CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

A Novel Heuristic Framework for Offline IMA Schedule Generation for Multicore Platforms

Alexandre Esper*
Jatin Arora*
Geoffrey Nelissen
Eduardo Tovar*

*CISTER Research Centre
CISTER-TR-240503

2024/06/11

A Novel Heuristic Framework for Offline IMA Schedule Generation for Multicore Platforms

Alexandre Esper*, Jatin Arora*, Geoffrey Nelissen, Eduardo Tovar*

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: alexandre.esper@capgemini.com, jatin@isep.ipp.pt, gnn@isep.ipp.pt, emt@isep.ipp.pt

https://www.cister-labs.pt

Abstract

Ensuring temporal predictability is one of the most important factors while designing applications for the avionics domain. Consequently, time-triggered scheduling (TT) is prevalent in safety-critical systems because TT scheduling is more predictable as the schedule is constructed at design time and is enforced at run-time. This allows system designers to determine the precise timing of each event, which is particularly important, for instance, in the design of control systems. Among others, one of the most important challenges of solutions implementing TT scheduling of IMA applications is scalability, since the next-generation avionics systems must be able to handle an increasingly large number of applications running on top of their embedded multi/many-core platforms. The existing approaches are efficient for smaller problems but do not scale well when the search space becomes large. To fill this gap, this paper proposes a novel scheduling heuristic framework for the next-generation avionics systems, which can efficiently generate the schedule for a large number of ARINC-653 compliant IMA applications running on top of multi/many-core platforms. The experimental results reveal that the proposed framework can outperform the state-of-the-art by improving the schedulability ratio up to 46% even for the threshold timeout limit, i.e., the maximum time allowed to find a solution, of 4 hours.

A Novel Heuristic Framework for Offline IMA Schedule Generation for Multicore Platforms

Alexandre Esper[*†], Jatin Arora[§†], Geoffrey Nelissen[‡], Eduardo Tovar[†]
[*]Capgemini Engineering, Porto, Portugal [†]CISTER, ISEP, Porto, Portugal [§]VORTEX CoLab, Portugal
[‡]Eindhoven University of Technology, Eindhoven, the Netherlands

I. Introduction

In recent years, the industry has been confronted with the inevitable trend towards multicore processing platforms, which allows to greatly improve the performance/cost ratio of the system. Concurrently, the industry has shown an increasing interest in developing methods and tools to implement, deploy, validate, and certify independently developed applications of different "criticalities" on the same computing node. Such integrated systems are commonly referred to as mixed-criticality systems (MCS) [5], [30]. In our previous work [11], we provided an industrial view on the notion of mixed-criticality systems and showed that some of the existing works that are built upon the Vestal model [30] have some limitations, e.g, it considers that lower criticality tasks can be suspended in case higher criticality tasks overshoot their execution budget. This can be problematic, as safety-critical systems such as avionics generally require strict *space and time partitioning* among tasks of different criticalities executing on the same platform. In the aeronautical domain, this partitioning approach is referred to as the Integrated Modular Avionics (IMA) concept. ARINC-653 [2] is a standard widely adopted in the avionics industry for the development of IMA systems to enforce strong time and space partitioning. This allows applications of different criticalities (also known as Design Assurance Levels) to be developed and run independently on the same hardware platform. Although the ARINC-653 was originally defined for single-core architectures, it has been extended to multicore computing platforms [1] [16].

Ensuring temporal predictability is one of the most important factors while designing applications for the avionics domain. Consequently, *time-triggered scheduling* (TT) is prevalent in safety-critical systems such as [22], because TT scheduling is more predictable as the schedule is constructed at design time and is enforced at run-time. This allows system designers to determine the precise timing of each event, which is particularly important, for instance, in the design of control systems. Among others, some of the most important goals for solutions implementing TT scheduling of IMA applications are: 1) generation ARINC-653 compliant TT schedule; 2) efficient generation of a TT schedule, i.e., within a reasonable time; 3) generation of a TT schedule that efficiently utilizes the computing platform; 4) generation of a TT schedule that is scalable as per the requirements of modern avionic systems.

Several approaches exist in the literature [7], [8], [18], [22], [24], [31] that focus on TT scheduling of IMA applications. However, these existing approaches are either not compliant with ARINC-653 specifications [31] or are not scalable to large IMA applications and the number of processing cores [7], [8], [24]. This can be problematic as the aeronautics industry is witnessing an unprecedented increase in the complexity of aircraft-embedded computers [12]. Consequently, the traditional aviation development processes are having difficulties keeping up with the development requirements of large-scale complex avionics systems, mainly in terms of cost, time, and reusability [9]. This trend suggests that future avionics systems will require also more sophisticated methods and tools that will enable handling larger systems with a higher number of cores [20]. For instance, the Boeing 787, which also uses an IMA architecture, hosts over 80 different applications in the core processor cabinet [13]. With such an increase in size and complexity, the importance of efficient, scalable, and effective real-time scheduling solutions becomes even more critical.

To achieve this goal, this paper proposes a *novel heuristic framework* for the next-generation avionics systems that can run a large number of ARINC-653 compliant IMA applications on top of multi/many core platform. The proposed framework can be used to efficiently generate a TT schedule for a large number of ARINC-653 IMA applications running on a large-scale multi/many-core platforms. Furthermore, the proposed framework allows partition instances of the same IMA application to be executed in any core to efficiently utilize the computing platform. The experimental results reveals that the proposed framework can outperform the state-of-the-art [24] by improving the schedulability ratio up to 46% even for the threshold timeout limit, i.e., the maximum time allowed to find a solution, of 4 hours.

The main **contributions** of the proposed heuristic framework are:

1. A novel algorithm for building a graph of the hierarchy of IMA applications partitions instances in an efficient manner based on a set of defined rules, which takes into account the impact of IMA partitions instances with smaller execution demands on IMA partitions instances with larger execution demands.

2. A novel algorithm to build a graph that abstracts the schedule in a hierarchy of smaller schedule intervals (sub-intervals), which is built based on a set of defined rules and that allows us to transform a large NP-Hard problem into a series of smaller problems that are relatively straightforward to solve;

3. A novel multi-core schedulability test that allows us to efficiently search for suitable sub-intervals in the schedule sub-intervals graph to allocate the partition instances of each IMA application; and

4. A novel scheduling strategy that allows to efficiently build the schedule, by scheduling the partition instances assigned to each sub-interval of the schedule sub-intervals graph.

II. System Model

We consider a multicore platform comprising m identical cores, denoted by $\Pi_1, \Pi_2, \ldots \Pi_m$, with $m \in \mathbb{Z}^+$, where $\mathbb{Z}^+$ is the set of positive integers. We consider a set of n IMA hosted applications denoted by $\alpha = \{\alpha_1, \alpha_2, ..., \alpha_n\}$, where each application α_i, $1 \leq i \leq n$, with $n \in \mathbb{Z}^+$. Associated with the n applications we have n processing partitions $P_1, ..., P_n$. Each application α_i is associated to a single processing partition P_i. Each P_i is defined by an activation period T_i, an execution budget B_i, a relative deadline D_i and an offset O_i in relation to the start of the period T_i, which means that the partitions are *asynchronous* in relation to the start of the schedule. The value of B_i can be computed using existing methods [29]. We assume that each P_i has constrained deadline, i.e., $D_i \leq T_i$. We assume that each P_i releases an infinite number of processing partitions instances, which we denote as $P_{i,j}$. Each instance $P_{i,j}$ of a processing partition of an application α_i is released periodically with period T_i until the end of the Major Frame (MAF). The length of the MAF is the *lcm* of all partition periods, i.e., $MAF := lcm_{\alpha_i \in \alpha}\{T_i\}$. The absolute release time (resp. start time) of the j^{th} instance of processing partition P_i is denoted as $r_{i,j}$ (resp. $s_{i,j}$). The absolute deadline of the j^{th} instance of P_i is denoted as $d_{i,j}$. We designate this set of input parameters of each $P_{i,j}$ as $\mathcal{P}_{i,j}$ and the set of all $\mathcal{P}_{i,j}$ as $\mathcal{P}$.

As we consider an offset O_i associated to each application α_i, we need to ensure that our schedule can be successfully repeated towards infinity. Hence, we extend the MAF definition to an observation window OBW. The OBW is defined as $OBW = O_{max} + MAF$, where $O_{max} = max_{\alpha_i \in \alpha} O_i$. To build our repetitive schedule, we only consider that $\forall P_{i,j},\ r_{i,j} < OBW$. Within OBW, we also define the total utilization u_{tot} as $\forall_i : u_{tot} = \sum_1^n B_i^T/(m \times OBW)$, where B_i^T is the sum of the B_i of all α_i partitions instances $P_{i,j}$ within OBW, i.e., with $r_{i,j} < OBW$.

We assume that each $P_{i,j}$ of an α_i can execute on any of the m cores. This property brings an advantage over existing solutions [18], [24] that restrict the $P_{i,j}$ of an α_i to run on the same core, because it allows more efficient utilization of the computing platform. We assume in this work that the IMA processes of an α_i that run inside the $P_{i,j}$ will always resume its execution in the same state after a migration (either on the same or another core). Ensuring that the processor cache is always flushed whenever a migration occurs is one of the methods that can be used to ensure that these migrations can be safely performed. The discussion of other methods that can be used in conjunction with cache flushing to ensure safe migration are left outside the scope of this work.

To ensure deterministic scheduling of the partitions, we define the following **set of assumptions**:

- Partitions are scheduled on a fixed cyclic basis - a Major Frame (MAF) of fixed duration is maintained by the OS scheduler, which is usually defined as a multiple of the least common multiple of all partition periods;
- The partitions are then allocated to one or more execution windows within the MAF;
- Partitions are activated according to the defined offset from the start of the MAF and remain active for the duration of their execution windows;
- The sequence of activation of the partitions are defined during design time using configuration tables;
- The configuration table for the partition schedule contains the order of activation and the length of the execution windows within the MAF;
- A partition periodically releases a potential infinite number of “partition instances";
- The processing partitions can be mapped to any of the available processing cores;
- Mapping of a partition instance between cores is not allowed, but each processing partition instance released by the same application may run on different cores.

The problem of synchronizing access to I/O resources is out of the scope of this paper and is kept for future work.

Although our heuristic framework does not use a CP approach, we do formulate some **constraints** that must be respected by our implementation. First, we need to ensure that the start time of all $P_{i,j}$ of all α_i is not negative (Constraint (1)) and that the deadlines of each α_i are always respected, i.e., the completion time of each $P_{i,j}$ is no later than its corresponding application's absolute deadline (Constraint (2)). Since partitions are executed periodically, the corresponding $P_{i,j}$ cannot be released before the beginning of each application period (Constraint (3)).

$$\forall P_{i,j} : s_{i,j} \geq 0 \tag{1}$$

$$\forall P_{i,j} : s_{i,j} + B_i \leq D_i \tag{2}$$

$$\forall P_{i,j} : r_{i,j} \geq (j-1) \times T_i \tag{3}$$

Knowing that only one $P_{i,j}$ can execute on a given core m at a time, we must ensure that the $P_{i,j}$ of different partitions allocated to that core do not overlap with each other. This is enforced by defining Constraint (4), where j and l denotes the j^{th} and l^{th} instances of processing partitions P_i and P_k, respectively, with $1 \leq i \leq n$ and $1 \leq k \leq n$.

$$\begin{aligned} \forall m, \forall P_{i,j}, \forall P_{k,l} | P_{i,j} \neq P_{k,l} : s_{i,j} \geq s_{k,l} + B_{k,l} \quad \vee \\ s_{k,l} \geq s_{i,j} + B_{i,j} \end{aligned} \tag{4}$$

The number of constraints defined by (4) can rapidly grow with the number of applications, thus reducing the effectiveness of the CP approach, especially with higher number of cores under high load. Our proposed heuristic framework addresses this problem by efficiently breaking down this NP-hard problem [15] into a series of smaller and simpler problems that are relatively straightforward to solve, as discussed in the next section.

III. Proposed Heuristic Framework

The goal of our heuristic framework is to efficiently generate an offline IMA-compliant schedule, where each $P_{i,j}$ is mapped to any of the m cores and is assigned a start time $s_{i,j}$, such that all the previously defined constraints are respected. Our heuristic consists of a set of deterministic algorithms that run sequentially, i.e., given a defined set of inputs, the heuristic will always produce exactly the same schedule. We use directed acyclic graphs (DAG), and more specifically *augmented trees*, to create hierarchical abstractions of partitions instances and schedule sub-intervals. This heuristic process consists of **four phases**. In **Phase 1**, we construct a graph reflecting the $P_{i,j}$ hierarchy. This graph is used to decide the order in which we schedule each $P_{i,j}$. In **Phase 2**, we build another graph, which abstracts the schedule in a hierarchy of smaller schedule intervals, thus simplifying the scheduling problem. Then in **Phase 3**, we traverse the graph built in Phase 2 and allocate each $P_{i,j}$ to a schedule interval. Finally, in **Phase 4**, we allocate each $P_{i,j}$ to one of the m cores within the allocated intervals and assign a start time $s_{i,j}$ to it. The referred phases are explained in detail in the following sections.

A. Phase 1 - IMA Partitions Hierarchy Graph Construction

This phase takes input a set of parameters of each IMA application α_i, namely $\mathcal{P}$. Based on these inputs, we create the IMA Partitions Hierarchy Graph, P_{graph}, which is a DAG object with its properties and methods (functions), which is built upon a set of defined rules. P_{graph} is used in our heuristic to determine the order in which we allocate the partitions to schedule sub-intervals. P_{graph} is formally defined as $P_{graph} = (\mathcal{V}_p, \mathcal{E}_p)$, where each vertex $v_p \in \mathcal{V}_\mathcal{P}$ is an object representing an IMA partition instance $P_{i,j}$, with $p \in [1..T^P]$, where $T^P = |\mathcal{P}|$. Each v_p also has properties and methods associated to it. We store several important data in each v_p object, including the input parameters from $\mathcal{P}$. We use the "." operator to access the data stored in v_p. For example, to retrieve the deadline of the $P_{i,j}$ stored in a v_p object, we perform the following operation: $r_{i,j} \leftarrow v_p.deadline$. Note that given a $P_{i,j}$, it is possible to access the respective v_p object from P_{graph} through the following operation: $v_p \leftarrow P_{graph}[p]$. The edges e_p of P_{graph} connect the vertices

v_p in a hierarchical way, according to a set of defined criteria that will be explained in the following sections.

Figure 1a depicts an example schedule with the intuition behind P_{graph} construction process. In this scenario, we consider a set of random partitions $P1$ to $P6$ with their respective releases $r_{i,j}$ and deadlines $d_{i,j}$ in absolute time units. From this simple example, we can intuitively verify that $P1$ is more constrained by $P4$, $P5$, and $P6$, and not so much by $P2$ and $P3$. This principle leads to the idea of creating a graph of the partitions hierarchy, where each partition corresponds to a vertex of the graph, connected by edges in such a way that a hierarchy of those partitions is formed.

Equation 5 defines the condition to determine the hierarchical relationship between partitions. The equation establishes that for a partition $P_{k,l}$ to be considered a child of another partition $P_{i,j}$, two conditions must be met. Firstly, $P_{i,j}$ must encompass $P_{k,l}$, i.e., it must have an earlier release time and a later deadline than $P_{k,l}$. Secondly, there should not exist any other distinct $P_{s,t}$ that simultaneously encompasses $P_{k,l}$ and is encompassed by $P_{i,j}$.

$$\begin{aligned}&\forall P_{k,l}, (\exists P_{i,j}, i \neq k, \mid r_{i,j} \leq r_{k,l} \wedge d_{k,l} \leq d_{i,j}) \wedge \\ &(\nexists\, P_{s,t}, s \neq k, s \neq i, \mid ((\, r_{i,j} \leq r_{s,t} \wedge d_{s,t} \leq d_{i,j}) \wedge \\ &(\, r_{s,t} \leq r_{k,l} \wedge d_{k,l} \leq d_{s,t})))\end{aligned} \tag{5}$$

Note that according to Equation 5, a parent partition $P_{i,j}$, can have only one child $P_{k,l}$ in the interval $[r_{k,l}, d_{k,l}]$, but one child partition $P_{k,l}$ is allowed to have several parents $P_{i,j}$. For example, in Figure 1a, if the deadline of $P2$ were equal to 26, then $P2$ would also be considered as a parent of $P6$, thus both $P2$ and $P3$ would be considered as parents of $P6$.

Next, we describe *Phase 1* sub-phases to create P_{graph}.

Sub-Phase 1.1 - Generation of a list of schedule events and Initialization of partitions hierarchy graph. The purpose of this sub-phase of our heuristic is twofold: the generation of the schedule events data set $\mathcal{E}$ and the initialization of P_{graph} vertices v_p. $\mathcal{E}$ is defined as $\mathcal{E} = \{\mathcal{T}_1, .., \mathcal{T}_q\}$, $q \in \mathbb{Z}^+$. Each $\mathcal{T}_x \in \mathcal{E}$, $x \in [1, q]$, is defined as $\mathcal{T}_x = \{t_x, \{E_1, .., E_l\}\}$, $l \in \mathbb{Z}^+$, where $0 \leq t_x \leq d_{last}$, and where d_{last} is the latest absolute deadline among all $P_{i,j}$ with $r_{i,j} < OBW$. Each $E_p \in \mathcal{T}$ associated with a $P_{i,j}$ with identifier p is represented as $E_p = \{\text{"release"} \vee \text{"deadline"}, p\}$, with $p \in [1..T^P]$. For the sake of simplicity, we designate each E_p as either E_p^r or E_p^d, to represent the release or deadline events of a $P_{i,j}$.

This sub-phase takes $\mathcal{P}$ as input, and performs the following actions:

- initialize P_{graph} vertex with identifier p
- iterate over all $P_{i,j} \in \mathcal{P}$ to:
  - add the vertices objects v_p representing each $P_{i,j}$ to P_{graph};
  - add the E_p^r or E_p^d of $P_{i,j}$ to $\mathcal{E}$;
- sort $\mathcal{E}$ in ascending order of time stamps t to yield $\mathcal{E}^O$.

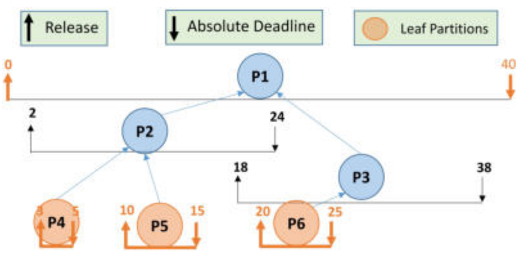
It is important to highlight that at this stage the vertices v_p are not connected by the respective edges.

Sub-Phase 1.2 - Construction of P_{graph}. In this sub-phase, we use function *ConstructPartitionHierarchyGraph()* (Algorithm 1) to build P_{graph}. This function iterates over all $\mathcal{T} \in \mathcal{E}^O$ to determine the hierarchical parents of each $P_{k,l}$ that satisfy the condition given by Equation 5. The analysis of each $\mathcal{T} \in \mathcal{E}^O$ is composed of three parts. In the first part (lines 2 to 6), we iterate over all $E_p \in \mathcal{T}_x$, and determine the set $\mathcal{A}_x$ containing the identifiers b of the $P_{i,j}$ that are active at time $t_x \in \mathcal{T}_x$. A $P_{i,j}$ is considered to be active when $r_{i,j} \leq t_x \leq d_{i,j}$. Note that being active at t_x is a pre-condition for a $P_{i,j}$ to be a parent of $P_{k,l}$, with $i \neq k$. In the second part of the analysis we determine the parent(s) of each $P_{k,l}$ (lines 7 to 34). We iterate again over all $E \in \mathcal{T}$ (line 8) to determine the $P_{i,j}$ with identifier b, with $b \in E_p$, that can be a parent of each $P_{k,l}$ with identifier a. The auxiliary set $\mathcal{Q}$ initialized in line 7 is simply used to prevent a child vertex from being added as a parent of its parent. In line 9 we get the identifier a of the $P_{k,l}$. If $e \in E$ is a deadline event and $\mathcal{A}_x$ is not empty (line 10), we initialize the set $\mathcal{A}_y$ (line 11) that will store the candidate parents $P_{i,j}$ of $P_{k,l}$. Then in line 12, we iterate over all active $P_{i,j}$ with identifier $b \in \mathcal{A}_x$. The condition in lines 13 and 14 filters the $P_{i,j}$ that are candidate parents. In line 15 we test the special case where a $P_{k,l}$ with identifier a has exactly the same release and deadline as that of the $P_{i,j}$ with identifier k. The operations $v_b.r$, $v_a.r$ and $v_b.d$, $v_a.d$ allow us to retrieve the releases and deadlines stored in the objects v_b and v_a, respectively. If that is the case, we can add it straight away as parent of $P_{k,l}$ (lines 16 and 17), reset $\mathcal{A}_y$ (line 18), and add a to $\mathcal{Q}$, to prevent $P_{k,l}$ from being added as parent of $P_{i,j}$. We then break the loop (line 20) and proceed to the analysis of the next event E_p. Otherwise, if the

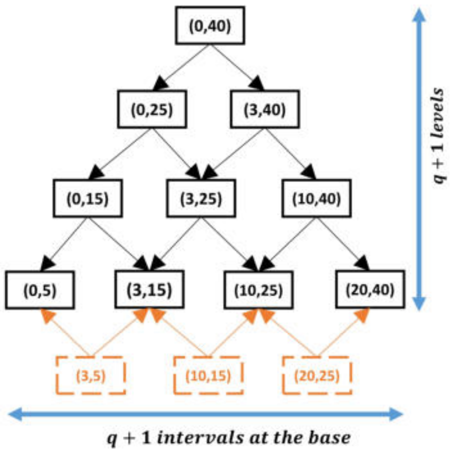
Algorithm 1: Sub-phase 1.2: P_{graph} construction

Output: P_{graph} construction completed

1 **Function** `ConstructPartitionHierarchyGraph(`$\mathcal{P}$, $\mathcal{E}^O$, P_{graph}`):`
2   $\mathcal{A}_x \leftarrow \{\ \}$
3   **for** $\mathcal{T} \in \mathcal{E}^O$ **do**
4   **for** $E \in \mathcal{T}$ **do**
5   **if** *Event $e \in E$ is a "release"* **then**
6   Get $P_{i,j}$ identifier $b \in E$ and append to $\mathcal{A}_x$
7     $\mathcal{Q} \leftarrow \{\ \}$
8   **for** $E \in \mathcal{T}$ **do**
9   Get $P_{k,l}$ identifier $a \in E$
10   **if** *(Event $e \in E$ is a "deadline")* $\wedge$ $(|\mathcal{A}_x| > 0)$ **then**
11         $\mathcal{A}_y \leftarrow \{\ \}$
12   **for** $b \in \mathcal{A}_x$ **do**
13   **if** $(a \neq b) \wedge (b \notin \mathcal{Q}) \wedge$
14           $(v_b.r \leq v_a.r \wedge v_a.d \leq v_b.d)$ **then**
15   **if** *$(v_b.r = v_a.r \wedge v_a.d = v_b.d)$* **then**
16   *P_{graph}.AddParent(v_b, v_a)*
17   *P_{graph}.AddChild(v_a, v_b)*
18               $\mathcal{A}_y \leftarrow \{\ \}$
19   Append a to $\mathcal{Q}$
20   **break**
21   **else**
22   Append b to $\mathcal{A}_y$
23         $\mathcal{A}_z \leftarrow \mathcal{A}_y$
24         $p\_found = FALSE$
25   **for** $b \in \mathcal{A}_y$ **do**
26   **if** $p\_found = FALSE$ **then**
27   **for** $c \in \mathcal{A}_y$ **do**
28   **if** $(c \neq b) \wedge$
29               $(v_b.r \leq v_c.r \wedge v_c.d \leq v_b.d)$ **then**
30   **if** $|\mathcal{A}_z| = 1$ **then**
31                   $p\_found = TRUE$
32   **break**
33   **else**
34   Remove c from $\mathcal{A}_z$
35   **for** $b \in \mathcal{A}_z$ **do**
36   *P_{graph}.AddParent(v_b, v_a)*
37   *P_{graph}.AddChild(v_a, v_b)*
38   **for** $E \in \mathcal{T}$ **do**
39   **if** *Event $e \in E$ is a "deadline"* **then**
40   *Get $P_{i,j}$ identifier $b \in E$*
41   **if** $b \in \mathcal{A}_x$ **then**
42   Remove b from $\mathcal{A}_x$



(a) Intuition behind P_{graph} construction.



(b) Intuition behind I_{graph} construction.

Fig. 1: Examples describing the intuition behind the construction process of P_{graph} and I_{graph}.

condition in line 15 is not satisfied, we add b to $\mathcal{A}_y$ (line 22) and proceed to the next stage (lines 23 to 34) to select the parent(s) among the candidates in $\mathcal{A}_y$. To achieve that goal, we must test all combinations among the $P_{i,j}$ stored in $\mathcal{A}_y$ through the for loops in lines 25 and 27, to check if they are a parent of one another, thus violating the condition given by Equation 5 (lines 28 and 29). But before that, as preparation, we make a copy $\mathcal{A}_z$ of $\mathcal{A}_y$ (line 23) to avoid interfering with the iteration control variable and initialize the flag to detect when a parent has been found (line 24). If the condition in lines 28 and 29 is true, we remove the $P_{i,j}$ with identifier c from $\mathcal{A}_z$ (line 34). We proceed with this iteration until all parents are found or until only one parent is left in $\mathcal{A}_z$ (line 30). Once this analysis is finalized, we simply add the $P_{i,j}$ with identifier $b \in \mathcal{A}_z$ as parent(s) of the $P_{k,l}$ with identifier a (lines 35 to 37). The final part of the analysis (lines 38 to 42), consists simply of deactivating all $P_{i,j}$ with deadline events E^d_k at time t associated with event subset $\mathcal{T}$. The approach is the same as the one used in lines 3 to 6 but with a deadline event instead of a release.

B. Phase 2 - IMA Partitions Interval Hierarchy Graph Construction

Once P_{graph} has been built, the next phase is the construction of the IMA partitions intervals hierarchy graph, which is also a DAG object with its properties and methods (functions), similar to P_{graph}. We define this DAG as $I_{graph} = (\mathcal{V}_k, \mathcal{E}_k)$. Each vertex $v_k \in \mathcal{V}_k$ is an object representing a IMA schedule sub-interval I_k, defined as a tuple (t_b, t_e), where $0 \leq t_b < t_e \ \wedge \ t_b < t_e \leq OBW$. Given an I_k, it is possible to retrieve the respective v_i object from I_{graph} through the following operation: $v_k \leftarrow I_{graph}[I_k]$. The edges e_k of I_{graph} connect the vertices v_k in a hierarchical way, according to a set of defined criteria. The construction process of I_{graph} is implemented by Algorithm 2.

Phase 2.1 Construction of $\mathcal{LP}$ and $\mathcal{LI}$. The purpose of this sub-phase is twofold: (i) the creation of the set $\mathcal{LP}$, which contains the list of identifiers p associated with the leaf vertex $v_p \in P_{graph}$; (ii) the initialization of I_{graph} leaf vertices. A $v_p \in P_{graph}$ is defined as a leaf iff v_p.*getChild()*$= \emptyset$, where *getChild()* is a function of v_p that returns the set of identifiers of the children of v_p. Similarly, a $v_k \in I_{graph}$ is defined as a leaf iff v_k.*getChild()*$= \emptyset$. We designate each interval of $\mathcal{LP}$ as L_k, with $k \in [0..|\mathcal{LI}| - 1]$. This sub-phase is implemented by function *constructLeafPartitionList()* (line 4 of Algorithm 2), which performs the the following procedure steps:

- Iterate over each $v_p \in P_{graph}$:
  - If v_p.*getChild*$= \emptyset$:
    - Add p to $\mathcal{LP}$;
    - Add the interval I_k =(v_p.*release*, v_p.*deadline*) to $\mathcal{LI}$;
    - Create the vertex v_k associated with I_k in I_{graph};
- Sort $\mathcal{LI}$ in ascending order of interval start time t.

Algorithm 2: Phase 2.1: I_{graph} construction

Output: I_{graph} construction completed

```
1  Function
   constructIntervalHierarchyGraph(𝒫, P_graph, ℰ^O, 𝒫, ℐℬ, ℒℐ):
2      SCHEDULE_START ← 0
3      SCHEDULE_END ← max(getEventsTime(ℰ^O))
4      constructLeafPartitionList (𝒫, ℒ𝒫, ℒℐ)
5      constructIntervalGraphBaseList(ℒℐ, ℐℬ, SCHEDULE_START,
        SCHEDULE_END)
6      I_graph ← createIntervalGraph()
7      graph_depth ← | ℐℬ |
8      constructRightDiagonalVertices(ℐℬ, I_graph, graph_depth)
9      constructLeftDiagonalVertices(ℐℬ, I_graph, graph_depth)
10     addLeafIntervals(ℒℐ, ℐℬ, I_graph)
11     return
```

Phase 2.2 Construction of $\mathcal{IB}$. In this sub-phase we use the set $\mathcal{LI}$ to build another set $\mathcal{IB}$, which will contain the intervals at the base of I_{graph}. This sub-phase is implemented by function *constructIntervalGraphBaseList()* (line 5 of Algorithm 2), which performs the following procedure steps:

- Build the first interval as $I_{first} = (\mathit{SCHEDULE\_START}, I_0[1])$, where $I_0[1]$ means the upper bound of the first interval $I_0 \in \mathcal{LI}$;
- append I_{first} to $\mathcal{IB}$ and add it to I_{graph};
- Iterate over each $I_k \in \mathcal{LI}$, with $k \in [0..|\mathcal{LI}| - 1]$:
  - Build the subsequent intervals, except for the last interval, according to the following rule: $(I_k[0], I_{k+1}[1])$;
  - Append each I_k to $\mathcal{IB}$ and add it to I_{graph};
- Build the last interval as $I_{last} = (I_e[0], \mathit{SCHEDULE\_END})$, where $e = |\mathcal{LI}| - 1$;
- Append each I_{last} to $\mathcal{IB}$ and add it to I_{graph};

Phase 2.3 Construction of I_{graph} right diagonal vertices. In this sub-phase we create and connect the remaining vertices of I_{graph}, taking the set as the starting point $\mathcal{IB}$. This sub-phase is implemented by the function *constructRightDiagonalVertices()* (line 8 of Algorithm 2). Our strategy consists in building the graph diagonally. Formally, we iterate over all intervals $J_k \in \mathcal{IB}$, with $k \in [0..|\mathcal{IB}| - 1]$, compute the new interval I_k as $(J_k[0], J_{k+1}[1])$, add I_k to I_{graph} and connect I_k with I_{k-1}.

Phase 2.4 Construction of I_{graph} left diagonal vertices. Since in the previous sub-phase we have created all vertices of I_{graph}, to finalize the construction process of I_{graph} we just need to connect the remaining vertices of I_{graph}. We follow the same process from the previous sub-phase, but iterating now from the last interval of $\mathcal{IB}$ to the first. This sub-phase is implemented by the function *constructLeftDiagonalVertices()* (line 9 of Algorithm 2).

We now use Figures 1a and 1b to illustrate the intuition behind the overall construction concept of I_{graph}. According to sub-phase 2.1, we take all the *leaves* of the partitions hierarchy graph P_{graph} (i.e., $P4$, $P5$ and $P6$) to build $\mathcal{LP}$. From $\mathcal{LP}$, we extract the releases and deadlines of $P4$, $P5$ and $P6$, and build $\mathcal{LI}$, i.e., $(3,5),(10,15),(20,25)$ in Figure 1b. Then according to phase 2.2, we take the leaf intervals $\mathcal{LI}$, augment by instants $t = 0$ (i.e. *SCHEDULE_START*) and $t = OBW$ (i.e., *SCHEDULE_END* $= 40$), to build $\mathcal{IB}$, which is given by intervals $(0,5),(3,15),(10,25),(20,40)$. To build interval $(0,5)$, we take the beginning of the schedule (i.e., $t = 0$) as the start time of the interval, and the upper bound of the first interval in $\mathcal{LI}$, i.e. the upper bound of $(3,5)$. To build the next interval $(3,15)$, we take the lower bound of the first interval in $\mathcal{LI}$, and the upper bound of the second interval $(10,15)$. We continue with this process until the last interval of $\mathcal{LI}$, i.e., $(20,25)$. To build last interval $(20,40)$ of $\mathcal{IB}$, we take the lower bound of the last interval $(20,25)$ of $\mathcal{LI}$ and augment it with *SCHEDULE_END*. Then we build the I_{graph} right diagonal vertices based on $\mathcal{IB}$ as per the process defined in sub-phase 2.2. To build the right diagonals of this example graph, we start with interval $(0,5)$, then we create interval $(0,15)$ and connect it with $(0,5)$. Then we proceed with $(0,25)$ up to $(0,40)$. Then we follow the same process to build the remaining right diagonals, restarting at $(3,15)$ up to (3.40), and so on. To finalize the graph according to phase 2.4, we construct the left diagonal using the same process, which means connecting intervals $(20,40),(10,40),(3,40),(0,40)$, then $(10,25),(3,25),(0,25)$ and so on, until all vertices are connected.

C. Phase 3 - Allocating Partitions to Schedule Sub-Intervals

Phase 3.1 P_{graph} **traversal.** The goal of this this phase is to decide which $P_{i,j}$ will be assigned next to an interval I_k in I_{graph} or not. Our strategy to traverse P_{graph} consists of a bottom-up approach described in Algorithm 3. To perform the traversal, we use two FIFO queues, which we designate as *p_queue* and *p_wait_queue*, and that we initialized in lines 2 and 3. We start the process by iterating over all leaf $P_{i,j}$ whose identifiers p are stored in $\mathcal{LP}$ (line 4), assigning them to their parent interval in the base intervals $\mathcal{IB}$ with the largest lower bound t_b and enqueuing their parents (lines 5 to 8). The interval search process is implemented by function $intervalSearch()$ (line 15 and line 24), which is explained in **Phase 3.2**. In case during the first attempt to find an interval to assign each $P_{i,j}$ we detect that we have two options to choose from in the search path (lines 11 to 21), we opt not to assign $P_{i,j}$ to any of those two intervals yet, and function $intervalSearch()$ returns $TRUE$ (line 15). We then append p to *p_wait_queue* (line 17). This strategy allows us to reduce the number of decisions that we need to take, by giving all $P_{i,j}$ a first chance to try to find an interval as low as possible towards the base of I_{graph}. We proceed iterating over *p_queue* (lines 10 to 21) and *p_wait_queue* (lines 22 to 27) in this order, until both queues are empty, meaning that all $P_{i,j}$ have been assigned to an I_k. Note that the return value in line 24 will never be used, since we don't give a third chance for finding a suitable interval for q.

Phase 3.2 I_{graph} **traversal.** After selecting the $P_{i,j}$ that must be assigned to an interval in the previous phase, we now initiate or resume the interval downward search (top-down) process to allocate the $P_{i,j}$ to an I_k in I_{graph}. This process is implemented by Algorithm 4. This algorithm implements a set of four criteria that allow us to decide which downward path we will follow, i.e., whether a child interval I_k will be allowed in the *i_queue* or not. Next, we define each criterion.

Criteria 1. Enqueue child I_k in *i_queue* iff: *(msst = TRUE)* $\wedge$ $((I_k \in \mathcal{LI} \wedge I_k \notin \mathcal{IB})$ *= FALSE)*. **Rationale**: we enqueue the child I_k if it passes the *msst* and if it is not a leaf interval. For example, in Figure 1b, the leaf intervals are $(3,5)$, $(10,15)$ and $(20,25)$.

Criteria 2. Enqueue in *i_queue* the child I_k with the highest value stored in the set A^x_{list}, given by $max(A^x_{list})$. The set A^x_{list} is defined is section IV. **Rationale**: we enqueue the child I_k with the highest available CPU processing time to increase the likelihood of being able to run the $P_{i,j}$ in that I_k. Note that in the rest of this paper we use the terms CPU processing time and CPU time interchangeably.

Algorithm 3: Phase 3.1: P_{graph} traversal

Output: All $P_{i,j}$ assigned to an interval in I_{graph}
1 **Function** `reversePartitionGraphTraversal(`P_{graph}, $\mathcal{LP}$`):`
2 $p\_queue \leftarrow \{\}$
3 $p\_wait\_queue \leftarrow \{\}$
4 **for** $p \in \mathcal{LP}$ **do**
5 $p\_int = (P_{graph}[p].getRelease(), P_{graph}[p].getDeadline())$
6 Assign p to the parent in $\mathcal{IB}$ with the largest t_b
7 $p\_parents = P_{graph}[p].getParent()$
8 Add $p\_parents$ to p_queue
9 **while** $|p\_queue| > 0 \wedge |p\_wait\_queue| > 0$ **do**
10 $temp\_p\_queue \leftarrow p\_queue$
11 **for** $p \in temp\_p\_queue$ **do**
12 **if** $P_{graph}[p].allChildScheduled() = FALSE$ **then**
13 Add p to the end of the p_queue
14 **break**
15 $p\_has\_options = intervalSearch(p)$
16 **if** $p\_has\_options = TRUE$ **then**
17 Append p to p_wait_queue
18 **else**
19 $p\_parents = P_{graph}[p].getParent()$
20 Append $p\_parents$ to p_queue
21 Remove p from p_queue
22 $temp\_p\_wait\_queue \leftarrow p\_wait\_queue$
23 **for** $q \in temp\_p\_wait\_queue$ **do**
24 $q\_has\_options = intervalSearch(q)$
25 $q\_parents = P_{graph}[q].getParent()$
26 Append $q\_parents$ to p_queue
27 Remove q from p_wait_queue
28 **return**

Criteria 3. Enqueue in *i_queue* the child I_k with the highest value of A^x_{max}. If we designate any pair of child vertices as v_l, v_r, then A^x_{max} is computed as follows: $A^x_{max} = max(max(A^l_{list}), max(A^r_{list}))$. A^x_{max} is computed during the construction of I_{graph} for all I_k and stored in each v_k. **Rationale**: The value of A^{max}_x is a metric that allows us to choose a search path towards child intervals with potentially larger values of available CPU time.

Criteria 4. Given two child intervals I_l and I_r, such that $I_l[0] < I_r[0]$, then enqueue $I_r[0]$ in *i_queue*. **Rationale**: at this point we must force a decision, so we select the child interval with the latest largest lower bound.

Next, we described Algorithm 4 in detail. To perform the interval search, we also use a FIFO queue, which we designate as *i_queue*. This algorithm takes as inputs the $P_{i,j}$ identifier p and the flag p_from_queue. If this flag is $TRUE$, it means the $P_{i,j}$ comes from p_queue, otherwise, it comes from p_wait_queue.

If it comes from p_queue (line 4), we need first to compute the interval search starting point for $P_{i,j}$ in I_{graph}, which we designate as $I_{top} = (L_{bound}, U_{bound})$. To compute I_{top} lower and upper bounds, we use function $getTopInterval()$ (line 5). This function takes the $P_{i,j}$ release $r_{i,j}$ and deadline $d_{i,j}$, and iterates over $\mathcal{IB}$ to search for the intervals $I_k, I_l \in \mathcal{IB}$, with $k < l$, whose lower and upper bounds satisfy respectively the following rules:

$$L_{bound} = I_k[0] \mid I_k[0] \leq r_{i,j} < I_{k+1}[0] \quad (6)$$
$$U_{bound} = I_l[1] \mid I_{l-1}[1] < d_{i,j} \leq I_l[1] \quad (7)$$

For instance, in Figure 1, if we apply the above rule to compute the I_{top} of $P2$ in Figure 1a, it would yield the interval $(0, 25)$ in Figure 1b.

Algorithm 4: Phase 3.2: I_{graph} traversal

Output: All $P_{i,j}$ assigned to an interval in I_{graph}
Data: $\mathcal{E}^O$, P_{graph}

1 **Function** `intervalSearch(`I_{graph}, p, *p_from_queue*`)`**:**
2   $p\_bi = I_{graph}[p].getBudget()$
3   $i\_queue \leftarrow \{\}$
4   **if** $p\_from\_queue = TRUE$ **then**
5     $p\_top\_int = getTopInterval(p)$
6     $p\_top\_msst\_result, A^{tot}, A^x_{list} = performMSST(p, p\_top\_int, p\_bi)$
7   **if** $p\_top\_msst\_result = TRUE$ **then**
8       $update\_Ax\_params(p\_top\_int, p, A^{tot}, A^x_{list})$
9   Append p_top_int to i_queue
10   **else**
11   EXIT - application set not schedulable
12   **else**
13     $p\_restart\_int = P_{graph}[p].getIntervals$
14   Append $p\_restart\_int$ to i_queue
15   **while** $|i\_queue| > 0$ **do**
16     $next\_i = pop(i\_queue)$
17     $i\_child = I_{graph}[next\_i].getSortedChild()$
18     $candidate\_ints \leftarrow \{\}$
19   **if** $|i\_child| > 0$ **then**
20   **for** $each\_child \in i\_child$ **do**
21         $p\_msst\_result, A^{tot}, A^x_{list} = performMSST(next\_p, c, p\_bi)$
22   **if** $Criteria\_1 = TRUE$ **then**
23   Append $each\_child$ to $candidate\_ints$
24   **if** $(|candidate\_ints| = 2) \wedge (p\_from\_queue = TRUE)$ **then**
25       $assignPtoInt(p, next\_i)$
26   **return** TRUE
27   **else if** $(|candidate\_ints| = 2) \wedge (p\_from\_queue = FALSE)$ **then**
28   **if** *(Evaluation of Criteria 2 was successful)* **then**
29         $update\_Ax\_params(next\_i, p, A^{tot}, A^x_{list})$
30   Append selected $candidate\_ints$ to i_queue
31   **else if** *(Evaluation of Criteria 3 was successful)* **then**
32         $update\_Ax\_params(next\_i, p, A^{tot}, A^x_{list})$
33   Append selected $candidate\_ints$ to i_queue
34   **else**
35   Apply Criteria 4
36   Append selected $candidate\_ints$ to i_queue
37   **else if** $(|candidate\_ints| = 0) \wedge (p\_from\_queue = TRUE)$ **then**
38       $assignPtoInt(p, next\_i)$
39   **return** FALSE
40   **else if** $(|candidate\_ints| = 0) \wedge (p\_from\_queue = FALSE)$ **then**
41   **if** $next\_i \neq p\_restart\_int$ **then**
42         $assignPtoInt(p, next\_i)$
43   **return** FALSE
44   **else**
45       $update\_Ax\_params(next\_i, p, A^{tot}, A^x_{list})$
46   Append selected $candidate\_ints$ to i_queue

Once we have determined the I_{top} of a $P_{i,j}$, before we can assign the $P_{i,j}$ to I_{top}, we must check if the interval I_{top} has sufficient CPU time available to run the candidate $P_{i,j}$. To perform this check, we have developed a novel schedulability test for TT systems that we designate simply as *msst*. This is implemented by function $performMSST()$ (line 6 of Algorithm 4). Due to *msst* complexity, and because it is one of the key contributions of this paper, we opt to describe it in the dedicated section IV. At this point, it suffices to say whether an attempt to assign a $P_{i,j}$ to an interval I_k was successful or not, i.e., whether it has passed the *msst* or not. The *msst* is a function that returns the result of the test (pass or fail) and a set of parameters that are used to determine whether an I_k has sufficient CPU time available in any of the cores to run a certain $P_{i,j}$ or not. Now, we designate these parameters as A^{tot} and A^x_{list}, which are stored in the I_{graph} vertices v_k associated with an I_k, and that help us track the available CPU time in each I_k. These parameters will be explained in detail in section IV. In line 7 we check the *msst* result. If it is a pass, we use function *update_Ax_params()* to update the A^{tot} and A^x_{list} parameters of I_k and to store them in v_k (line 8). We then append the p_top_int to i_queue to continue the search process (line 9). For the case where the *msst* fails when trying to assign a $P_{i,j}$ to its I_{top} (line 10), we halt the process and deem the system as unschedulable by our heuristic.

In case the condition in line 4 yields $FALSE$, it means that the $P_{i,j}$ that comes from the *p_wait_queue*, so we don't need to recompute its I_{top}, because we simply resume the downward search from the interval where the $P_{i,j}$ first search attempt was halted, when more than one option was detected. Hence, we just retrieve the interval where the search was previously halted (line 13) and append it to i_queue (line 14). We are now ready to initiate the downward search for an interval to assign $P_{i,j}$ by iterating over i_queue (lines 15 to 46). In this part of the algorithm, we take each child interval and apply the four defined Criteria, taking into consideration the number of selected children, and whether $P_{i,j}$ comes from the p_queue queue or not.

D. Phase 4 - Scheduling of Partitions

The last phase of our heuristic is the scheduling of the partitions assigned to each interval of I_{graph}, which means that we assign a start time $s_{i,j}$ to each partition $P_{i,j}$ and a core for it to execute. Algorithm 5 defines our scheduling strategy. It consists in building the schedule backward, by traversing the I_{graph} "diagonally" from right to left and from bottom to top. For example, in the graph in Figure 1b, we would schedule the $P_{i,j}$ assigned to each interval in the following order of intervals: (20,40), (10,40), (3,40), (0,40), then (10,25), (3,25), (0,25), and so on.

Since we build the schedule backward, we start Algorithm 5 by storing the reversed $\mathcal{IB}$ in *reversed_*$\mathcal{IB}$ (line 2) and by storing the end of the schedule in EOS, which corresponds to the upper bound of the last interval of $\mathcal{IB}$, given by *reversed_*$\mathcal{IB}[0][1]$ (line 3). In line 4 we initialize all m elements of the schedule tracker set $\mathcal{S}$ with the value EOS. We define $\mathcal{S}$ as a set of size m, whose purpose is to keep track of the start time $s_{i,j}$ of the last $P_{i,j}$ scheduled in the m cores. Next we iterate over the *reversed_*$\mathcal{IB}$ to schedule the $P_{i,j}$ in each interval $I \in I_{graph}$ (lines 5). But before we initiate the scheduling of the $P_{i,j}$ assigned to I, we need to check if any of the schedule tracker values in $\mathcal{S}$ is larger than the upper bound of I, given by $I[1]$. This is performed in the procedure in lines 7 to 10. First, we check if it is not the last interval in the schedule (line 6), and then we iterate over all elements of $\mathcal{S}$ (line 7) and check if the schedule tracker values $\mathcal{S}[i]$ are larger than the interval upper bound $I[1]$ (line 8). If this condition is true, we update the respective $\mathcal{S}[i]$ value with the $I[1]$ (line 9). Next, we initiate the iteration over all intervals of the I_{graph} diagonal to schedule the $P_{i,j} \in I$, starting from the base intervals $I \in reversed\_\mathcal{IB}$ (line 10) until I has no parent interval (i.e., we reach the top of I_{graph}). In each I, we perform the iteration over all the $P_{i,j} \in I$, and schedule each one of them (line 22) in the core that has the largest available CPU time, given by function $max(\mathcal{S})$ (line 27). If the computed P_{start} occurs before the release time P_r of the $P_{i,j}$ (check at line 31 fails), we deem the application set as unschedulable as per our heuristic (line 32).

Algorithm 5: Phase 4: generateSchedule

Output: All $P_{i,j}$ assigned a start time $s_{i,j}$ and a core m

1 **Function** `generateSchedule`($P_{graph}, I_{graph}, \mathcal{IB}$):
2 $\quad$ *reversed_IB* $\leftarrow$ *reverseBaseList(IB)*
3 $\quad$ $EOS \leftarrow reversed\_\mathcal{IB}[0][1]$
4 $\quad$ Let a set $\mathcal{S}, |\mathcal{S}| = m, \forall w \in \mathcal{S}, w \leftarrow EOS$
5 $\quad$ **for** $I \in reversed\_\mathcal{IB}$ **do**
6 $\quad\quad$ **if** $I \neq reversed\_\mathcal{IB}[0]$ **then**
7 $\quad\quad\quad$ **for** $i \in [0..|\mathcal{S}|-1]$ **do**
8 $\quad\quad\quad\quad$ **if** $\mathcal{S}[i] > I[1]$ **then**
9 $\quad\quad\quad\quad\quad$ $\mathcal{S}[i] \leftarrow I[1]$
10 $\quad\quad$ **while** *I has a parent interval* **do**
11 $\quad\quad\quad$ $\mathcal{J} \leftarrow I_{graph}[I]$*.getCandidates()*
12 $\quad\quad\quad$ **if** $\mathcal{J} \neq \emptyset$ **then**
13 $\quad\quad\quad\quad$ $\mathcal{B} \leftarrow \{\}$
14 $\quad\quad\quad\quad$ $\mathcal{D} \leftarrow \{\}$
15 $\quad\quad\quad\quad$ **for** $J \in \mathcal{J}$ **do**
16 $\quad\quad\quad\quad\quad$ $\mathcal{B}_{temp} \leftarrow \{P_{graph}[j]$*.getBudget()*, $J\}$
17 $\quad\quad\quad\quad\quad$ $\mathcal{D}_{temp} \leftarrow \{P_{graph}[j]$*.getDeadline()*, $J\}$
18 $\quad\quad\quad\quad\quad$ Append $\mathcal{B}_{temp}$ to $\mathcal{B}$
19 $\quad\quad\quad\quad\quad$ Append $\mathcal{D}_{temp}$ to $\mathcal{D}$
20 $\quad\quad\quad\quad$ Sort $\mathcal{B}$ in descending order of B_i
21 $\quad\quad\quad\quad$ Sort $\mathcal{D}$ in descending order of deadline
22 $\quad\quad\quad\quad$ **for** $d \in [0..|\mathcal{D}|-1]$ **do**
23 $\quad\quad\quad\quad\quad$ $P \leftarrow \mathcal{D}[d][1]$
24 $\quad\quad\quad\quad\quad$ $P_d \leftarrow \mathcal{D}[d][0]$
25 $\quad\quad\quad\quad\quad$ $P_b \leftarrow P_{graph}[P]$*.getBudget()*
26 $\quad\quad\quad\quad\quad$ $P_r \leftarrow P_{graph}[P]$*.getRelease()*
27 $\quad\quad\quad\quad\quad$ *latest_core* $\leftarrow$ *(index of the max(S))*
28 $\quad\quad\quad\quad\quad$ **if** $P_d < max(\mathcal{S})$ **then**
29 $\quad\quad\quad\quad\quad\quad$ $\mathcal{S}[latest\_core] \leftarrow P_d$
30 $\quad\quad\quad\quad\quad$ $P_{start} \leftarrow \mathcal{S}[latest\_core] - P_b$
31 $\quad\quad\quad\quad\quad$ **if** $P_{start} < P_r$ **then**
32 $\quad\quad\quad\quad\quad\quad$ EXIT - the system is not schedulable by our heuristic
33 $\quad\quad\quad\quad\quad$ $P_{graph}[P]$*.scheduleP*$(P_{start}$, *latest_core*$)$
34 $\quad\quad\quad\quad\quad$ $\mathcal{S}[latest\_core] \leftarrow P_{start}$
35 $\quad\quad\quad\quad$ $I \leftarrow I_{parent} \mid I[1] = I_{parent}[1]$
36 $\quad\quad\quad\quad$ **break**
37 $\quad\quad\quad$ **else**
38 $\quad\quad\quad\quad$ $I \leftarrow I_{parent} \mid I[1] = I_{parent}[1]$
39 $\quad\quad\quad\quad$ **break**
40 $\quad$ **return**

IV. Proposed Multi-core Schedulability Test

A. MSST Formalization

Our heuristic consists of traversing I_{graph} in the search for a suitable I_k to assign a $P_{i,j}$ with B_i. For each $I_k \in I_{graph}$ traversed during the search process, we run the *msst* to check if I_k has sufficient CPU time available in the m cores to execute the $P_{i,j}$. Instead of deciding upfront the allocation of the tasks to $P_{i,j}$ to the m cores, and tracking the available processing time in each core, we propose an efficient approach by reducing this two-dimensional problem to a single-dimensional one. We explore the temporal properties of the system as a whole, instead of tracking the available CPU time in each core individually. Through this approach, we compress the information in such a way that it summarizes a lot of possible solutions, i.e., each time we assign a $P_{i,j}$ to an I_k, we may have one or more possible cores to execute it.

Whenever we assign a $P_{i,j}$ to an interval I_k, we consider that I_k changes its state from s to $s+1$, with $s \in \mathbb{Z}^+$. Each state s of interval I_k is characterized by a set of temporal parameters (attributes) stored in each v_k. Whenever a transition occurs from state s to $s+1$, we use these attributes to decide whether an interval I_k has sufficient CPU time available to run a $P_{i,j}$ or not. Next, we define those attributes. Given an interval $I_k = (t_b, t_e)$ and m cores, the key parameters A_s^x, $\forall x$, $x \in [1, m]$, $m \in \mathbb{Z}^+$, are defined as the maximum CPU time units that can be simultaneously available in x cores in the interval $[t_b, t_b + A_s^x]$ in a defined state s of I_k. The intuition behind A_s^x, for $x = 3$, is depicted in Figure 2 for an $I_k = (0, 100)$ and $m = 6$. The blue color in each core means available CPU time. $A_s^3 = 60$ means a maximum of 60 time units are available simultaneously in 3 cores. A_s^x is initialized as follows: $\forall I_k, \forall x, A_s^x = (t_e - t_b)$ time units. The next key parameter, A_s^{tot}, is defined as the total available CPU time available in all m cores within I_k at a state s. In Figure 2, A_s^{tot} corresponds to the sum of the total CPU time available in blue color in all cores. A_s^{tot} is initialized as $\forall I_k, A_s^{tot} = m \times (t_e - t_b)$ time units.

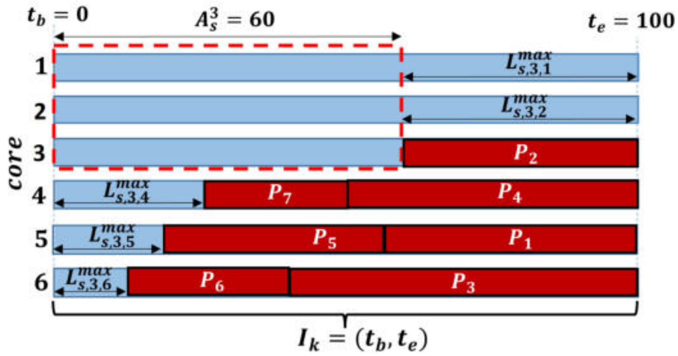


Fig. 2: Intuition behind key *msst* parameters

We now define the parameters $L_{s,x,y}^{max}$, $L_{s,x,z}^{max}$. Given an I_k at state s, $L_{s,x,y}^{max}$ is defined in Equation 8 as the maximum available CPU time available in core y, $\forall y < x$, with $x, y \in [1, m]$, in addition to A_s^x. $L_{s,x,y}^{max}$ for $x = 3$ can be visualized in Figure 2, for cores $y = 1$ and $y = 2$. We also define $L_{s,x,y}$, which is an approximation towards $L_{s,x,y}^{max}$, where $L_{s,x,y} \leq L_{s,x,y}^{max}$, and that is computed by means of a heuristic described later in this section.

$$\forall x \in [1, m], \forall y < x, \; L_{s,x,y}^{max} = A_s^y - A_s^x \tag{8}$$

Similarly, given an I_k at state s, $L_{s,x,z}^{max}$ is defined as a constraint given by Equation 9.

$$\forall x \in [1, m], \forall z > x, \; L_{s,x,z}^{max} < A_s^x \tag{9}$$

We also define $L_{s,x,z}$, which is an approximation towards $L_{s,x,z}^{max}$, where $L_{s,x,z} \leq L_{s,x,z}^{max}$, and that is also computed by means of a heuristic described later. Hereinafter, we use the term "L parameters" to generically refer to $L_{s,x,y}$ and $L_{s,x,z}$.

Next, we define $A_{s,x}^{other}$, $\forall x \in [1, m]$, which is the total available CPU time in x cores in addition to A_s^x, at state s of interval I_k, and that is computed according to Equation 10. The intuition behind $A_{s,x}^{other}$, for $x = 3$, can be visualized in Figure 2. In Equation 10, the term $x \times A_s^x$ yields the total maximum CPU time available in all x cores within I_k, which corresponds to the sum of the CPU time available in blue color inside the red dashed lines. If we subtract this value from the total available CPU time in all cores, given by $A_{s,x}^{tot}$, we obtain the value of $A_{s,x}^{other}$ for a given x.

$$\forall x \in [1, n], \; A_{s,x}^{other} = A_s^{tot} - x \times A_s^x \tag{10}$$

We also define Equation 11 to aid in our computations. By looking at Figure 2, we can intuitively see that this equation holds true.

$$\forall x \in [1, m], \; \sum_{\forall y < x} L_{s,x,y}^{max} + \sum_{\forall z > x} L_{s,x,z}^{max} = A_{s,x}^{other} \tag{11}$$

As previously referred, since $L_{s,x,y} \leq L_{s,x,y}^{max}$ and $L_{s,x,z} \leq L_{s,x,z}^{max}$, Equations 12 or 13 also hold true.

$$\sum_{\forall y<x} L_{s,x,y} + \sum_{\forall z>x} L_{s,x,z} = A_{s,x}^{other} \tag{12}$$

$$\sum_{\forall y<x} L_{s,x,y} + \sum_{\forall z>x} L_{s,x,z} < A_{s,x}^{other} \tag{13}$$

For the previous definitions to hold, the constraint in Equation 14 must be respected. In Figure 2 it can be easily visualized that if this constraint is violated, the definition of A_s^x becomes invalid.

$$\forall x \in [1, m],\, 0 \leq A_s^{x+1} \leq A_s^x \tag{14}$$

Algorithm 6: Multicore Schedulability Test Algorithm

Output: $TRUE \vee FALSE,\ A_{s+1}^{tot},\ A_{list}^{x}$
Data: $A_s^{tot},\ m,\ A_s^x, x \in [1, m]$

1 **Function** `performMSST(`B_i, I_k`):`
2 *partition_schedulable* $\leftarrow FALSE$
3 **if** $A_s^{tot} > B_i$ **then**
4 $x \leftarrow 1$
5 **if** $A_s^x \geq B_i$ **then**
6 *partition_schedulable* $\leftarrow TRUE$
7 **while** $x \leq m$ **do**
8 Compute $A_{s,x}^{other}$ (Eq.10)
9 **if** $A_{s,x}^{other} > 0$ **then**
10 $y \leftarrow 1$
11 **while** $y < x$ **do**
12 Compute $L_{x,y}^{max}$ (Eq.8)
13 $y \leftarrow y + 1$
14 $z \leftarrow x + 1$
15 **while** $z \leq n$ **do**
16 Compute $L_{x,z}$ (Eq.16)
17 $z \leftarrow z + 1$
18 $y \leftarrow x - 1$
19 $A_{s_{tmp}}^{other} \leftarrow A_{s,x}^{other}$
20 **while** $y \geq 1$ **do**
21 Compute $L_{x,y}$ (Eq.17)
22 $A_{s_{tmp}}^{other} \leftarrow A_{s_{tmp}}^{other} - L_{x,y}$
23 $y \leftarrow y - 1$
24 **if** *((Eq. 12* $= FALSE$*)* $\wedge$ *(Eq. 13* $=$ *FALSE))* $\vee$ *((Eq. 12* $= TRUE$*)* $\wedge$ *(*$x = 1$*))* **then**
25 Compute A_{s+1}^x (Eq.15)
26 **break**
27 **else if** *(Eq. 12* $= TRUE) \wedge (x \neq 1) \wedge (L_{s,x,1} < B_i)$ **then**
28 Compute A_{s+1}^x (Eq.19)
29 **else**
30 Compute A_{s+1}^x (Eq.18)
31 **else**
32 Compute A_{s+1}^x (Eq.15)
33 **break** `// exit the while loop`
34 $x \leftarrow x + 1$
35 $A_{list}^x \leftarrow$ *SortDescending* $(A_{s+1}^x, \forall x)$
36 $A_{s+1}^{tot} \leftarrow A_s^{tot} - B_i$
37 **return** *partition_schedulable*, A_{s+1}^{tot}, A_{list}^x

B. MSST Heuristic Intuition

Whenever we try to allocate a $P_{i,j}$ to an I_k, our strategy consists in preserving as much as possible the CPU time reserved in the A_s^x parameters, giving priority to the cores with lower values of x, by consuming first the CPU time available in the L parameters, according to a defined strategy. By maximizing A_s^x towards state $s+1$, we increase the likelihood of being able to allocate the next candidate $P_{i,j}$ to I_k. We can visualize the effect of this strategy in Figure 2, where core 1 will always have the largest available amount of CPU time and core 6 will always have the lowest. We show in this figure an example of a possible allocation of the $P_{i,j}$ among the 6 cores in red color. But it is important to emphasize that when executing the *msst*, we are not concerned yet about where the $P_{i,j}$ will be executed. Through this approach, we compress the information in such a way that it summarizes a lot of possible solutions, i.e., each time we assign a $P_{i,j}$ to an I_k, we may have one or more possible cores to execute it. Therefore, by keeping track of the values of the A_s^x and A_s^{tot} parameters, and by updating them accordingly every time we allocate a $P_{i,j}$ to an I_k, it is possible to establish an effective *msst*.

C. MSST Heuristic Implementation

Our solution is implemented by the function $performMSST$ in Algorithm 6, which receives as input the execution time B_i of a candidate $P_{i,j}$ and the target interval I_k to be tested. The data parameters used by the algorithm are A_s^{tot}, the set of the A_s^x parameters, and the number of cores m. This function returns a boolean variable indicating whether it was possible to assign the candidate $P_{i,j}$ to I_k or not, as well as the updated values of A_s^{tot} and $A_s^x, \forall x$, for the next state s of I_k. The values of $A_s^x, \forall x$, are stored in the set A_{list}^x. Next we explain the computation steps performed by function $performMSST$.

Step 1. Check that $A_s^{tot} \geq B_i$ (line 3) and then check that $A_{s_{I,i}}^1 \geq B_{s_{I,i}}^{job}$ (lines 4 and 5). If both conditions are satisfied, that means we have at least one core with sufficient available CPU time to execute the $P_{i,j}$, so we can set the boolean variable *partition_schedulable* $\leftarrow TRUE$ in line 6.

Step 2. Compute $A_{s,x}^{other}$ and the L parameters by iterating over all values of x (line 7). We further divide Step 2 into four sub-steps.

Step 2.1. Compute $A_{s,x}^{other}, \forall x$, and check if $A_{s,x}^{other} > 0$ (line 8 and 9). If $A_{s,x}^{other} = 0$, it means no CPU time is available in addition to A_s^x for core x. In this case, it is not worth continuing with the computation of the L parameters for core x, because they are equal to zero. So here we have no choice but to consume the CPU time available in the A_s^x parameter, which is updated according to Equation 15 (line 32).

$$A_{s+1}^x \leftarrow A_s^x - B_i \tag{15}$$

By updating the value of A_s^x, we know that at least one core with sufficient processing time to execute the $P_{i,j}$ (remember Step 1) exists. Hence, we stop looping over the values of x (line 33) and proceed to Step 6 described later. For the cases when $A_{s,x}^{other} > 0$ (line 9), it implies that we can be able to use the available CPU time in the L parameters, thus maximizing A_s^x for the next state $s+1$. To achieve this, we proceed to **Step 2.2**, where we compute the parameter $L_{s,x,y}^{max}$ $,\forall y < x$, according to Equation 8 (lines 10 to 13). In **Step 2.3**, we compute the value of $L_{s,x,z}, \forall z > x$ (lines 14 to 17) according to Equation 16. Since we do not know how the value of $A_{s,x}^{other}$ is distributed among the L parameters, we make an assumption that $L_{s,x,z}, \forall z > x$, is just below B_i, i.e, $B_i - 1$. Through this assumption, we prioritize the consumption of the CPU time available in the cores with smaller values of x, because their $L_{s,x,y}$ will always contain the largest CPU time reserves. This increases the possibility of scheduling the $P_{i,j}$ with larger B_i values. Later in the heuristic, we evaluate if this assumption was accurate or not. We divide Equation 16 in two parts, to improve the accuracy of the computation, by subtracting in the second part the sum of the previously computed values of $L_{s,x,z}$ from $A_{s,x}^{other}$.

$$\forall x, \forall z > x, L_{s,x,z} = \begin{cases} \min(B_i - 1, A_{s,x}^{\text{other}}), \text{if } z = x+1 \\ \min(B_i - 1, A_{s,x}^{\text{other}} - \sum_{k=x+1}^{z-1} L_{s,x,k}), \text{if } z > x+1 \end{cases} \tag{16}$$

By knowing the values of $A^{other}_{s,x}$ and $L_{s,x,z}$, $\forall z > x$, we proceed to **Step 2.4,** where we compute $L_{s,x,y}$, $\forall y < x$ (lines 18 to 23). The computation method to try to maximize $L_{s,x,y}$ is defined by Equation 17 (line 21). Here our assumption is that in a worst case, the values of $L_{s,x,y}$, $\forall y < x$, are equally distributed among the y cores. After the computation of $L_{s,x,y}$, for each y, we update the value of $A^{other}_{s,x}$ (line 22) by subtracting the computed value of $L_{s,x,y}$ in each iteration to improve the accuracy.

$$\forall x, \forall y, L_{s,x,y} = \max\Big(\min\Big(\frac{A^{\text{other}}_{s,x} - \sum_{\forall z>x} L_{s,x,z}}{y}, L^{\max}_{s,x,y}\Big), 0\Big) \tag{17}$$

Step 3. Having computed the value of $A^{other}_{s,x}$ and of the L parameters, next we decide how the A^x_s parameters should be updated, so that their value is maximized for $s+1$. The three methods to update the value of the A^x_s parameters are defined by Equations 15, 18 and 19, and are implemented in lines 24 to 30.

$$A^x_{s+1} \leftarrow A^x_s \tag{18}$$

$$A^x_{s+1} \leftarrow \max\left(A^x_s - (B_i - \ L_{s,x,1});\ 0\right) \tag{19}$$

The first part of the condition in line 24 checks if computation of the L parameters was optimistic, i.e., if Equations 12 and 13 are violated. In that case, we don't have a safe bound, so we take a conservative approach and update A^x_{s+1} according to Equation 15. The second part of the condition in line 24 checks the special case for $x = 1$, because $L_{s,x,1} = 0$, which implies that $\sum_{\forall z>x} L_{s,x,z} = A^{other}_x$, thus violating our assumption in Eq. 16 that $L_{s,x,z}$ is always smaller than B_i, $\forall z > x$. Hence we have no choice but to update A^1_{s+1} according to Equation 15. If the conditions in line 24 do not hold, we test in line 27 Equation 12 for $x > 1$ and $L_{s,x,1} < B_i$. If this happens, it means that we have CPU time available in the $L_{s,x,1}$ parameter, which will always have the largest CPU time reserve among all $L_{s,x,y}$, but it is not sufficient to completely execute the $P_{i,j}$. So we update A^x_{s+1} according to Equation 19, where we first consume all the CPU time available in the $L_{s,x,1}$ parameter, and the remaining we take it from A^x_s, but always ensuring that the term $A^x_s - (B_i - L_{s,x,1})$ does not lead to a negative value of A^x_s. Finally, if no in lines 24 and 27 hold true, it means that we have sufficient CPU time available in at least one of the $L_{s,x,y}$ parameters, hence we can safely keep the values of A^x_s for the next state $s+1$, according to Equation 18.

Step 5. After computing the A^x_{s+1} parameters, $\forall x$ at state s, depending on the value of B_i, nothing prevents the case where $A^p_{s+1} - B_i < A^q_{s+1}$, $\forall p, q \in [1, m]$, with $p < q$. If this case would happen, this would imply that $A^p_{s+1} < A^q_{s+1}$, which would be a violation of the constraint defined by Equation 14. Therefore, to prevent this situation from happening, we define a reordering function that reorders the computed values of A^x_{s+1}, $\forall x$, in descending order, named *SortDescending()* (line 35). This function takes as input all computed values of A^x_{s+1} and outputs the reordered values to be used for state $s+1$, which are stored in the set A^x_{list}.

Step 6. In this final step we update the value of A^{tot}_s for the next state, according to Equation 20 (line 36). Once this final update is performed, we return variable *partition_schedulable* equal to $TRUE$ or $FALSE$ (line 37), and the computed A^x_{list}.

$$A^{tot}_{s+1} \leftarrow A^{tot}_s - B_i \tag{20}$$

V. Experimental Results

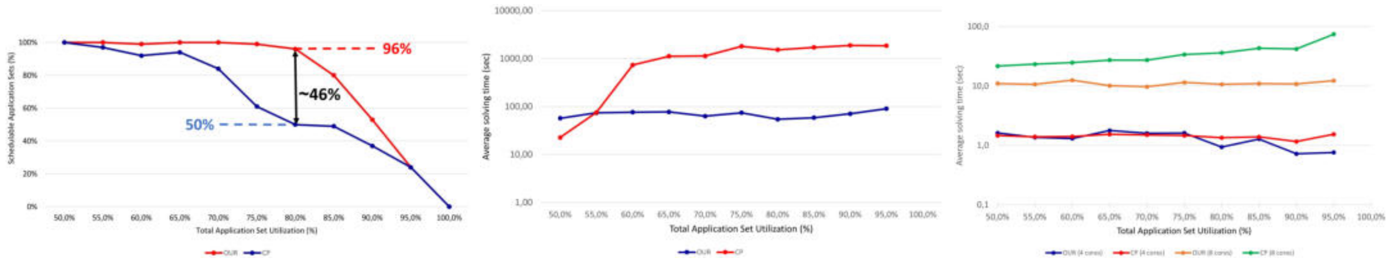
In this section, we discuss the experimental results to evaluate the effectiveness of the proposed work. As explained in [22], due to the NP-hardness of the general periodic scheduling problems, it is a common approach to compare the performance of heuristic solutions against formal approaches that obtain optimal solutions (e.g., ILP, SMT, or CP). This approach allows us to estimate the quality of the heuristic solution compared to the optimal solution. To the best of our knowledge, we are the first ones to compare with [24].

Experimental Setup: Our framework is implemented in a simulation environment that runs on Ubuntu 16.04 running on an Intel® Core™ i7-6700K CPU 4.2 GHz with 64GB RAM. For the default configuration, we generated synthetic data sets with the following parameters: $m = 16$, 60 IMA applications per application set α, with non-harmonic periods T_i randomly chosen from [10; 20; 30; 50; 60; 90; 100] * 1000. The applications' utilizations (u_i) are randomly chosen between 10% and 50% according to a uniform distribution using the randfixedsum [27] algorithm. The B_i of each α_i was computed as $B_i = T_i \times u_i$. A random offset O_i was assigned to each application in relation to the start of T_i such that $0 \leq O_i \leq T_i$. For each run, the same input data sets were provided to the proposed heuristic framework and the CP approach in [24], using CPLEX optimization studio. We also defined a threshold timeout t_{out} of 4 hours, i.e., the maximum time allowed for both approaches to find a solution. This is a common practice [24] as the solving time for the CP approach can drastically increase with the search space, so, the threshold limits the maximum time to find the solution.

We compare the proposed approach against the existing CP approach [24] by evaluating the schedulability ratio, i.e., the percentage of application sets deemed schedulable, average solving time, i.e., the average time required to find a valid schedule, and varying the number of cores. In all the experiments, our approach is marked as "**OUR**" and the existing approach of [24] is marked as "**CP**". In all the experiments, the x-axis represents the total application set utilization u_{tot}.

1. Schedulability Ratio: In this experiment, we vary the total application set utilization u_{tot} in the range [50%, 100%] with a step size of 5% and evaluated the schedulability ratio using the proposed framework and existing CP-based approach [24] as plotted in Figure 3a. We can see in Figure 3a that the schedulability ratio using both approaches reduces with the increase in u_{tot}. This happens because the increase in u_{tot} results in an increase of the u_i of each α_i, which increases B_i as $B_i = T_i \times u_i$. Consequently, there is an increase in the system workload, which degrades the schedulability ratio. However, we can see in Figure 3a that the proposed approach was able to schedule up to 46% more applications compared to the CP approach [24]. This gain is mainly observed because, for most of the runs, the CP solution could not find a solution within $t_{out} = 4$ hours for the default configuration, i.e., $m = 16$ with 60 IMA applications, whereas the solving time for the proposed approach was mostly within the t_{out} limit. We observed that the gain of the proposed approach over the CP approach increases significantly for a higher number of cores, e.g, $m = 32$, but we have not reported it in the paper due to space constraints.

2. Average Solving Time: In this experiment, we evaluate the average solving time in relation to the u_{tot} considering default configuration. For this, we varied u_{tot} in the range [50%, 100%] with a step size of 5% and plotted the resulting average solving time using the proposed approach and CP approach in Figure 3b. For this experiment, to plot each point, we only consider the cases in which both approaches were able to find a valid schedule, e.g., if out of 100 runs, an approach can find a valid for only 30 runs, we take the average solving time for those 30 runs. This is the reason that the average solving time for both approaches does not significantly increase with the increase in the u_{tot} value as shown in Figure 3b, since the number of successful runs diminishes as the workload



(a) Schedulability Ratio (b) Average Solving Time for $m = 16$ (c) Average Solving Time for $m = 4, 8$

Fig. 3: Experimental Results

increases. Nonetheless, it presents a big picture of the average time consumed by both approaches to find a valid schedule. We can see in Figure 3b that for the successful cases, the proposed approach was generally able to find a solution 10x faster than the existing CP solution.

3. Number of Cores and IMA applications: In this experiment, we redo experiment 2 by varying the value of m and IMA partitions as $m = 4$ with 20 IMAs, $t_{out} = 10$ minutes; and $m = 8$ with 40 IMAs, $t_{out} = 10$ minutes. The average solving time for successful runs using both approaches is plotted in Figure 3c. We can observe in Figure 3c that the difference between the proposed approach and the CP approach is marginal because the existing CP approach can efficiently find a solution for a smaller search space.

VI. Related Work

The multiprocessor scheduling can be broadly categorized into event-driven scheduling, i.e., scheduling decisions are made at run-time based on different parameters, and TT scheduling, i.e., a system-level schedule is constructed at design time which is then enforced at run time, (see surveys [21], [22]). The TT scheduling is proven to be more predictable, as the schedule is constructed at design time so the precise information of each event that will take place at run time is known at the design time. As a consequence, TT scheduling is preferred for designing safety-critical systems (e.g. avionics control systems) due to higher predictability. It also simplifies the process of design, verification, and (re-)certification. Furthermore, designing such systems requires strict *space and time partitioning* among applications of different criticalities executing on the same platform, as mandated by the ARINC-653 standard. Such partitioning ensures sufficient isolation between applications of different criticalities (possibly running on different cores), in such a way that they can be modified/upgraded independently, thus minimizing the system re-certification costs. Considering this, a plethora of works in the literature [6]–[8], [14], [17]–[19], [22], [24], [28], [31] focus on building solutions to generate TT schedule for tasks/IMA applications on multicore platform.

Xu et al. [31] presented a scheduling algorithm based on a branch and bound heuristic to find a feasible non-preemptive schedule on M identical processors. However, in contrast to the proposed work, the work in [31] does not comply with the specifications of ARINC-653, which requires a static allocation of IMA partitions to cores. Deroche et al. [7] proposed an exhaustive branch-and-bound heuristic based approach to build a TT schedule for IMA applications running on multicore platform. Even though their solution is important, it suffers from the problem of scalability as their approach does not scale well for systems with a large number of avionics functions distributed in a limited number of processors. To overcome this challenge, in their subsequent work, Deroche et al. [8] propose an improvement by eliminating the backtracking during the decision tree search process, using a greedy heuristic. To achieve this goal they choose the most promising valid MAF set in each node of the search tree, based on a metric that takes into consideration the margin of a communication chain, which is the difference between the chain end-to-end delay constraint and the current delay of each chain. The authors performed a comparison with the exhaustive (optimal) approach from [7] and showed improvement in terms of solving time. Although their approach is efficient, it is limited to IMA partitions with synchronous harmonic periods, which is a much less complex problem than the one we are trying to solve (i.e., non-harmonic asynchronous case). Furthermore, contrary to our approach, the solution from [8] does not consider the migration of tasks among cores.

Other existing approaches use Constraint Programming (CP) or Integer Linear Programming (ILP) to build a TT schedule of IMA partitions [4], [10], [23]–[26]. Among all these approaches, the solution in [24] is the closest to the proposed work in terms of contribution and assumptions. Puffitsch et al. [24] presents a CP approach for the generation of TT schedule of real-time dependent periodic non-preemptive asynchronous task sets on multi/many-core platforms for IMA systems. Their solution considers a) precedence constraints between partitions of different IMA applications; b) spatial mapping of IMA application to cores; and c), mapping of communication buffers in the message passing area. Even though the existing CP-based approach [24] can efficiently find a solution for a relatively smaller problem, it does not scale well with the increase in the search space. If a valid TT schedule is not found by an approach within a reasonable time, it will directly impact the schedulability ratio as the taskset will be deemed unschedulable if the schedule cannot be found within a reasonable time. As reported in Section V (see Figure 3a), even with 4 hours of threshold limit, the proposed approach outperformed approach [24] by improving the schedulability ratio up to 46%. Furthermore, the work in [24] uses commercial constraint solver [3] which is limited to internal undisclosed search algorithms. Our tool on the other hand offers many possibilities for future improvements, such as the adoption of different strategies to traverse P_{graph}, I_{graph}, and scheduling strategies.

VII. Conclusion and Future Work

In this paper, we proposed a novel heuristic framework for configuring and generating IMA-compliant schedules which is efficient in terms of finding a valid schedule, scalable to a large number of IMA applications, and efficiently utilizes the computing platform. The experimental results reveal that our solution can outperform the state-of-the-art CP solution by [24] in terms of solving time, memory usage and does not perform significantly poorly compared to the CP-based optimal solution. In the future, we plan to extend our framework to a) include precedence relations between partitions; and b) consider synchronization of access to system I/O resources by mapping the I/O partitions to a dedicated I/O core.

References

[1] *Avionics Application Software Standard Interface (Part 1): Required Services, ARINC Specification ARINC653P1-4, 2015*. SAE International, 2015.

[2] *Avionics Application Software Standard Interface (Part 0): Overview of ARINC 653, ARINC Specification ARINC653P0-3*. SAE International, 2021.

[3] Ibm ilog cplex optimization studio. https://www.ibm.com/products/ilog-cplex-optimization-studio?mhsrc=ibmsearch_a&mhq=ilog, 2023. Accessed: 2023-11-24.

[4] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24, 2016.

[5] Alan Burns and Robert Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, pages 1–69, 2013.

[6] Jinchao Chen, Chenglie Du, Fei Xie, and Zhenkun Yang. Schedulability analysis of non-preemptive strictly periodic tasks in multi-core real-time systems. *Real-Time Syst.*, 52(3):239–271, may 2016.

[7] Emilie Deroche, Jean-Luc Scharbarg, and Christian Fraboul. Mapping real-time communicating tasks on a distributed ima architecture. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2016.

[8] Emilie Deroche, Jean-Luc Scharbarg, and Christian Fraboul. A greedy heuristic for distributing hard real-time applications on an ima architecture. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–8. IEEE, 2017.

[9] Xiaoyan Du, Chenglie Du, Jinchao Chen, Mei Yang, and Wenquan Yu. A simulation and verification platform for avionics systems based on future airborne capability environment architecture. *Applied Sciences*, 12(22):11533, 2022.

[10] Friedrich Eisenbrand, Karthikeyan Kesavan, Raju S. Mattikalli, Martin Niemeier, Arnold W. Nordsieck, Martin Skutella, José Verschae, and Andreas Wiese. Solving an avionics real-time scheduling problem by advanced ip-methods. In Mark de Berg and Ulrich Meyer, editors, *Algorithms – ESA 2010*, pages 11–22, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[11] Alexandre Esper et al. An industrial view on the common academic understanding of mixed-criticality systems. *Real-Time Systems*, 54, 2018.

[12] D. Aubrey et al. The future of avionics: High performance, machine-learned and certified. White paper, October 2023. Available at: https://www.intel.com/content/www/us/en/content-details/791291/the-future-of-avionics-high-performance-machine-learned-and-certified.html.

[13] Thomas Gaska, Chris Watkins, and Yu Chen. Integrated modular avionics past, present, and future (vol 30, pg 12, 2015). *IEEE Aerospace and Electronic Systems Magazine*, 30(11):11–11, 2015.

[14] Jia Huang, Jan Olaf Blech, Andreas Raabe, Christian Buckl, and Alois Knoll. Static scheduling of a time-triggered network-on-chip based on smt solving. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 509–514, 2012.

[15] Kevin Jeffay et al. On non-preemptive scheduling of periodic and sporadic tasks. In *IEEE real-time systems symposium*, pages 129–139, 1991.

[16] Hyun-Chul Jo, Joo-Kwang Park, Hyun-Wook Jin, Hyung-Sik Yoon, and Sang Hun Lee. Portable and configurable implementation of arinc-653 temporal partitioning for small civilian uavs. *IEEE Access*, 7:142478–142487, 2019.

[17] Omar Kermia. An efficient approach for the multiprocessor non-preemptive strictly periodic task scheduling problem. *Journal of Systems Architecture*, 79:31–44, 2017.

[18] Jung-Eun Kim et al. Integrated modular avionics (ima) partition scheduling with conflict-free i/o for multicore avionics systems. In *38th Annual Computer Software and Applications Conference*, pages 321–331, 2014.

[19] H. Kopetz. Time-triggered real-time computing. *Annual Reviews in Control*, 27(1):3–13, 2003.

[20] Georgios Kornaros. *Multi-Core Embedded Systems*. CRC Press, 2018.

[21] C. Maiza et al. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Computing Surveys*, 52(3):1–38, June 2019.

[22] Anna Minaeva and Zdeněk Hanzálek. Survey on periodic scheduling for time-triggered hard real-time systems. *ACM CSUR*, 54(1):1–32, 2021.

[23] Clément Pira and Christian Artigues. Line search method for solving a non-preemptive strictly periodic scheduling problem. *Journal of Scheduling*, 19, 07 2014.

[24] Wolfgang Puffitsch et al. Off-line mapping of multi-rate dependent task sets to many-core platforms. *Real-Time Systems*, 51:526–565, 2015.

[25] Eike Schweissguth, Peter Danielis, Dirk Timmermann, Helge Parzyjegla, and Gero Mühl. Ilp-based joint routing and scheduling for time-triggered networks. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, RTNS '17, page 8–17, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Ahmad Sheikh, O. Brun, Pierre-Emmanuel Hladik, and Balakrishna Prabhu. Strictly periodic scheduling in ima-based architectures. *Real-Time Systems*, 48, 07 2012.

[27] Roger Stafford. Random vectors with fixed sum. https://www.mathworks.com/matlabcentral/fileexchange/9700-random-vectors-with-fixed-sum, 2023. Accessed: 2023-11-19.

[28] Yasin Unlu and Scott J Mason. Evaluation of mixed integer programming formulations for non-preemptive parallel machine scheduling problems. *Computers & Industrial Engineering*, 58(4):785–800, 2010.

[29] Steven H VanderLeest and Samuel R Thompson. Measuring the impact of interference channels on multicore avionics. In *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, pages 1–8. IEEE, 2020.

[30] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE international real-time systems symposium (RTSS 2007)*, pages 239–243. IEEE, 2007.

[31] Jia Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on software engineering*, 19(2):139–154, 1993.