorithm performs a first fit approach: it selects the next task in l and schedules it on the least loaded core. The `allocateAndRemoveIfBefore` procedure, lines $10-17$, also updates l with tasks having a direct dependency on the currently allocated task. This procedure stops the scheduling in two cases: 1) if a task is scheduled after its maximum start time xs, lines $12-13$, 2) if a the total idle time is more than the maximum possible, line 7, according to the formula line 21.

Algorithm 2 is *greedy* since if a task τ implies an idle time, the algorithm tries to schedule before τ the tasks τ_b having $ns < predFinishTime(\tau)$, without delaying τ. This is the purpose of lines $26-30$ in Algorithm 2. The test line 14 ensures that τ_b can be executed without delaying τ, and if not it prevents to allocate it. Although Algorithm 2 is *greedy*, it is not subject to the Dhall's effect [13]. Dahll's effect occurs on multicore processors when two tasks are ready at the same time and have the same deadline, but the allocation order prevents to allocate both tasks because the smallest task may be allocated first on the least loaded core, not leaving enough space for the other task. In Algorithm 2, as tasks are sorted by average start time from ASAP and ALAP scheduling, the biggest task will have a shorter average start time and thus will appear sooner in the list of tasks ready to be scheduled.

The complexity of Algorithm 2 is upper bounded by the number of edges in G^* and by the linearithmic cost of the sorting operation on the vertices: $O(\#E^* + \#V^*(m + \log(\#V^*)))$. The number of cores m appears as a factor of $\#V^*$ since the list of cores c must remain sorted to select the least loaded core for every vertex in the ready queue l. The cost of transforming G in G^* is not included in this complexity, it is upper bounded by $\#E^*$. It is a standard transformation, already implemented in the PREESM tool for example.

6 EVALUATION

This section discusses how Algorithms 1 and 2 can be used for the design process of CPSs, and presents an evaluation.

6.1 Partially periodic CPSs applications

Only a few CPSs use-cases are presented in the literature as *partially periodic* SDF graphs. Indeed it is often assumed that every component is periodic in order to ease the analysis and the code generation. Due to the lack of available partially periodic implementations, and to the simplicity of the existing ones, the necessary conditions for schedulability have not been practically evaluated. However, two small examples are given hereafter. A pacemaker [36] has been studied and modeled in the Architecture Analysis Design Language (AADL)[3] and it is a partially periodic system. A critical subpart of this pacemaker is described using the Cyclo-Static Data Flow (CSDF) model [6], an extension of SDF. In this subpart, two sensors (Motion and EKG) periodically send data to a processing component, each with its own period. A second example is in the telecommunication domain: the LTE standard has been studied and partially modeled with SDF graphs [34]. In the LTE standard, the signals retrieved by the antenna are down-sampled and periodically sent to the decoder. To the best of our knowledge, no open source benchmark exists that is explicitly partially periodic. Yet, the StreamIt [42] benchmark contains dozens of signal processing applications in the SDF model. Periods are not specified in StreamIt but signal processing applications usually have one periodic input actor and another periodic output.

[3] http://www.aadl.info

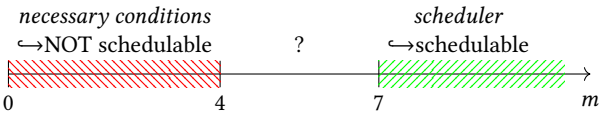


Figure 6: Scheduling bounds on the number of cores m.

Finally it is possible to generate random SDF graphs with SDF^3 [40] and Turbine [7], but they do not generate partially periodic constraints. Thus, our experiments, detailed in Sections 6.3 and 6.4, have been performed on ten random DAGs generated by Turbine, and on one existing use-case of the SDF^3 data set. For the first experiment, partially periodic constraints are added to these generated graphs as a post-processing step that follows the graph generation.

6.2 Schedulability check and scheduling

The practical usage of the presented algorithms is summarized in Figure 6. In terms of the number of cores m, the necessary conditions (Algorithm 1) give a schedulability lower bound (left part) while the scheduler (Algorithm 2) gives an upper bound (right part). For example, the lower bound is 4 cores while the upper bound is 7 cores in Figure 6.

The algorithms have to be run iteratively with different values of m to tighten the bounds. A starting point can be derived using the length of the critical path as done in the experiments on the scheduler, presented in Section 6.3.

Regarding the scheduler, notice that it can actually be run for any unconnected DAG and thus is more generic than for SDF graphs with partially periodic constraints. The next subsections report two experiments done on the algorithms presented in this paper.

6.3 Gap between necessary conditions and scheduler

This experiment measures the gap between the proposed necessary conditions, the proposed scheduler, and the optimal solution. The gap is measured in number of cores required to synthesize a schedule while respecting all periodic constraints.

6.3.1 Dataset. The dataset contains ten random SDF DAG generated with the Turbine tool [7]. Table 1 details the characteristics of the generated graphs. The first five graphs are small, with only ten actors, in order to make comparison with the optimal number of cores computed by a Constraint Programming (CP) solver. The generated graphs do not contain any delay.

Algorithm 2: Scheduling of tasks

procedure addReadyTasks(l, τ, *nbAllocations*) ▷ Add tasks in the schedule queue only if their predecessors are allocated.
  remove(l, τ); *nbAllocations* ← *nbAllocations* + 1;
  forall $e \in OE(\tau)$ **do** ▷ Visit all successors of newly allocated τ.
    dest ← dst(e); *nbVisits*(*dest*) ← *nbVisits*(*dest*) + 1 *predFinishTime*(*dest*) ← max{*predFinishTime*(*dest*), *finishTime*(τ)};
    if *nbVisits*(*dest*) = #*IE*(*dest*) **then**
      push(l, *dest*); ▷ If all predecessors of dst(e) have visited it, dst(e) is added in the queue.

procedure casIdleTime(*startTimeDif*, *currentIdleTime*, *maxIdleTime*) ▷ Check and set remaining idle time.

function isThereACoreIdlingBefore(c, *deadline*) ▷ Returns true if one or multiple cores idle before the deadline.

function possibleAllocationsBefore(l, *deadline*) ▷ Returns tasks in l which can start before the given deadline, ensuring that the selected tasks total execution time is not higher than the current idle time to the given deadline.

procedure allocateAndRemoveIfBefore(l, τ, c, *deadline*, *maxIdleTime*, *currentIdleTime*)
  coresHead ← *head*(c); *startTime* ← max{*predFinishTime*(τ), *finishTime*(*coresHead*)};
  if *startTime* > xs(τ) **then**
    raise Scheduling failed.;
  if *startTime* + C_τ > *deadline* **then**
    return;
  casIdleTime(*startTime* − *finishTime*(*coresHead*), *currentIdleTime*, *maxIdleTime*) ; ▷ Check and set idle time.
  finishTime(τ) ← *startTime* + C_τ; *push*(*coresHead*, τ); *addReadyTasks*(l, τ); ▷ Update τ, c and l.

procedure schedule(*tasks*, *graphPeriod*, m)
  forall $\tau \in$ *tasks* **do**
    nbVisits(τ) ← 0; *predFinishTime*(τ) ← ns(τ) ; ▷ Sets initial properties of each task.
  maxIdleTime ← m ∗ *graphPeriod* − $\sum_\tau C_\tau$; *currentIdleTime* ← 0; *nbAllocations* ← 0;
  l ← all tasks without incoming edges, always maintained by increasing average start time i.e. $\frac{\text{xs} + \text{ns}}{2}$;
  c ← schedule of each core, always maintained by increasing finish time;
  while $l \neq \emptyset$ **do**
    τ ← *head*(l); *prevNbAllocations* ← *nbAllocations*;
    if *isThereACoreIdlingBefore*(c, *predFinishTime*(τ)) **then** ▷ *predFinishTime* is the finish time of direct predecessors.
      forall $\tau_b \in$ *possibleAllocationsBefore*(l, c, *predFinishTime*(τ)) **do**
        allocateAndRemoveIfBefore(l, τ_b, c, *predFinishTime*(τ), *maxIdleTime*, *currentIdleTime*, *nbAllocations*);
      if *prevNbAllocations* < *nbAllocations* **then**
        continue ; ▷ We restart the loop since new tasks may be ready now.
    allocateAndRemoveIfBefore(l, τ, c, ∞, *maxIdleTime*, *currentIdleTime*, *nbAllocations*);

Name	#actors	avg. WCET	#firings	#deps. in G^*	$\vec{r}[\pi]$	T_G	Time Alg.1	Time Alg.2	Time Choco
RandomDAG1	10	100	141	280	5	2445	7 ms	27 ms	T/O (12 h)
RandomDAG2	10	100	184	373	7	4270	4 ms	13 ms	320327 ms
RandomDAG3	10	100	139	370	13	9412	4 ms	10 ms	993116 ms
RandomDAG4	10	100	143	329	3	1557	3 ms	11 ms	T/O (12 h)
RandomDAG5	10	100	136	251	3	1377	4 ms	7 ms	9046 ms
RandomDAG6	100	200	3226	6093	7/4	13496	10 ms	134 ms	–
RandomDAG7	100	200	2824	6239	5/10	15040	6 ms	68 ms	–
RandomDAG8	100	200	2978	6341	7/7	11711	5 ms	61 ms	–
RandomDAG9	100	200	2567	5600	6/6	8496	10 ms	89 ms	–
RandomDAG10	100	200	3358	6535	10/10	16680	8 ms	15 ms	–

Table 1: Details of the random directed acyclic SDF graphs generated by Turbine, and execution time of the algorithms. RandomDAG1-5 contain one periodic actor π. RandomDAG6-10 contain two periodic actors.

One periodic actor π is set in the middle of the longest path of each SDF graph; largest number of firings breaks the tie. A second periodic actor is set on RandomDAG6-10. Their periods T_π are set manually, being the smallest integer such that a solution exists. Formally, T_π is the smallest integer which ensures: $\forall \tau \in G^*,\ \mathrm{ns}(\tau) < \mathrm{xs}(\tau)$.

6.3.2 Standard Integer Linear Programming (ILP) formulation with Choco. A scheduling model using ILP formulation has been developed in order to compare the performance with Algorithm 2. Although the formulation is purely ILP, the generic Choco[4] CP solver has been used since there is no objective function in the formulation: the goal is only to test if there exists a valid schedule for a given number of cores m. Choco stops on the first valid schedule encountered, and otherwise enumerates all possible schedules in order to prove that there is no solution.

The model size is bounded by the size of a transient Boolean matrix storing mapping overlap of each couple of tasks: $\Theta(m \times \#V \times \#V)$. A transitive closure of the DAG G^* is computed before the model construction in order to reduce the size of this Boolean matrix. The transitive closure prevents to add useless free variables and redundant constraints to the matrix stating mapping overlap: overlap between two tasks is checked only if there is no transitive precedence between the two tasks. The number of constraints is reduced by up to 16% thanks to the transitive closure. Last but not least, symmetries of the homogeneous cores are broken by enforcing some properties on the mapping matrix, see [41].

6.3.3 Implementation of algorithms in PREESM. Algorithms 1 and 2 and the Choco model have all been implemented in the PREESM open-source[5] tool dedicated to the design of embedded systems from applications modeled as SDF graphs. PREESM automatically generates the SRDF graph G^* from a given SDF graph; the generation time of G^* is not included in the experiments. Experiments have been run on an Intel i7-7820HQ processor.

6.3.4 Evaluation results. The valuation has been performed as following. For each scheduling algorithm, the result is the smallest number of cores ensuring a valid schedule. For the necessary conditions, the result is the smallest number of cores ensuring that no valid schedule exists for all lower number of cores. All algorithms are run iteratively with an increasing number of cores. Diagram in Figure 7 presents the results for the five small random graphs. The processor utilization factor *Utot* of the graph is given as reference on the left column. The execution time of the algorithms are given in the right part of Table 1 (in millisecond ms, and hour h). While the proposed algorithms always run in less than a second, Choco takes hours for the small graphs.

Choco is optimal but cannot solve problems with too many firings or cores. It actually reaches timeout (T/O) of 12 hours for RandomDAG1 and RandomDAG4 with 9 cores, and it also takes multiple hours to prove that the same graphs have no solutions for 8 cores. The timeout of Choco is specified by an error interval, materialized by a small black line in Figure 7. The gap between the optimal solution and the proposed scheduler is at most two cores, for RandomDAG4, and may be zero as for RandomDAG5.

[4]http://www.choco-solver.org/
[5]https://preesm.github.io/

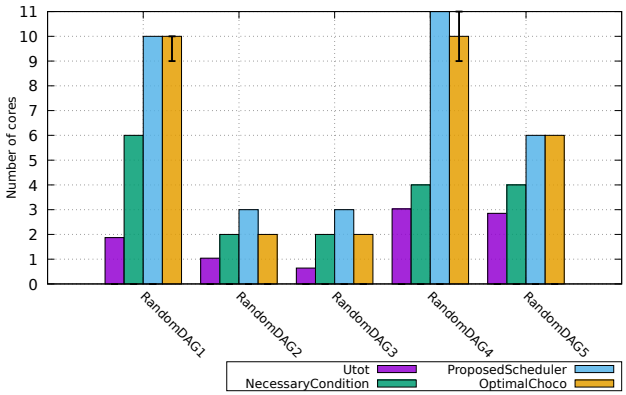


Figure 7: Evaluation of the schedulability gap on the small random graphs RandomDAG1-5.

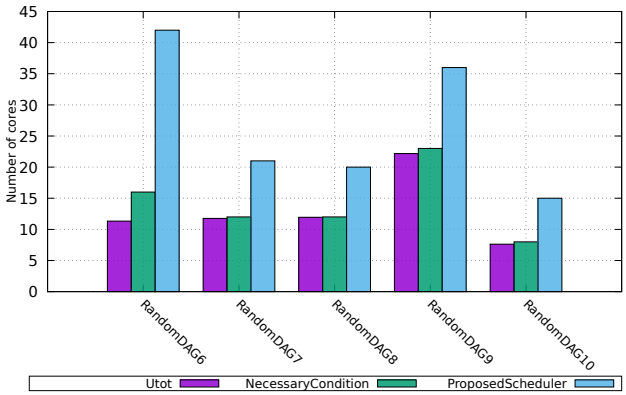


Figure 8: Evaluation of the schedulability gap on the large random graphs RandomDAG6-10.

The results of the five large graphs are pictured in Figure 8, without comparison with Choco because of the size of the problem (it would timeout in any case). On all ten graphs, the necessary conditions appear to be weakly discriminating: in most of the case, it states that the system is possibly schedulable as soon as $m > U_{tot}$. The necessary conditions are discriminating only for RandomDAG1 and RandomDAG6. Note that RandomDAG1 and RandomDAG6 are also the graphs requiring a longer run time of the proposed scheduler. Yet two reasons may increase the complexity of the scheduler: more tasks ready at the same time (which increases the size of the sorted list l), or more idle time (which triggers the execution of lines 26-30 in Algorithm 2). Further investigation are needed to characterize this phenomenon.

6.4 Gap between the proposed scheduler and preemptive Earliest Deadline First (EDF)

This experiment measures the graph period gap between the proposed non-preemptive offline scheduler Algorithm 2 and a standard preemptive real-time scheduler: EDF. The ADFG tool [21] is used as a reference, using Global EDF scheduling [4] (with an algorithm

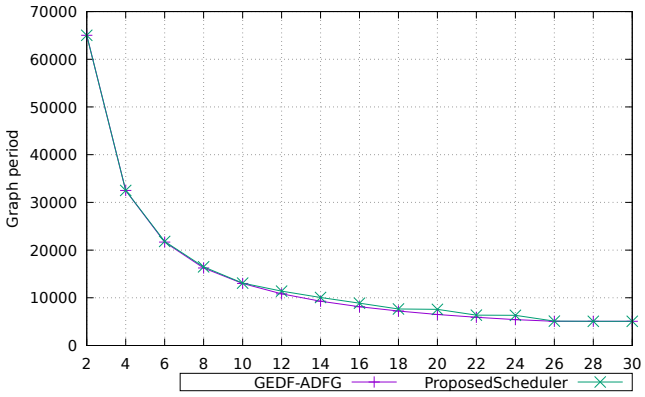


Figure 9: Evaluation of the graph period computed by Algorithm 2 compared to ADFG with global EDF policy. Number of cores in abscissa.

adapted from the forced-forward demand bound function). ADFG considers that all actors are periodic and computes their optimal smallest period for a given number of cores. ADFG also computes delays on the buffers; these delays are kept in the input of Algorithm 2. Then Algorithm 2 is initially run with a graph period equal to the sum of all WCET, and the actual latency[6] found by Algorithm 2 is kept as a result. Indeed the latency found is the smallest graph period achievable by the proposed scheduler and it may be smaller than the input graph period since Algorithm 2 is greedy and uses idle time as soon as possible.

The results are depicted in Figure 9, for the Beamformer application from the StreamIt benchmark. Beamformer graph contains 57 actors and 70 edges, each actor is fired only once. For each number of cores, Algorithm 2 finds a graph period close to the optimal, and even optimal from 28 cores. The optimal graph period is equal to the greatest WCET (5076) since numerous delays are added by ADFG and break the data dependencies.

7 RELATED WORK

To our knowledge, only the MAPS [9] tool accepts SDF graphs with partially periodic constraints as specified in this paper. However, MAPS does not exactly compute a schedule, but instead it checks if execution traces can be executed on the targeted architecture. As MAPS is not freely accessible, we did not compare to it. Deadlock [12] and consistency [14] analysis of SDF graphs with partially periodic constraints have also been studied. Other tools and papers are closely related to our scheduling problem and are listed in the three next paragraphs, according to there category.

Schedulers of real-time tasks with precedence constraints. Real-time systems have been widely studied for online periodic scheduling, the most common schedulers being EDF and Fixed Priority (FP). Yet an offline part is still needed in most of the online schedulers: either to compute the deadlines as in the Chetto's algorithm [10] to respect precedence constraints under EDF, or to compute the task priorities in the case of FP. Some online schedulers also take into account periodic and aperiodic tasks [30]; they still may need an offline pre-schedule [16], and may rely on EDF [23]. Regarding offline non-preemptive scheduling, there exists an ILP formulation [44] for sporadic and periodic tasks under EDF and FP. A CP solution for periodic tasks only has also been formulated [38]. Both ILP and CP formulations have a high complexity and thus are not scalable.

Schedulers of SDF graphs with periodic constraints. The Darts tool [2] is able to schedule SDF graphs under a throughput constraint, equivalent to a graph period constraint, for EDF and FP schedulers. ADFG [21] is similar to Darts, but it optimizes the throughput under a total buffer size constraint. SDF graphs can be modeled with synchronous languages such as Prelude [33], which generates code for EDF and FP schedulers. Still using synchronous language, activation clocks with precedences [11]. can be composed and checked. Yet, in all the aforementioned tools of this paragraph, all SDF actors are periodic. Minimal actor periods can be be computed independently from the scheduling policy [1], but only for SRDF graphs. Note that a polynomial algorithm [39] exists for the unicore processor case under EDF, with real-time tasks being specific SDF graphs. Other papers [5] specify throughput constraints on the only input or output actor of an SDF graph G, which is equivalent to specify a graph period T_G.

Schedulers of SDF graphs with latency constraints. A latency constraint on an SDF graph G is equivalent to a throughput constraint or graph period if and only if the scheduler assumes Assumption 1 and there is no delay in G (except to break cycles). Indeed if delays are present, as in Figure 2, the latency may be higher than the graph period. Latency constraints for SDF graphs have been heavily studied, especially symbolically. For example, the latency has been analyzed either without scheduling assumption to derive upper and lower bounds [24], or with self-time scheduling of SDF [18] and SRDF [32] graphs. Practically, the Ptolemy [15] tool supporting SDF graphs has been extended to perform timing verification, as latency, through system simulation [20]. Finally, there exists an offline scheduler accepting throughput and latency constraints on SDF graphs [29]; it takes into account communications and computes the static schedule with ILP and heuristics. Thus all these tools tackle only a small subset of partially periodic constraints: the specific case of one graph period without any delay on the graph.

8 CONCLUSION AND FUTURE WORK

A few necessary conditions and an offline non-preemptive scheduling algorithm have been presented in order to analyze and synthesize the scheduling of partially periodic SDF graphs. These results hold under a weak assumption on the execution of the systems: the presence of barriers at each graph period. The proposed algorithms have, at most, a linearithmic complexity and can thus be used on large cyber-physical systems modeled as SDF graphs. Experiments show that the proposed non-preemptive scheduler is fast, scalable, and efficient. Next step is to extend the schedulability analysis to model heterogeneous hardware, that is becoming the new standard for embedded systems. Another direction for future work is to take into account inter-core communication time while assessing the real-time schedulability.

[6] In this paragraph, the *latency* is taken as the finish time of the last task in the schedule during one scheduler iteration, instead of during one graph iteration as in Section 7.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement №732105 and from the Région Bretagne (France) under grant ARED 2017 ADAMS. We would like to thank J. Castrillon for his kind answers to our questions about the MAPS tool.

REFERENCES

[1] H. I. Ali, B. Akesson, and L. M. Pinho. 2015. Generalized Extraction of Real-Time Parameters for Homogeneous Synchronous Dataflow Graphs. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 701–710. https://doi.org/10.1109/PDP.2015.57

[2] M. Bamakhrama and T. Stefanov. 2011. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*. 195–204. https://doi.org/10.1145/2038642.2038672

[3] Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming* 19, 2 (1992), 87 – 152. https://doi.org/10.1016/0167-6423(92)90005-V

[4] Marko Bertogna and Sanjoy Baruah. 2011. Tests for Global EDF Schedulability Analysis. *J. Syst. Archit.* 57, 5 (May 2011), 487–497. https://doi.org/10.1016/j.sysarc.2010.09.004

[5] S. S. Bhattacharyya and W. S. Levine. 2006. Optimization of signal processing software for control system implementation. In *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*. 1562–1567. https://doi.org/10.1109/CACSD-CCA-ISIC.2006.4776874

[6] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. 1996. Cycle-static Dataflow. *Trans. Sig. Proc.* 44, 2 (Feb. 1996), 397–408. https://doi.org/10.1109/78.485935

[7] Bruno Bodin, Youen Lesparre, Jean-Marc Delosme, and Alix Munier-Kordon. 2014. Fast and Efficient Dataflow Graph Generation. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems (SCOPES '14)*. ACM, New York, NY, USA, 40–49. https://doi.org/10.1145/2609248.2609258

[8] B. Bodin, A. Munier-Kordon, and B. D. de Dinechin. 2012. K-Periodic schedules for evaluating the maximum throughput of a Synchronous Dataflow graph. In *2012 International Conference on Embedded Computer Systems (SAMOS)*. 152–159. https://doi.org/10.1109/SAMOS.2012.6404169

[9] J. Castrillon, R. Leupers, and G. Ascheid. 2013. MAPS: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs. *IEEE Transactions on Industrial Informatics* 9, 1 (Feb 2013), 527–545. https://doi.org/10.1109/TII.2011.2173941

[10] H. Chetto, M. Silly, and T. Bouchentouf. 1990. Dynamic Scheduling of Real-time Tasks Under Precedence Constraints. *Real-Time Syst.* 2, 3 (Sept. 1990), 181–194. https://doi.org/10.1007/BF00365326

[11] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. 2006. N-synchronous Kahn Networks: A Relaxed Model of Synchrony for Real-time Systems. *SIGPLAN Not.* 41, 1 (Jan. 2006), 180–193. https://doi.org/10.1145/1111320.1111054

[12] P. Derler, K. Ravindran, and R. Limaye. 2016. Specification of precise timing in synchronous dataflow models. In *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 85–94. https://doi.org/10.1109/MEMCOD.2016.7797751

[13] Sudarshan K. Dhall and C. L. Liu. 1978. On a Real-Time Scheduling Problem. *Operations Research* 26, 1 (1978), 127–140. http://www.jstor.org/stable/169896

[14] Paul Dubrulle, Christophe Gaston, Nikolai Kosmatov, Arnault Lapitre, and Stéphane Louise. 2019. A Data Flow Model with Frequency Arithmetic. In *Fundamental Approaches to Software Engineering*, Reiner Hähnle and Wil van der Aalst (Eds.). Springer International Publishing, Cham, 369–385.

[15] J. Eker, J. W. Janneck, E. A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. 2003. Taming heterogeneity - the Ptolemy approach. *Proc. IEEE* 91, 1 (Jan 2003), 127–144. https://doi.org/10.1109/JPROC.2002.805829

[16] Gerhard Fohler. 1995. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings 16th IEEE Real-Time Systems Symposium*. 152–161. https://doi.org/10.1109/REAL.1995.495205

[17] T. Gee, J. James, W. Van Der Mark, P. Delmas, and G. Gimel'farb. 2016. Lidar guided stereo simultaneous localization and mapping (SLAM) for UAV outdoor 3-D scene reconstruction. In *2016 International Conference on Image and Vision Computing New Zealand (IVCNZ)*. 1–6. https://doi.org/10.1109/IVCNZ.2016.7804433

[18] A. H. Ghamarian, S. Stuijk, T. Basten, M. C. W. Geilen, and B. D. Theelen. 2007. Latency Minimization for Synchronous Data Flow Graphs. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*. 189–196. https://doi.org/10.1109/DSD.2007.4341468

[19] T. Grandpierre, C. Lavarenne, and Y. Sorel. 1999. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Hardware/Software Codesign, 1999. (CODES '99) Proceedings of the Seventh International Workshop on*. 74–78. https://doi.org/10.1145/301177.301489

[20] L. Guo, Q. Zhu, P. Nuzzo, R. Passerone, A. Sangiovanni-Vincentelli, and E. A. Lee. 2014. Metronomy: A function-architecture co-simulation framework for timing verification of cyber-physical systems. In *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 1–10. https://doi.org/10.1145/2656075.2656093

[21] Alexandre Honorat, Hai Nam Tran, Loïc Besnard, Thierry Gautier, Jean-Pierre Talpin, and Adnan Bouakaz. 2017. ADFG: a scheduling synthesis tool for dataflow graphs in real-time systems. In *International Conference on Real-Time Networks and Systems* . Grenoble, France, 1–10. https://doi.org/10.1145/3139258.3139267

[22] W. A. Horn. 1974. Some simple scheduling algorithms. *Naval Research Logistics Quarterly* 21, 1 (1974), 177–185. https://doi.org/10.1002/nav.3800210113

[23] D. Isovic and G. Fohler. 2000. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *Proceedings 21st IEEE Real-Time Systems Symposium*. 207–216. https://doi.org/10.1109/REAL.2000.896010

[24] Jad Khatib, Alix Munier-Kordon, Enagnon Cedric Klikpo, and Trabelsi-Colibet Kods. 2016. Computing latency of a real-time system modeled by Synchronous Dataflow Graph. In *Real-Time Networks and Systems RTNS*. Brest, France, 87 – 96. https://doi.org/10.1145/2997465.2997479

[25] Yu-Kwong Kwok and Ishfaq Ahmad. 1999. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.* 31, 4 (Dec. 1999), 406–471. https://doi.org/10.1145/344588.344618

[26] Yu-Kwong Kwok, I. Ahmad, and Jun Gu. 1996. FAST: a low-complexity algorithm for efficient scheduling of DAGs on parallel processors. In *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*, Vol. 2. 150–157 vol.2. https://doi.org/10.1109/ICPP.1996.537394

[27] E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (Sept 1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

[28] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*. ACM.

[29] Jing Lin, Andreas Gerstlauer, and Brian L. Evans. 2012. Communication-aware Heterogeneous Multiprocessor Mapping for Real-time Streaming Systems. *Journal of Signal Processing Systems* 69, 3 (01 Dec 2012), 279–291. https://doi.org/10.1007/s11265-012-0674-6

[30] Giuseppe Lipari and Giorgio Buttazzo. 2000. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal of Systems Architecture* 46, 4 (2000), 327 – 338. https://doi.org/10.1016/S1383-7621(99)00009-0

[31] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (Jan. 1973), 46–61. https://doi.org/10.1145/321738.321743

[32] Orlando M. Moreira and Marco J. G. Bekooij. 2007. Self-Timed Scheduling Analysis for Real-Time Applications. *EURASIP Journal on Advances in Signal Processing* 2007, 1 (2007), 083710. https://doi.org/10.1155/2007/83710

[33] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. 2011. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems* 21, 3 (2011), 307–338. https://hal.inria.fr/inria-00638936

[34] Maxime Pelcat. 2010. *Rapid Prototyping and Dataflow-Based Code Generation for the 3GPP LTE eNodeB Physical Layer mapped onto Multi-Core DSPs*. Theses. INSA de Rennes. https://tel.archives-ouvertes.fr/tel-00578043

[35] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi. 2014. Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*. 36–40. https://doi.org/10.1109/EDERC.2014.6924354

[36] Rodolfo Pellizzoni, Patrick Meredith, Min-Young Nam, Mu Sun, Marco Caccamo, and Lui Sha. 2009. Handling Mixed-criticality in SoC-based Real-time Embedded Systems. In *Proceedings of the Seventh ACM International Conference on Embedded Software (EMSOFT '09)*. ACM, New York, NY, USA, 235–244. https://doi.org/10.1145/1629335.1629367

[37] Jonathan Piat, Philippe Fillatreau, Daniel Tortei, Francois Brenot, and Michel Devy. 2018. HW/SW co-design of a visual SLAM application. *Journal of Real-Time Image Processing* (16 Nov 2018). https://doi.org/10.1007/s11554-018-0836-2

[38] Wolfgang Puffitsch, Eric Noulard, and Claire Pagetti. 2015. Off-line mapping of multi-rate dependent task sets to many-core platforms. *Real-Time Systems* 51, 5 (01 Sep 2015), 526–565. https://doi.org/10.1007/s11241-015-9232-1

[39] Abhishek Singh, Pontus Ekberg, and Sanjoy Baruah. 2018. Uniprocessor scheduling of real-time synchronous dataflow tasks. *Real-Time Systems* (21 May 2018). https://doi.org/10.1007/s11241-018-9310-2

[40] S. Stuijk, M.C.W. Geilen, and T. Basten. 2006. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*. IEEE Computer Society Press, Los Alamitos, CA, USA, 276–278. https://doi.org/10.1109/ACSD.2006.23

[41] Pranav Tendulkar, Peter Poplavko, and Oded Maler. 2013. Symmetry Breaking for Multi-criteria Mapping and Scheduling on Multicores. In *Formal Modeling and Analysis of Timed Systems*, Víctor Braberman and Laurent Fribourg (Eds.).

Springer Berlin Heidelberg, Berlin, Heidelberg, 228–242.

[42] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, London, UK, UK, 179–196. http://dl.acm.org/citation.cfm?id=647478.727935

[43] Shuhuan Wen, Miao Sheng, Chunli Ma, Zhen Li, H. K. Lam, Yongsheng Zhao, and Jingrong Ma. 2018. Camera Recognition and Laser Detection based on EKF-SLAM in the Autonomous Navigation of Humanoid Robot. *Journal of Intelligent & Robotic Systems* 92, 2 (01 Oct 2018), 265–277. https://doi.org/10.1007/s10846-017-0712-5

[44] J. Xiao, S. Altmeyer, and A. Pimentel. 2017. Schedulability Analysis of Non-preemptive Real-Time Scheduling for Multicore Processors with Shared Caches. In *2017 IEEE Real-Time Systems Symposium (RTSS)*. 199–208. https://doi.org/10.1109/RTSS.2017.00026

[45] Xinzheng Zhang, Ahmad B. Rad, and Yiu-Kwong Wong. 2012. Sensor Fusion of Monocular Cameras and Laser Rangefinders for Line-Based Simultaneous Localization and Mapping (SLAM) Tasks in Autonomous Mobile Robots. In *Sensors*.