Technical Report

A Multi-DAG Model for Real-Time Parallel Applications with Conditional Execution

José Fonseca*
Vincent Nélis*
Gurulingesh Raravi
Luis Miguel Pinho*

CISTER-TR-141207

2015/04/13

A Multi-DAG Model for Real-Time Parallel Applications with Conditional Execution

José Fonseca*, Vincent Nélis*, Gurulingesh Raravi, Luis Miguel Pinho*

*CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: jcnfo@isep.ipp.pt, nelis@isep.ipp.pt, guhri@isep.ipp.pt, lmp@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Owing to the current trends for higher performance and the ever growing availability of multiprocessors in the embedded computing (EC) domain, there is nowadays a strong push towards the parallelization of modern embedded applications. Several real-time task models have recently been proposed to capture different forms of parallelism. However, they do not deal explicitly with control flow information as they assume that all the threads of a parallel task must execute every time the task is activated. In contrast, in this paper, we present a multi-DAG model where each task is characterized by a set of execution flows, each of which represents a different execution path throughout the task code and is modeled as a DAG of sub-tasks. We propose a two-step solution that computes a single synchronous DAG of servers for a task modeled by a multi-DAG and show that these servers are able to supply every execution flow of that task with the required cpu-budget so that the task can execute entirely, irrespective of the execution flow taken at run-time, while satisfying its precedence constraints. As a result, each task can be modeled by its single DAG of servers, which facilitates in leveraging the existing single-DAG schedulability analyses techniques for analyzing the schedulability of parallel tasks with multiple execution flows.

A Multi-DAG Model for Real-Time Parallel Applications with Conditional Execution

José Carlos Fonseca[†], Vincent Nélis[†], Gurulingesh Raravi[§] and Luís Miguel Pinho[†]
[†]CISTER/INESC-TEC, ISEP, Portugal [§]Xerox Research Center, India
{jcnfo, nelis, lmp}@isep.ipp.pt gurulingesh.raravi@xerox.com

ABSTRACT

Owing to the current trends for higher performance and the ever growing availability of multiprocessors in the embedded computing (EC) domain, there is nowadays a strong push towards the parallelization of modern embedded applications. Several real-time task models have recently been proposed to capture different forms of parallelism. However, they do not deal explicitly with control flow information as they assume that all the threads of a parallel task must execute every time the task is activated. In contrast, in this paper, we present a multi-DAG model where each task is characterized by a set of execution flows, each of which represents a different execution path throughout the task code and is modeled as a DAG of sub-tasks. We propose a two-step solution that computes a single synchronous DAG of servers for a task modeled by a multi-DAG and show that these servers are able to supply every execution flow of that task with the required cpu-budget so that the task can execute entirely, irrespective of the execution flow taken at run-time, while satisfying its precedence constraints. As a result, each task can be modeled by its single DAG of servers, which facilitates in leveraging the existing single-DAG schedulability analyses techniques for analyzing the schedulability of parallel tasks with multiple execution flows.

1. INTRODUCTION

Forty years ago, in 1973, Chang Liu and James Layland proposed in [17] a simplistic model to characterize the timing behavior of time-critical control and monitor functions that they termed "pure process control". Their model abstracts each of these pure process controls by two numbers: a worst-case execution requirement and an execution rate (called period). Since then the real-time (RT) community has developed an extensive set of task models, platform models and scheduling techniques by relaxing some of the assumptions originally made by Liu and Layland to incorporate the requirements of new systems, applications and architectures [10, 23]. Although researchers have built an impressive body of knowledge that gives a deep understanding of the broad variety of scheduling problems, today's applications in the EC domain are neither designed nor implemented the way it used to be back in 1973, and most of the models available today are not expressive enough to correctly model the timing behavior of most of the contemporary embedded applications.

SAC'15 April 13-17, 2015, Salamanca, Spain.
Copyright 2015 ACM 978-1-4503-3196-8/15/04...$15.00.
http://dx.doi.org/10.1145/2695664.2695808

Given that the modern embedded applications increasingly require higher performance and with the ever growing need of deploying such applications on multiprocessors, there is a strong push towards the parallelization of these applications. The parallelization is typically achieved by splitting applications into multiple smaller computation units (called sub-tasks) which may run simultaneously on different cores. In order to capture this parallelization opportunity and overcome the limitations of the *sequential* task models, several *parallel* task models have recently been proposed for real-time embedded systems, which we shall revise now.

Related Work. There has been extensive work on scheduling real-time *sequential* tasks atop multiprocessor systems (see [11] for an excellent survey). The problem of scheduling *parallel* tasks has been studied significantly by the high performance computing (HPC) community [12, 7, 6]. In the RT community, where timing constraints and worst-case scenarios are the primary concerns, this problem is currently receiving substantial attention. Researchers have started developing schedulability analyses over more general task models to cope with the task parallelism generated by prominent parallel programming models (e.g., OpenMP [20]). Parallel task models can be broadly classified *according to the restrictions imposed on the internal task structure.*

At one extreme, the fork/join model [14, 2, 13] imposes the *most severe* restrictions: a task is represented as an alternate sequence of sequential and parallel segments, always starting with a sequential segment. Thus, nested parallelism is not allowed. Usually, the number of sub-tasks spawned in each parallel segment is fixed and may not exceed the number of cores in the system. On the other end of the spectrum, the DAG model [5, 8, 15, 21, 3, 16] represents each task as a directed acyclic graph (DAG), where each node is a sequential sub-task, and each directed edge defines a precedence constraint between two nodes. In this model, a node becomes ready for execution as soon as all its predecessor nodes (if any) have completed. In between, the synchronous parallel task model [22, 1, 19, 9, 18] generalizes the fork/join model by allowing successive parallel segments and arbitrary number of sub-tasks in each segment. Still, this model imposes synchronization points at every segment's boundary, meaning that all the sub-tasks in a segment may begin execution only when all the sub-tasks of the previous segment have finished their execution.

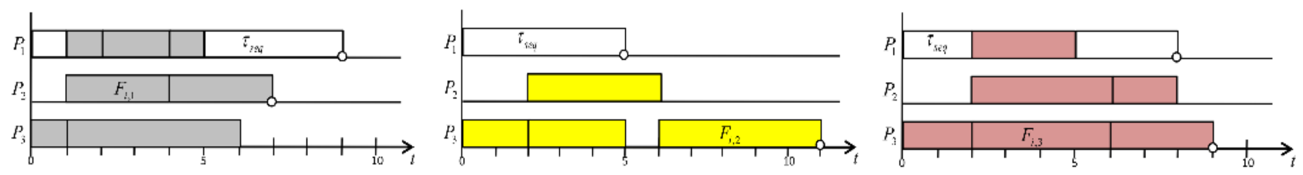

Figure 1: A hypothetical global fixed-priority schedule of the 3 different execution flows $F_{i,j}$ of the task τ_i (see Fig. 3) and a sequential task τ_{seq} with $c_{seq} = 5$ on 3 cores.

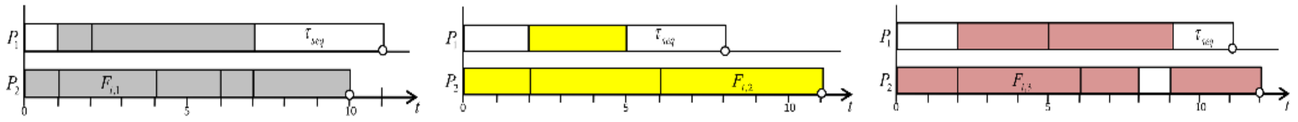

Figure 2: A hypothetical global fixed-priority schedule of the 3 different execution flows $F_{i,j}$ of the task τ_i (see Fig. 3) and a sequential task τ_{seq} with $c_{seq} = 5$ on 2 cores.

Typically, due to control structures within the tasks code, different activations of a same task may execute different parts of its code. Unfortunately, none of the aforementioned models explicitly capture these different flows of execution that a parallel task most likely will take during its recurrent activity. Instead, all these models represent a task as a single graph for which *all the nodes (the sub-tasks) must execute each time the task is released*. In other words, none of these models is able to express any control-flow information, such as conditional statements, because they assume a single non-variable internal task structure that has to be fully scheduled and executed at every task release. This paper takes a first step in addressing this limitation by proposing a multi-DAG model which facilitates the schedulability analysis of parallel tasks with multiple execution flows on multi-core systems.

Motivation. Let us consider a simple program starting with a call to a function A followed by an if-then-else statement. If the condition of the statement is satisfied, then the program executes in parallel four instances of a function B, otherwise it executes in parallel two instances of a function C. It is trivial to see that an execution of this code can only take one of the two execution paths (referred to as "execution flows" hereafter), i.e., it can either take the *if* execution flow or the *else* execution flow, but never both. If each of the instances of these functions is modeled as a sub-task, then the mutual exclusion between sub-tasks of B and C (resulting from the conditional statement) cannot be expressed in the aforementioned models; either they model this task as a single graph of seven sub-tasks (1 sub-task for function A, 4 for B and 2 for C), with no consideration whatsoever for the mutual exclusions between B and C, or they model only the "worst-case" execution flow of that task, which we shall discuss now.

On a *single-core* platform, the fact that a task can execute different execution flows during different runs is not a major issue for performing schedulability analysis. To analyze the schedulability of a task set on a *single-core* it is known that for each task, only the flow with the highest workload (defined in Section 2) must be considered in the analysis because such flow (i) executes for the longest time and thus (ii) causes the highest interference on the other tasks. Consequently, the number of scheduling scenarios to be considered stays within reasonable bounds. In contrast, in a *multi-core* settings the interference between two or more graphs of sub-tasks is much more difficult to capture and analyze. This is because the interference not only depends on the execution time of all the sub-tasks in each graph, but also on the inherent structure of the graphs themselves. As it can be seen in Fig. 1 and Fig. 2, when analyzing a possible schedule of a high priority multi-DAG task (τ_i as in Fig. 3) and a low priority sequential task (τ_{seq} with WCET of 5 time units), their worst-case response times are found on different executions flows depending on the circumstances[1] (in this case we first assume that 3 cores are available in Fig. 1 and then only 2 cores in Fig. 2). Therefore, for a given scheduling algorithm, platform and set of parallel tasks, where each task may take different execution flows at run-time, an exact schedulability test would have to consider every feasible interference scenario between all combinations of execution flows from each task. Clearly, this would lead to a *combinatorial explosion* in the number of scenarios to be considered, which is *prohibitively expensive* in terms of computational time.

To the best of our knowledge, the RT research community so far has only managed to address this scheduling problem to a limited extent. That is, although current works do not explicitly deal with conditional execution of sub-tasks, some works have derived results that may still hold under such circumstances due to the fact that the internal structure of the graphs is completely ignored. For example, the authors of [15] have derived a capacity augmentation bound based on the workload and the critical path length of a task modeled as a single DAG of sub-tasks. In the case of a task with multiple execution flows, where each flow is modeled as a separate DAG, identical results can be derived considering the maximum workload and critical path length among all the task's execution flows; the pessimism in the analysis itself reduces the number of scenarios to be considered.

This research. In this work, we ask the question: is it possible to represent a general parallel task with multiple execution flows as a single DAG to leverage the existing DAG-based scheduling techniques for analyzing the schedulability of such tasks? We answer the question affirmatively by making the following contributions. This paper (c1) proposes a multi-DAG model in which each real-time parallel task is characterized by a collection of execution flows, each of which is explicitly modeled as a separate DAG; (c2) presents a two-step algorithm to construct a single synchronous DAG

[1]For simplicity, here we assume that tasks have harmonic periods and synchronous releases.

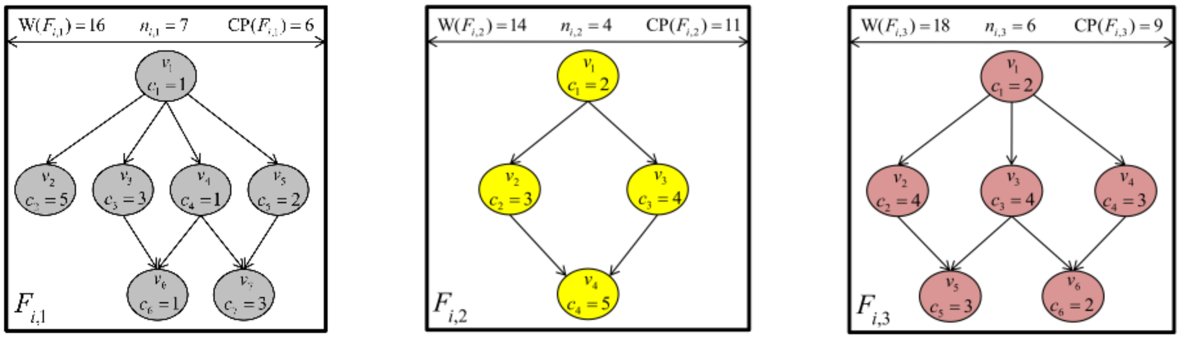

Figure 3: Example of a task τ_i with $\mathcal{F}_i = \{F_{i,1}, F_{i,2}, F_{i,3}\}, T_i = 30, D_i = 20$. Note that in each execution flow, node v_k denotes the k'th node in the flow and thus v_1 (for instance) of these three flows may or may not refer to the same sub-task in τ_i.

of servers[2] for each task; (c3) defines a run-time mapping rule to assign sub-tasks to servers for execution; and (c4) proves that (c2) together with (c3) ensure that the DAG of servers derived for each task always provides the required cpu-budget to its sub-tasks and preserves their precedence constraints, irrespective of which execution flow is taken at run-time.

We believe that the significance of this research is as follows: (s1) this is the first work to model real-time parallel tasks with multiple execution flows on multi-core platforms; (s2) our approach reduces the number of interference scenarios to be considered while deriving schedulability tests for the tasks under consideration on multi-cores by orders of magnitude; and (s3) with this approach, any existing scheduling techniques for the DAG model (synchronous or not) can be applied to analyze such multi-DAG tasks.

2. MODEL AND DEFINITIONS

We assume a multi-core platform π comprising m identical cores, and where each real-time parallel application is modeled by a sporadic task τ_i , $i \in [1, n]$, which is characterized by a 3-tuple $(\mathcal{F}_i, T_i, D_i)$ with the following interpretation. Task τ_i is a recurrent process that releases a (potentially) infinite sequence of "jobs", with the first job released at any time during the system execution and subsequent jobs released at least T_i time units apart. Each job released by a task τ_i has to complete its execution within D_i time units from its release — we assume $D_i \leq T_i$ (constrained-deadline task model). Every job of a task τ_i executes the code of τ_i from its (unique) entry point to its exit point(s). Due to the control structures within τ_i's code (e.g., the "if-then-else" statements), two different jobs of τ_i may execute two different parts of the code, and we call an *"execution flow"* the path taken by a job throughout the task's code during its execution. Every execution flow of a task τ_i is modeled by a DAG of computing units that may execute simultaneously: we refer to such units as *sub-tasks*. Each such DAG is denoted by $F_{i,j}$ and the task parameter $\mathcal{F}_i$ denotes the set of all execution flows of τ_i, i.e. $\mathcal{F}_i = \{F_{i,1}, F_{i,2}, \ldots, F_{i,n_i}\}$ where n_i is the number of execution flows of τ_i.

[2]The term "server" is employed here with the same meaning as in [4], for instance. Servers are the entities to be scheduled on the cores. Each server has a pre-defined cpu-budget to be "consumed" through the execution of ready tasks, every time a server is granted a core. A task cannot execute within a server if its budget is exhausted.

Each execution flow $F_{i,j} \in \mathcal{F}_i$ is thus a DAG of sub-tasks, i.e. $F_{i,j} = \langle \mathcal{V}_{i,j}, \mathcal{E}_{i,j} \rangle$, where $\mathcal{V}_{i,j}$ is a set of $n_{i,j}$ nodes and $\mathcal{E}_{i,j}$ is a set of directed edges. Each node $v_k \in \mathcal{V}_{i,j}$ represents a sub-task that executes sequentially and is characterized by a worst-case execution time (WCET) denoted by c_k. For ease of understanding, we assume that every sub-task v_k executes for *exactly* c_k time units (we will further discuss this assumption at the end of Section 4). Each directed edge $(v_a, v_b) \in \mathcal{E}_{i,j}$ denotes a precedence constraint between the two sub-tasks v_a and v_b with the interpretation that successor v_b cannot start executing before predecessor v_a completes its execution. A sub-task is then said to be "ready-to-execute" (or simply "ready") if and only if all its predecessors have finished their execution. Fig. 3 illustrates our multi-DAG model for a task τ_i comprised of three execution flows. We further define some notations and terminology that will be extensively used throughout the paper.

Definition 1 (Workload). *The workload* $\mathrm{W}(F_{i,j})$ *of an execution flow* $F_{i,j}$ *of a task* τ_i *is defined as the cumulative amount of work*[3] *of all the sub-tasks in* $\mathcal{V}_{i,j}$*, i.e.*

$$\mathrm{W}(F_{i,j}) \stackrel{\text{def}}{=} \sum_{v_k \in \mathcal{V}_{i,j}} c_k$$

Definition 2 (Path). *In a given execution flow* $F_{i,j}$ *of a task* τ_i *, a path* $p = \{v_1, \ldots, v_k\}$ *is a sequence of sub-tasks* $\in \mathcal{V}_{i,j}$ *such that (1) there is an edge* $\in \mathcal{E}_{i,j}$ *connecting every two adjacent sub-tasks of* p *and (2) the first and the last sub-tasks* v_1 *and* v_k *in* p *are an entry and an exit node of the DAG* $F_{i,j}$*.*

Definition 3 (Critical path length). *For an execution flow* $F_{i,j}$ *of a task* τ_i*, the critical path length* $\mathrm{CP}(F_{i,j})$ *is defined as the cumulative execution requirement of the longest path*[4] *in* $F_{i,j}$*, i. e.*

$$\mathrm{CP}(F_{i,j}) \stackrel{\text{def}}{=} \max_{p \in F_{i,j}} \sum_{v_k \in p} c_k$$

In the next section, we show how to transform each $F_{i,j}$ in a synchronous DAG of servers and introduce a mapping rule to arbitrate the assignment of sub-tasks to servers at

[3]Note that the workload also gives the WCET of a DAG on a single-core platform.
[4]The critical path length can also be seen as the WCET of a DAG on a platform comprising an infinite number of cores.

run-time. We also define some properties to assert the correctness of that transformation, setting this way the basis for section 4 in which we will present a method that merges all the DAGs of servers defined for every execution flow $F_{i,j}$ into a single synchronous DAG of servers for each task τ_i.

3. PER-FLOW SERVER GRAPH

For each execution flow $F_{i,j}$ of every task τ_i, we derive a synchronous DAG of servers referred to as *synchronous server graph* (SSG) and is denoted by $F_{i,j}^{\mathrm{SSG}}$. Formally, we define an SSG as follows:

DEFINITION 4 (SSG). *A Synchronous Server Graph is a synchronous DAG of nodes (here the nodes are the servers) organized as a set $\{\sigma_1, \sigma_2, \ldots, \sigma_r\}$ of r segments. Each segment σ_ℓ with $\ell \in [1, r]$ is characterized by a pair $\langle b_\ell, q_\ell \rangle$, where q_ℓ is the number of servers in σ_ℓ and b_ℓ is the cpu-budget associated to each of these q_ℓ servers. Directed edges exist only between nodes of adjacent segments. Specifically, every node within a segment is connected to every node of the next segment (if any).*

Informally, the purpose of the method developed in this section is to be able to represent each execution flow of a given task τ_i as a synchronous DAG of servers such that, when τ_i takes one of its execution flows $F_{i,j}$ at run-time, the corresponding SSG $F_{i,j}^{\mathrm{SSG}}$ provides the required budget to finish the execution of all the sub-tasks of $F_{i,j}$ without violating any precedence constraint. This is an *intermediate* step in our approach; in the next section we develop a second step (based on this first one) that assigns a single synchronous DAG to *each task*, rather than one SSG for each flow.

The mechanism to handle these per-flow SSG at run-time works as follows: each segment of an SSG is a collection of servers whose budget is used to execute exclusively the ready sub-tasks of the execution flow from which the SSG has been derived. The servers are the entities to compete for, and to be scheduled on, the m cores by the scheduling algorithm of the operating system. Each time a server is granted a core, its budget is used to execute a ready sub-task. Each time a job of a task is released (and thus executes one of its execution flows), the first segment of the corresponding SSG "releases" all its servers, in the sense that they become ready to provide budget to the sub-tasks of that particular flow. Then, each of the subsequent segments releases all its servers only after all the servers from the previous segment have exhausted their budgets. That is, servers belonging to a segment σ_ℓ are allowed to provide cpu-budget to the sub-tasks of the dedicated execution flow only when all the servers from segment $\sigma_{\ell-1}$ have exhausted their budget. Since at some point in time we may have several sub-tasks from the same execution flow that are ready-to-execute and several servers in the corresponding SSG that are ready to provide budget, there must be a mapping rule to define which sub-task is granted budget from which server.

Firstly, we state the generic conditions that assert the *validity* of an SSG toward a given execution flow through Property 1. Then, we define a simple, yet efficient, mapping rule which is used throughout the paper to arbitrate the assignment of ready sub-tasks to servers. From that point onward, every time we refer to a valid SSG it implies that the mapping rule given by Definition 5 is enforced. Finally, we present an algorithm to construct an SSG for each execution flow and prove its correctness.

PROPERTY 1 (VALIDITY). *For a platform π, a scheduling algorithm A, a mapping rule R, and an execution flow $F_{i,j}$ of a task τ_i, an SSG $F_{i,j}^{\mathrm{SSG}}$ is said to be valid for $F_{i,j}$ according to R if and only if for any schedule of the servers of $F_{i,j}^{\mathrm{SSG}}$ produced by A on π, at run-time all the nodes of $F_{i,j}$ are guaranteed to be mapped by R to the server nodes of $F_{i,j}^{\mathrm{SSG}}$ in such a way that (1) all the dependencies between the nodes of $F_{i,j}$ are satisfied, and (2) $F_{i,j}$ receives the required budget to execute all its nodes.*

DEFINITION 5 (MAPPING RULE). *Let $F_{i,j}$ be an execution flow of a task τ_i and let $F_{i,j}^{\mathrm{SSG}}$ be the corresponding SSG constructed using Algorithm 1. A server $s_{\ell,x} \in \sigma_\ell \subseteq F_{i,j}^{\mathrm{SSG}}$, with $x \in [1, q_\ell]$, can execute a ready sub-task $v_k \in \mathcal{V}_{i,j}$ if and only if v_k has not been executed by a server $s_{\ell,y} \neq s_{\ell,x}$ such that $s_{\ell,y} \in \sigma_\ell$ as well.*

Algorithm 1: generateSSG($F_{i,j}$)

```
Input  : $F_{i,j}$ - An execution flow of task $\tau_i$
Output: $F_{i,j}^{\mathrm{SSG}}$ - An SSG for $F_{i,j}$
1  $F_{i,j}^{\mathrm{SSG}} \leftarrow \emptyset$ ;
2  while $\mathcal{V}_{i,j} \neq \emptyset$ do
3      $S^{\mathrm{curr}} \leftarrow \{v_k \in \mathcal{V}_{i,j} \mid \mathrm{pred}(v_k) = \emptyset\}$;
4      $C^{\min} \leftarrow \min\{c_k \mid v_k \in S^{\mathrm{curr}}\}$ ;
5      $F_{i,j}^{\mathrm{SSG}} \leftarrow F_{i,j}^{\mathrm{SSG}} \otimes \{\langle C^{\min}, |S^{\mathrm{curr}}|\rangle\}$ ;
6      foreach $v_k \in S^{\mathrm{curr}}$ do
7          $c_k \leftarrow c_k - C^{\min}$;
8          if $c_k = 0$ then
9              $\mathcal{V}_{i,j} \leftarrow \mathcal{V}_{i,j} \setminus \{v_k\}$ ;
10             $\mathcal{E}_{i,j} \leftarrow \mathcal{E}_{i,j} \setminus \{(v_k, *)\}$ ;
11         end
12     end
13 end
14 return $F_{i,j}^{\mathrm{SSG}}$;
```

The pseudo-code of the SSG creation algorithm is shown in Algorithm 1, whereas Fig. 4 depicts the resulting SSG[5] for the execution flow $F_{i,1}$ illustrated in Fig. 3. This algorithm takes an execution flow $F_{i,j}$ of task τ_i as input and outputs an SSG $F_{i,j}^{\mathrm{SSG}}$ for that flow, working as follows. The algorithm traverses the DAG $F_{i,j}$ by starting at its unique entry node (first iteration at line 3). At each iteration in the while loop, the algorithm adds a new segment at the end of $F_{i,j}^{\mathrm{SSG}}$ (line 5). The addition is represented by the operator $\otimes \langle b, q \rangle$ which appends a segment of q servers, each with a budget of b. This new segment has as many servers as there are sub-tasks with no predecessor(s) in $\mathcal{V}_{i,j}$ (i.e., ready sub-tasks) and each of these servers is assigned a budget equal to the minimum execution requirement among these sub-tasks (computed at line 4). The algorithm then proceeds by updating the DAG $F_{i,j}$ and "simulating" the execution of its sub-tasks within the created servers, i.e. for each sub-task with no predecessor, its execution time is decreased by $C^{\min}$ time units (line 7), thus reflecting its execution within that dedicated server. The number of servers per segment is basically tied to the number of sub-tasks that are guaranteed to be ready at that point in time, at run-time. Sub-tasks reaching zero execution requirement are removed from the input DAG $F_{i,j}$, as well as their respective outgoing edges

[5]As a coincidence, in this example, all segments of the SSG have unitary budgets although it may not be the case in general.

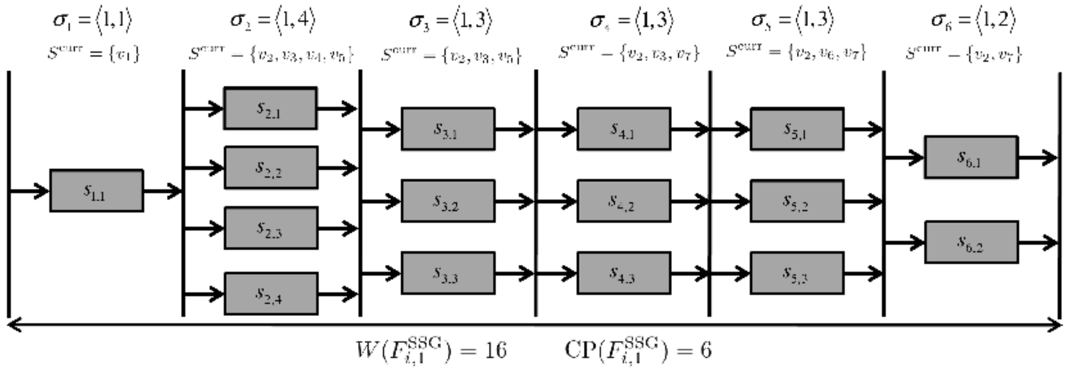
$\sigma_1 = \langle 1,1 \rangle$
$S^{\text{curr}} = \{v_1\}$
$\sigma_2 = \langle 1,4 \rangle$
$S^{\text{curr}} = \{v_2, v_3, v_4, v_5\}$
$\sigma_3 = \langle 1,3 \rangle$
$S^{\text{curr}} = \{v_2, v_3, v_5\}$
$\sigma_4 = \langle 1,3 \rangle$
$S^{\text{curr}} = \{v_2, v_3, v_7\}$
$\sigma_5 = \langle 1,3 \rangle$
$S^{\text{curr}} = \{v_2, v_6, v_7\}$
$\sigma_6 = \langle 1,2 \rangle$
$S^{\text{curr}} = \{v_2, v_7\}$
$s_{1,1}$
$s_{2,1}$
$s_{2,2}$
$s_{2,3}$
$s_{2,4}$
$s_{3,1}$
$s_{3,2}$
$s_{3,3}$
$s_{4,1}$
$s_{4,2}$
$s_{4,3}$
$s_{5,1}$
$s_{5,2}$
$s_{5,3}$
$s_{6,1}$
$s_{6,2}$
$W(F_{i,1}^{\text{SSG}}) = 16$
$\text{CP}(F_{i,1}^{\text{SSG}}) = 6$

Figure 4: SSG $F_{i,1}^{\text{SSG}}$ obtained by running Algorithm 1 with input $F_{i,1}$.

(lines 8–10). Algorithm 1 is guaranteed to terminate as $\mathcal{V}_{i,j}$ eventually becomes empty.

We now prove that the SSG output by Algorithm 1 is always valid (see 1) for its input execution flow.

LEMMA 1. $\text{W}(F_{i,j}^{\text{SSG}}) = \text{W}(F_{i,j})$

PROOF. At each iteration in the while loop in Algorithm 1, $|S^{\text{curr}}| \times C^{\min}$ units of workload are added to $F_{i,j}^{\text{SSG}}$ (at lines 5), and the same amount is then iteratively subtracted from $F_{i,j}$ (lines 6 and 7). From this and the while loop termination condition, the claim trivially holds. □

THEOREM 1. *The SSG $F_{i,j}^{\text{SSG}}$, obtained by running Algorithm 1 with input $F_{i,j}$, is valid for the execution flow $F_{i,j}$.*

PROOF. According to the validity property, we need to show that (a) all the dependencies in $F_{i,j}$ are preserved and (b) $F_{i,j}$ is provided the required budget to finish the execution of all its sub-tasks.

Proof of (a): It can be easily seen that all the precedence constraints in $F_{i,j}$ are preserved by construction since the servers are created for only those sub-tasks which are ready to execute (lines 3–5 in Algorithm 1), and the mapping rule ensures that no sub-task is assigned to more than one server within the same segment.

Proof of (b): Let us recall the run-time management mechanism of an SSG: all the servers within a segment of an SSG become "ready" to provide budgets only when all the servers from the previous segment have exhausted their budgets. Given this run-time mechanism, we prove by induction on the number of segments that no budget provided by the servers is wasted, i.e. all the servers of each segment use their entire budget to execute sub-tasks of $F_{i,j}$ that are ready-to-execute. Therefore, since at the end no budget is wasted and the total amount of budget provided by the SSG is equal to the workload of $F_{i,j}$ (from Lemma 1) the claim holds true. The detailed proof follows.

Base case. In the first iteration of the while loop, there is only one sub-task with no predecessor (remember that there is only one entry point to any execution flow) and thus only one server is created and added to $F_{i,j}^{\text{SSG}}$ at line 5. This server has a budget $C^{\min}$ equal to the WCET c_k of that sub-task (line 4), and at run-time this single server will provide budget to that single sub-task as soon as it is released, i.e. when $F_{i,j}$ is taken for execution. Hence, this first sub-task will execute entirely within the budget of that first server and no budget is wasted in this first segment. In addition, the algorithm "simulates" the completion of this first sub-task as it is removed from $F_{i,j}$ at line 9 and 10, implying that at the next iteration $\mathcal{V}_{i,j}$ will contain only the sub-tasks that have not completed yet.

Inductive step. Assume that at run-time, the ℓ'th segment just released all its servers and no budget has been wasted by the servers of all the previous segments. Also (as mentioned above), at the ℓ'th iteration of the while loop, $\mathcal{V}_{i,j}$ contains only the sub-tasks that have not completed yet and S^{curr} therefore contains the set of all the uncompleted sub-tasks that are ready-to-execute at the release of the servers of the ℓ'th segment. As seen in line 5, Algorithm 1 creates in segment σ_ℓ as many servers as there are ready sub-tasks, i.e. $|S^{\text{curr}}|$ servers are created in σ_ℓ. Each of these $|S^{\text{curr}}|$ servers is assigned a budget of $C^{\min}$, which corresponds to the minimum remaining WCET of all the ready sub-tasks. At run-time, the mapping rule guarantees that each one of the $|S^{\text{curr}}|$ ready sub-tasks will be allocated to one (and only one) of the $|S^{\text{curr}}|$ servers, and they will all execute for $C^{\min}$ time units, which is "simulated" at lines 6 and 7 of Algorithm 1. Here again, no budget is wasted in the ℓ'th segment and since the tasks that complete at the end of this segment are removed from $F_{i,j}$ at lines 8–10, at the next iteration of the while loop, $\mathcal{V}_{i,j}$ will once more contain only the uncompleted sub-tasks.

The algorithm terminates when $\mathcal{V}_{i,j}$ is empty, which means that there are no more uncompleted sub-tasks. In every segment, no budget has been wasted and since we have $\text{W}(F_{i,j}^{\text{SSG}}) = \text{W}(F_{i,j})$ by Lemma 1, it holds that all the sub-tasks of $F_{i,j}$ have been executed entirely. □

Note that upon applying Algorithm 1 to each execution flow of a task τ_i, we obtain a set of SSGs for that task, where each SSG is defined and proven valid for one of τ_i's execution flow. With that, we now describe how to construct a single synchronous DAG of servers for each *task* which accommodates all of its execution flows through its SSGs.

4. PER-TASK SERVER GRAPH

The algorithm presented in the previous section has paved the way for the second and final step of our approach. In this section, we present how to merge all the SSGs $F_{i,j}^{\text{SSG}}$, created for a task τ_i , into a single synchronous DAG of servers, called *"global synchronous server graph"* (GSSG) and denoted by $\mathcal{F}_i^{\text{GSSG}}$. Such a GSSG must ensure that every execution flow $F_{i,j}$ of task τ_i can be entirely executed within its servers, i.e. $\mathcal{F}_i^{\text{GSSG}}$ must be valid for every exe-

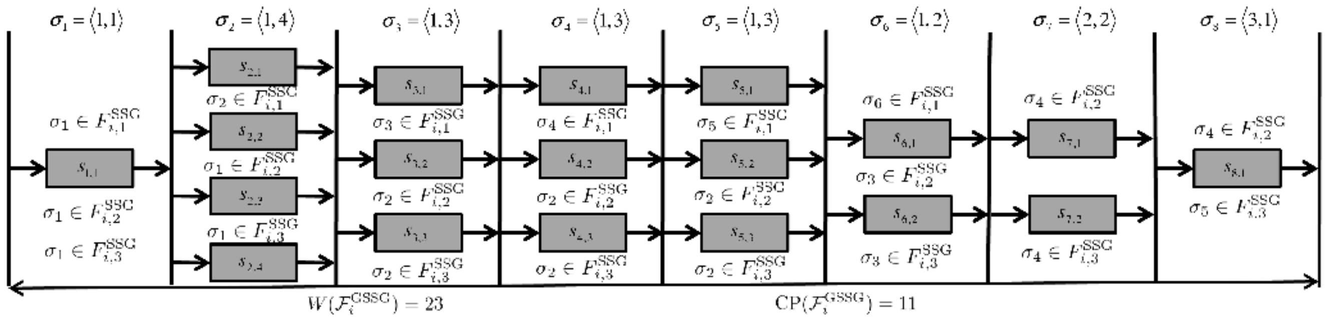
$\sigma_1 = \langle 1,1 \rangle$
$\sigma_2 = \langle 1,4 \rangle$
$\sigma_3 = \langle 1,3 \rangle$
$\sigma_4 = \langle 1,3 \rangle$
$\sigma_5 = \langle 1,3 \rangle$
$\sigma_6 = \langle 1,2 \rangle$
$\sigma_7 = \langle 2,2 \rangle$
$\sigma_8 = \langle 3,1 \rangle$
$s_{1,1}$
$\sigma_1 \in F_{i,1}^{\text{SSG}}$
$\sigma_1 \in F_{i,2}^{\text{SSG}}$
$\sigma_1 \in F_{i,3}^{\text{SSG}}$
$s_{2,1}$
$\sigma_2 \in F_{i,1}^{\text{SSG}}$
$s_{2,2}$
$\sigma_1 \in F_{i,2}^{\text{SSG}}$
$s_{2,3}$
$\sigma_1 \in F_{i,3}^{\text{SSG}}$
$s_{2,4}$
$s_{3,1}$
$\sigma_3 \in F_{i,1}^{\text{SSG}}$
$s_{3,2}$
$\sigma_2 \in F_{i,2}^{\text{SSG}}$
$s_{3,3}$
$\sigma_2 \in F_{i,3}^{\text{SSG}}$
$s_{4,1}$
$\sigma_4 \in F_{i,1}^{\text{SSG}}$
$s_{4,2}$
$\sigma_2 \in F_{i,2}^{\text{SSG}}$
$s_{4,3}$
$\sigma_2 \in F_{i,3}^{\text{SSG}}$
$s_{5,1}$
$\sigma_5 \in F_{i,1}^{\text{SSG}}$
$s_{5,2}$
$\sigma_2 \in F_{i,2}^{\text{SSG}}$
$s_{5,3}$
$\sigma_2 \in F_{i,3}^{\text{SSG}}$
$\sigma_6 \in F_{i,1}^{\text{SSG}}$
$s_{6,1}$
$\sigma_3 \in F_{i,2}^{\text{SSG}}$
$s_{6,2}$
$\sigma_3 \in F_{i,3}^{\text{SSG}}$
$\sigma_4 \in F_{i,2}^{\text{SSG}}$
$s_{7,1}$
$s_{7,2}$
$\sigma_4 \in F_{i,3}^{\text{SSG}}$
$\sigma_4 \in F_{i,2}^{\text{SSG}}$
$s_{8,1}$
$\sigma_5 \in F_{i,3}^{\text{SSG}}$
$W(\mathcal{F}_i^{\text{GSSG}}) = 23$
$\text{CP}(\mathcal{F}_i^{\text{GSSG}}) = 11$

Figure 5: GSSG $\mathcal{F}_i^{\text{GSSG}}$ obtained by running Algorithm 2 with input $F_{i,j}^{\text{SSG}} \in L$ where $j \in [1,2,3]$. The notation $\sigma_\ell \in F_{i,j}^{\text{SSG}}$ shows which segments of the SSGs are mapped to the different segments of $\mathcal{F}_i^{\text{GSSG}}$.

cution flow $F_{i,j}$ of τ_i. With that, state-of-the-art schedulability techniques and analysis that have been developed for DAG tasks can be straightforwardly applied over a new task set comprised of GSSGs (one derived for each task), as our approach transforms our multi-DAG model into a common parallel synchronous task model. That is, for each task τ_i , its set $\mathcal{F}_i$ of DAGs is executed within a single synchronous DAG $\mathcal{F}_i^{\text{GSSG}}$ (period and deadline are inherited). Note that a GSSG is still an SSG (as defined in Definition 4) and therefore all the definitions and properties presented in Section 3 remain in effect.

Algorithm 2: generateGSSG(L)

```
   Input  : $L$ - A list with a valid SSG $F_{i,j}^{\text{SSG}}$ for each
            execution flow $F_{i,j}$ of task $\tau_i$
   Output: $\mathcal{F}_i^{\text{GSSG}}$ - A GSSG for task $\tau_i$
1  $\mathcal{F}_i^{\text{GSSG}} \leftarrow \emptyset$ ;
2  while $L \neq \emptyset$ do
3      $B^{\min} \leftarrow \infty$ ;
4      $Q^{\max} \leftarrow 0$ ;
5      foreach $F_{i,j}^{\text{SSG}} \in L$ do
6          $\sigma_{i,j}^{\text{curr}} \leftarrow \{\sigma_\ell \in F_{i,j}^{\text{SSG}} \mid \text{pred}(\sigma_\ell) = \emptyset\}$;
7          if $B^{\min} > b_{i,j}^{\text{curr}}$ then $B^{\min} \leftarrow b_{i,j}^{\text{curr}}$ ;
8          if $Q^{\max} < q_{i,j}^{\text{curr}}$ then $Q^{\max} \leftarrow q_{i,j}^{\text{curr}}$ ;
9      end
10     $\mathcal{F}_i^{\text{GSSG}} \leftarrow \mathcal{F}_i^{\text{GSSG}} \otimes \{\langle B^{\min}, Q^{\max} \rangle\}$ ;
11     foreach $F_{i,j}^{\text{SSG}} \in L$ do
12         if $b_{i,j}^{\text{curr}} - B^{\min} = 0$ then $F_{i,j}^{\text{SSG}} \leftarrow F_{i,j}^{\text{SSG}} \setminus \sigma_{i,j}^{\text{curr}}$ ;
13         else $\sigma_{i,j}^{\text{curr}} \leftarrow (b_{i,j}^{\text{curr}} - B^{\min}, q_{i,j}^{\text{curr}})$ ;
14         if $F_{i,j}^{\text{SSG}} = \emptyset$ then $L \leftarrow L \setminus \{F_{i,j}^{\text{SSG}}\}$ ;
15     end
16 end
17 return $\mathcal{F}_i^{\text{GSSG}}$ ;
```

Algorithm 2 shows the pseudo-code of the GSSG creation algorithm, whereas Fig. 5 depicts the resulting GSSG for task τ_i once provided its execution flows have been converted into SSGs by Algorithm 1, as exemplified in Fig. 4. For a given task τ_i , this algorithm takes as input the SSGs $F_{i,j}^{\text{SSG}}$ derived by Algorithm 1 for each of its execution flows $F_{i,j}$, and outputs a unique GSSG $\mathcal{F}_i^{\text{GSSG}}$ that can accommodate all the referred flows, working as follows. The algorithm keeps on iterating in the while loop at line 2 until the list L of SSGs given as input is empty. At each of these iterations, a new segment of servers is added to the output GSSG (line 10). This new segment is composed of $Q^{\max}$ servers, each with a budget of $B^{\min}$. These two parameters $B^{\min}$ and $Q^{\max}$ are computed in lines 3–9. Specifically, line 6 records in $\sigma_{i,j}^{\text{curr}}$ the segment from every input $F_{i,j}^{\text{SSG}}$ that has no predecessors, therefore implicitly providing the pair $(b_{i,j}^{\text{curr}}, q_{i,j}^{\text{curr}})$. Here $b_{i,j}^{\text{curr}}$ is the remaining budget of any of the $q_{i,j}^{\text{curr}}$ servers. Note that by the definition of an SSG (see Definition 4) all the servers within a segment will always have the same initial budget and only the servers of one segment are ready at any time instant. On line 7, $B^{\min}$ is set to the minimum remaining budget of all these servers in the "not-yet-processed" segment ($\sigma_{i,j}^{\text{curr}}$) of all SSGs and $Q^{\max}$ records the maximum number of servers in all these segments.

The algorithm then updates all the SSGs (lines 11–13). For every SSG $F_{i,j}^{\text{SSG}}$, if the remaining budget of all the servers in the "not-yet-processed" segment $\sigma_{i,j}^{\text{curr}}$ is equal to $B^{\min}$ then all the servers of that segment are now considered as "processed", as $Q^{\max} \geq q_{i,j}^{\text{curr}}$ servers of budget $B^{\min}$ have been added to the output GSSG at line 10. All these servers are thus removed from their SSG $F_{i,j}^{\text{SSG}}$, and here we can see that the next iteration of the foreach loop (lines 5–9) will again give at line 6 a $\sigma_{i,j}^{\text{curr}}$ for each $F_{i,j}^{\text{SSG}}$ equal to its next segment. Otherwise, if the remaining budget of all the servers in the "not-yet-processed" segment $\sigma_{i,j}^{\text{curr}}$ is higher than $B^{\min}$ (it cannot be lower by definition of $B^{\min}$ at line 7), then these remaining budgets are simply decremented by $B^{\min}$ units of workload. At line 14, if all the servers have been removed from $F_{i,j}^{\text{SSG}}$ then this SSG is removed from the list L.

We now prove that Algorithm 2 produces a valid GSSG for task τ_i. To do so, we define two operations called *splitting* and *expanding* that transform an input SSG derived from Algorithm 1 into another SSG, and we show that both operations preserve the validity of the input SSG. Then, we prove in Theorem 2 that the output GSSG $\mathcal{F}_i^{\text{GSSG}}$ from Algorithm 2 can always be obtained from any of its input SSGs $F_{i,j}^{\text{SSG}}$ by applying a sequence of splitting and expanding operations and therefore $\mathcal{F}_i^{\text{GSSG}}$ is also valid for all the execution flows $F_{i,j}$ of task τ_i.

DEFINITION 6 (SPLITTING OPERATION). *A splitting operation replaces any segment $\sigma_\ell = (b_\ell, q_\ell)$ of an SSG $F_{i,j}^{\text{SSG}}$ with a list of consecutive segments $(\sigma_\ell^1, \sigma_\ell^2, \ldots, \sigma_\ell^r)$ such that $\forall k \in [1,r]$ it holds that $\sigma_\ell^k = \langle b_\ell^k, q_\ell^k \rangle$, where*

$$q_\ell^k = q_\ell \quad (1)$$

$$\sum_{k=1}^{r} b_\ell^k = b_\ell \quad (2)$$

Lemma 2. *Let $F_{i,j}^{\text{SSG}}$ be a valid SSG derived by Algorithm 1 for a given execution flow $F_{i,j}$ of task τ_i and let $F_{i,j}^{\text{SSG}'}$ be the SSG obtained from $F_{i,j}^{\text{SSG}}$ after applying an arbitrary series of splitting operations on the segments of $F_{i,j}^{\text{SSG}}$. It holds that $F_{i,j}^{\text{SSG}'}$ is also valid for $F_{i,j}$. That is, the splitting operation preserves the validity of the original SSG $F_{i,j}^{\text{SSG}}$.*

Proof. In Theorem 1, the validity of the SSG $F_{i,j}^{\text{SSG}}$ obtained by Algorithm 1 for the execution flow $F_{i,j}$ is proven by showing that (i) $\mathrm{W}(F_{i,j}^{\text{SSG}}) = \mathrm{W}(F_{i,j})$ and (ii) no budget of $F_{i,j}^{\text{SSG}}$ is wasted at run-time. Regarding the workload of the SSG $F_{i,j}^{\text{SSG}'}$, for any segment $\sigma_\ell \in F_{i,j}^{\text{SSG}}$ that has been broken into a series $(\sigma_\ell^1, \sigma_\ell^2, \ldots, \sigma_\ell^r) \in F_{i,j}^{\text{SSG}'}$, it holds from Eq. 2 that $\sum_{k=1}^{r} b_\ell^k = b_\ell$ and since from Eq. 1 $q_\ell^k = q_\ell$ for all $k \in [1, r]$, it is easy to see that $\mathrm{W}(F_{i,j}^{\text{SSG}'}) = \mathrm{W}(F_{i,j}^{\text{SSG}}) = \mathrm{W}(F_{i,j})$.

Second, it is shown in Theorem 1 that every segment of $F_{i,j}^{\text{SSG}}$ has as many servers as the number of sub-tasks that will be ready-to-execute at run-time when the segment will be allowed to provide budget. Therefore, since no budget of $F_{i,j}^{\text{SSG}}$ is wasted, for any segment $\sigma_\ell \in F_{i,j}^{\text{SSG}}$ that has been broken into a series $(\sigma_\ell^1, \sigma_\ell^2, \ldots, \sigma_\ell^r) \in F_{i,j}^{\text{SSG}'}$, there will be exactly q_ℓ ready sub-tasks of remaining WCET $\geq b_\ell$ competing for these q_ℓ servers of budget b_ℓ. From Eq. 1 and 2, and by the mapping rule, there will also be q_ℓ ready-sub tasks competing for the q_ℓ servers of every segment σ_ℓ^k, with $k \in [1, r]$, and such that at every segment σ_ℓ^k the remaining WCET of these q_ℓ ready sub-tasks will be $\geq b_\ell^k$. As a result, no budget will ever be wasted in these new segments σ_ℓ^k, $k \in [1, r]$. □

Definition 7 (Expanding operation). *An expanding operation consists in supplying any segment $\sigma_\ell = (b_\ell, q_\ell)$ of an SSG $F_{i,j}^{\text{SSG}}$ with an arbitrary number of extra servers of budget b_ℓ.*

Lemma 3. *Let $F_{i,j}^{\text{SSG}}$ be a valid SSG for a given execution flow $F_{i,j}$ of task τ_i obtained by Algorithm 1 (and possibly after an arbitrary series of splitting operations). Let $F_{i,j}^{\text{SSG}'}$ be the SSG obtained from $F_{i,j}^{\text{SSG}}$ after applying an arbitrary series of expanding operations on the segments of $F_{i,j}^{\text{SSG}}$. It holds that $F_{i,j}^{\text{SSG}'}$ is also valid for $F_{i,j}$.*

Proof. Adding new servers to a segment $\sigma_\ell \in F_{i,j}^{\text{SSG}}$ leads to $\mathrm{W}(F_{i,j}^{\text{SSG}'}) > \mathrm{W}(F_{i,j}^{\text{SSG}})$ and at run-time q_ℓ becomes greater than the number of ready sub-tasks at segment σ_ℓ. However, the mapping rule of Definition 5 enforces that no sub-task is assigned to two different servers within a same segment. As a consequence the extra servers will simply be ignored at run-time, and the precedence constraints between the sub-tasks of $F_{i,j}$ will still be satisfied as the order of execution of the sub-tasks remains unchanged. In short, the budget of all the servers added to $F_{i,j}^{\text{SSG}'}$ will be entirely wasted and thus the validity is preserved. □

Theorem 2. *The GSSG $\mathcal{F}_i^{\text{GSSG}}$ obtained by running Algorithm 2 is valid for task τ_i as it is valid for every of its execution flows $F_{i,j}$.*

Proof. The proof is a direct consequence of Lemmas 2 and 3, and the fact that the output $\mathcal{F}_i^{\text{GSSG}}$ of Algorithm 2 can be obtained from every input $F_{i,j}^{\text{SSG}} \in L$ by applying a series of splitting and expanding operations. Let $F_{i,j}^{\text{SSG}}$ be *any* of the input SSGs and let $\sigma_{i,j}^{\text{curr}}$ be its current segment with no predecessor (line 6). By definition of $B^{\min}$ at line 7, we have $b_{i,j}^{\text{curr}} \geq B^{\min}$ and by definition of $Q^{\max}$ at line 8, we have $q_{i,j}^{\text{curr}} \leq Q^{\max}$. Therefore, the addition of a new segment of $Q^{\max}$ servers of budget $B^{\min}$ at line 10 (alongside the corresponding reduction of $B^{\min}$ units at line 13) can be seen as a splitting operation performed on segment $\sigma_{i,j}^{\text{curr}}$. Also, if $q_{i,j}^{\text{curr}} < Q^{\max}$ then the addition of $Q^{\max}$ servers can be seen as an expanding operation. Finally, note that if $F_{i,j}^{\text{SSG}}$ is not the last SSG to be removed from L at line 14, then the addition of extra segments to the output $\mathcal{F}_i^{\text{GSSG}}$ in the next iterations can also be seen as applying an expanding operation on arbitrary empty segments. □

As a last result, the following corollary holds true from the validity of the GSSGs created by Algorithm 2. The corollary states that the schedulability of the original task set composed of parallel tasks with multiple execution flows can now be assessed by applying any existing schedulability test (the only restriction on the scheduling algorithm is to be work-conserving) over the set of derived GSSGs as long as the test conforms with the final parallel model. The results from [9, 18] are examples of such tests.

Corollary 1. *If a valid GSSG $\mathcal{F}_i^{\text{GSSG}}$ is deemed schedulable by a schedulability test of a scheduling algorithm A, so does the task τ_i from which it was derived.*

We shall now briefly discuss the assumption made in Section 2, according to which every sub-task v_k executes for exactly c_k time units. It is certain that at run-time the sub-tasks will most of the time execute for less than their WCET and unfortunately our methodology is not sustainable in these circumstances (we omit the proof here due to space limitation). Nevertheless, the results obtained in this paper can be made sustainable w.r.t. WCET through the addition of run-time mechanisms. For example, consider the following mechanism: when a sub-task v_k completes earlier than indicated by its WCET c_k, the mechanism checks the GSSG of the corresponding task and determines in how many segments, say r, that sub-task v_k was supposed to complete its execution. Then, the mechanism immediately locks all the successor sub-tasks of v_k (thus preventing them to become ready) for the next r segments. This way, the system behaves like if all the sub-tasks execute for their WCET.

5. DISCUSSION AND CONCLUSIONS

We firmly believe that it has nowadays become crucial to investigate new models and techniques to schedule contemporary applications subject to real-time requirements, especially given the new trend towards parallelization to achieve higher performance. The theoretical results and investigations carried out in this work should be seen as a step in that direction. Some of the practical concerns and implementation details have been set aside in this work, but will be the focus of our future work.

We have presented a technique to compute a single DAG of servers $\mathcal{F}_i^{\text{GSSG}}$ for a task τ_i with different execution flows, and showed that these servers are able to supply every execution flow of τ_i with the required cpu-budget so that the task can execute entirely, irrespective of the execution flow taken at run-time. Therefore, the multi-DAG parameter $\mathcal{F}_i$ assumed in the task model can be replaced for its corresponding GSSG $\mathcal{F}_i^{\text{GSSG}}$, while the period and the deadline remain unchanged. With this, there is no need to consider every feasible interference scenario between all combinations of execution flows of all the tasks in order to derive a schedulability test based on the internal structure of the tasks, as $\mathcal{F}_i^{\text{GSSG}}$ naturally upper-bounds the on-core interference that a task τ_i causes on the other tasks. Moreover, a GSSG is a special case of the synchronous parallel task model, which in turn is a special case of the DAG model. Therefore, existing multi-core scheduling techniques suited for any of these classes of parallel tasks can be leveraged to ascertain the schedulability of a task set modeled as discussed in this work. Additionally, note that a GSSG has a very peculiar structure (fair progression and synchronous behavior), which offers opportunities for new theoretical analysis.

From a schedulability point of view, current scheduling techniques for parallel tasks can be broadly categorized into two categories: decomposition method and direct analysis. In decomposition method, each sub-task of a DAG is assigned an intermediate offset and a deadline based on the structure of the DAG. With this, each sub-task can be treated as an individual sequential task. The parallel task scheduling problem then reduces to the traditional sequential task schedulability problem on a multiprocessor system, for which there is a plethora of scheduling algorithms and schedulability tests in the literature. In direct analysis, schedulability conditions are derived directly from the properties of the DAG. Some analysis techniques consider the precedence constraints on the DAG to study the execution requirements at different time instants, whereas others simply rely on the workload and critical path length values to create a synthetic worst-case scenario that upper-bounds the interference. For the latter case, our contribution brings no benefit since we end up increasing the maximum workload of the task. However, it has been shown in [21] that considering the internal structure of a DAG (as we do in this work) may improve the schedulability tests. Hence, for all the other cases which rely on the internal structure of the DAG (including the decomposition methods) our contribution directly enables the application of such schedulability analysis methods to generic real-time parallel applications with conditional execution without having to assume that all the sub-tasks of every flow must execute.

Although the SSGs output by the algorithms presented in this paper retain optimal critical path length values, we provide no proofs due to space constraints. Regarding the final workload values, it is worth noting that Algorithm 2 can be further improved with respect to tightening of the GSSG's workload. However, we chose to present this algorithm in its simplest form for ease of understanding and proving the validity of its output. As a final remark, it is also worth mentioning that there exists a trade-off between the critical path length and the workload of the GSSG output by Algorithm 2, in the sense that it is sometimes possible to reduce its workload by increasing its critical path length, and vice versa, while preserving its validity. Besides evaluating our approach through extensive experiments in terms of schedulability gains comparatively to existing techniques that neglect the internal structure of the tasks, our future work will also explore and try to exploit this trade-off in order to influence the interference between tasks by fine-tuning these two parameters, thereby reducing the worst-case response time of some tasks to improve the system schedulability.

Acknowledgments

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project(s) FCOMP-01-0124-FEDER-037281 (CISTER), FCOMP-01-0124-FEDER-020447 (REGAIN); by the European Union, under the Seventh Framework Programme (FP7/2007-2013), grant agreement nr. 611016 (P-SOCRATES).

6. REFERENCES

[1] B. Andersson and D. de Niz. Analyzing global-edf for multiprocessor scheduling of parallel tasks. In *Principles of Distributed Systems*, volume 7702 of *Lecture Notes in Computer Science*, pages 16–30. 2012.

[2] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Dobel, and H. Hartig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *ECRTS*, pages 215–224, July 2013.

[3] S. Baruah. Improved multiprocessor global schedulability analysis of sporadic dag task systems. In *ECRTS*, pages 97–105, July 2014.

[4] S. Baruah, J. Goossens, and G. Lipari. Implementing constant-bandwidth servers upon multiprocessor platforms. In *RTAS*, pages 154–163, 2002.

[5] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese. A generalized parallel task model for recurrent real-time processes. In *RTSS*, pages 63–72, 2012.

[6] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, 1999.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[8] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese. Feasibility analysis in the sporadic dag task model. In *ECRTS*, pages 225–233, July 2013.

[9] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin. Global edf schedulability analysis for synchronous parallel tasks on multicore platforms. In *ECRTS*, pages 25–34, July 2013.

[10] R. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Survey*, 43(4):35:1–35:44, Oct. 2011.

[11] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, Oct. 2011.

[12] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 1–34, 1997.

[13] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *Cyber-Physical Systems (ICCPS), 2013 ACM/IEEE International Conference on*, pages 31–40, April 2013.

[14] K. Lakshmanan, S. Kato, and R. R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *RTSS*, pages 259–268, 2010.

[15] J. Li, K. Agrawal, C. Lu, and C. D. Gill. Analysis of global EDF for parallel tasks. In *ECRTS*, pages 3–13, 2013.

[16] J. Li, J.-J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *ECRTS*, July 2014.

[17] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[18] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho. Response-time analysis of synchronous parallel tasks in multiprocessor systems. In *RTNS*, pages 3–12, 2014.

[19] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *ECRTS*, pages 321–330, July 2012.

[20] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, 2013.

[21] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet. Global edf scheduling of directed acyclic graphs on multiprocessor systems. In *RTNS*, pages 287–296, 2013.

[22] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *RTSS*, pages 217–226, 2011.

[23] L. Sha, T. Abdelzaher, K. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.